Discovering Significant Communities on Bipartite Graphs: An Index-based Approach

Journal:	*Transactions on Knowledge and Data Engineering*
Manuscript ID	TKDE-2020-12-1465
Manuscript Type:	Regular
Keywords:	Bipartite graph, Cohesive subgraph, Community search, Indexing

Cover Letter

Discovering Significant Communities on Bipartite Graphs: An Index-based Approach

Kai Wang, Shuting Wang, Wenjie Zhang, Ying Zhang, Lu Qin and Yuting Zhang

Dear Editor and Fellow Reviewers,

We are writing to submit our paper "Discovering Significant Communities on Bipartite Graphs: An Index-based Approach" to the prestigious journal, IEEE TKDE, for possible publication.

This submission is a substantial extension of the conference version "Efficient and Effective Community Search on Large-scale Bipartite Graphs" which is accepted by the International Conference on Data Engineering 2021 (IEEE ICDE 2021).

We summarize the major new materials in this submission as follows:

1. We have updated the abstract, introduction, related work, and conclusion for the new materials.
2. We have added new Section 3.3 to discuss the *size-aware indexing techniques* to handle the scenarios where a (memory) space budget is given. We have analyzed and compared the space complexity of the proposed indexes, and the new size-aware index can be efficiently constructed and fitted into a limited space.
3. We have added an entirely new Section 4 to discuss the *index maintenance for dynamic graphs* which is commonly encountered in real applications. When graphs are updated dynamically (i.e., vertices/edges are inserted or removed), it is inefficient to reconstruct the indexes from scratch. We have proposed new incremental algorithms to handle the edge insertion/deletion cases which are very efficient since they do not need to access and update most of the index entries.
4. In Section 6, we have added new experiments to evaluate the size-aware index techniques and index maintenance algorithms. The results of the evaluation of the size-aware indexing techniques are shown in Fig-12. The efficiency and the scalability of the new index maintenance algorithms are evaluated in Fig-13.

Please kindly let us know if further information is required. Thank you for your valuable reviews and comments!

Best Regards,

Kai Wang, Shuting Wang, Wenjie Zhang, Ying Zhang, Lu Qin, and Yuting Zhang

Discovering Significant Communities on Bipartite Graphs: An Index-based Approach

Kai Wang, Shuting Wang, Wenjie Zhang, Ying Zhang, Lu Qin, and Yuting Zhang

Abstract—Bipartite graphs are widely used to model relationships between two types of entities. Community search retrieves densely connected subgraphs containing a query vertex, which has been extensively studied on unipartite graphs. However, it remains largely unexplored on bipartite graphs. Moreover, all existing cohesive subgraph models on bipartite graphs only measure the structure cohesiveness while overlooking the edge weight. In this paper, we study the significant (α, β)-community search problem on weighted bipartite graphs. Given a query vertex q, we aim to find the significant (α, β)-community $\mathcal{R}$ of q which adopts (α, β)-core to characterize the engagement level of vertices, and maximizes the minimum edge weight (significance) within $\mathcal{R}$. To support fast retrieval of $\mathcal{R}$, we first obtain the maximal connected subgraph of (α, β)-core containing q (the (α, β)-community), and the search space is limited to this subgraph with a much smaller size than the original graph. A novel index structure is presented to support retrieving the (α, β)-community in optimal time. Efficient index maintenance techniques are also proposed to handle dynamic graphs. To further obtain $\mathcal{R}$, we develop peeling and expansion algorithms. The experimental results on real graphs validate the effectiveness and efficiency of our proposed techniques.

Index Terms—Bipartite graph, Cohesive subgraph, Community search, Indexing

1 Introduction

In many real-world applications, relationships between two different types of entities are modeled as bipartite graphs, such as customer-product networks [1], user-page networks [2] and collaboration networks [3]. Community structures naturally exist in these practical networks and *community search* has been extensively explored and proved useful on unipartite graphs [4], [5], [6], [7], [8], [9], [10], [11]. Given a query vertex q, *community search* aims to find communities (connected subgraphs) containing q which satisfy specific cohesive constraints. In the literature, fair clustering methods [12], [13], [14] are used to find communities (i.e., clusters) under fairness constraints on bipartite graphs. However, they aim to find a set of clusters under a global optimization goal and do not aim to search a personalized community for a specific user. Nevertheless, no existing work has studied the *community search* problem on bipartite graphs. On bipartite graphs, various dense subgraph models are designed (e.g., (α, β)-core [15], [16], bitruss [17], [18], [19] and biclique [20]) which can be used as the cohesive measurement of a community. However, simply applying these cohesive measurements only ensures the structure cohesiveness of communities but ignores another important characteristic, the weight (or significance) of interactions between the two sets of vertices. For example, (α, β)-core is defined as the maximal subgraph where each vertex in upper layer has at least α neighbors and each vertex in lower layer has at least β neighbors. In the customer-movie network shown in Figure 1, each edge has a weight denoting the rating of a user to a movie. If the (α, β)-core model is applied to search a community of "Eric", e.g., the maximal connected subgraph of (3, 2)-core containing "Eric", we will get the community formed by the four users and the five movies on the left side. Note that, this community includes "Alien" (not liked by "Andy" or "Kane") and "Taylor" (who has less interest in this genre of movies).

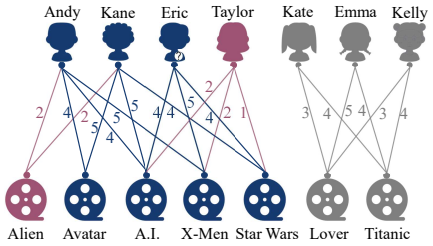


Fig. 1: A user-movie network

In this paper, we study the significant community search problem on weighted bipartite graphs, which is the first to study community search on bipartite graphs. Here, in a weighted bipartite graph G, each edge is associated with an edge weight. In addition, the weight (significance) of a community is measured by the minimum edge weight in it. A community with a high weight value indicates that every edge in the community represents a highly significant interaction. We propose the significant (α, β)-community model, which is the maximal connected subgraph containing the query vertex q that satisfies the vertex degree constraint from (α, β)-core, and has the highest graph significance. The intuition behind the new significant (α, β)-community model is to capture structure cohesiveness as well as interactions (edges) with high significance. In addition, if we

- *Kai Wang, Wenjie Zhang and Yuting Zhang are with the University of New South Wales, Australia. E-mail: kai.wang@unsw.edu.au, zhangw@cse.unsw.edu.au, yutingz@cse.unsw.edu.au.*
- *Shuting Wang is with Zhejiang Gongshang University, China. Email: wangshuting@zjgsu.edu.cn.*
- *Ying Zhang and Lu Qin are with the University of Technology Sydney, Australia. Email: ying.zhang@uts.edu.au, lu.qin@uts.edu.au.*

Manuscript received Jan, 2021; revised XXXX, 2021.
Kai Wang and Shuting Wang are the joint first authors.

maximize the weight value under given α and β, we can find the most significant subgraph while preserving the structure cohesiveness. For example, in Figure 1, the subgraph in blue color, which excludes "Alien" and "Taylor", is the significant (3, 2)-community of "Eric".

Applications. Finding the significant (α, β)-community has many applications and we list some of them below.

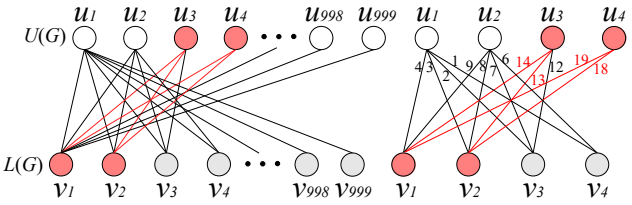
• *Personalized Recommendation.* In user-item networks, users leave reviews for items with ratings. Examples include viewer-movie network in `IMDB`, reader-book network in `goodreads`. The platforms can utilize the significant (α, β)-community model to provide personalized recommendations. For example, based on the community found in Figure 1, we can put the people who give common high ratings ("Andy" and "Kane") on the recommended friend list of the query user ("Eric"). We can also recommend the movie ("Avatar") which the user is likely to be interested in to the query user ("Eric").

• *Fraud Detection.* In e-commerce platforms such as Amazon and Alibaba, customers and items form a customer-item bipartite graph in which an edge represents a customer purchased an item, and the edge weight measures the number of purchases or the total transaction amount. Fraudsters and the items they promote are prone to form cohesive subgraphs [15], [17]. Since the cost of opening fake accounts is increased with the improvement of fraud detection techniques, frauds cannot rely on many fake accounts [2]. Thus, the number of purchases or the total transaction amount per account is increased. Given a suspicious item or customer as the query vertex, our significant (α, β)-community model allows us to find the most suspicious fraudsters and related items in the customer-item bipartite graphs.

• *Team Formation.* In a bipartite graph formed by developers and projects, an edge between a developer and a project indicates that the developer participates in the project, and the edge weight shows the corresponding contribution (e.g., number of tasks accomplished). A developer may wish to assemble a team with a proven track record of contributions in related projects, which can be supported by a significant (α, β)-community search over the bipartite graph.

Challenges. To obtain the significant (α, β)-community, we can iteratively remove the vertices without enough neighbors and the edges with small weights from the graph. However, when the graph size is large and there are many vertices and edges that need to be removed, this approach is inefficient. For example, Figure 2(a) shows the graph G with 2,003 edges. We need to remove 1,999 edges from G to get the significant $(2, 2)$-community of u_3 with only 4 edges.

In this paper, we focus on indexing-based approaches. Our intuition is to reduce the search space of the query algorithms by indexing necessary results. A straightforward idea is precomputing all the significant (α, β)-communities for all α, β, and q combinations. This idea is impractical since both structure cohesiveness and significance need to be considered. For different query vertex q and α, β values, the significant (α, β)-communities can be different and there does not exist hierarchical relationships among them. Therefore, we resort to a two-step approach. In the first step, we observe that the (α, β)-community always contains the significant (α, β)-community for a query vertex q. Here, (α, β)-community is the maximal connected subgraph containing



(a) A weighted bipartite graph G, $w(u, v) = 5 \cdot u.id - v.id$. (b) The (2, 2)-community of u_3 in G.

Fig. 2: An example graph, the significant $(2, 2)$-community of u_3 is marked in red color

q in the (α, β)-core (without considering the edge weights). For example, Figure 2(b) shows the $(2, 2)$-community of u_3 which contains the significant $(2, 2)$-community of u_3 and is much smaller than the original graph G. Therefore, we try to index all (α, β)-communities and use the one containing q as the starting point when querying w.r.t. q. In the second step, we compute the significant (α, β)-community based on the (α, β)-community obtained in the first step. To make our ideas practically applicable, we need to address the following challenges.

1) How to build an index to cover all (α, β)-communities.
2) How to bound the index size and the indexing time.
3) How to efficiently maintain the index when the vertices/edges are dynamically updated.
4) How to efficiently obtain the significant (α, β)-community from the (α, β)-community of a vertex q.

Our approaches. To address Challenge 1, we first propose the index I_{bs}^{α} to store all the (α, β)-communities. It is observed that the model of (α, β)-core has a hierarchical property. In other words, (α, β)-core $\subseteq$ (α', β')-core if $\alpha \geq \alpha'$ and $\beta \geq \beta'$. Motivated by this observation, all the $(1, \beta)$-community with $\beta \geq 1$ can be organized hierarchically in the $(1, 1)$-core. Then, when querying a $(1, \beta)$-community with $\beta \geq 1$, we only need to take the vertices and edges in this community using breath-first search. By organizing all the $(\alpha, 1)$-cores where $\alpha \in [1, \alpha_{max}]$, I_{bs}^{α} can cover all the (α, β)-communities. Similarly, we can also build the index I_{bs}^{β} which stores all the $(1, \beta)$-core where $\beta \in [1, \beta_{max}]$ to cover all the (α, β)-communities. Here α_{max} and β_{max} are the maximal valid α and β values in G respectively.

Reviewing I_{bs}^{α} and I_{bs}^{β}, we observe that $I_{bs}^{\alpha}(I_{bs}^{\beta})$ can be very large when high degree vertices exist in $U(G)(L(G))$. For example, I_{bs}^{α} needs to store 999 copies of neighbors of u_1 since u_1 is contained in $(999, 1)$-core. The same issue occurs when I_{bs}^{β} stores v_1's neighbors. To handle this issue and address Challenge 2, we further propose the degeneracy-bounded index I_{δ}. Here, the degeneracy (δ) is the largest number where the (δ, δ)-core is nonempty in G. Note that for each nonempty (α, β)-core (or (α, β)-community), we must have $min(\alpha, \beta) \leq \delta$. This is because it contradicts the definition of δ if an (α, β)-core with $\alpha > \delta$ and $\beta > \delta$ exists. In addition, according to the hierarchical property of the (α, β)-core model, all (α, β)-communities with $\alpha \leq \beta$ can be organized in the (α, α)-core and all (α, β)-communities with $\beta < \alpha$ can be organized in the (β, β)-core. In this manner, I_{δ} only needs to store all the (τ, τ)-cores for each $\tau \in [1, \delta]$ to cover all the (α, β)-communities. For example, in Figure 2, unlike I_{bs}^{α} which needs to store $(1, 1)$-core to $(999, 1)$-core, I_{δ} only needs to store $(1, 1)$-core, $(2, 2)$-core and $(3, 3)$-core since δ=3. Since the size of each (τ, τ)-core ($\tau \in [1, \delta]$) is

bounded by $O(m)$, I_δ can be built in $O(\delta \cdot m)$ time and takes $O(\delta \cdot m)$ space to index all the (α, β)-communities. We also propose a size-aware index whose size can be adjusted under a given space budget.

To address Challenge 3, we propose effective index maintenance techniques to handle scenarios where graphs are dynamically updated. Specifically, we discuss how the index I_δ is updated when inserting or deleting an edge. We theoretically show that all the affected vertices are limited in a small subgraph and techniques are devised to update the affected index entries efficiently.

To address Challenge 4, after retrieving the (α, β)-community $C_{\alpha,\beta}(q)$, we first propose the peeling algorithm SCS-Peel which iteratively removes the edge with the minimal weight from $C_{\alpha,\beta}(q)$ to obtain $\mathcal{R}$. Observing that $\mathcal{R}$ can be much smaller than $C_{\alpha,\beta}(q)$ in many cases, we also propose the expansion algorithm SCS-Expand which iteratively adds the edge with maximal weights into an empty graph until $\mathcal{R}$ is found. In SCS-Expand, we derive several rules to avoid excessively validating $\mathcal{R}$.

Contribution. Our main contributions are listed as follows.

- We propose the model of significant (α, β)-community which is the first to study community search problem on (weighted) bipartite graphs.
- We develop a new two-step paradigm to search the significant (α, β)-community. Under this two-step paradigm, novel indexing techniques are proposed to support the retrieval of the (α, β)-community in optimal time. The index I_δ can be built in $O(\delta \cdot m)$ time and takes $O(\delta \cdot m)$ space where δ is bounded by $\sqrt{m}$ and is much smaller in practice. Note that, the proposed indexing techniques can also be directly applied to retrieve the (α, β)-community on unweighted bipartite graphs in optimal time.
- We propose effective index maintenance techniques to handle dynamic graphs.
- We propose efficient query algorithms to extract the significant (α, β)-community from the (α, β)-community.
- We conduct comprehensive experiments on 11 real weighted bipartite graphs to evaluate the effectiveness and the efficiency of the proposed techniques.

2 Problem Definition

TABLE 1: The summary of notations

Notation	Definition
G	a bipartite graph
$V(G)/E(G)$	the vertex/edge set of G
$U(G), L(G)$	the upper layer and lower layer of G
u, v, x	a vertex in a bipartite graph
$(u, v), e$	an edge in a bipartite graph
$R_{\alpha,\beta}$	the (α, β)-core of G
$C_{\alpha,\beta}(q)$	(α, β)-community
$\mathcal{R}$	significant (α, β)-community
$N(u, G)$	the set of neighbors of u on G
n, m	the number of vertices and edges in G ($m > n$)

Our problem is defined over an undirected weighted bipartite graph $G(V{=}(U, L), E)$, where $U(G)$ denotes the set of vertices in the upper layer, $L(G)$ denotes the set of vertices in the lower layer, $U(G) \cap L(G) = \emptyset$, $V(G) = U(G) \cup L(G)$ denotes the vertex set, $E(G) \subseteq U(G) \times L(G)$ denotes the edge set. An edge e between two vertices u and v in G is denoted as (u, v) or (v, u). The set of neighbors of a vertex u in G is denoted as $N(u, G) = \{v \in V(G) \mid (u, v) \in E(G)\}$, and the degree of u is denoted as $deg(u, G) = |N(u, G)|$. We use n and m to denote the number of vertices and edges in G, respectively, and we assume each vertex has at least one incident edge. Each edge $e = (u, v)$ has a weight $w(e)$ (or $w(u, v)$). The size of G is denoted as size$(G) = |E(G)|$.

Before formally defining the problem, we introduce the following critical concepts.

Definition 1. ((α, β)-core) *Given a bipartite graph G and degree constraints α and β, a subgraph $R_{\alpha,\beta}$ is the (α, β)-core of G if (1) $deg(u, R_{\alpha,\beta}) \geq \alpha$ for each $u \in U(R_{\alpha,\beta})$ and $deg(v, R_{\alpha,\beta}) \geq \beta$ for each $v \in L(R_{\alpha,\beta})$; (2) $R_{\alpha,\beta}$ is maximal, i.e., any supergraph $G' \supset R_{\alpha,\beta}$ is not an (α, β)-core.*

Definition 2. ((α, β)-Connected Component) *Given a bipartite graph G and its (α, β)-core $R_{\alpha,\beta}$, a subgraph $C_{\alpha,\beta}$ is a (α, β)-connected component if (1) $C_{\alpha,\beta} \subseteq R_{\alpha,\beta}$ and $C_{\alpha,\beta}$ is connected; (2) $C_{\alpha,\beta}$ is maximal, i.e., any supergraph $G' \supset C_{\alpha,\beta}$ is not a (α, β)-connected component.*

Definition 3. ((α, β)-Community) *Given a vertex q, we call the (α, β)-connected component containing q the (α, β)-community, denoted as $C_{\alpha,\beta}(q)$.*

Definition 4. (Bipartite Graph Weight) *Given a bipartite graph G, the weight value of G denoted by $f(G)$ is defined as the minimum edge weight in G.*

Definition 5. (Significant (α, β)-Community) *Given a weighted bipartite graph G, degree constraints α, β and query vertex q, a subgraph $\mathcal{R}$ is the significant (α, β)-community of G if it satisfies the following constraints:*

1) **Connectivity Constraint.** *$\mathcal{R}$ is a connected subgraph which contains q;*
2) **Cohesiveness Constraint.** *Each vertex $u \in U(\mathcal{R})$ satisfies $deg(u, \mathcal{R}) \geq \alpha$ and each vertex $v \in L(\mathcal{R})$ satisfies $deg(v, \mathcal{R}) \geq \beta$;*
3) **Maximality Constraint.** *There exists no other $G' \subseteq C_{\alpha,\beta}(q)$ satisfying constraints 1) and 2) with $f(G') > f(\mathcal{R})$. In addition, there exists no other supergraph $G'' \supset \mathcal{R}$ satisfying constraints 1) and 2) with $f(G'') = f(\mathcal{R})$.*

Problem Statement. Given a weighted bipartite graph G, parameters α, β and a query vertex q, the *significant (α, β)-community search* problem aims to find the significant (α, β)-community (SC) in G.

Solution Overview. According to Definition 3 and Definition 5, we have the following lemma.

Lemma 1. *Given a weighted bipartite graph G, the significant (α, β)-community is unique, which is a subgraph of the (α, β)-community.*

Following the above lemma, we can use indexing techniques to efficiently find the (α, β)-community first. In this manner, the search space is limited to a much smaller subgraph compared to G. Then, we further search on the (α, β)-community to identify the significant (α, β)-community. According to this two-step algorithmic framework, we present our techniques in the following sections.

3 Time-optimal (α, β)-community Retrieval

In this section, we explore indexing techniques to retrieve the (α, β)-community in an efficient way.

3.1 Basic Indexes

In [15], the authors propose the bicore index which can obtain the vertex set of the (α, β)-core (i.e., $V(R_{\alpha,\beta})$) in optimal time. However, to obtain $C_{\alpha,\beta}(q)$ after having $V(R_{\alpha,\beta})$, we still need to traverse all the neighbors of each vertex in $C_{\alpha,\beta}(q)$ (starting from the query vertex) including those neighbors which are not in $C_{\alpha,\beta}(q)$. This process needs $O(|V(C_{\alpha,\beta}(q))| \cdot \sum_{v \in V(C_{\alpha,\beta}(q))} deg(v,G))$ time and when $\frac{|\text{size}(C_{\alpha,\beta}(q))|}{\sum_{v \in V(C_{\alpha,\beta}(q))} deg(v,G)}$ is small, it may need to access many additional edges not in the queried community. Motivated by this, we explore how to construct an index to support optimal retrieval of the (α, β)-community (i.e., optimal retrieval of (α, β)-connected components).

By Definition 1, we have the following lemma.

Lemma 2. *(α, β)-core $\subseteq (\alpha', \beta')$-core if $\alpha \geq \alpha'$ and $\beta \geq \beta'$.*

We also define the α-offset and the β-offset of a vertex as follows.

Definition 6. (α-/β-offset) *Given a vertex $u \in V(G)$ and an α value, its α-offset denoted as $s_a(u, \alpha)$ is the maximal β value where u can be contained in an (α, β)-core. If u is not contained in $(\alpha, 1)$-core, $s_a(u, \alpha) = 0$. Symmetrically, the β-offset $s_b(u, \beta)$ of u is the maximal α value where u can be contained in an (α, β)-core.*

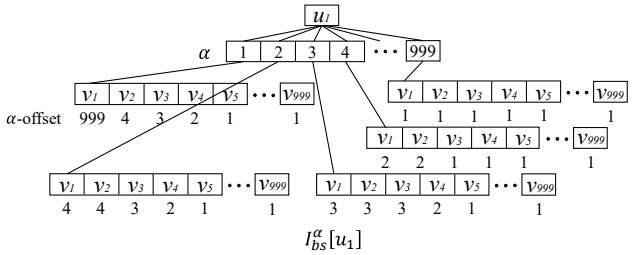
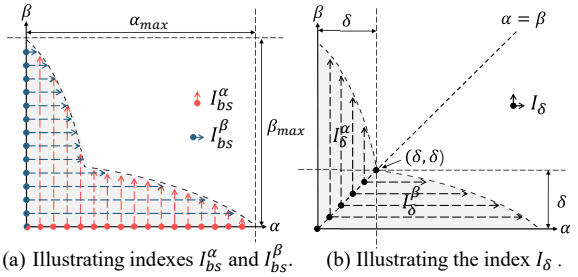


Fig. 3: $I^{\alpha}_{bs}[u_1]$ of G, edge weights are omitted

Algorithm 1: Index Construction of I^{α}_{bs}

Input: G
Output: I^{α}_{bs}
1 $\alpha \leftarrow 1$;
2 $\alpha_{max} \leftarrow$ the maximal vertex degree in $U(G)$;
3 **while** $\alpha \leq \alpha_{max}$ **do**
4 compute $s_a(u, \alpha)$ for each vertex $u \in V(G)$;
5 **foreach** $u \in (\alpha, 1)$-*core* **do**
6 **foreach** $v \in N(u, G)$ **do**
7 **if** $s_a(v, \alpha) \geq 1$ **then**
8 $I^{\alpha}_{bs}[u][\alpha] \leftarrow \{v, w(u, v), s_a(v, \alpha)\}$;
9 sort $I^{\alpha}_{bs}[u][\alpha]$ in decreasing order of their α-offsets;
10 $\alpha \leftarrow \alpha + 1$;
11 **return** I^{α}_{bs};

Since (α, β)-core follows a hierarchical structure according to Lemma 2, an index can be constructed in the following way. For each vertex u, its α-offset indicates that u is contained in the $(\alpha, s_a(u, \alpha))$-core and is not contained in the $(\alpha, s_a(u, \alpha)+1)$-core. According to Lemma 2, if u is contained in the $(\alpha, s_a(u, \alpha))$-core, it is also contained in the



(a) Illustrating indexes I^{α}_{bs} and I^{β}_{bs}. (b) Illustrating the index I_{δ}.

Fig. 4: Illustrating the ideas of indexing techniques

(α, β)-core with $\beta \leq s_a(u, \alpha)$. As shown in Figure 4(a), the shaded area represents all the valid combinations of α and β where an (α, β)-community exists. As illustrated, we can organize the (α, β)-cores hierarchically and construct the basic index I^{α}_{bs} as shown in Algorithm 1. Firstly, we obtain α_{max} which is the maximal α value such that an $(\alpha, 1)$-core exists and it is equal to the maximal vertex degree in $U(G)$. We then compute the α-offset for each vertex. For each vertex u and α combination (where u exists in $(\alpha, 1)$-core), we create an adjacent list $I^{\alpha}_{bs}[u][\alpha]$ to store its neighbors. In $I^{\alpha}_{bs}[u][\alpha]$, we sort u's neighbors in non-increasing order of their α-offsets and remove these neighbors with α-offsets equal to zero. Figure 3 is an example which shows $I^{\alpha}_{bs}[u_1]$ of G in Figure 2(a). We can see that $I^{\alpha}_{bs}[u_1]$ contains the neighbors of u_1 of different α values.

Algorithm 2: Query based on I^{α}_{bs}

Input: G, q, α, β, I^{α}_{bs};
Output: $C_{\alpha,\beta}(q)$
1 $Q \leftarrow q$;
2 $visited(q) \leftarrow true$;
3 **while** Q *is not empty* **do**
4 $u \leftarrow Q.pop()$;
5 **foreach** $v \in I^{\alpha}_{bs}[u][\alpha]$ **do**
6 **if** $s_a(v, \alpha) \geq \beta$ **then**
7 $C_{\alpha,\beta}(q) \leftarrow (u, v)$ if $u \in L(G)$;
8 **if** $visited(v) = false$ **then**
9 $Q.push(v)$;
10 $visited(v) \leftarrow true$;
11 **else**
12 **break**;
13 **return** $C_{\alpha,\beta}(q)$;

Optimal retrieval of $C_{\alpha,\beta}(q)$ based on I^{α}_{bs}. Given a query vertex q, Algorithm 2 illustrates the query process of the (α, β)-community (i.e., $C_{\alpha,\beta}(q)$) based on I^{α}_{bs}. When querying $C_{\alpha,\beta}(q)$, we first put the query vertex into the queue. Then, we pop the vertex u from the queue, and visit the adjacent list $I^{\alpha}_{bs}[u][\alpha]$ to obtain the neighbors of u with α-offset $\geq \beta$. For each valid neighbor v, we add the edge (u, v) into $C_{\alpha,\beta}(q)$ if $u \in L(G)$ to avoid duplication. Then, we put these valid neighbors into the queue and repeat this process until the queue is empty. Since the neighbors are sorted in non-increasing order of their α-offsets, we can early terminate the traversal of the adjacent list when the α-offset of a vertex is smaller than the given β.

Lemma 3. *Given a bipartite graph G and a vertex q, Algorithm 2 computes $C_{\alpha,\beta}(q)$ in $O(\text{size}(C_{\alpha,\beta}(q)))$ time, which is optimal.*

Proof. In Algorithm 2, for each $u \in Q$, since there is no duplicate vertex in $I^{\alpha}_{bs}[u][\alpha]$ and only its neighbor $v \in I^{\alpha}_{bs}[u][\alpha]$ with $s_a(v, \alpha) \geq \beta$ can be accessed, each u and v combination corresponds to an edge in $C_{\alpha,\beta}(q)$. In addition, since each

vertex can be only added once into Q according to lines 8 - 10, Algorithm 2 computes $C_{\alpha,\beta}(q)$ in $O(\mathsf{size}(C_{\alpha,\beta}(q)))$ time, which is optimal as it is linear to the result size. □

Example 1. *Considering the graph in Figure 2 and $I^{\alpha}_{bs}[u_1]$ in Figure 3, if we want to get the $(3,3)$-community of u_1 $C_{3,3}(u_1)$, we first traverse $I^{\alpha}_{bs}[u_1][3]$ to get all the neighbors with α-offsets ≥ 3 which are v_1, v_2 and v_3. The edges (u_1, v_1), (u_1, v_2) and (u_1, v_3) will be added into $C_{3,3}(u_1)$. Then, we go to the index nodes $I^{\alpha}_{bs}[v_1][3]$, $I^{\alpha}_{bs}[v_2][3]$ and $I^{\alpha}_{bs}[v_3][3]$ to get unvisited vertices u_2 and u_3 with α-offsets ≥ 3. The edges (u_2, v_1), (u_2, v_2), (u_2, v_3), (u_3, v_1), (u_3, v_2), (u_3, v_3) will be added into $C_{3,3}(u_1)$ when accessing $I^{\alpha}_{bs}[u_2][3]$ and $I^{\alpha}_{bs}[u_3][3]$.*

In addition, apart from I^{α}_{bs}, we can construct an index I^{β}_{bs} similarly based on β-offsets which also achieves optimal query processing. For each vertex u and β combination, we create an adjacent list to store its neighbors and we sort its neighbors in non-increasing order of their β-offsets (removing these neighbors with β-offsets = 0). When querying the $C_{\alpha,\beta}(q)$, we first go to the adjacent list indexing by q and β, and obtain the neighbors of q with β-offset $\geq \alpha$. Then we run a similar breadth-first search as Algorithm 2 shows. Using I^{β}_{bs}, we can also achieve optimal retrieval of $C_{\alpha,\beta}(q)$ which can be proved similarly as Lemma 3.

Complexity analysis of basic indexes. Storing I^{α}_{bs} needs $\mathsf{size}(I^{\alpha}_{bs}) = O(\sum_{\alpha=1}^{\alpha_{max}}(\mathsf{size}((\alpha,1)\text{-}core))$ space. Since $\sum_{\alpha=1}^{\alpha_{max}}(\mathsf{size}((\alpha,1)\text{-}core) \leq \sum_{\alpha=1}^{\alpha_{max}}(\mathsf{size}((1,1)\text{-}core))$, $\mathsf{size}(I^{\alpha}_{bs})$ is also bounded by $O(\alpha_{max} \cdot m)$. Similarly, I^{β}_{bs} needs $O(\sum_{\beta=1}^{\beta_{max}}(\mathsf{size}((1,\beta)\text{-}core)) = O(\beta_{max} \cdot m)$ space. In addition, the time complexity of constructing I^{α}_{bs} is $\mathsf{TC}(I^{\alpha}_{bs})$ = $O(\alpha_{max} \cdot m)$ as discussed in [21].

3.2 The Degeneracy-bounded Index I_δ

Reviewing I^{α}_{bs} and I^{β}_{bs}, we can see that it is hard to handle high degree vertices in $U(G)(L(G))$ using $I^{\alpha}_{bs}(I^{\beta}_{bs})$. This is because if these vertices exist in an (α, β)-core with large α (or β) value, according to Lemma 2, I^{α}_{bs} or I^{β}_{bs} may need large space to store several copies of the neighbors of these high degree vertices. For example, in Figure 3, I^{α}_{bs} needs to store multiple copies of neighbors of u_1 since u_1 is contained in $(999,1)$-core. The same issue occurs when I^{β}_{bs} stores v_1's neighbors. Thus, in this part, we explore how to effectively handle these high degree vertices and build an index with smaller space consumption.

Firstly, we give the definition of degeneracy as follows.

Definition 7. (Degeneracy) *Given a bipartite graph G, the degeneracy of G denoted as δ is the largest number where (δ,δ)-core is nonempty in G.*

Note that, δ is bounded by $\sqrt{m}$ and in practice, it is much smaller than $\sqrt{m}$ [15].

Lemma 4. *Given a bipartite graph G, a nonempty (α,β)-core in G must have $min(\alpha,\beta) \leq \delta$.*

Proof. The detailed proof can be found in [21]. □

Based on Lemma 4, we can observe that, given query parameters α and β, a partial index of I^{α}_{bs} which only stores adjacent lists of u for each u and α combinations with $\alpha \leq \delta$ is enough to handle queries when $\alpha = min(\alpha,\beta)$. Similarly, a partial index of I^{β}_{bs} which only stores adjacent lists under (u,β) combinations with $\beta \leq \delta$ is enough to handle queries when $\beta = min(\alpha,\beta)$. Based on the above observation, we propose the index I_δ as follows.

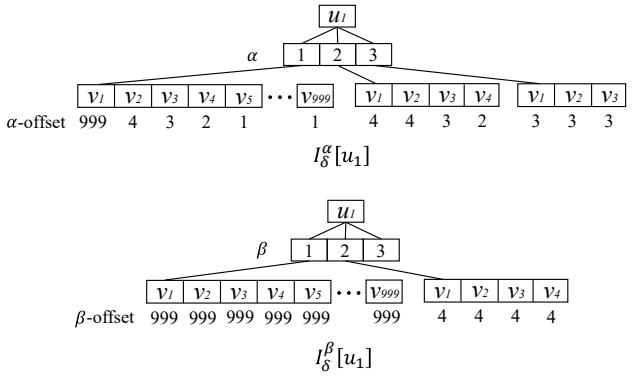


Fig. 5: $I_\delta[u_1]$ of G, edge weights are omitted

Index Overview. I_δ contains two parts I^{α}_{δ} and I^{β}_{δ} to cover all the (α,β)-communities as illustrated in Figure 4(b).

In I^{α}_{δ}, for each vertex u and $\alpha \leq \delta$ where u exists in the (α,α)-core, we create an adjacent list $I^{\alpha}_{\delta}[u][\alpha]$ to store its neighbors. Note that, the neighbors are sorted in non-increasing order of their α-offsets and the neighbors with α-offsets less than α are removed.

In I^{β}_{δ}, for each vertex u and $\beta \leq \delta$ where u exists in the (β,β)-core, we create an adjacent list $I^{\beta}_{\delta}[u][\beta]$ to store its neighbors with β-offsets larger than β. The neighbors are sorted in non-increasing order of their β-offsets and the neighbors with β-offsets less or equal than β are removed. Figure 5 is an example of $I_\delta[u_1]$ of G in Figure 2(a). We can see that it consists of two parts $I^{\alpha}_{\delta}[u_1]$ and $I^{\beta}_{\delta}[u_1]$.

Optimal retrieval of $C_{\alpha,\beta}(q)$ based on I_δ. The query processing of $C_{\alpha,\beta}(q)$ based on I_δ is similar to the query processing based on the basic indexes. The difference is that we need to choose to use I^{α}_{δ} or I^{β}_{δ} at first. If the query parameter $\alpha \leq \delta$, we use I^{α}_{δ} to support the query process. Otherwise, we go for I^{β}_{δ} to obtain the $C_{\alpha,\beta}(q)$. Since only valid edges are touched using I_δ, we can also obtain $C_{\alpha,\beta}(q)$ in $O(\mathsf{size}(C_{\alpha,\beta}(q)))$ time which is optimal. The proof of optimality is similar as Lemma 3 and we omit it here due to the space limit.

Example 2. *Considering G in Figure 2 and $I_\delta[u_1]$ in Figure 5, if we want to get the $(3,3)$-community of u_1 $C_{3,3}(u_1)$, since $\alpha = \beta$, we first traverse $I^{\alpha}_{\delta}[u_1][3]$ to get all the neighbors with α-offsets ≥ 3, which are v_1, v_2 and v_3. The edges (u_1, v_1), (u_1, v_2) and (u_1, v_3) will be added into $C_{3,3}(u_1)$. Then, we go to the index nodes $I^{\alpha}_{\delta}[v_1][3]$, $I^{\alpha}_{\delta}[v_2][3]$ and $I^{\alpha}_{\delta}[v_3][3]$ to get unvisited vertices u_2 and u_3 with α-offsets ≥ 3. The edges (u_2, v_1), (u_2, v_2), (u_2, v_3), (u_3, v_1), (u_3, v_2), (u_3, v_3) will be added into $C_{3,3}(u_1)$ when accessing $I^{\alpha}_{\delta}[u_2][3]$ and $I^{\alpha}_{\delta}[u_3][3]$.*

Lemma 5. *The space complexity of I_δ denoted as $\mathsf{size}(I_\delta)$ is $O(2 \cdot \sum_{\tau=1}^{\delta}\mathsf{size}(R_{\tau,\tau})) = O(\delta \cdot m)$.*

Proof. For each $\alpha \in [1,\delta]$ and $u \in R_{\alpha,\alpha}$, we need to store at most $deg(u, R_{\alpha,\alpha})$ u's neighbors in I^{α}_{δ}. Thus, I^{α}_{δ} needs $O(\sum_{\alpha=1}^{\delta}\sum_{u\in R_{\alpha,\alpha}} deg(u, R_{\alpha,\alpha}))$ = $O(\sum_{\alpha=1}^{\delta}\mathsf{size}(R_{\alpha,\alpha}))$=$O(\delta\cdot m)$ space. Similarly, I^{β}_{δ} also needs $O(\sum_{\beta=1}^{\delta}(\mathsf{size}(R_{\beta,\beta})) = O(\delta\cdot m)$ space. In total, the space for storing I_δ is $O(\delta \cdot m)$. □

Algorithm 3: Index Construction of I_δ

```
Input: G
Output: I_δ
τ ← 1;
compute δ using the k-core decomposition algorithm;
while τ ≤ δ do
    compute α-offset s_a(u,τ) and β-offset s_b(u,τ) for each vertex u ∈ V(G);
    foreach u ∈ (τ,τ)-core do
        foreach v ∈ N(u,G) do
            if s_a(v,τ) ≥ τ then
                I_δ^α[u][τ] ← {v, w(u,v), s_a(v,τ)};
            if s_b(v,τ) > τ then
                I_δ^β[u][τ] ← {v, w(u,v), s_b(v,τ)};
        sort I_δ^α[u][τ] in decreasing order of their α-offsets;
        sort I_δ^β[u][τ] in decreasing order of their β-offsets;
    τ ← τ + 1;
return I_δ;
```

Index Construction. The construction algorithm of I_δ is shown in Algorithm 3. We first compute δ using the k-core decomposition algorithm in [22] since δ is equal to the maximum core number in G. Then, for each vertex u, we compute its α-offset for each $\alpha \leq \delta$ and its β-offset for each $\beta \leq \delta$. These values can be obtained by the peeling algorithm in [16]. Then, we loop τ from 1 to δ and add the valid neighbors of the vertices in the (τ,τ)-$core$ into I_δ.

Lemma 6. *The time complexity of Algorithm 3 is $O(\delta \cdot m)$.*

Proof. The detailed proof can be found in [21]. □

3.3 Size-aware Indexing Techniques

In this part, we discuss how to construct a size-aware index to handle the scenario where a space budget is given. Based on the three indexes proposed in the above parts, we have the following lemmas.

Lemma 7. *In a bipartite graph G, $\sum_{\tau=1}^{\delta}(\mathsf{size}((\tau,\tau)\text{-}core) \leq min(\sum_{\alpha=1}^{\alpha_{max}}(\mathsf{size}((\alpha,1)\text{-}core)), \sum_{\beta=1}^{\beta_{max}}(\mathsf{size}((1,\beta)\text{-}core))$.*

Proof. From Lemma 4, we have $\delta \leq \alpha_{max}$. From Lemma 2, for each $1 \leq \tau \leq \delta$, (τ,τ)-$core) \subseteq (\tau,1)$-$core)$. Thus, we can have $\sum_{\tau=1}^{\delta}(\mathsf{size}((\tau,\tau)\text{-}core) \leq \sum_{\alpha=1}^{\alpha_{max}}(\mathsf{size}((\alpha,1)\text{-}core)$. We can also prove $\sum_{\tau=1}^{\delta}(\mathsf{size}((\tau,\tau)\text{-}core) \leq \sum_{\beta=1}^{\beta_{max}}(\mathsf{size}((1,\beta)\text{-}core)$ in a similar way. Thus, we can get $\sum_{\tau=1}^{\delta}(\mathsf{size}((\tau,\tau)\text{-}core) \leq min(\sum_{\alpha=1}^{\alpha_{max}}(\mathsf{size}((\alpha,1)\text{-}core)), \sum_{\beta=1}^{\beta_{max}}(\mathsf{size}((1,\beta)\text{-}core))$. □

Lemma 8. $\mathsf{size}(I_\delta) \leq 2 \cdot min(\mathsf{size}(I_{bs}^{\alpha}), \mathsf{size}(I_{bs}^{\alpha}))$.

Proof. This lemma directly follows from Lemma 5 and Lemma 7. □

According to Lemma 6 and 8, the size of I_δ is bounded by $O(\delta \cdot m)$ and it is at most two times of the minimal size of the basic indexes. In some circumstances, when α_{max} or β_{max} is not much larger than δ, the naive indexes may have a smaller size than I_δ. Thus, we propose the size-aware indexing technique which can not only choose to construct an index with the minimal size among I_{bs}^{α}, I_{bs}^{β} and I_δ, but also can determine the number of entries constructed in the index if a space limitation is given.

Choose the index with the minimal size in $O(m)$ time. To compute which index is the one with the minimal size among I_{bs}^{α}, I_{bs}^{β} and I_δ, we have the following lemma.

Lemma 9. *Given a bipartite graph G, the values of $\sum_{\alpha=1}^{\alpha_{max}}(\mathsf{size}((\alpha,1)\text{-}core)$, $\sum_{\beta=1}^{\beta_{max}}(\mathsf{size}((1,\beta)\text{-}core)$ and $\sum_{\tau=1}^{\delta}(\mathsf{size}((\tau,\tau)\text{-}core)$ can be computed in $O(m)$ time.*

Proof. Since a vertex $u \in U(G)$ is contained in $(deg(u,G),1)$-core, all of neighbors of u are also contained in $(deg(u,G),1)$-core. Thus, u contributes $deg(u,G)$ edges to the $(deg(u,G),1)$-core. In addition, according to Lemma 2, u also contributes $deg(u,G)$ edges for each $(\alpha,1)$-core where $\alpha \in [1, deg(u,G)]$. According to Definition 1, an $(\alpha,1)$-core is formed by all the incident edges of vertices in $U(G)$ with degree larger than or equal to α. Thus, we can get that $\sum_{\alpha=1}^{\alpha_{max}}(\mathsf{size}((\alpha,1)\text{-}core) = \sum_{u\in U(G)} deg(u,G)^2$ which can be easily computed in $O(m)$ time.

Similarly, $\sum_{\beta=1}^{\beta_{max}}(\mathsf{size}((1,\beta)\text{-}core) = \sum_{v\in L(G)} deg(v,G)^2$ which can also be easily computed in $O(m)$ time.

In addition, using the core decomposition algorithm proposed in [22], we can compute the size of each (τ,τ)-$core$ where $\tau \in [1,\delta]$. Thus, $\sum_{\tau=1}^{\delta}(\mathsf{size}((\tau,\tau)\text{-}core)$ can be computed in $O(m)$ time. □

Based on Lemma 9, we can pre-compute the sizes of indexes I_{bs}^{α}, I_{bs}^{β} and I_δ with a very small overhead (i.e., $O(m)$ time), and construct the one with the minimal size denoted as I_*.

Fit into limited space. In most real-world applications, the query for less cohesive communities is rare. Also, as evaluated in our experiments, the bicore index can efficiently support the retrieval of these less cohesive connected components since only a small number of vertices are not in the (α, β)-core when the query parameters α and β are small. Thus, we propose the hybrid indexing techniques which combine I_* and the bicore index I_v to handle these scenarios when the space budget is given.

Note that all our proposed indexes have high flexibility, that is, given a parameter τ, we only need to build $I_{*,\geq\tau}$ to support optimal queries of $C_{\alpha,\beta}(q)$ when parameters α and/or β are larger than a threshold. When $I_* = I_\delta$, $I_{*,\geq\tau}$ can support optimal queries when parameters $\alpha \geq \tau$ and $\beta \geq \tau$ which requires $O((\delta-\tau)\cdot m)$ space and $O((\delta-\tau)\cdot m)$ construction time. Because of the hierarchical structure of (α, β)-cores, size ($I_{*,\geq\tau}$) can be much smaller than $\mathsf{size}(I_*) \cdot \frac{\delta-\tau}{\delta}$. Similarly, when $I_* = I_{bs}^{\alpha}$, $I_{*,\geq\tau}$ can support optimal queries when parameters $\alpha \geq \tau$ which needs $O((\alpha_{max} - \tau)\cdot m)$ space and construction time. When $I_* = I_{bs}^{\beta}$, $I_{*,\geq\tau}$ can support optimal queries when parameters $\beta \geq \tau$ which needs $O((\beta_{max} - \tau)\cdot m)$ space and construction time. In addition, we build bicore index to handle the other queries. Note that, the bicore index needs $O(m)$ space and takes $O(\delta \cdot m)$ construction time [15].

Lemma 10. *Given a space budget D which is enough to store the bicore index I_v, we can choose the largest integer τ to construct $I_{*,\geq\tau}$ which can fit in $D - \mathsf{size}(I_v)$ space in $O(m \cdot log(max(\alpha_{max}, \beta_{max}))$.*

Proof. According to Lemma 9, the index size of I_* can be computed in $O(m)$ time. Thus, we can use binary search to find the largest τ where $I_{*,\geq\tau}$ can be fitted in the given space. Since $\tau \in [0, max(\alpha_{max}, \beta_{max})]$, the binary search algorithm needs $O(m \cdot log(max(\alpha_{max}, \beta_{max}))$ time. □

Remark. Since we are dealing with the weighted bipartite graphs in this paper, the weights of edges can be dynamically changed in some circumstances. Note that changing the edge weights does not affect the α/β-offsets of the vertices, so We only need to find and update the index entries related to the edge with the changed weight.

4 Index Maintenance for Dynamic Graphs

When graphs are updated dynamically (i.e., vertices/edges are inserted or removed), it is inefficient to reconstruct the indexes from scratch. In this section, we present incremental algorithms to maintain I_δ. Other indexes in this paper can be maintained similarly. Note that we mainly discuss the insertion/removal of one edge since the insertion/removal of vertices can be considered as inserting/removing a sequence of edges.

4.1 Edge insertion

Suppose an edge (u, v) is inserted into G and we denote the graph after insertion as G^+. Firstly, we need to track the changes of α-offsets and β-offsets of the affected vertices. Then, we adjust the index according to the new α/β-offsets.

Tracking the changes of α/β-offsets. By the definition of (α, β)-community, we have the following lemmas.

Lemma 11. *Suppose an edge (u, v) is inserted into G. For each α, only the α-offsets of the vertices in $S^+_\alpha = V(C_{\alpha,s_a(u,\alpha)}(u)) \cup V(C_{\alpha,s_a(v,\alpha)}(v))$ can be increased. Similarly, for each β, only the β-offsets of the vertices in $S^+_\beta = V(C_{s_b(u,\beta),\beta}(u)) \cup V(C_{s_b(v,\beta),\beta}(v))$ can be increased.*

Proof. When fixing α, for each vertex $\notin S^+_\alpha$, it either does not connect to $u(v)$ or $u(v)$ already exists in some (α, β)-connected component it belongs to. Similarly, only the vertices in S^+_β can be affected when fixing β. □

Lemma 12. *Suppose an edge (u, v) is inserted into G. For each α, the α-offset of a vertex $w \in S^+_\alpha \backslash \{u\}$ can be increased by at most 1. For each β, the β-offset of a vertex $w \in S^+_\beta \backslash \{v\}$ can be increased by at most 1.*

Proof. Since there is only one edge inserted, by the structure of (α, β)-core, this lemma holds. □

The above lemmas provide a search scope of affected vertices and how much their offsets can be increased except u (or v). Note that, as proved in [23], when inserting an edge (u, v), for each α, the α-offset of u can be increased to b_α or $b_{\alpha+1}$ where $b_\alpha = max\ x\ s.t.\ |\{w \in N_{G^+}(u) \cap w \in C_{\alpha,x}(u)\}| \geq \alpha$. For each β, the β-offset of v can be increased to b_β or $b_{\beta+1}$ where $b_\beta = max\ x\ s.t.\ |\{w \in N_{G^+}(v) \cap w \in C_{x,\beta}(v)\}| \geq \beta$. In addition, the following lemma is proved in [23] by the definition of (α, β)-core.

Lemma 13. *[23] Suppose an edge (u, v) is inserted into G. For each α, except u, only the α-offsets of the vertices in the (α, ϕ_α)-core can be increased, where $\phi_\alpha = min\{b_\alpha, s_a(v, \alpha)\}$. Similarly, for each β, except v, only the β-offsets of the vertices in the (ϕ_β, β)-core can be increased, where $\phi_\beta = min\{s_b(u, \beta)\}, b_\beta\}$.*

According to all the above lemmas, we can have the following lemma.

Lemma 14. *Suppose an edge (u, v) is inserted into G. For each α, if $s_a(v, \alpha) \leq s_a(u, \alpha)$, only the α-offsets of the vertices in $C_{\alpha,s_a(v,\alpha)}(v)$ can be increased by at most 1; otherwise, except u, only the α-offsets of the vertices in $C_{\alpha,\phi_\alpha}(u)$ can be increased by at most 1. For each β, if $s_b(u, \beta) \leq s_b(v, \beta)$, only the β-offsets of the vertices in $C_{s_b(u,\beta),\beta}(u)$ can be increased by at most 1; otherwise, except v, only the β-offsets of the vertices in $C_{\phi_\beta,\beta}(v)$ can be increased by at most 1.*

Proof. This lemma directly follows from Lemma 12, Lemma 13 and Lemma 14. □

According to the above lemma, the scope of the affected vertices can be significantly reduced. Clearly, the vertices with changed offsets exist in a connected component containing u (or v). Therefore, the local search algorithm proposed in [23] can be adopted here to find the affected vertices and their changed offsets.

Adjusting the index entries. After obtaining all the affected vertices and their changed offsets, now we discuss how to adjust the index entries. We denote the increased α-offset (β-offset) of a vertex w as $s^+_a(w, \alpha)$ ($s^+_b(w, \beta)$). Suppose an edge (u, v) is inserted into G. According to the structure of I_δ, for each α and a vertex w with $s^+_a(w, \alpha) \geq \alpha$, if $I^\alpha_\delta[w][\alpha]$ is empty, we need to create $I^\alpha_\delta[w][\alpha]$ to store each of its neighbor x with $s^+_a(x, \alpha) \geq \alpha$ in I^α_δ. For each β and a vertex w with $s^+_b(w, \beta) \geq \beta$, if $I^\alpha_\delta[w][\beta]$ is empty, we need to create $I^\beta_\delta[w][\beta]$ to store each of its neighbor x with $s^+_b(x, \alpha) \geq \beta$ in I^β_δ. The existing index entries need to be changed based on the following lemma.

Lemma 15. *Suppose an edge (u, v) is inserted into G. For each α and a vertex w with $s^+_a(w, \alpha) \geq \alpha \wedge s^+_a(w, \alpha) > s_a(w, \alpha)$, we only need to update the existing index entries in $I^\alpha_\delta[x][\alpha]$ if $x \in N(w, G^+)$ and $s^+_a(x, \alpha) \geq \alpha$. For each β and a vertex w with $s^+_b(w, \beta) \geq \beta \wedge s^+_b(w, \beta) > s_b(w, \beta)$, we only need to update the existing index entries in $I^\beta_\delta[x][\beta]$ if $x \in N(w, G^+)$ and $s^+_b(x, \beta) \geq \beta$.*

Proof. By the construction process of I_δ (i.e., Algorithm 3), the above lemma holds. □

According to the above lemma, we can create new index entries and adjust all the affected existing index entries for each α and w with changed offsets. However, it is cost-prohibitive if we process each of the vertices with a changed offset and update the corresponding affected index entries. Note that, for one adjacent list of the index (e.g., $I^\alpha_\delta[x][\alpha]$), it can be updated more than once. Motivated by this observation, we try to update the index entries in a batch way. We take the updating of I^α_δ as an example and the updating of I^β_δ can be performed in a similar way. For each α, we process each vertex w with $s^+_a(w, \alpha) \geq \alpha$, and add w into an array $P[x]$ if $x \in N(w, G^+)$ and $s^+_a(x, \alpha) \geq \alpha$. After that, for each vertex x with non-empty $P[x]$, we adjust $I_\delta[x][\alpha]$. By Lemma 14, we know that the vertices in $P[x]$ (except u) are with the same α-offsets which are increased by exactly 1. In addition, since the index entries of $I_\delta[x][\alpha]$ are sorted

in descending order of their α-offsets, we can perform a binary search on $I_\delta[x][\alpha]$ to find the range of index entries which have the same α-offset as the vertices in $P[x]$ (before increasing). Then, we update these index entries of $I_\delta[x][\alpha]$ according to each vertex in $P[x]$. For each index entry with increased α-offset after updating, we exchange it with the first index entry in this range to keep $I_\delta[x][\alpha]$ sorted.

Algorithm 4: Maintaining I_δ for Edge Insertion

Input: G, I_δ, and (u, v)
Output: I_δ of G^+
1 $\delta \leftarrow$ the largest number where (δ, δ)-core is nonempty in G;
2 $G^+ \leftarrow$ insert (u, v) into G;
3 **for** $\alpha = 1$ **to** δ **do**
4 **if** $s_a(u, \alpha) \geq \alpha \wedge s_a(v, \alpha) \geq \alpha$ **then**
5 add u into $I_\delta^\alpha[v][\alpha]$;
6 add v into $I_\delta^\alpha[u][\alpha]$;
7 compute the vertices with increased α-offsets and put them into Y;
8 initialize an array P with empty;
9 **foreach** $w \in Y$ **do**
10 **if** $s_a^+(w, \alpha) \geq \alpha$ **then**
11 **if** $I_\delta^\alpha[w][\alpha]$ *is not constructed* **then**
12 construct $I_\delta^\alpha[w][\alpha]$;
13 $P[x] \leftarrow w$;
14 **foreach** *vertex* x *with nonempty* $P[x]$ **do**
15 update $I_\delta^\alpha[x][\alpha]$ for each $w \in P[x]$;
16 update I_δ^β similarly as lines 3 - 15;
17 **if** $(\delta + 1, \delta + 1)$*-core exists in* G^+ **then**
18 compute the index entries $I_\delta[\cdot][\delta + 1]$ according to Algorithm 3 lines 4 - 13;
19 **return** I_δ;

The IM-Ins algorithm. According to the above discussions, we propose the IM-Ins algorithm as shown in Algorithm 4 to handle the edge insertion case. Note that for each vertex u, we keep its α-offset $s_a(u, \tau)$ and β-offset $s_b(u, \tau)$ for each $\tau \leq \delta$ in the memory for efficient index maintenance. Suppose an edge (u, v) is inserted into G. We first add $u(v)$ into the adjacent lists of $v(u)$ in I_δ, respectively. Specifically, for each $\alpha \leq \delta$, we add $u(v)$ into $I_\delta^\alpha[v][\alpha]$ ($I_\delta^\alpha[u][\alpha]$) if $s_a(u, \alpha) \geq \alpha$ and $s_a(v, \alpha) \geq \alpha$. I_δ^β is updated in a similar way. After that, we track the changes of the α-offsets of the vertices according to Lemma 14. Then, we get the new α/β-offset of each affected vertex by adopting the local search algorithm in [23]. According to Lemma 16, for each $\alpha(\beta)$ and w with increased $s_a^+(w, \alpha) \geq \alpha$ ($s_b^+(w, \beta) \geq \beta$), we update the adjacent lists in I_δ^α (I_δ^β) as discussed before. Finally, we check whether δ is increased by 1 after inserting (u, v). If it is, we construct the index entries for each vertices in the (δ, δ)-core similarly as Algorithm 1. Note that this can only happen when u and v are both contained in the (δ, δ)-core according to Lemma 12.

4.2 Edge removal

Suppose an edge (u, v) is removed from G and we denote the graph after removal as G^-. Similar as the insertion case, for each α, only the α-offsets of the vertices in $S_\alpha^- = V(C_{\alpha,1}(u) \backslash C_{\alpha,s_a(u,\alpha)+1}(u)) \cup V(C_{\alpha,1}(v) \backslash C_{\alpha,s_a(v,\alpha)+1}(v))$ can be decreased. For each β, only the β-offsets of the vertices in $S_\beta^- = V(C_{1,\beta}(u) \backslash C_{s_b(u,\beta)+1,\beta}(u)) \cup V(C_{1,\beta}(v) \backslash C_{s_b(v,\beta)+1,\beta}(v)))$ can be decreased. In addition, we can further shrink the search scope similarly as Lemma 16. Thus, we can also adopt the local search approach in [23] to compute the changes of α/β-offsets. The new α-offset(β-offset) for a vertex w is denoted as $s_a^-(w, \alpha)$ ($s_b^-(w, \beta)$), respectively. After that, we can update the existing index entries based on the following lemma.

Lemma 16. *Suppose an edge (u, v) is removed from G. For each α and a vertex w with $s_a(w, \alpha) \geq \alpha \wedge s_a^-(w, \alpha) < s_a(w, \alpha)$, we need to update the index entries in $I_\delta^\alpha[x][\alpha]$ if $x \in N(w, G)$ and $s_a(x, \alpha) \geq \alpha$. If $s_a^-(w, \alpha) < \alpha$, we remove $I_\delta^\alpha[w][\alpha]$. For each β and a vertex w with $s_b(w, \beta) \geq \beta \wedge s_b^-(w, \beta) < s_b(w, \beta)$, we need to update the index entries in $I_\delta^\beta[x][\beta]$ if $x \in N(w, G)$ and $s_b(x, \beta) \geq \beta$. If $s_b^-(w, \beta) < \beta$, we remove $I_\delta^\beta[w][\beta]$.*

Proof. By the construction of I_δ, this lemma holds. □

The IM-Rem algorithm. Suppose an edge (u, v) is removed from G. We first remove $u(v)$ from the adjacent lists of $v(u)$ in I_δ, respectively. Specifically, for each $\alpha \leq \delta$, we first remove $u(v)$ from $I_\delta^\alpha[v][\alpha]$ ($I_\delta^\alpha[u][\alpha]$) if $s_a(u, \alpha) \geq \alpha$ and $s_a(v, \alpha) \geq \alpha$. I_δ^β is updated in a similar way. After that, we get the new α/β-offset of each affected vertex by adopting the local search algorithm in [23]. Then, similar as the insertion case, for each $\alpha(\beta)$ and an affected vertex w with decreased offset $s_a^-(w, \alpha) \geq \alpha - 1$ ($s_b^-(w, \beta) \geq \beta - 1$), we update the adjacent lists in I_δ^α (I_δ^β). Finally, we check whether δ is decreased by 1 after removing (u, v). If it is, we remove the index entries in each adjacent list $I_\delta[\cdot][\delta]$. Note that this can only happen when u and v are both contained in the (δ, δ)-core.

Complexity Analysis. The time complexity of both IM-Ins and IM-Rem is approximately the same as Algorithm 3 since they need to update the whole index in the worst case. However, they are much faster in practice since they do not need to access and update most of the index entries. As shown in our experiments, they can achieve up to 2 orders of magnitude faster than Algorithm 3.

5 Query the Significant (α, β)-Community

According to the definition of significant (α, β)-community, the subgraph $C_{\alpha,\beta}(q)$ obtained from the index already satisfies the connectivity constraint and the cohesiveness constraint. Thus, here we introduce two query algorithms to obtain the significant (α, β)-community from $C_{\alpha,\beta}(q)$ to further satisfy the maximality constraint. The first algorithm is the peeling approach SCS-Peel which iteratively removes the edge with the minimal weight from $C_{\alpha,\beta}(q)$, and SCS-Expand follows an expansion way which iteratively adds the edge with the maximal weight into an empty graph until the significant (α, β)-community is found.

5.1 Peeling Approach

Here, we introduce the peeling approach. Firstly, we retrieve $C_{\alpha,\beta}(q)$ based on the indexes proposed in Section 3. Note that if all the edge weights are equal in $C_{\alpha,\beta}(q)$, we can just return $C_{\alpha,\beta}(q)$ as the result. Otherwise, we sort the edges in $C_{\alpha,\beta}(q)$ in non-decreasing order by weights and we initialize an edge set S and a queue Q to empty. After that, we run the peeling process on $C_{\alpha,\beta}(q)$. In each iteration, we remove each edge (u, v) with the minimal weight in

$C_{\alpha,\beta}(q)$. Also, we add (u, v) into an edge set S which records the edges removed in this iteration. Due to the removal of (u, v), there may exist many vertices which do not have enough degree to stay in $C_{\alpha,\beta}(q)$ (i.e., for vertex $u \in U(C_{\alpha,\beta}(q))$, $deg(u, C_{\alpha,\beta}(q)) < \alpha$ or for vertex $v \in L(C_{\alpha,\beta}(q))$, $deg(v, C_{\alpha,\beta}(q)) < \beta$), we also remove the edges of these vertices and add the edges into S. We run the peeling process until q does not satisfy the degree constraint. Then, we create G'= $S \cup C_{\alpha,\beta}(q)$ since the edges removed in this iteration need to be recovered to form the $\mathcal{R}$. Finally, we remove the vertices without enough degree in G' and run a breath-first search from q on G' to get the connected subgraph containing q which is $\mathcal{R}$.

Analysis of the SCS-Peel **algorithm.** Below we show the correctness and complexity analysis of SCS-Peel.

Theorem 1. *The* SCS-Peel *algorithm correctly solves the significant (α, β)-community search problem.*

Proof. According to Lemma 1, $\mathcal{R}$ is a subgraph of $C_{\alpha,\beta}(q)$. Suppose there is a $G' \subseteq C_{\alpha,\beta}(q)$ satisfying the connected constraint and the cohesiveness constraint and has $f(G') > f(\mathcal{R})$. Since we always peel the edge with the minimal weight, G' will be found after $\mathcal{R}$. Since we peel $C_{\alpha,\beta}(q)$ until the degree of q is not enough, $q \in G'$ will not have enough degree which contradicts the cohesiveness constraint. For the same reason, there exists no $G'' \supset \mathcal{R}$ with $f(G'') = f(\mathcal{R})$. Thus, this theorem holds. □

Time complexity. SCS-Peel has three phases. Retrieving $C_{\alpha,\beta}(q)$ based on the index needs $(\mathsf{size}(C_{\alpha,\beta}(q)))$ time. Then, sorting the edges in $C_{\alpha,\beta}(q)$ needs $\mathsf{sort}(C_{\alpha,\beta}(q))$ time which will be $O(\mathsf{size}(C_{\alpha,\beta}(q)) \cdot (log(\mathsf{size}(C_{\alpha,\beta}(q)))))$ if we use quick sort or $O(m')$ if we use bin sort where m' equals to the maximal weight in $C_{\alpha,\beta}(q)$. After that, the whole peeling process requires $O(\mathsf{size}(C_{\alpha,\beta}(q))$ time. In total, the time complexity of SCS-Peel is $O(\mathsf{sort}(C_{\alpha,\beta}(q)) + \mathsf{size}(C_{\alpha,\beta}(q)))$.
Space complexity. In the SCS-Peel algorithm, we need only $O(\mathsf{size}(C_{\alpha,\beta}(q)))$ space to store the edges in $C_{\alpha,\beta}(q)$ apart from the space used by the indexes.

5.2 Expansion Approach

Unlike the peeling approach which iteratively removes the edge with the minimal weight from $C_{\alpha,\beta}(q)$, in this part, we introduce the expansion approach SCS-Expand. SCS-Expand first initializes a subgraph G^* as empty. Then it iteratively adds the edges with the maximal weight to G^* (from $C_{\alpha,\beta}(q)$) until G^* contains $\mathcal{R}$. In this manner, if $\mathsf{size}(\mathcal{R})$ is much smaller than $\mathsf{size}(C_{\alpha,\beta}(q))$, SCS-Expand can retrieve $\mathcal{R}$ in a more efficient way compared to SCS-Peel.

Following the above idea, we add edges with the maximal weight in $C_{\alpha,\beta}(q)$ to G^* (and remove them from $C_{\alpha,\beta}(q)$) in each iteration. However, when adding an edge into G^*, it may not connect to q. Note that, we cannot discard these edges immediately since they may be connected to q due to the later coming edges. Thus, the connected subgraphs in G^* should be maintained in each iteration. With the help of union-find data structure [24], the connected subgraphs in G^* can be maintained in constant amortized time, and we can efficiently obtain the connected subgraph containing q in G^*.

Checking the existence of $\mathcal{R}$ in C^*. Suppose C^* is the connected subgraph containing q in G^*, we can easily observe that $\mathcal{R}$ can only be found in the iteration where C^* is changed. In addition, we have the following bounds which can let us know whether $\mathcal{R}$ is contained in C^*.

Lemma 17. *Given a connected subgraph C^*, if $\mathcal{R} \subseteq C^*$, we have:*

$$\alpha\beta - \alpha - \beta \leq |E(C^*)| - |U(C^*)| - |L(C^*)|$$

Proof. The detailed proof can be found in [21]. □

Lemma 18. *Given a connected subgraph $C^* \subseteq G$, if $\mathcal{R} \subseteq C^*$, it must contain α vertices where each vertex u of them has $deg(u, C^*) \geq \beta$, and it must contain β vertices where each vertex v of them has $deg(v, C^*) \geq \alpha$. In addition, the query vertex should be one of these vertices.*

Proof. This lemma directly follows from Definition 5. □

Based on the above lemmas, we can skip checking the existence of $\mathcal{R}$ if the constraints are not satisfied. It is still costly if we check each C^* satisfies the constraints since we need to perform the peeling algorithm on C^* using $O(\mathsf{size}(C^*))$ time. To mitigate this issue, we set an expansion parameter $\epsilon > 1$ to control the number of checks. Firstly, we check C^* when it first satisfies the constraints in the Lemma 17 and Lemma 18. After that, we only check C^* if its size is at least ϵ times than the size of its last check. Here we choose $\epsilon = 2$ as analyzed in [21]. The details of the expansion algorithm SCS-Expand can be found in [21].

Analysis of the SCS-Expand **algorithm.** Below we show the correctness and time/space complexities of SCS-Expand.

Theorem 2. *The* SCS-Expand *algorithm correctly solves the significant (α, β)-community search problem.*

Proof. The proof is similar as Theorem 1. □

Time complexity. In SCS-Expand, retrieving $C_{\alpha,\beta}(q)$ based on the index needs $O(\mathsf{size}(C_{\alpha,\beta}(q)))$ time. Then, sorting the edges in $C_{\alpha,\beta}(q)$ needs $O(\mathsf{sort}(C_{\alpha,\beta}(q)))$ time. After that, the whole expansion process requires $O(\sum_{i=1}^{d} \mathsf{size}(C_i^*))$ time where d is the number of subgraphs need to be peeled. In total, the time complexity of SCS-Expand is $O(\mathsf{sort}(C_{\alpha,\beta}(q)) + \sigma_{i=1}^{d}\mathsf{size}(C_i^*))$.
Space complexity. In the SCS-Expand algorithm, we need $O(\mathsf{size}(C_{\alpha,\beta}(q)))$ space to store the edges in $C_{\alpha,\beta}(q)$ except the space used by indexes.

TABLE 2: Summary of Datasets

Dataset	$\|E\|$	$\|U\|$	$\|L\|$	δ	α_{max}	β_{max}	$\|R_{\delta,\delta}\|$
BS	433K	77.8K	186K	13	8,524	707	13.6K
GH	440K	56.5K	121K	39	884	3,675	21.5K
SO	1.30M	545K	96.6K	22	4,917	6,119	13.0K
LS	4.41M	992	1.08M	164	55,559	773	177K
DT	5.74M	1.62M	383	73	378	160,047	30.5K
AR	5.74M	2.15M	1.23M	26	12,180	3,096	36.6K
PA	8.65M	1.43M	4.00M	10	951	119	639
ML	25.0M	162K	59.0K	636	32,202	81,491	2.12M
DUI	102M	833K	33.8M	183	24,152	29,240	2.30M
EN	122M	3.82M	21.5M	254	1,916,898	62,330	1.03M
DTI	137M	4.51M	33.8M	180	1,057,753	6,382	242K

6 Experiments

In this section, we first evaluate the effectiveness of the significant (α, β)-community model. Then, we evaluate the efficiency of the techniques for retrieving (α, β)-communities and significant (α, β)-communities.

6.1 Experiments setting

Algorithms. Our empirical studies are conducted against the following designs:

- *Techniques to retrieve the (α, β)-community.* The query algorithms: 1) the online query algorithm Q_o in [16], and the query algorithms based on the following indexes: 2) Q_v based on the bicore index I_v proposed in [15], 3) Q_{opt} based on I_δ in Section 3.2. The indexes: 1) the bicore index I_v, 2) basic indexes I^α_{bs} and I^β_{bs}, 3) I_δ and 4) the size-aware index I_*. In addition, we evaluate the index maintenance algorithms IM-Ins and IM-Rem.
- *Algorithms to retrieve the significant (α, β)-community.* 1) the peeling algorithm SCS-Peel, 2) the expansion algorithm SCS-Expand in Section 5 and 3) a baseline algorithm SCS-Baseline which iteratively expands the edges (with larger weight value) from the connected component containing q of the whole graph rather than from $C_{\alpha,\beta}(q)$.

The algorithms are implemented in C++ and the experiments are run on a Linux server with Intel Xeon 2650 v3 2.3GHz processor and 768GB main memory. *We terminate an algorithm if the running time is more than 10^4 seconds.*

Datasets. We use 11 real datasets in our experiments which are `Bookcrossing` (BS), `Github` (GH), `StackOverflow` (SO), `Lastfm` (LS), `Discogs` (DT), `Amazon` (AR), `DBLP` (PA), `MovieLens` (ML), `Delicious-ui` (DUI), `Wikipedia-en` (EN) and `Delicious-ti` (DTI). All the datasets we use can be found in KONECT (http://konect.uni-koblenz.de). Note that, for the datasets without weights (i.e., DT and PA), we use the random walk with restart model [25] to compute the node relevance and generate the weights.

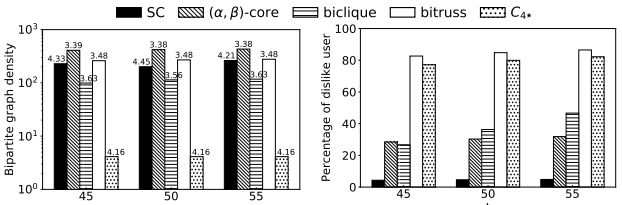
The summary of datasets is shown in Table 2. U and L are vertex layers, $|E|$ is the number of edges. δ is the degeneracy. α_{max} and β_{max} are the largest value of α and β where a $(\alpha, 1)$-core or $(1, \beta)$-core exists, respectively. $|R_{\delta,\delta}|$ denotes the number of edges in $R_{\delta,\delta}$ in each dataset. In addition, M denotes 10^6 and K denotes 10^3.

6.2 Effectiveness evaluation

In this section, we evaluate the effectiveness of our model on `MovieLense` which contains 25M ratings (ranging from 1 to 5) from 162K users (U) on 59K movies (L).

We compare the significant (α, β)-community model with the (α, β)-core, k-bitruss (setting $k = \alpha \cdot \beta$) [18] and maximal biclique [20] models. We also add a community $C_{4\star}$ which is the induced subgraph of all the movies with average ratings at least 4. Note that, we use the connected components of the query vertex as the result when considering different models.

Evaluating the community quality. Suppose a user wants to find some friends who are also fans of comedy movies. We extract the subgraph formed by the ratings on comedy movies and perform community search algorithms. Figure 6(a) shows the bipartite graph density which is computed as $d(G) = |E(G)|/\sqrt{|U(G)||L(G)|}$ [26]. We can see that the communities produced by (α, β)-core, bitruss, biclique and SC all have high densities comparing with $C_{4\star}$ since the structure cohesiveness is considered in these models. Thus, the users in $C_{4\star}$ are loosely connected with each other and have fewer interactions. In addition, the average ratings (i.e., the numbers on the top of each bar) indicate that SC can always return a group of users with higher average ratings than (α, β)-core, bitruss and biclique. We also show the number of dislike users in Figure 6(b). A user is a dislike user if he/she gives fewer than 0.6α good ratings (i.e., rating ≥ 4), who is not likely to be a fan of comedies. We can see that SC contains fewer number of dislike users comparing with all the other models because both weight and structure cohesiveness are considered. Thus, the users in SC are considered as good candidates to be recommended to the query user. Note that the percentage of dislike users in bitruss and $C_{4\star}$ is very high. This is because bitruss ensures the structure cohesiveness using the butterfly (i.e., 2×2-biclique) and a user can exist in a k-bitruss with a large k value if he/she only watched a few number of hot movies. In addition, $C_{4\star}$ does not ensure the structure cohesiveness and there exist many users who only watched few high rating movies.



(a) Bipartite graph density (b) Percentage of dislike users

Fig. 6: Evaluating the community quality, varying $\alpha, \beta = t$

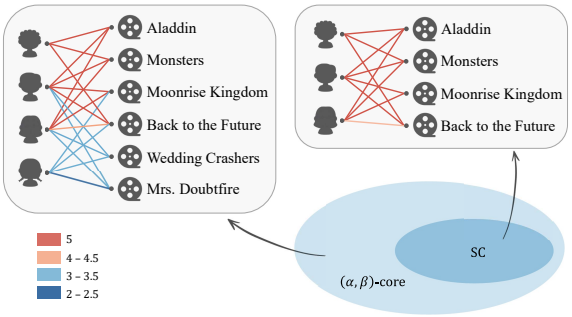


Fig. 7: Representative components of real-life communities

TABLE 3: Statistics of query results, $q = 6{,}778$

Models	$\|U\|$	$\|M\|$	R_{avg}	R_{min}	M_{avg}	Sim (%)
SC	2,127	670	4.81	4.50	63.47	100
(α, β)-core	34,466	2491	3.39	0.5	110.03	7.57
bitruss	158,183	2,985	3.48	0.5	35.87	1.74
biclique	65	45	3.45	0.5	45	2.39
$C_{4\star}$	114,915	387	4.16	0.5	2.39	1.82

Case study. We conduct queries using parameters $q = 6778, \alpha = 45, \beta = 45$ on comedy movies. The statistics of query results are shown in Table 3. $|U|$ and $|M|$ denote the total number of users and movies in the community, respectively. R_{avg} and R_{min} denote the average and minimal rating in the community, respectively. M_{avg} is the average

number of movies a user watched in the community and Sim is the jaccard similarity between each community and SC. For the biclique model, here we use a maximal biclique containing q with at least 45 vertices in each layer. We can see that SC contains reasonable number of users and vertices with higher average rating and minimal rating in the community than the others. We also show the representative components of the communities using (α, β)-core and SC in Figure 7. We can see that (α, β)-core contains users who do not like such movies and movies that are not liked by such users. This is because (α, β)-core only considers structure cohesiveness and ignores the edge weights. We can observe that M_{avg} of $C_{4\star}$ is only 2.39 since the structure cohesiveness is not considered in $C_{4\star}$. Thus, $C_{4\star}$ contains many users who only watched a few number of high rating movies and these users are loosely connected with the query user. Among these models, only SC considers both weight and structure cohesiveness, which is not similar to other communities compared here. In SC, the quality of the users and movies found can be guaranteed.

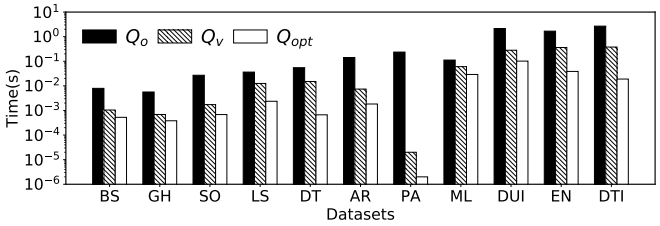
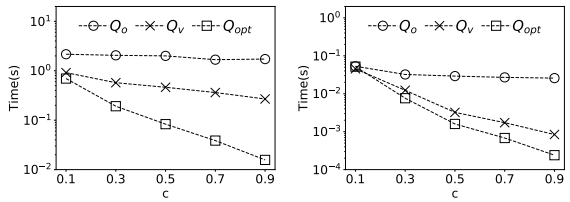
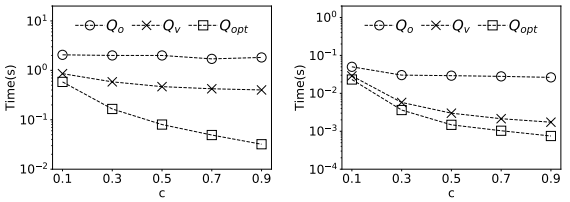


Fig. 8: Retrieving the (α, β)-communities



(a) EN, $\alpha, \beta = c \cdot \delta$ (b) SO, $\alpha, \beta = c \cdot \delta$



(c) EN, $\alpha = 0.5 \cdot \delta, \beta = c \cdot \delta$ (d) SO, $\alpha = c \cdot \delta, \beta = 0.5 \cdot \delta$

Fig. 9: Retrieving the (α, β)-communities, varying α and β

6.3 Retrieving (α, β)-communities

In this part, we evaluate the proposed indexing techniques to retrieve (α, β)-communities.

Query time. *1) Performance on all the datasets.* We first evaluate the performance on all the datasets by setting α and β to 0.7δ. In Figure 8, we can observe that Q_{opt} significantly outperforms Q_o and Q_v on all the datasets. This is because Q_{opt} is based on I_δ which can achieve optimal retrieval of (α, β)-communities. Especially, on large datasets such as DUI, EN and DTI, the Q_{opt} algorithm is one to two orders of magnitude faster than Q_o and is up to 20× faster than Q_v.

2) Varying α and β. We also vary α and β to assess the performance of these algorithms. In Figure 9(a) and (b), α and β are varied simultaneously. We can observe that when α and β are small, the performance of these algorithms is similar. This is because only a few number of edges are removed from the original graph when the query parameters are small. When α and β are large, the resulting (α, β)-communities are much smaller than the original graph. Thus, Q_{opt} is much faster than Q_o and Q_v. In Figure 9(c) and (d), we fix α (or β) and the trends are similar.

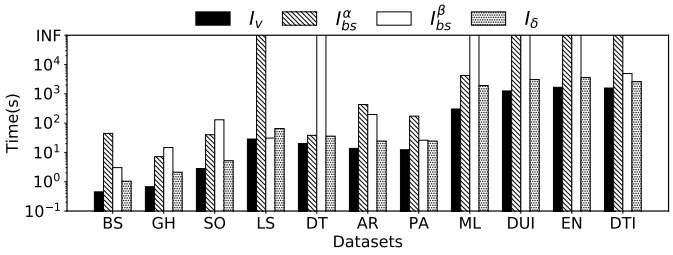


Fig. 10: Index construction time

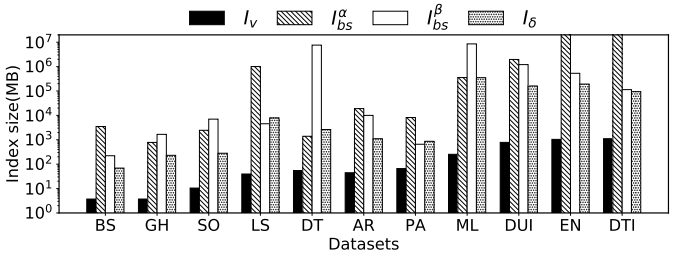


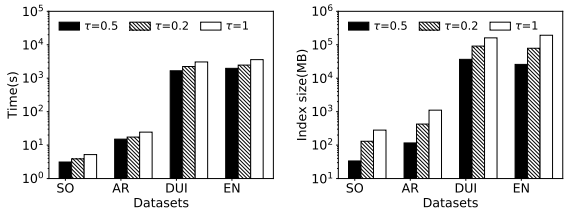
Fig. 11: Index size

Evaluating index construction time and index size. In this part, we evaluate the index size and index construction time. We omit the index I_* here as it will always choose the index with the minimal size between I_{bs}^{α}, I_{bs}^{β} and I_δ under a very small overhead.

1) Index construction time. In Figure 10, we can see that I_δ can be efficiently constructed on all the datasets since it only needs the same low constructing time complexity as I_v ($O(\delta m)$). In addition, constructing I_δ is slightly slower than constructing I_v which is reasonable since I_v only contains vertex information of (α, β)-cores while I_δ contains edge information which can support optimal retrieval of (α, β)-communities. The time for constructing I_{bs}^{α} and I_{bs}^{β} highly depends on α_{max} and β_{max}. Thus, it is very slow (or even unaccomplished) on the datasets where these two values are large such as DUI and EN.

2) Index size. In Figure 11, we evaluate the size of these indexes. If an index cannot be built within the time limit, we report the expected size of it. We can see that size(I_δ) is smaller than size(I_{bs}^{α}) and size(I_{bs}^{β}) on almost all the datasets. I_v is the index with the minimal size since it only contains vertex information. This result also validates that $\text{size}(I_\delta) \leq 2 \times min(\text{size}(I_{bs}^{\alpha}), \text{size}(I_{bs}^{\beta}))$.

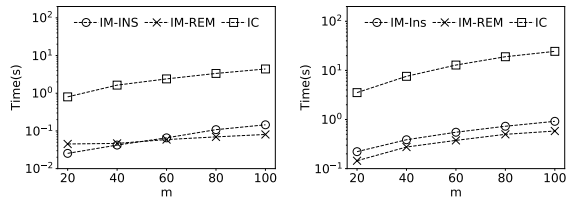
Evaluating the effect of τ in I_*. In Figure 12, we evaluate the effect of τ in I_* on four datasets. We can see that the size of $I_{*,\geq 0.5}$ and $I_{*,\geq 0.2}$ are much smaller than I_* on these datasets. However, it can still support the optimal query when the given query parameters are larger than the threshold. Also as validated in Figure 9, when the query



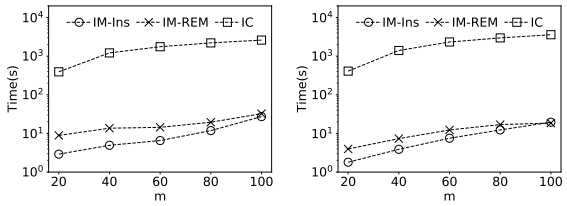
(a) Effect of τ on time (b) Effect of τ on index size

Fig. 12: Evaluating the effect of τ

parameters α and β are small, I_v is efficient enough to handle these queries.



(a) SO (b) AR



(c) DTI (d) EN

Fig. 13: Edge insertion and removal, varying scalability

Evaluating the scalability of index maintenance. In Figure 13, we evaluate the scalability of index maintenance algorithms. When varying m, we randomly sample 20% to 100% edges of the original graphs. Here IC denotes the index construction algorithm of I_δ. We randomly insert or remove 200 distinct edges and report the average processing time. We can observe that our proposed algorithms are scalable. In addition, we can see that the processing time of our index maintenance algorithms IM-Ins and IM-Rem is much smaller than reconstructing the index. This is because our proposed solutions do not need to modify the whole index, and a lot of unnecessary computation is reduced.

6.4 Retrieving significant (α, β)-communities

Here we evaluate the performance of the algorithms (SCS-Baseline, SCS-Peel, and SCS-Expand) for querying significant (α, β)-communities. In these algorithms, we use Q_{opt} to support the optimal retrieval of (α, β)-community. In each test, we randomly select 100 queries and take the average.

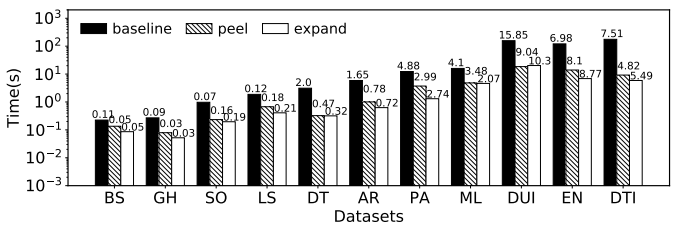
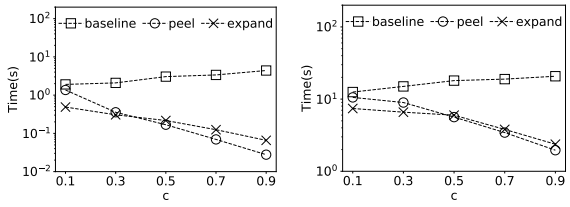
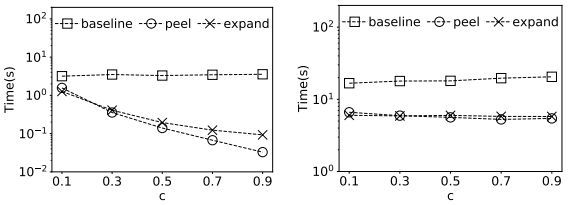


Fig. 14: Query performance on different datasets

Evaluating the performance on all the datasets. In Figure 14, we evaluate the performance of SCS-Baseline, SCS-Peel, and SCS-Expand on all the datasets. We also report the standard deviation on the top of each bar. We can see that SCS-Expand and SCS-Peel are significantly faster than SCS-Baseline, especially on large datasets. This is because, with the help of the two-step framework, the search space of SCS-Peel and SCS-Expand is limited in $C_{\alpha,\beta}(q)$, while SCS-Baseline needs to consider all edges in the connected component containing q of the whole graph. We can also see in Table 2 that $|R_{\delta,\delta}|$ is much smaller than $|E|$. Since $C_{\delta,\delta}(q) \subseteq R_{\delta,\delta}$, when we choose relatively larger parameters, the search space of SCS-Peel and SCS-Expand is much smaller than SCS-Baseline. In addition, we can see that on most datasets, SCS-Expand is on average more efficient than SCS-Peel. However, the standard deviations of SCS-Expand and SCS-Peel are large. This is because SCS-Peel and SCS-Expand both need more time to handle the cases when α and β are small and SCS-Expand is usually much faster than SCS-Peel in these cases.



(a) DT, $\alpha, \beta = c \cdot \delta$ (b) ML, $\alpha, \beta = c \cdot \delta$



(c) DT, $\alpha = c \cdot \delta, \beta = 0.5 \cdot \delta$ (d) ML, $\alpha = 0.5 \cdot \delta, \beta = c \cdot \delta$

Fig. 15: Effect of α and β

Evaluating the effect of query parameters α and β. In Figure 15, we vary α and β on two datasets DT and ML. From Figure 15(a) and (b), we can see that, when α and β are small, SCS-Expand is more efficient than SCS-Peel. In addition, the running time of SCS-Peel and SCS-Expand decreases as α (or β) increases. Note that the efficiency of these two algorithms largely depends on the size of the (α, β)-community containing q (i.e., size$(C_{\alpha,\beta}(q))$, which determines the search space) and the size of the final result (i.e., size$(\mathcal{R})$, which relates to the actual computation cost). In most cases, when α and β are large, the size of $C_{\alpha,\beta}(q)$ is small and $\mathcal{R}$ is expected to be large since more edges are needed in $\mathcal{R}$ to satisfy the cohesiveness constraints. Thus, the edges need to be peeled are usually few and SCS-Peel is more efficient than SCS-Expand. When α and β are small, the search space (i.e., $C_{\alpha,\beta}(q)$) can be large and $\mathcal{R}$ is expected to be small. Thus, SCS-Expand is usually more efficient than SCS-Peel in these cases. In most cases, we can determine to use SCS-Peel or SCS-Expand according to the choice of α and β.

Evaluate the effect of weight distribution. In Table 4, we evaluate the effect of weight distribution on DT dataset. We test four weight distributions: (1) AE: the weights are all equal; (2) RW: the weights are generated using the random

TABLE 4: Running time under different weight distribution

Algorithms	AE	RW	UF	SK
SCS-Baseline	0.03s	3.12s	4.42s	4.31s
SCS-Peel	0.03s	0.34s	0.48s	0.45s
SCS-Expand	0.03s	0.31s	0.41s	0.36s

walk with restart model [25]; (3) UF: the weights follow uniform distribution; (4) SK: the weights follow skewed normal distribution with skewness = 1.02. When all the edge weights are equal (AE) which can be considered as a special case, all three algorithms can just return $C_{\alpha,\beta}(q)$ after efficiently scanning $C_{\alpha,\beta}(q)$. Note that the performances of these three algorithms are not very sensitive to the other three distributions. This is because both weight and structure cohesiveness are considered in our problem and the impact of RW/SK/UF weight distributions are limited.

7 Related Work

To the best of our knowledge, this paper is the first to study community search over bipartite graphs. Below we review two closely related areas, community search on unipartite graphs and cohesive subgraph models on bipartite graphs.

Community search on unipartite graphs. On unipartite graphs, community search is conducted based on different cohesiveness models such as k-core [4], [5], [6], [7], [27], [28], k-truss [8], [9], [10], [29], [30], k-clique [31]. Interested readers can refer to [11] for a recent comprehensive survey.

Based on k-core, [4] and [5] study online algorithms for k-core community search on unipartite graphs. In [6], Barbieri et al. propose a tree-like index structure for the k-core community search. Using k-core, Fang et al. [7] further integrate the attributes of vertices to identify community and the spatial locations of vertices are considered in [27], [28]. An core-based index is proposed in [27], and the index maintenance technique is studied in [32]. In [33], [34], [35], the algorithms for core number maintenance are studied. For the truss-based community search, [8], [29] study the triangle-connected model and [9] studies the closest model. In addition, a truss-based community search solution is proposed in [10] for attributed graphs. In [31], the authors study the problem of densest clique percolation community search. However, the edge weights are not considered in any of the above works and their techniques cannot be easily extended to solve our problem. On edge-weighted unipartite graphs, the k-core model is applied to find cohesive subgraphs in [36], [37]. They use a function to associate the edge weights with vertex degrees and the edge weights are not considered as a second factor apart from the graph structure. Thus, these works do not aim to find a cohesive subgraph with both structure cohesiveness and high weight (significance). Under their settings, a subgraph with loose structure can be found in the result. For example, a vertex can be included in the result if it is only incident with one large-weight edge. In [30], the k-truss model is adopted on edge-weighted graphs to find communities. However, the k-truss model is based on the triangle structure which does not exist on bipartite graphs. The graph projection method [38] is not appropriate to be used here as discussed in [21].

Finding cohesive subgraphs on bipartite graphs. On bipartite graphs, several existing works [15], [16] extend the k-core model on unipartite graph to the (α, β)-core model. An effective index is proposed in [15] to retrieve the (α, β)-core, and the index maintenance technique is studied in [23] to handle dynamic graphs. [17], [18], [19] study the bitruss model in bipartite graphs which is the maximal subgraph where each edge is contained in at least k butterflies. [20] studies the biclique enumeration problem. However, the above works only consider the structure cohesiveness and ignore the edge weights which are important as validated in the experiments. In the literature, fair clustering problems [12], [13], [14] are studied to find communities (i.e., clusters) under fairness constraints on bipartite graphs. The problem is inherently different and the techniques are not applicable to the problem studied in this paper. An interesting work in [39] studies the paper matching problem in peer-review process which also finds dense subgraphs on bipartite graphs. However, their flow-based techniques are often used to solve a matching problem which can not be used here.

8 Conclusion

In this paper, we study the significant (α, β)-community search problem. To solve this problem efficiently, we follow a two-step framework which first retrieves the (α, β)-community, and then identifies the significant (α, β)-community from the (α, β)-community. We develop a novel index I_δ to retrieve the (α, β)-community in optimal time. In addition, we propose efficient peeling and expansion algorithms to obtain the significant (α, β)-community. We conduct extensive experiments on real-world graphs, and the results demonstrate the effectiveness of the significant (α, β)-community model and the proposed techniques.

References

[1] J. Wang, A. P. De Vries, and M. J. Reinders, "Unifying user-based and item-based collaborative filtering approaches by similarity fusion," in *SIGIR*. ACM, 2006, pp. 501–508.
[2] A. Beutel, W. Xu, V. Guruswami, C. Palow, and C. Faloutsos, "Copycatch: stopping group attacks by spotting lockstep behavior in social networks," in *WWW*. ACM, 2013, pp. 119–130.
[3] M. Ley, "The DBLP computer science bibliography: Evolution, research issues, perspectives," in *Proc. Int. Symposium on String Processing and Information Retrieval*, 2002, pp. 1–10.
[4] W. Cui, Y. Xiao, H. Wang, and W. Wang, "Local serach of communities in large graphs," in *SIGMOD*, 2014, pp. 991–1002.
[5] M. Sozio and A. Gionis, "The community-search problem and how to plan a succesful cocktail party," in *SIGKDD*, 2010, pp. 939–948.
[6] N. Barbieri, F. Bonchi, E. Galimberti, and F. Gullo, "Efficient and effective community search," *Data mining and knowledge discovery*, vol. 29, no. 5, pp. 1406–1433, 2015.
[7] Y. Fang, R. Cheng, S. Luo, and J. Hu, "Effective community search for large attributed graphs," *PVLDB*, vol. 9, no. 12, pp. 1233–1244, 2016.
[8] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu, "Querying k-truss community in large and dynamic graphs," in *SIGMOD*, 2014, pp. 1311–1322.
[9] X. Huang, L. V. S. Lakshmanan, J. X. Yu, and H. Cheng, "Approximate closest community search in networks," *PVLDB*, vol. 9, no. 4, pp. 276–287, 2015.
[10] X. Huang and L. V. Lakshmanan, "Attribute-driven community search," *PVLDB*, vol. 10, no. 9, pp. 949–960, 2017.
[11] Y. Fang, X. Huang, L. Qin, Y. Zhang, R. Cheng, and X. Lin, "A survey of community search over big graphs," *VLDB J.*, vol. 29, no. 1, pp. 353–392, 2020.
[12] F. Chierichetti, R. Kumar, S. Lattanzi, and S. Vassilvitskii, "Fair clustering through fairlets," in *Advances in Neural Information Processing Systems*, 2017, pp. 5029–5037.
[13] S. Ahmadi, S. Galhotra, B. Saha, and R. Schwartz, "Fair correlation clustering," *arXiv preprint arXiv:2002.03508*, 2020.

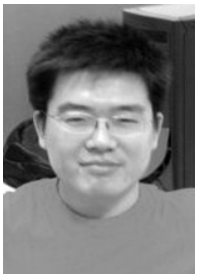
[14] S. Ahmadian, A. Epasto, M. Knittel, R. Kumar, M. Mahdian, B. Moseley, P. Pham, S. Vassilvtiskii, and Y. Wang, "Fair hierarchical clustering," *arXiv preprint arXiv:2006.10221*, 2020.
[15] B. Liu, L. Yuan, X. Lin, L. Qin, W. Zhang, and J. Zhou, "Efficient (α, β)-core computation: An index-based approach," in *WWW*. ACM, 2019, pp. 1130–1141.
[16] D. Ding, H. Li, Z. Huang, and N. Mamoulis, "Efficient fault-tolerant group recommendation using alpha-beta-core," in *CIKM*, 2017, pp. 2047–2050.
[17] K. Wang, X. Lin, L. Qin, W. Zhang, and Y. Zhang, "Efficient bitruss decomposition for large-scale bipartite graphs," in *ICDE*. IEEE, 2020, pp. 661–672.
[18] Z. Zou, "Bitruss decomposition of bipartite graphs," in *DASFAA*. Springer, 2016, pp. 218–233.
[19] A. E. Sarıyüce and A. Pinar, "Peeling bipartite networks for dense subgraph discovery," in *WSDM*. ACM, 2018, pp. 504–512.
[20] Y. Zhang, C. A. Phillips, G. L. Rogers, E. J. Baker, E. J. Chesler, and M. A. Langston, "On finding bicliques in bipartite graphs: a novel algorithm and its application to the integration of diverse biological data types," *BMC bioinformatics*, vol. 15, no. 1, p. 110, 2014.
[21] K. Wang, W. Zhang, X. Lin, Y. Zhang, L. Qin, and Y. Zhang, "Efficient and effective community search on large-scale bipartite graphs," 2020.
[22] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo, "K-core decomposition of large networks on a single pc," *Proceedings of the VLDB Endowment*, vol. 9, no. 1, pp. 13–23, 2015.
[23] B. Liu, L. Yuan, X. Lin, L. Qin, W. Zhang, and J. Zhou, "Efficient (α, β)-core computation in bipartite graphs," *The VLDB Journal*, pp. 1–25, 2020.
[24] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.
[25] H. Tong, C. Faloutsos, and J.-Y. Pan, "Fast random walk with restart and its applications," in *Sixth international conference on data mining (ICDM'06)*. IEEE, 2006, pp. 613–622.
[26] R. Kannan and V. Vinay, *Analyzing the structure of large graphs*. Rheinische Friedrich-Wilhelms-Universität Bonn Bonn, 1999.
[27] Y. Fang, R. Cheng, X. Li, S. Luo, and J. Hu, "Effective community search over large spatial graphs," *PVLDB*, vol. 10, no. 6, pp. 709–720, 2017.
[28] K. Wang, X. Cao, X. Lin, W. Zhang, and L. Qin, "Efficient computing of radius-bounded k-cores," in *ICDE*. IEEE, 2018, pp. 233–244.
[29] E. Akbas and P. Zhao, "Truss-based community search: a truss-equivalence based indexing approach," *PVLDB*, vol. 10, no. 11, pp. 1298–1309, 2017.
[30] Z. Zheng, F. Ye, R.-H. Li, G. Ling, and T. Jin, "Finding weighted k-truss communities in large networks," *Information Sciences*, vol. 417, pp. 344–360, 2017.
[31] L. Yuan, L. Qin, W. Zhang, L. Chang, and J. Yang, "Index-based densest clique percolation community search in networks," *TKDE*, vol. 30, no. 5, pp. 922–935, 2017.
[32] Y. Fang, R. Cheng, Y. Chen, S. Luo, and J. Hu, "Effective and efficient attributed community search," *The VLDB Journal*, vol. 26, no. 6, pp. 803–828, 2017.
[33] A. E. Saríyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek, "Streaming algorithms for k-core decomposition," *Proceedings of the VLDB Endowment*, vol. 6, no. 6, pp. 433–444, 2013.
[34] A. E. Sarıyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek, "Incremental k-core decomposition: algorithms and evaluation," *The VLDB Journal*, vol. 25, no. 3, pp. 425–447, 2016.
[35] Y. Zhang, J. X. Yu, Y. Zhang, and L. Qin, "A fast order-based approach for core maintenance," in *ICDE*. IEEE, 2017, pp. 337–348.
[36] A. Garas, F. Schweitzer, and S. Havlin, "A k-shell decomposition method for weighted networks," *New Journal of Physics*, vol. 14, no. 8, p. 083030, 2012.
[37] M. Eidsaa and E. Almaas, "S-core network decomposition: A generalization of k-core analysis to weighted networks," *Physical Review E*, vol. 88, no. 6, p. 062819, 2013.
[38] M. E. Newman, "Scientific collaboration networks. i. network construction and fundamental results," *Physical review E*, vol. 64, no. 1, p. 016131, 2001.
[39] A. Kobren, B. Saha, and A. McCallum, "Paper matching with local fairness constraints," in *SIGKDD*, 2019, pp. 1247–1257.

Kai Wang received the BEng degree in Computer Science from Zhejiang University in 2016, and the PhD degree in Computer Science from the University of New South Wales in 2020. He is currently a research associate in the School of Computer Science and Engineering, University of New South Wales. His research interests lie in big data analytics, especially for the graph/network and spatial data.

Shuting Wang is currently working at Zhejiang Gongshang University. She is also conducting research as a volunteer researcher at the East China Normal University in data science Lab. She received Bachelor degree in 2014 from Jiangxi Normal university and Research-Master degree in 2017 from Zhejiang Gongshang University.

Wenjie Zhang received the PhD degree in computer science and engineering from the University of New South Wales, in 2010. She is currently a professor and ARC DECRA (Australian Research Council Discovery Early Career Researcher Award) fellow in the School of Computer Science and Engineering, the University of New South Wales, Australia. Her research interests lie in big data management and processing.

Ying Zhang is a Professor and ARC Future Fellow (2017- 2021) at Australia Artificial Intelligence Institute (AAII), the University of Technology, Sydney (UTS). He received his BSc and MSc degrees in Computer Science from Peking University, and PhD in Computer Science from the University of New South Wales. His research interests include query processing and analytics on large-scale data with focus on graphs and high dimensional data.



Lu Qin received the BEng degree from the Department of Computer Science and Technology, Renmin University of China, in 2006, and the PhD degree from the Department of Systems Engineering and Engineering Management, Chinese University of Hong Kong, in 2010. He is currently an associate professor with the Centre for Artificial Intelligence, University of Technology, Sydney. His research interests include big graph analytics and graph query processing.

Yuting Zhang received the BEng degree in Computer Science and Technology from Zhejiang A&F University in 2019. She is currently a Mphil candidate in the School of Computer Science and Engineering, University of New South Wales. Her research interests lie in graph databases, particularly in graph data analytics.

Efficient and Effective Community Search on Large-scale Bipartite Graphs

Kai Wang†, Wenjie Zhang†, Xuemin Lin†, Ying Zhang⋆, Lu Qin⋆, Yuting Zhang†
†University of New South Wales, ⋆University of Technology Sydney
kai.wang@unsw.edu.au, {zhangw, lxue}@cse.unsw.edu.au, {ying.zhang, lu.qin}@uts.edu.au, ytzunsw@gmail.com

***Abstract*—Bipartite graphs are widely used to model relationships between two types of entities. Community search retrieves densely connected subgraphs containing a query vertex, which has been extensively studied on unipartite graphs. However, community search on bipartite graphs remains largely unexplored. Moreover, all existing cohesive subgraph models on bipartite graphs can only be applied to measure the structure cohesiveness between two sets of vertices while overlooking the edge weight in forming the community. In this paper, we study the significant (α, β)-community search problem on weighted bipartite graphs. Given a query vertex q, we aim to find the significant (α, β)-community $\mathcal{R}$ of q which adopts (α, β)-core to characterize the engagement level of vertices, and maximizes the minimum edge weight (significance) within $\mathcal{R}$.**

To support fast retrieval of $\mathcal{R}$, we first retrieve the maximal connected subgraph of (α, β)-core containing the query vertex (the (α, β)-community), and the search space is limited to this subgraph with a much smaller size than the original graph. A novel index structure is presented which can be built in $O(\delta \cdot m)$ time and takes $O(\delta \cdot m)$ space where m is the number of edges in G, δ is bounded by $\sqrt{m}$ and is much smaller in practice. Utilizing the index, the (α, β)-community can be retrieved in optimal time. To further obtain $\mathcal{R}$, we develop peeling and expansion algorithms to conduct searches by shrinking from the (α, β)-community and expanding from the query vertex, respectively. The experimental results on real graphs not only demonstrate the effectiveness of the significant (α, β)-community model but also validate the efficiency of our query processing and indexing techniques.

I. Introduction

In many real-world applications, relationships between two different types of entities are modeled as bipartite graphs, such as customer-product networks [1], user-page networks [2] and collaboration networks [3]. Community structures naturally exist in these practical networks and *community search* has been extensively explored and proved useful on unipartite graphs [4]–[11]. Given a query vertex q, *community search* aims to find communities (connected subgraphs) containing q which satisfy specific cohesive constraints. In the literature, fair clustering methods [12]–[14] are used to find communities (i.e., clusters) under fairness constraints on bipartite graphs. However, they aim to find a set of clusters under a global optimization goal and do not aim to search a personalized community for a specific user. Nevertheless, no existing work has studied the *community search* problem on bipartite graphs. On bipartite graphs, various dense subgraph models are designed (e.g., (α, β)-core [15], [16], bitruss [17]–[19] and biclique [20]) which can be used as the cohesive measurement of a community. However, simply applying these cohesive measurements only ensures the structure cohesiveness of communities but ignores another important characteristic, the weight (or significance) of interactions between the two sets of vertices. For example, (α, β)-core is defined as the maximal subgraph where each vertex in upper layer has at least α neighbors and each vertex in lower layer has at least β neighbors. In the customer-movie network shown in Figure 1, each edge has a weight denoting the rating of a user to a movie. If the (α, β)-core model is applied to search a community of "Eric", e.g., the maximal connected subgraph of (3, 2)-core containing "Eric", we will get the community formed by the four users and the five movies on the left side. Note that, this community includes "Alien" (not liked by "Andy" or "Kane") and "Taylor" (who has less interest in this genre of movies).

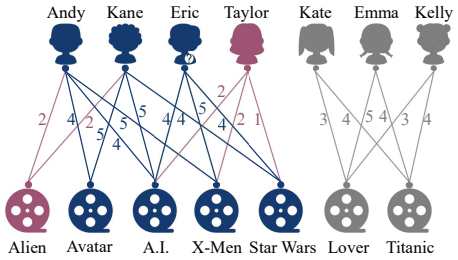


Fig. 1: A user-movie network

In this paper, we study the significant community search problem on weighted bipartite graphs, which is the first to study community search on bipartite graphs. Here, in a weighted bipartite graph G, each edge is associated with an edge weight. In addition, the weight (significance) of a community is measured by the minimum edge weight in it. A community with a high weight value indicates that every edge in the community represents a highly significant interaction. We propose the significant (α, β)-community model, which is the maximal connected subgraph containing the query vertex q that satisfies the vertex degree constraint from (α, β)-core, and has the highest graph significance. The intuition behind the new significant (α, β)-community model is to capture structure cohesiveness as well as interactions (edges) with high significance. In addition, if we maximize the weight value under given α and β, we can find the most significant subgraph while preserving the structure cohesiveness. For example, in Figure 1, the subgraph in blue color, which excludes "Alien" and "Taylor", is the significant (3, 2)-community of "Eric".

Applications. Finding the significant (α, β)-community has many real-world applications and we list some of them below.

• *Personalized Recommendation.* In user-item networks, users leave reviews for items with ratings. Examples include viewer-movie network in `IMDB` (https://www.imdb.com), reader-book network in `goodreads` (https://www.goodreads.com), etc.

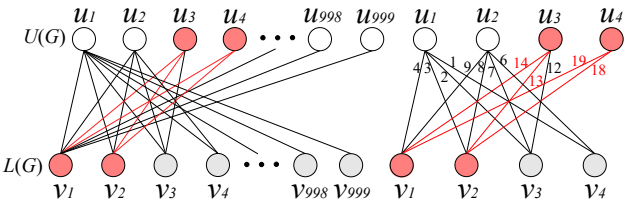
The platforms can utilize the significant (α, β)-community model to provide personalized recommendations. For example, based on the community found in Figure 1, we can put the people who give common high ratings ("Andy" and "Kane") on the recommended friend list of the query user ("Eric"). We can also recommend the movie ("Avatar") which the user is likely to be interested in to the query user ("Eric").

• *Fraud Detection.* In e-commerce platforms such as Amazon and Alibaba, customers and items form a customer-item bipartite graph in which an edge represents a customer purchased an item, and the edge weight measures the number of purchases or the total transaction amount. Fraudsters and the items they promote are prone to form cohesive subgraphs [15], [17]. Since the cost of opening fake accounts is increased with the improvement of fraud detection techniques, frauds cannot rely on many fake accounts [2]. Thus, the number of purchases or the total transaction amount per account is increased. Given a suspicious item or customer as the query vertex, our significant (α, β)-community model allows us to find the most suspicious fraudsters and related items in the customer-item bipartite graphs and reduce false positives.

• *Team Formation.* In a bipartite graph formed by developers and projects, an edge between a developer and a project indicates that the developer participates in the project, and the edge weight shows the corresponding contribution (e.g., number of tasks accomplished). A developer may wish to assemble a team with a proven track record of contributions in related projects, which can be supported by a significant (α, β)-community search over the bipartite graph.

Challenges. To obtain the significant (α, β)-community, we can iteratively remove the vertices without enough neighbors and the edges with small weights from the original graph. However, when the graph size is large and there are many vertices and edges that need to be removed, this approach is inefficient. For example, Figure 2(a) shows the graph G with 2,003 edges. We need to remove 1,999 edges from G to get the significant $(2, 2)$-community of u_3 with only 4 edges.

In this paper, we focus on indexing-based approaches. A straightforward idea is precomputing all the significant (α, β)-communities for all α, β, and q combinations. This idea is impractical since both structure cohesiveness and significance need to be considered. For different q and α, β values, the significant (α, β)-communities can be different and there does not exist hierarchical relationships among them. Therefore, we resort to a two-step approach. In the first step, we observe that the (α, β)-community always contains the significant (α, β)-community for a query vertex q. Here, (α, β)-community is the maximal connected subgraph containing q in the (α, β)-core (without considering the edge weights). For example, Figure 2(b) shows the $(2, 2)$-community of u_3 which contains the significant $(2, 2)$-community of u_3 and is much smaller than the original graph G. Therefore, we try to index all (α, β)-communities and use the one containing q as the starting point when querying w.r.t. q. In the second step, we compute the significant (α, β)-community based on the (α, β)-community obtained in the first step. To make our ideas practically applicable, we need to address the following challenges.



(a) A weighted bipartite graph G, $w(u, v) = 5 \cdot u.id - v.id$.

(b) The (2, 2)-community of u_3 in G.

Fig. 2: An example graph, the significant $(2, 2)$-community of u_3 is marked in red color

1) How to build an index to cover all (α, β)-communities.
2) How to bound the index size and the indexing time.
3) How to efficiently obtain the significant (α, β)-community from the (α, β)-community of a query vertex.

Our approaches. To address Challenge 1, we first propose the index I_{bs}^{α} to store all the (α, β)-communities. It is observed that the model of (α, β)-core has a hierarchical property. In other words, (α, β)-core $\subseteq (\alpha', \beta')$-core if $\alpha \geq \alpha'$ and $\beta \geq \beta'$. For example, in Figure 2, G itself is the $(1, 1)$-core, the induced subgraph of $\{u_1, ..., u_{999}, v_1, v_2, v_3, v_4\}$ is the $(1, 2)$-core and we can obtain the $(1, 3)$-core from the $(1, 2)$-core by excluding v_4. Motivated by this observation, all the $(1, \beta)$-community with $\beta \geq 1$ can be organized hierarchically in the $(1, 1)$-core. For each vertex existing in $(1, 1)$-core, we sort its neighbors according to the maximal β value where they exist in the $(1, \beta)$-core in non-increasing order. Then, when querying a $(1, \beta)$-community with $\beta \geq 1$, we only need to take the vertices and edges in this community using breath-first search. For example, if we want to query the $(1, 2)$-community of u_1, we first take the neighbors $\{v_1, v_2, v_3, v_4\}$ of u_1 and get u_1 to u_{999} after searching from v_1. By organizing all the $(\alpha, 1)$-cores where $\alpha \in [1, \alpha_{max}]$ in this manner, I_{bs}^{α} can cover all the (α, β)-communities. Similarly, we can also build the index I_{bs}^{β} which stores all the $(1, \beta)$-core where $\beta \in [1, \beta_{max}]$ to cover all the (α, β)-communities. Here α_{max} and β_{max} are the maximal valid α and β values in G respectively.

Reviewing I_{bs}^{α} and I_{bs}^{β}, we observe that $I_{bs}^{\alpha}(I_{bs}^{\beta})$ can be very large when high degree vertices exist in $U(G)(L(G))$. For example, I_{bs}^{α} needs to store 999 copies of neighbors of u_1 since u_1 is contained in $(999, 1)$-core. The same issue occurs when I_{bs}^{β} stores v_1's neighbors. To handle this issue and address Challenge 2, we further propose the degeneracy-bounded index I_{δ}. Here, the degeneracy (δ) is the largest number where the (δ, δ)-core is nonempty in G. Note that for each nonempty (α, β)-core (or (α, β)-community), we must have $min(\alpha, \beta) \leq \delta$. This is because it contradicts the definition of δ if an (α, β)-core with $\alpha > \delta$ and $\beta > \delta$ exists. In addition, according to the hierarchical property of the (α, β)-core model, all (α, β)-communities with $\alpha \leq \beta$ can be organized in the (α, α)-core and all (α, β)-communities with $\beta < \alpha$ can be organized in the (β, β)-core. In this manner, I_{δ} only needs to store all the (τ, τ)-cores for each $\tau \in [1, \delta]$ to cover all the (α, β)-communities. For example, in Figure 2, unlike I_{bs}^{α} which needs to store $(1, 1)$-core to $(999, 1)$-core, I_{δ} only needs to store $(1, 1)$-core, $(2, 2)$-core and $(3, 3)$-core since $\delta = 3$. Since the size of each (τ, τ)-core ($\tau \in [1, \delta]$) is bounded by $O(m)$, I_{δ} can be built in $O(\delta \cdot m)$ time and takes $O(\delta \cdot m)$ space to index all the (α, β)-communities.

To address Challenge 3, after retrieving the (α, β)-community $C_{\alpha,\beta}(q)$, we first propose the peeling algorithm SCS-Peel which iteratively removes the edge with the minimal weight from $C_{\alpha,\beta}(q)$ to obtain $\mathcal{R}$. For example, in Figure 2(b), to obtain the significant $(2,2)$-community of u_3, the edge (u_1, v_4) is the first edge to be removed in SCS-Peel. Observing that $\mathcal{R}$ can be much smaller than $C_{\alpha,\beta}(q)$ in many cases, we also propose the expansion algorithm SCS-Expand which iteratively adds the edge with maximal weights into an empty graph until $\mathcal{R}$ is found. In SCS-Expand, we derive several rules to avoid excessively validating $\mathcal{R}$.

Contribution. Our main contributions are listed as follows.

- We propose the model of significant (α, β)-community which is the first to study community search problem on (weighted) bipartite graphs.
- We develop a new two-step paradigm to search the significant (α, β)-community. Under this two-step paradigm, novel indexing techniques are proposed to support the retrieval of the (α, β)-community in optimal time. The index I_δ can be built in $O(\delta \cdot m)$ time and takes $O(\delta \cdot m)$ space where δ is bounded by $\sqrt{m}$ and is much smaller in practice. Note that the proposed indexing techniques can also be directly applied to retrieve the (α, β)-community on unweighted bipartite graphs in optimal time.
- We propose efficient query algorithms to extract the significant (α, β)-community from the (α, β)-community.
- We conduct comprehensive experiments on 11 real weighted bipartite graphs to evaluate the effectiveness of the proposed model and the efficiency of our algorithms.

II. Problem Definition

Our problem is defined over an undirected weighted bipartite graph $G(V{=}(U, L), E)$, where $U(G)$ denotes the set of vertices in the upper layer, $L(G)$ denotes the set of vertices in the lower layer, $U(G) \cap L(G) = \emptyset$, $V(G) = U(G) \cup L(G)$ denotes the vertex set, $E(G) \subseteq U(G) \times L(G)$ denotes the edge set. An edge e between two vertices u and v in G is denoted as (u, v) or (v, u). The set of neighbors of a vertex u in G is denoted as $N(u, G) = \{v \in V(G) \mid (u, v) \in E(G)\}$, and the degree of u is denoted as $deg(u, G) = |N(u, G)|$. We use n and m to denote the number of vertices and edges in G, respectively, and we assume each vertex has at least one incident edge. Each edge $e = (u, v)$ has a weight $w(e)$ (or $w(u, v)$). The size of G is denoted as $\mathsf{size}(G) = |E(G)|$.

Definition 1. ((α, β)-core) *Given a bipartite graph G and degree constraints α and β, a subgraph $R_{\alpha,\beta}$ is the (α, β)-core of G if (1) $deg(u, R_{\alpha,\beta}) \geq \alpha$ for each $u \in U(R_{\alpha,\beta})$ and $deg(v, R_{\alpha,\beta}) \geq \beta$ for each $v \in L(R_{\alpha,\beta})$; (2) $R_{\alpha,\beta}$ is maximal, i.e., any supergraph $G' \supset R_{\alpha,\beta}$ is not an (α, β)-core.*

Definition 2. ((α, β)-Connected Component) *Given a bipartite graph G and its (α, β)-core $R_{\alpha,\beta}$, a subgraph $C_{\alpha,\beta}$ is a (α, β)-connected component if (1) $C_{\alpha,\beta} \subseteq R_{\alpha,\beta}$ and $C_{\alpha,\beta}$ is connected; (2) $C_{\alpha,\beta}$ is maximal, i.e., any supergraph $G' \supset C_{\alpha,\beta}$ is not a (α, β)-connected component.*

Definition 3. ((α, β)-Community) *Given a vertex q, we call the (α, β)-connected component containing q the (α, β)-community, denoted as $C_{\alpha,\beta}(q)$.*

Definition 4. (Bipartite Graph Weight) *Given a bipartite graph G, the weight value of G denoted by $f(G)$ is defined as the minimum edge weight in G.*

After introducing the (α, β)-core and bipartite graph weight, we define the significant (α, β)-community as below.

Definition 5. (Significant (α, β)-Community) *Given a weighted bipartite graph G, degree constraints α, β and query vertex q, a subgraph $\mathcal{R}$ is the significant (α, β)-community of G if it satisfies the following constraints:*

1) **Connectivity Constraint.** *$\mathcal{R}$ is a connected subgraph which contains q;*
2) **Cohesiveness Constraint.** *Each vertex $u \in U(\mathcal{R})$ satisfies $deg(u, \mathcal{R}) \geq \alpha$ and each vertex $v \in L(\mathcal{R})$ satisfies $deg(v, \mathcal{R}) \geq \beta$;*
3) **Maximality Constraint.** *There exists no other $G' \subseteq C_{\alpha,\beta}(q)$ satisfying constraints 1) and 2) with $f(G') > f(\mathcal{R})$. In addition, there exists no other supergraph $G'' \supset \mathcal{R}$ satisfying constraints 1) and 2) with $f(G'') = f(\mathcal{R})$.*

Problem Statement. Given a weighted bipartite graph G, parameters α, β and a query vertex q, the *significant (α, β)-community search* problem aims to find the significant (α, β)-community (SC) in G.

Example 1. *Consider the bipartite graph G in Figure 2(a). Figure 2(b) shows the (2,2)-community of u_3. In addition, the significant (2,2)-community of u_3 is shown in Figure 2(b) (in red color) which is formed by the edges (u_3, v_1), (u_3, v_2), (u_4, v_1) and (u_4, v_2).*

Solution Overview. According to Definition 3 and Definition 5, we have the following lemma.

Lemma 1. *Given a weighted bipartite graph G, the significant (α, β)-community is unique, which is a subgraph of the (α, β)-community.*

Proof. Suppose there exist two different significant (α, β)-communities $\mathcal{R}_1$ and $\mathcal{R}_2$ where $f(\mathcal{R}_1) = f(\mathcal{R}_2)$, $\mathcal{R}_1 \not\subseteq \mathcal{R}_2$ and $\mathcal{R}_2 \not\subseteq \mathcal{R}_1$. Then $\mathcal{R}_3 = \mathcal{R}_1 \cup \mathcal{R}_2$ satisfies constraints 1) and 2) in Definition 5 with $f(\mathcal{R}_3) = f(\mathcal{R}_1) = f(\mathcal{R}_2)$. This violates the maximality constraint in Definition 5. Thus, the significant (α, β)-community is unique and is a subgraph of the (α, β)-community by definition. □

Following the above lemma, we can use indexing techniques to efficiently find the (α, β)-community first. In this manner, the search space is limited to a much smaller subgraph compared to G. Then, we further search on the (α, β)-community to identify the significant (α, β)-community. According to this two-step algorithmic framework, we present our techniques in the following sections.

III. Retrieve the (α, β)-community in optimal time

In this section, we explore indexing techniques to retrieve the (α, β)-community in an efficient way.

A. Basic Indexes

In [15], the authors propose the bicore index which can obtain the vertex set of the (α, β)-core (i.e., $V(R_{\alpha,\beta})$)

in optimal time. However, to obtain $C_{\alpha,\beta}(q)$ after having $V(R_{\alpha,\beta})$, we still need to traverse all the neighbors of each vertex in $C_{\alpha,\beta}(q)$ (starting from the query vertex) including those neighbors which are not in $C_{\alpha,\beta}(q)$. This process needs $O(|V(C_{\alpha,\beta}(q))| \cdot \sum_{v \in V(C_{\alpha,\beta}(q))} deg(v,G))$ time and when $\frac{|\text{size}(C_{\alpha,\beta}(q))|}{\sum_{v \in V(C_{\alpha,\beta}(q))} deg(v,G)}$ is small, it may need to access many additional edges not in the queried community. Motivated by this, we explore how to construct an index to support optimal retrieval of the (α, β)-community (i.e., optimal retrieval of (α, β)-connected components).

By Definition 1, we have the following lemma.

Lemma 2. *(α,β)-core $\subseteq$ (α',β')-core if $\alpha \geq \alpha'$ and $\beta \geq \beta'$.*

We also define the α-offset and the β-offset of a vertex as follows.

Definition 6. **(α-/β-offset)** *Given a vertex $u \in V(G)$ and an α value, its α-offset denoted as $s_a(u,\alpha)$ is the maximal β value where u can be contained in an (α, β)-core. If u is not contained in $(\alpha,1)$-core, $s_a(u,\alpha)$ = 0. Symmetrically, the β-offset $s_b(u,\beta)$ of u is the maximal α value where u can be contained in an (α, β)-core.*

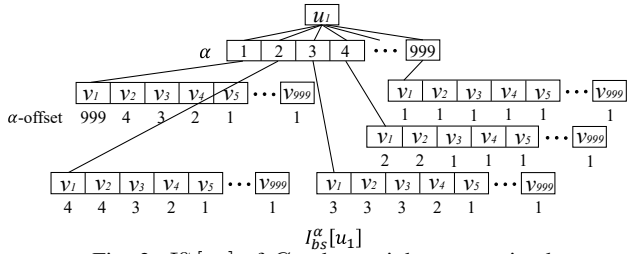
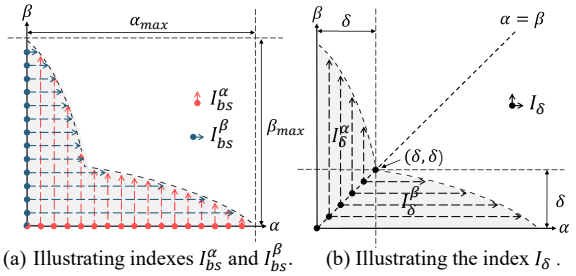


Fig. 3: $I^{\alpha}_{bs}[u_1]$ of G, edge weights are omitted

Algorithm 1: Index Construction of I^{α}_{bs}

Input: G
Output: I^{α}_{bs}
1 $\alpha \leftarrow 1$;
2 $\alpha_{max} \leftarrow$ the maximal vertex degree in $U(G)$;
3 **while** $\alpha \leq \alpha_{max}$ **do**
4 compute $s_a(u,\alpha)$ for each vertex $u \in V(G)$;
5 **foreach** $u \in (\alpha,1)$-*core* **do**
6 **foreach** $v \in N(u,G)$ **do**
7 **if** $s_a(v,\alpha) \geq 1$ **then**
8 $I^{\alpha}_{bs}[u][\alpha] \leftarrow \{v, w(u,v), s_a(v,\alpha)\}$;
9 sort $I^{\alpha}_{bs}[u][\alpha]$ in decreasing order of their α-offsets;
10 $\alpha \leftarrow \alpha + 1$;
11 **return** I^{α}_{bs};

Since (α, β)-core follows a hierarchical structure according to Lemma 2, an index can be constructed in the following way. For each vertex u, its α-offset indicates that u is contained in the $(\alpha, s_a(u,\alpha))$-core and is not contained in the $(\alpha, s_a(u,\alpha)+1)$-core. According to Lemma 2, if u is contained in the $(\alpha, s_a(u,\alpha))$-core, it is also contained in the (α, β)-core with $\beta \leq s_a(u,\alpha)$. As shown in Figure 4(a), the shaded area represents all the valid combinations of α and β where an (α, β)-community exists. As illustrated, we can organize the (α, β)-cores hierarchically and construct the basic index I^{α}_{bs} as shown in Algorithm 1. Firstly, we obtain α_{max} which is the maximal α value such that an $(\alpha,1)$-core exists and



(a) Illustrating indexes I^{α}_{bs} and I^{β}_{bs}. (b) Illustrating the index I_{δ} .

Fig. 4: Illustrating the ideas of indexing techniques

it is equal to the maximal vertex degree in $U(G)$. We then compute the α-offset for each vertex. For each vertex u and α combination (where u exists in $(\alpha,1)$-core), we create an adjacent list $I^{\alpha}_{bs}[u][\alpha]$ to store its neighbors. In $I^{\alpha}_{bs}[u][\alpha]$, we sort u's neighbors in non-increasing order of their α-offsets and remove these neighbors with α-offsets equal to zero. Figure 3 is an example which shows $I^{\alpha}_{bs}[u_1]$ of G in Figure 2(a). We can see that $I^{\alpha}_{bs}[u_1]$ contains the neighbors of u_1 of different α values.

Algorithm 2: Query based on I^{α}_{bs}

Input: G, q, α, β, I^{α}_{bs};
Output: $C_{\alpha,\beta}(q)$
1 $Q \leftarrow q$;
2 $visited(q) \leftarrow true$;
3 **while** Q *is not empty* **do**
4 $u \leftarrow Q.pop()$;
5 **foreach** $v \in I^{\alpha}_{bs}[u][\alpha]$ **do**
6 **if** $s_a(v,\alpha) \geq \beta$ **then**
7 $C_{\alpha,\beta}(q) \leftarrow (u,v)$ if $u \in L(G)$;
8 **if** $visited(v) = false$ **then**
9 $Q.push(v)$;
10 $visited(v) \leftarrow true$;
11 **else**
12 **break**;
13 **return** $C_{\alpha,\beta}(q)$;

Optimal retrieval of $C_{\alpha,\beta}(q)$ based on I^{α}_{bs}. Given a query vertex q, Algorithm 2 illustrates the query process of the (α, β)-community (i.e., $C_{\alpha,\beta}(q)$) based on I^{α}_{bs}. When querying $C_{\alpha,\beta}(q)$, we first put the query vertex into the queue. Then, we pop the vertex u from the queue, and visit the adjacent list $I^{\alpha}_{bs}[u][\alpha]$ to obtain the neighbors of u with α-offset $\geq \beta$. For each valid neighbor v, we add the edge (u,v) into $C_{\alpha,\beta}(q)$ if $u \in L(G)$ to avoid duplication. Then, we put these valid neighbors into the queue and repeat this process until the queue is empty. Since the neighbors are sorted in non-increasing order of their α-offsets, we can early terminate the traversal of the adjacent list when the α-offset of a vertex is smaller than the given β.

Lemma 3. *Given a bipartite graph G and a query vertex q, Algorithm 2 computes $C_{\alpha,\beta}(q)$ in $O(\text{size}(C_{\alpha,\beta}(q)))$ time, which is optimal.*

Proof. In Algorithm 2, for each $u \in Q$, since there is no duplicate vertex in $I^{\alpha}_{bs}[u][\alpha]$ and only its neighbor $v \in I^{\alpha}_{bs}[u][\alpha]$ with $s_a(v,\alpha) \geq \beta$ can be accessed, each u and v combination corresponds to an edge in $C_{\alpha,\beta}(q)$. In addition, since each vertex can be only added once into Q according to lines 8 - 10, Algorithm 2 computes $C_{\alpha,\beta}(q)$ in $O(\text{size}(C_{\alpha,\beta}(q)))$ time, which is optimal as it is linear to the result size. □

Example 2. *Considering the graph in Figure 2 and $I_{bs}^{\alpha}[u_1]$ in Figure 3, if we want to get the $(3,3)$-community of u_1 $C_{3,3}(u_1)$, we first traverse $I_{bs}^{\alpha}[u_1][3]$ to get all the neighbors with α-offsets ≥ 3 which are v_1, v_2 and v_3. The edges (u_1, v_1), (u_1, v_2) and (u_1, v_3) will be added into $C_{3,3}(u_1)$. Then, we go to the index nodes $I_{bs}^{\alpha}[v_1][3]$, $I_{bs}^{\alpha}[v_2][3]$ and $I_{bs}^{\alpha}[v_3][3]$ to get unvisited vertices u_2 and u_3 with α-offsets ≥ 3. The edges (u_2, v_1), (u_2, v_2), (u_2, v_3), (u_3, v_1), (u_3, v_2), (u_3, v_3) will be added into $C_{3,3}(u_1)$ when accessing $I_{bs}^{\alpha}[u_2][3]$ and $I_{bs}^{\alpha}[u_3][3]$.*

In addition, apart from I_{bs}^{α}, we can construct an index I_{bs}^{β} similarly based on β-offsets which also achieves optimal query processing. For each vertex u and β combination, we create an adjacent list to store its neighbors and we sort its neighbors in non-increasing order of their β-offsets (removing these neighbors with β-offsets = 0). When querying the $C_{\alpha,\beta}(q)$, we first go to the adjacent list indexing by q and β, and obtain the neighbors of q with β-offset $\geq \alpha$. Then we run a similar breadth-first search as Algorithm 2 shows. Using I_{bs}^{β}, we can also achieve optimal retrieval of $C_{\alpha,\beta}(q)$ which can be proved similarly as Lemma 3.

Complexity analysis of basic indexes. Storing I_{bs}^{α} needs $\mathsf{size}(I_{bs}^{\alpha}) = O(\sum_{\alpha=1}^{\alpha_{max}}(\mathsf{size}((\alpha,1)\text{-}core))$ space. Since $\sum_{\alpha=1}^{\alpha_{max}}(\mathsf{size}((\alpha,1)\text{-}core) \leq \sum_{\alpha=1}^{\alpha_{max}}(\mathsf{size}((1,1)\text{-}core))$, $\mathsf{size}(I_{bs}^{\alpha})$ is also bounded by $O(\alpha_{max} \cdot m)$. Similarly, I_{bs}^{β} needs $O(\sum_{\beta=1}^{\beta_{max}}(\mathsf{size}((1,\beta)\text{-}core)) = O(\beta_{max} \cdot m)$ space.

In addition, the time complexity of constructing I_{bs}^{α} is $\mathsf{TC}(I_{bs}^{\alpha}) = O(\alpha_{max} \cdot m)$. This is because for α from 1 to α_{max}, we can perform the peeling algorithm on each $(\alpha, 1)$-core to get the α-offset for each vertex first. This process needs $O(\alpha_{max} \cdot m)$ time. Then, for each vertex u, we create at most α_{max} adjacent lists to store its neighbors which needs $O(\alpha_{max} \cdot m)$ time. Similarly, the time complexity of constructing I_{bs}^{β} is $\mathsf{TC}(I_{bs}^{\alpha}) = O(\beta_{max} \cdot m)$.

B. The Degeneracy-bounded Index I_δ

Reviewing I_{bs}^{α} and I_{bs}^{β}, we can see that it is hard to handle high degree vertices in $U(G)(L(G))$ using $I_{bs}^{\alpha}(I_{bs}^{\beta})$. This is because if these vertices exist in an (α, β)-core with large α (or β) value, according to Lemma 2, I_{bs}^{α} or I_{bs}^{β} may need large space to store several copies of the neighbors of these high degree vertices. For example, in Figure 3, I_{bs}^{α} needs to store multiple copies of neighbors of u_1 since u_1 is contained in $(999, 1)$-core. The same issue occurs when I_{bs}^{β} stores v_1's neighbors. Thus, in this part, we explore how to effectively handle these high degree vertices and build an index with smaller space consumption.

Firstly, we give the definition of degeneracy as follows.

Definition 7. (Degeneracy) *Given a bipartite graph G, the degeneracy of G denoted as δ is the largest number where (δ, δ)-core is nonempty in G.*

Note that, δ is bounded by $\sqrt{m}$ and in practice, it is much smaller than $\sqrt{m}$ [15].

Lemma 4. *Given a bipartite graph G, a nonempty (α, β)-core in G must have $min(\alpha, \beta) \leq \delta$.*

Proof. We prove this lemma by contradiction. Suppose a nonempty (α, β)-core exists in G with $\alpha < \beta$ and $\alpha > \delta$. Then we will have $\alpha \geq \delta + 1$ and $\beta \geq \delta + 1$ which contradicts to the definition of δ. Similarly, we cannot have an nonempty (α, β)-core existing in G with $\beta < \alpha$ and $\beta > \delta$. Thus, a nonempty (α, β)-core in G must have $min(\alpha, \beta) \leq \delta$. □

Based on Lemma 4, we can observe that, given query parameters α and β, a partial index of I_{bs}^{α} which only stores adjacent lists of u for each u and α combinations with $\alpha \leq \delta$ is enough to handle queries when $\alpha = min(\alpha, \beta)$. Similarly, a partial index of I_{bs}^{β} which only stores adjacent lists under (u, β) combinations with $\beta \leq \delta$ is enough to handle queries when $\beta = min(\alpha, \beta)$. Based on the above observation, we propose the index I_δ as follows.

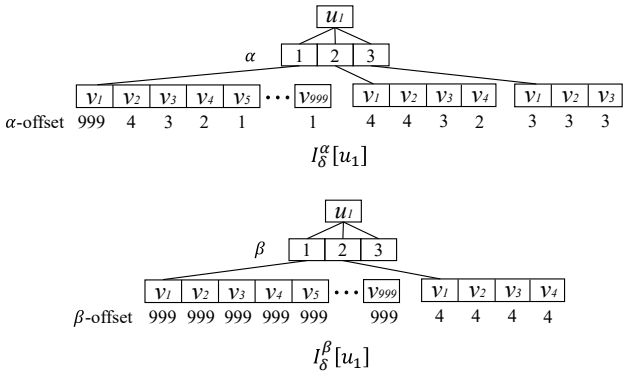


Fig. 5: $I_\delta[u_1]$ of G, edge weights are omitted

Index Overview. I_δ contains two parts I_δ^{α} and I_δ^{β} to cover all the (α, β)-communities as illustrated in Figure 4(b).

In I_δ^{α}, for each vertex u and $\alpha \leq \delta$ where u exists in the (α, α)-core, we create an adjacent list $I_\delta^{\alpha}[u][\alpha]$ to store its neighbors. Note that, the neighbors are sorted in non-increasing order of their α-offsets and the neighbors with α-offsets less than α are removed.

In I_δ^{β}, for each vertex u and $\beta \leq \delta$ where u exists in the (β, β)-core, we create an adjacent list $I_\delta^{\beta}[u][\beta]$ to store its neighbors with β-offsets larger than β. The neighbors are sorted in non-increasing order of their β-offsets and the neighbors with β-offsets less or equal than β are removed. Figure 5 is an example of $I_\delta[u_1]$ of G in Figure 2(a). We can see that it consists of two parts $I_\delta^{\alpha}[u_1]$ and $I_\delta^{\beta}[u_1]$.

Optimal retrieval of $C_{\alpha,\beta}(q)$ based on I_δ. The query processing of $C_{\alpha,\beta}(q)$ based on I_δ is similar to the query processing based on the basic indexes. The difference is that we need to choose to use I_δ^{α} or I_δ^{β} at first. If the query parameter $\alpha \leq \delta$, we use I_δ^{α} to support the query process. Otherwise, we go for I_δ^{β} to obtain the $C_{\alpha,\beta}(q)$. Since only valid edges are touched using I_δ, we can also obtain $C_{\alpha,\beta}(q)$ in $O(\mathsf{size}(C_{\alpha,\beta}(q)))$ time which is optimal. The proof of optimality is similar as Lemma 3 and we omit it here due to the space limit.

Example 3. *Considering G in Figure 2 and $I_\delta[u_1]$ in Figure 5, if we want to get the $(3,3)$-community of u_1 $C_{3,3}(u_1)$, since $\alpha = \beta$, we first traverse $I_\delta^{\alpha}[u_1][3]$ to get all the neighbors with α-offsets ≥ 3, which are v_1, v_2 and v_3. The edges (u_1, v_1), (u_1, v_2) and (u_1, v_3) will be added into $C_{3,3}(u_1)$. Then, we go to the index nodes $I_\delta^{\alpha}[v_1][3]$, $I_\delta^{\alpha}[v_2][3]$ and $I_\delta^{\alpha}[v_3][3]$ to get unvisited vertices u_2 and u_3 with α-offsets ≥ 3. The edges*

(u_2, v_1), (u_2, v_2), (u_2, v_3), (u_3, v_1), (u_3, v_2), (u_3, v_3) *will be added into* $C_{3,3}(u_1)$ *when accessing* $I_\delta^\alpha[u_2][3]$ *and* $I_\delta^\alpha[u_3][3]$.

Lemma 5. *The space complexity of* I_δ *denoted as* $\mathsf{size}(I_\delta)$ *is* $O(2 \cdot \sum_{\tau=1}^{\delta} \mathsf{size}(R_{\tau,\tau})) = O(\delta \cdot m)$.

Proof. For each $\alpha \in [1, \delta]$ and $u \in R_{\alpha,\alpha}$, we need to store at most $deg(u, R_{\alpha,\alpha})$ u's neighbors in I_δ^α. Thus, I_δ^α needs $O(\sum_{\alpha=1}^{\delta} \sum_{u \in R_{\alpha,\alpha}} deg(u, R_{\alpha,\alpha})) = O(\sum_{\alpha=1}^{\delta} \mathsf{size}(R_{\alpha,\alpha}))$=$O(\delta \cdot m)$ space. Similarly, I_δ^β also needs $O(\sum_{\beta=1}^{\delta} (\mathsf{size}(R_{\beta,\beta})) = O(\delta \cdot m)$ space. In total, the space for storing I_δ is $O(\delta \cdot m)$. □

Algorithm 3: Degeneracy-bounded Index Construction

```
Input: G
Output: I_δ
1  τ ← 1;
2  compute δ using the k-core decomposition algorithm;
3  while τ ≤ δ do
4      compute α-offset s_a(u, τ) and β-offset s_b(u, τ) for each
         vertex u ∈ V(G);
5      foreach u ∈ (τ, τ)-core do
6          foreach v ∈ N(u, G) do
7              if s_a(v, τ) ≥ τ then
8                  I_δ^α[u][τ] ← {v, w(u, v), s_a(v, τ)};
9              if s_b(v, τ) > τ then
10                 I_δ^β[u][τ] ← {v, w(u, v), s_b(v, τ)};
11         sort I_δ^α[u][τ] in decreasing order of their α-offsets;
12         sort I_δ^β[u][τ] in decreasing order of their β-offsets;
13     τ ← τ + 1;
14 return I_δ;
```

Index Construction. The construction algorithm of I_δ is shown in Algorithm 3. We first compute δ using the k-core decomposition algorithm in [21] since δ is equal to the maximum core number in G. Then, for each vertex u, we compute its α-offset for each $\alpha \leq \delta$ and its β-offset for each $\beta \leq \delta$. These values can be obtained by the peeling algorithm in [16]. Then, we loop τ from 1 to δ and add the valid neighbors of the vertices in the (τ, τ)-$core$ into I_δ.

Lemma 6. *The time complexity of Algorithm 3 is* $O(\delta \cdot m)$.

Proof. For each τ, we can first obtain the $(\tau, 1)$-core and the α-offsets of all the vertices can be computed using the core decomposition algorithm [21] in $O(m)$ time. The β-offsets of all the vertices can also be computed in $O(m)$ time similarly. Then, sorting $I_\delta^\alpha[u][\tau]$ and $I_\delta^\beta[u][\tau]$ for each vertex u also needs $O(m)$ time in total by using bin sort [21]. Since $\tau \in [1, \delta]$, the time complexity of Algorithm 3 is $O(\delta \cdot m)$. □

Discussion of index maintenance. When graphs are updated dynamically, it is inefficient to reconstruct the indexes from scratch. Thus, we discuss the main idea of the incremental algorithms for maintaining I_δ. Other indexes in this paper can be maintained in a similar way.

Edge insertion. Suppose an edge (u, v) is inserted into G. For each $\alpha \leq \delta$, we first add $u(v)$ into $I_\delta^\alpha[v][\alpha]$ ($I_\delta^\alpha[u][\alpha]$) if $s_a(u, \alpha) \geq \alpha$ ($s_a(v, \alpha) \geq \alpha$). Then, for each $\alpha \leq \delta$, we track changes of the α-offsets of the vertices. Note that, only the α-offsets of the vertices in $S_\alpha^+ = V(C_{\alpha, s_a(u,\alpha)}(u)) \cup V(C_{\alpha, s_a(v,\alpha)}(v))$ can be changed. This is because for each vertex not in S_α^+, it either does not connect to $u(v)$ or $u(v)$ already exists in any (α, β)-connected component it belongs to when fixing α. Thus, we obtain the induced subgraph of S_α^+ from I_δ and compute the new α-offsets of the vertices in S_α^+ by peeling the subgraph. If the α-offset of the vertex $u' \in S_\alpha^+$ is changed, we only need to update $I_\delta^\alpha[v'][\alpha]$ where $v' \in N(u', G)$. Similarly, for each $\beta \leq \delta$, only the β-offsets of the vertices in $S_\beta^+ = V(C_{s_b(u,\beta),\beta}(u)) \cup V(C_{s_b(v,\beta),\beta}(v))$ can be changed. We compute the new β-offsets of these vertices and update I_δ^β in a similar way. Note that after the new edge is inserted, the value of δ can be increased by 1. If δ is increased, we compute the new index elements for $\delta + 1$.

Edge removal. Suppose an edge (u, v) is removed from G. For each $\alpha \leq \delta$, we first remove $u(v)$ from $I_\delta^\alpha[v][\alpha]$ ($I_\delta^\alpha[u][\alpha]$) if $s_a(u, \alpha) \geq \alpha$ ($s_a(v, \alpha) \geq \alpha$). Similar as the insertion case, for each α, only the α-offsets of the vertices in $S_\alpha^- = V(C_{\alpha,1}(u) \backslash C_{\alpha, s_a(u,\alpha)+1}(u)) \cup V(C_{\alpha,1}(v) \backslash C_{\alpha, s_a(v,\alpha)+1}(v))$ can be changed. Thus, we recompute the α-offsets of these vertices and update I_δ^α. I_δ^β can also be updated similarly.

Remark. Although we are dealing with the weighted bipartite graph in this work, the indexing techniques proposed in this section can directly support finding the (α, β)-community on unweighted bipartite graph.

IV. Query the significant (α, β)-community

According to the definition of significant (α, β)-community, the subgraph $C_{\alpha,\beta}(q)$ obtained from the index already satisfies the connectivity constraint and the cohesiveness constraint. Thus, in this section, we introduce two query algorithms to obtain the significant (α, β)-community from $C_{\alpha,\beta}(q)$ to further satisfy the maximality constraint.

A. Peeling Approach

Algorithm 4: SCS-Peel

```
Input: G, q, α, β;
Output: R
1  get C_{α,β}(q) from the index;
2  S ← ∅; Q ← ∅;
3  sort edges of C_{α,β}(q) in non-decreasing order by weights;
4  while C_{α,β}(q) is not empty do
5      w_min ← the minimal edge weight in C_{α,β}(q)
6      foreach (u, v) ∈ C_{α,β}(q) with w(u, v) = w_min do
7          remove (u, v) from C_{α,β}(q);
8          S.add((u, v));
9          if deg(u, C_{α,β}(q)) < α ∧ u ∉ Q then
10             Q.push(u);
11         if deg(v, C_{α,β}(q)) < β ∧ v ∉ Q then
12             Q.push(v);
13     while Q is not empty do
14         u' ← Q.pop();
15         foreach v' ∈ N(u', C_{α,β}(q)) do
16             remove (u', v') from C_{α,β}(q);
17             S.add((u', v'));
18             if v' does not have enough degree then
19                 Q.push(v');
20                 if v'=q then
21                     G' ← S ∪ C_{α,β}(q);
22                     Obtain R from G';
23                     return R;
24     S = ∅;
```

Here, we introduce the peeling approach as shown in Algorithm 4. Firstly, we retrieve $C_{\alpha,\beta}(q)$ based on the indexes

proposed in Section III. Note that if all the edge weights are equal in $C_{\alpha,\beta}(q)$, we can just return $C_{\alpha,\beta}(q)$ as the result. Otherwise, we sort the edges in $C_{\alpha,\beta}(q)$ in non-decreasing order by weights and we initialize an edge set S and a queue Q to empty. After that, we run the peeling process on $C_{\alpha,\beta}(q)$. In each iteration, we remove each edge (u, v) with the minimal weight in $C_{\alpha,\beta}(q)$. Also, we add (u, v) into an edge set S which records the edges removed in this iteration. Due to the removal of (u, v), there may exist many vertices which do not have enough degree to stay in $C_{\alpha,\beta}(q)$ (i.e., for vertex $u \in U(C_{\alpha,\beta}(q))$, $deg(u, C_{\alpha,\beta}(q)) < \alpha$ or for vertex $v \in L(C_{\alpha,\beta}(q))$, $deg(v, C_{\alpha,\beta}(q)) < \beta$), we also remove the edges of these vertices and add the edges into S. We run the peeling process until q does not satisfy the degree constraint. Then, we create G'= $S \cup C_{\alpha,\beta}(q)$ since the edges removed in this iteration need to be recovered to form the $\mathcal{R}$. Finally, we remove the vertices without enough degree in G' and run a breath-first search from q on G' to get the connected subgraph containing q which is $\mathcal{R}$.

Theorem 1. *The* SCS-Peel *algorithm correctly solves the significant (α, β)-community search problem.*

Proof. According to Lemma 1, $\mathcal{R}$ is a subgraph of $C_{\alpha,\beta}(q)$. Suppose there is a $G' \subseteq C_{\alpha,\beta}(q)$ satisfying the connected constraint and the cohesiveness constraint and has $f(G') > f(\mathcal{R})$. Since we always peel the edge with the minimal weight, G' will be found after $\mathcal{R}$. Since we peel $C_{\alpha,\beta}(q)$ until the degree of q is not enough, $q \in G'$ will not have enough degree which contradicts the cohesiveness constraint. For the same reason, there exists no $G'' \supset \mathcal{R}$ with $f(G'') = f(\mathcal{R})$. Thus, this theorem holds. □

Time complexity. SCS-Peel has three phases. Retrieving $C_{\alpha,\beta}(q)$ based on the index needs $(\mathsf{size}(C_{\alpha,\beta}(q)))$ time. Then, sorting the edges in $C_{\alpha,\beta}(q)$ needs $\mathsf{sort}(C_{\alpha,\beta}(q))$ time which will be $O(\mathsf{size}(C_{\alpha,\beta}(q)) \cdot (log(\mathsf{size}(C_{\alpha,\beta}(q)))))$ if we use quick sort or $O(m')$ if we use bin sort where m' equals to the maximal weight in $C_{\alpha,\beta}(q)$. After that, the whole peeling process requires $O(\mathsf{size}(C_{\alpha,\beta}(q))$ time. In total, the time complexity of SCS-Peel is $O(\mathsf{sort}(C_{\alpha,\beta}(q)) + \mathsf{size}(C_{\alpha,\beta}(q)))$.
Space complexity. In the SCS-Peel algorithm, we need only $O(\mathsf{size}(C_{\alpha,\beta}(q)))$ space to store the edges in $C_{\alpha,\beta}(q)$ apart from the space used by the indexes.

B. Expansion Approach

Unlike the peeling approach which iteratively removes the edge with the minimal weight from $C_{\alpha,\beta}(q)$, in this part, we introduce the expansion approach SCS-Expand. SCS-Expand first initializes a subgraph G^* as empty. Then it iteratively adds the edges with the maximal weight to G^* (from $C_{\alpha,\beta}(q)$) until G^* contains $\mathcal{R}$. In this manner, if $\mathsf{size}(\mathcal{R})$ is much smaller than $\mathsf{size}(C_{\alpha,\beta}(q))$, SCS-Expand can retrieve $\mathcal{R}$ in a more efficient way compared to the peeling approach.

Following the above idea, we add edges with the maximal weight in $C_{\alpha,\beta}(q)$ to G^* (and remove them from $C_{\alpha,\beta}(q)$) in each iteration. However, when adding an edge into G^*, it may not connect to q. Note that, we cannot discard these edges immediately since they may be connected to q due to the later coming edges. Thus, the connected subgraphs in G^* should be maintained in each iteration. With the help of union-find data structure [22], the connected subgraphs in G^* can be maintained in constant amortized time, and we can efficiently obtain the connected subgraph containing q in G^*.

Checking the existence of $\mathcal{R}$ in C^*. Suppose C^* is the connected subgraph containing q in G^*, we can easily observe that $\mathcal{R}$ can only be found in the iteration where C^* is changed. In addition, we have the following bounds which can let us know whether $\mathcal{R}$ is contained in C^*.

Lemma 7. *Given a connected subgraph C^*, if $\mathcal{R} \subseteq C^*$, we have:*

$$\alpha\beta - \alpha - \beta \leq |E(C^*)| - |U(C^*)| - |L(C^*)|$$

Proof. Since C^* is a connected subgraph, we have $|E(C^*)| \geq |U(C^*)| + |L(C^*)| - 1$. According to the cohesiveness constraint of $\mathcal{R}$, $\mathcal{R}$ has at least $max\{\alpha \cdot |U(\mathcal{R})|, \beta \cdot |L(\mathcal{R})|\}$ edges. In addition, the number of incident edges of vertices in $V(C^*) \backslash V(\mathcal{R})$ is at least $|U(C^*)|+|L(C^*)|-|U(\mathcal{R})|-|L(\mathcal{R})|$ to ensure C^* is connected.

Hence, when $\alpha \cdot |U(\mathcal{R})| \geq \beta \cdot |L(\mathcal{R})|$, $|E(C^*)| \geq |U(C^*)| + |L(C^*)| - |U(\mathcal{R})| - |L(\mathcal{R})| + \alpha \cdot |U(\mathcal{R})|$. It is immediate that $\alpha \leq |L(\mathcal{R})|$ and $\beta \leq |U(\mathcal{R})|$. Thus, we have $(\alpha-1) \cdot |U(\mathcal{R})| - |L(\mathcal{R})| \leq |E(C^*)| - |U(C^*)| - |L(C^*)|$. By transformation, we have $(\alpha - 1) \cdot \beta - \alpha \leq |E(C^*)| - |U(C^*)| - |L(C^*)|$. Then, we get $\alpha\beta - \alpha - \beta \leq |E(C^*)| - |U(C^*)| - |L(C^*)|$.

When $\alpha \cdot |U(\mathcal{R})| < \beta \cdot |L(\mathcal{R})|$, $|E(C^*)| \geq |U(C^*)| + |L(C^*)| - |U(\mathcal{R})| - |L(\mathcal{R})| + \beta \cdot |L(\mathcal{R})|$, we can also get $\alpha\beta - \alpha - \beta \leq |E(C^*)| - |U(C^*)| - |L(C^*)|$. □

Lemma 8. *Given a connected subgraph $C^* \subseteq G$, if $\mathcal{R} \subseteq C^*$, it must contain α vertices where each vertex u of them has $deg(u, C^*) \geq \beta$, and it must contain β vertices where each vertex v of them has $deg(v, C^*) \geq \alpha$. In addition, the query vertex should be one of these vertices.*

Proof. This lemma directly follows from Definition 5. □

Based on the above lemmas, we can skip checking the existence of $\mathcal{R}$ if the constraints are not satisfied. It is still costly if we check each C^* satisfies the constraints since we need to perform the peeling algorithm on C^* using $O(\mathsf{size}(C^*))$ time. To mitigate this issue, we set an expansion parameter $\epsilon > 1$ to control the number of checks. Firstly, we check C^* when it first satisfies the constraints in the Lemma 7 and Lemma 8. After that, we only check C^* if its size is at least ϵ times than the size of its last check. Here we choose $\epsilon = 2$ and the reasons are as follows. Suppose for each C_i^* ($i \in [1, d]$, d is the total number of checks) which needs to be checked, $\mathsf{size}(C_i^*)$ is exactly ϵ times of $\mathsf{size}(C_{i-1}^*)$. Since we can find $\mathcal{R}$ in the final check, we have $\mathsf{size}(C_d^*) < \epsilon(\mathsf{size}(\mathcal{R}))$. The time complexity of using the peeling algorithm to check all these connected subgraphs is $O(\sigma_{i=1}^{d}\mathsf{size}(C_i^*))$, and $\sigma_{i=1}^{d}\mathsf{size}(C_i^*)$ = $\mathsf{size}(C^d) + \frac{1}{\epsilon}\mathsf{size}(C^d) + \frac{1}{\epsilon^2}\mathsf{size}(C^d) + ... + \frac{1}{\epsilon^d}\mathsf{size}(C^d)$, we can have $O(\sigma_{i=1}^{d}\mathsf{size}(C_i^*)) = O(\epsilon(\frac{1}{\epsilon-1})\mathsf{size}(\mathcal{R}))$. We choose $\epsilon = 2$ since $\frac{1}{\epsilon-1}$ achieves the smallest value at $\epsilon = 2$.

The SCS-Expand **Algorithm.** We present the SCS-Expand algorithm as shown in Algorithm 5. Firstly, we retrieve $C_{\alpha,\beta}(q)$ based on the indexes proposed in Section III. We can return

$C_{\alpha,\beta}(q)$ if all the edge weights are equal in $C_{\alpha,\beta}(q)$. Otherwise, we sort the edges in $C_{\alpha,\beta}(q)$ in non-increasing order by weights and we initialize G^* and C^* to empty. After that, we iteratively add each edge (u, v) with the maximal weight (in $C_{\alpha,\beta}(q)$) to G^* and remove the added edge from $C_{\alpha,\beta}(q)$. Note that the size and edges of the connected subgraphs in G^* will be maintained using the union-find structure. If C^* is changed, we will check whether C^* satisfies the constraints in the Lemma 7 and Lemma 8. After that, we will check if its size grows at least ϵ times. If it is, we run the peeling process to check whether $\mathcal{R}$ is contained by C^*. In this peeling process, we iteratively remove all the vertices without enough degree from C^*. If q is not removed from C^*, we run Algorithm 4 to obtain $\mathcal{R}$. The algorithm finishes if it finds $\mathcal{R}$ in C^*.

Algorithm 5: SCS-Expand

```
   Input: G, q, α, β, ϵ;
   Output: R
1  G* ← ∅; C* ← ∅; pre_size = 0;
2  get C_{α,β}(q) from the index;
3  sort edges of C_{α,β}(q) in non-increasing order by weights;
4  while C_{α,β}(q) is not empty do
5      w_max ← the maximal edge weight in C_{α,β}(q)
6      foreach (u, v) ∈ C_{α,β}(q) with w(u, v) = w_max do
7          remove (u, v) from C_{α,β}(q);
8          G*.add((u, v));
9          maintain the connected subgraphs in G*;
10     if C* is not changed or violates constraints in Lemma 7
         and Lemma 8 then
11         continue;
12     if size(C*) ≥ pre_size · ϵ then
13         pre_size ← size(C*);
14     else
15         continue;
16     Remove the vertices without enough degree from C*;
17     if q ∈ C* then
18         run Algorithm 4 lines 3 - 23, replace C_{α,β}(q) with a
             copy of C*
```

Theorem 2. *The* SCS-Expand *algorithm correctly solves the significant (α, β)-community search problem.*

Proof. According to Definition 5, $\mathcal{R}$ is a subgraph of $C_{\alpha,\beta}(q)$. Since we always expand the edge with the maximal weight, the connected subgraph C^* will always contain all the edges in $C_{\alpha,\beta}(q)$ which is connected to q with weights $\geq f(C^*)$. According to Theorem 1, SCS-Peel can correctly check whether $\mathcal{R}$ exists in C^*. Thus, this theorem holds. □

Time complexity. In SCS-Expand, retrieving $C_{\alpha,\beta}(q)$ based on the index needs $O(\mathsf{size}(C_{\alpha,\beta}(q)))$ time. Then, sorting the edges in $C_{\alpha,\beta}(q)$ needs $O(\mathsf{sort}(C_{\alpha,\beta}(q)))$ time. After that, the whole expansion process requires $O(\sum_{i=1}^{d} \mathsf{size}(C_i^*))$ time where d is the number of subgraphs which survive to Algorithm 5 line 16. In total, the time complexity of SCS-Expand is $O(\mathsf{sort}(C_{\alpha,\beta}(q)) + \sigma_{i=1}^{d}\mathsf{size}(C_i^*))$.

Space complexity. In the SCS-Expand algorithm, we need $O(\mathsf{size}(C_{\alpha,\beta}(q)))$ space to store the edges in $C_{\alpha,\beta}(q)$ except the space used by indexes.

Remark. One may also consider using binary search over the weights to find $\mathcal{R}$. To validate each weight, it still needs to run the peeling process which needs $O(m)$ time. In addition, if the result is found under a weight threshold, the algorithm stops and the search space does not need to be reduced anymore. Thus, this binary search method only needs to expand the search space which is similar to SCS-Expand. We implement the binary search approach and find its running time is similar to that of SCS-Expand (0.86×-1.08×) on all the datasets. Note that when the number of distinct weight values are small, SCS-Binary can have better performance than SCS-Expand.

TABLE I: Summary of Datasets

Dataset	$\|E\|$	$\|U\|$	$\|L\|$	δ	α_{max}	β_{max}	$\|R_{\delta,\delta}\|$
BS	433K	77.8K	186K	13	8,524	707	13.6K
GH	440K	56.5K	121K	39	884	3,675	21.5K
SO	1.30M	545K	96.6K	22	4,917	6,119	13.0K
LS	4.41M	992	1.08M	164	55,559	773	177K
DT	5.74M	1.62M	383	73	378	160,047	30.5K
AR	5.74M	2.15M	1.23M	26	12,180	3,096	36.6K
PA	8.65M	1.43M	4.00M	10	951	119	639
ML	25.0M	162K	59.0K	636	32,202	81,491	2.12M
DUI	102M	833K	33.8M	183	24,152	29,240	2.30M
EN	122M	3.82M	21.5M	254	1,916,898	62,330	1.03M
DTI	137M	4.51M	33.8M	180	1,057,753	6,382	242K

V. Experiments

In this section, we first evaluate the effectiveness of the significant (α, β)-community model. Then, we evaluate the efficiency of the techniques for retrieving (α, β)-communities and significant (α, β)-communities.

A. Experiments setting

Algorithms. Our empirical studies are conducted against the following designs:

• *Techniques to retrieve the (α, β)-community.* The query algorithms: 1) the online query algorithm Q_o in [16], and the query algorithms based on the following indexes: 2) Q_v based on the bicore index I_v proposed in [15], 3) Q_{opt} based on the degeneracy-bounded index I_δ in Section III-B. The indexes: 1) the bicore index I_v, 2) basic indexes I_{bs}^{α} and I_{bs}^{β}, 3) I_δ.

• *Algorithms to retrieve the significant (α, β)-community.* 1) the peeling algorithm SCS-Peel, 2) the expansion algorithm SCS-Expand in Section IV and 3) a baseline algorithm SCS-Baseline which iteratively expands the edges (with larger weight value) from the connected component containing q of the whole graph rather than from $C_{\alpha,\beta}(q)$.

The algorithms are implemented in C++ and the experiments are run on a Linux server with Intel Xeon 2650 v3 2.3GHz processor and 768GB main memory. *We terminate an algorithm if the running time is more than 10^4 seconds.*

Datasets. We use 11 real datasets in our experiments which are `Bookcrossing` (BC), `Github` (GH), `StackOverflow` (SO), `Lastfm` (LS), `Discogs` (DT), `Amazon` (AR), `DBLP` (PA), `MovieLens` (ML), `Delicious-ui` (DUI), `Wikipedia-en` (EN) and `Delicious-ti` (DTI). All the datasets we use can be found in KONECT (http://konect.uni-koblenz.de). Note that, for the datasets without weights (i.e., DT and PA), we choose the random walk with restart model [23] to compute the node relevance and generate the weights. Here several other models [24], [25] can also be applied.

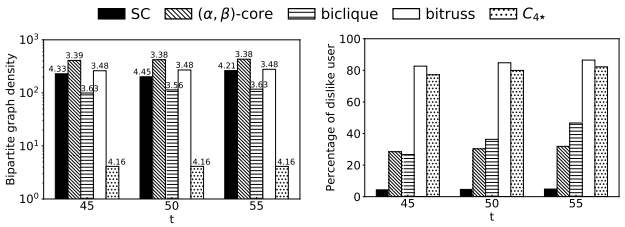
The summary of datasets is shown in Table I. U and L are vertex layers, $|E|$ is the number of edges. δ is the degeneracy.

α_{max} and β_{max} are the largest value of α and β where a $(\alpha, 1)$-core or $(1, \beta)$-core exists, respectively. $|R_{\delta,\delta}|$ denotes the number of edges in $R_{\delta,\delta}$ in each dataset. In addition, M denotes 10^6 and K denotes 10^3.

B. Effectiveness evaluation

In this section, we evaluate the effectiveness of our model on `MovieLense` which contains 25M ratings (ranging from 1 to 5) from 162K users (U) on 59K movies (L).

We compare the significant (α, β)-community model with the (α, β)-core, k-bitruss (setting $k = \alpha \cdot \beta$) [18] and maximal biclique [20] models. We also add a community $C_{4\star}$ which is the induced subgraph of all the movies with average ratings at least 4. Note that, we use the connected components of the query vertex as the result when considering different models.



(a) Bipartite graph density (b) Percentage of dislike users

Fig. 6: Evaluating the community quality, varying $\alpha, \beta = t$

Evaluating the community quality. Suppose a user wants to find some friends who are also fans of comedy movies. We extract the subgraph formed by the ratings on comedy movies and perform community search algorithms. Figure 6(a) shows the bipartite graph density which is computed as $d(G) = |E(G)|/\sqrt{|U(G)||L(G)|}$ [26]. We can see that the communities produced by (α, β)-core, bitruss, biclique and SC all have high densities comparing with $C_{4\star}$ since the structure cohesiveness is considered in these models. Thus, the users in $C_{4\star}$ are loosely connected with each other and have fewer interactions. In addition, the average ratings (i.e., the numbers on the top of each bar) indicate that SC can always return a group of users with higher average ratings than (α, β)-core, bitruss and biclique. We also show the number of dislike users in Figure 6(b). A user is a dislike user if he/she gives fewer than 0.6α good ratings (i.e., rating $\geq$ 4), who is not likely to be a fan of comedies. We can see that SC contains fewer number of dislike users comparing with all the other models because both weight and structure cohesiveness are considered. Thus, the users in SC are considered as good candidates to be recommended to the query user. Note that the percentage of dislike users in bitruss and $C_{4\star}$ is very high. This is because bitruss ensures the structure cohesiveness using the butterfly (i.e., 2×2-biclique) and a user can exist in a k-bitruss with a large k value if he/she only watched a few number of hot movies. In addition, $C_{4\star}$ does not ensure the structure cohesiveness and there exist many users who only watched few high rating movies.

Case study. We conduct queries using parameters $q = 6778, \alpha = 45, \beta = 45$ on comedy movies. The statistics of query results are shown in Table II. $|U|$ and $|M|$ denote the total number of users and movies in the community, respectively. R_{avg} and R_{min} denote the average and minimal rating in the community, respectively. M_{avg} is the average number of movies a user watched in the community and Sim is the jaccard similarity between each community and SC. For the biclique model, here we use a maximal biclique containing q with at least 45 vertices in each layer. We can see that SC contains reasonable number of users and vertices with higher average rating and minimal rating in the community than the others. We also show the representative components of the communities using (α, β)-core and SC in Figure 7. We can see that (α, β)-core contains users who do not like such movies and movies that are not liked by such users. This is because (α, β)-core only considers structure cohesiveness and ignores the edge weights. We can observe that M_{avg} of $C_{4\star}$ is only 2.39 since the structure cohesiveness is not considered in $C_{4\star}$. Thus, $C_{4\star}$ contains many users who only watched a few number of high rating movies and these users are loosely connected with the query user. Among these models, only SC considers both weight and structure cohesiveness, which is not similar to other communities compared here. In SC, each user has given at least 45 times 4.5-star ratings on these comedy movies and the movies are reviewed as 4.5-star at least 45 times by the users. Thus, the quality of the users and movies found by SC can be guaranteed and highly recommended to the query user.

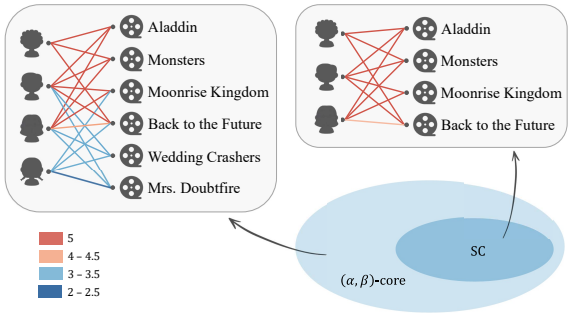


Fig. 7: Representative components of real-life communities

TABLE II: Statistics of query results, $q = 6{,}778$

Models	$\|U\|$	$\|M\|$	R_{avg}	R_{min}	M_{avg}	Sim (%)
SC	2,127	670	4.81	4.50	63.47	100
(α, β)-core	34,466	2491	3.39	0.5	110.03	7.57
bitruss	158,183	2,985	3.48	0.5	35.87	1.74
biclique	65	45	3.45	0.5	45	2.39
$C_{4\star}$	114,915	387	4.16	0.5	2.39	1.82

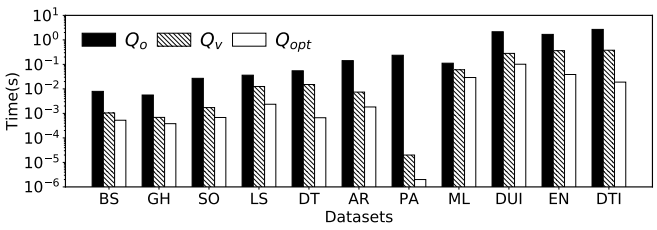
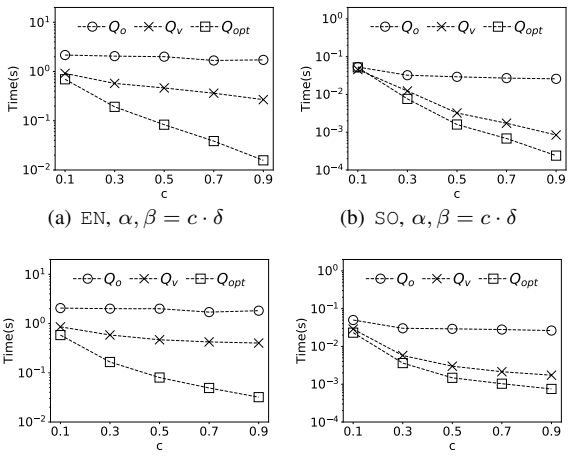


Fig. 8: Retrieving the (α, β)-communities

C. Evaluation of retrieving (α, β)-community

In this part, we evaluate the proposed indexing techniques to retrieve the (α, β)-community.

Query time. *1) Performance on all the datasets.* We first evaluate the performance on all the datasets by setting α and



(a) EN, $\alpha, \beta = c \cdot \delta$ (b) SO, $\alpha, \beta = c \cdot \delta$

(c) EN, $\alpha = 0.5 \cdot \delta$, $\beta = c \cdot \delta$ (d) SO, $\alpha = c \cdot \delta$, $\beta = 0.5 \cdot \delta$

Fig. 9: Retrieving the (α, β)-communities, varying α and β

β to 0.7δ. In Figure 8, we can observe that Q_{opt} significantly outperforms Q_o and Q_v on all the datasets. This is because Q_{opt} is based on I_δ which can achieve optimal retrieval of (α, β)-communities. Especially, on large datasets such as DUI, EN and DTI, the Q_{opt} algorithm is one to two orders of magnitude faster than Q_o and is up to 20× faster than Q_v.

2) Varying α and β. We also vary α and β to assess the performance of these algorithms. In Figure 9(a) and (b), α and β are varied simultaneously. We can observe that when α and β are small, the performance of these algorithms is similar. This is because only a few number of edges are removed from the original graph when the query parameters are small. When α and β are large, the resulting (α, β)-communities are much smaller than the original graph. Thus, Q_{opt} is much faster than Q_o and Q_v. In Figure 9(c) and (d), we fix α (or β) and vary the other one and the trends are similar.

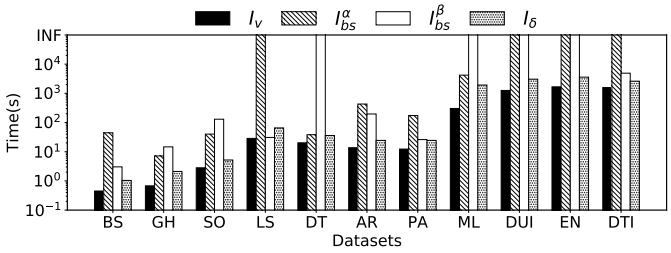


Fig. 10: Index construction time

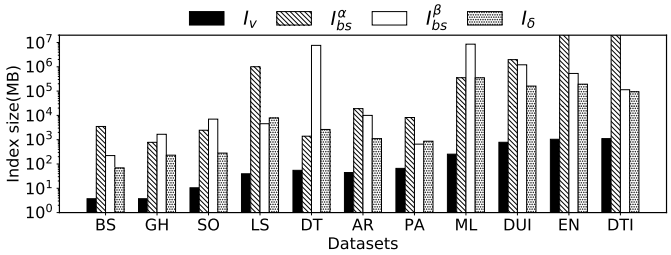


Fig. 11: Index size

Evaluating index construction time and index size. In this part, we evaluate the index size and index construction time.

1) Index construction time. In Figure 10, we can see that I_δ can be efficiently constructed on all the datasets since it only needs the same low constructing time complexity as I_v ($O(\delta m)$). In addition, constructing I_δ is slightly slower than constructing I_v which is reasonable since I_v only contains vertex information of (α, β)-cores while I_δ contains edge information which can support optimal retrieval of (α, β)-communities. The time for constructing I_{bs}^{α} and I_{bs}^{β} highly depends on α_{max} and β_{max}. Thus, it is very slow (or even unaccomplished) on the datasets where these two values are large such as DUI and EN.

2) Index size. In Figure 11, we evaluate the size of these indexes. If an index cannot be built within the time limit, we report the expected size of it. We can see that size(I_δ) is smaller than size(I_{bs}^{α}) and size(I_{bs}^{β}) on almost all the datasets. I_v is the index with the minimal size since it only contains vertex information.

D. Evaluation of retrieving significant (α, β)-community

Here we evaluate the performance of the algorithms (SCS-Baseline, SCS-Peel, and SCS-Expand) for querying significant (α, β)-communities. In these algorithms, we use Q_{opt} to support the optimal retrieval of (α, β)-community. In each test, we randomly select 100 queries and take the average.

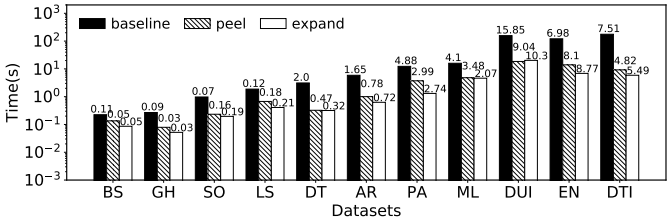
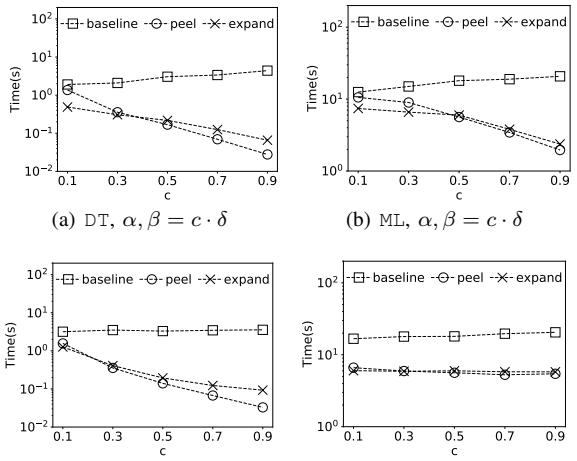


Fig. 12: Query performance on different datasets

Evaluating the performance on all the datasets. In Figure 12, we evaluate the performance of SCS-Baseline, SCS-Peel, and SCS-Expand on all the datasets. We also report the standard deviation on the top of each bar. We can see that SCS-Expand and SCS-Peel are significantly faster than SCS-Baseline, especially on large datasets. This is because, with the help of the two-step framework, the search space of SCS-Peel and SCS-Expand is limited in $C_{\alpha,\beta}(q)$, while SCS-Baseline needs to consider all edges in the connected component containing q of the whole graph. We can also see in Table I that $|R_{\delta,\delta}|$ is much smaller than $|E|$. Since $C_{\delta,\delta}(q) \subseteq R_{\delta,\delta}$, when we choose relatively larger parameters, the search space of SCS-Peel and SCS-Expand is much smaller than SCS-Baseline. In addition, we can see that on most datasets, SCS-Expand is on average more efficient than SCS-Peel. However, the standard deviations of SCS-Expand and SCS-Peel are large. This is because SCS-Peel and SCS-Expand both need more time to handle the cases when α and β are small and SCS-Expand is usually much faster than SCS-Peel.

Evaluating the effect of query parameters α and β. In Figure 13, we vary α and β on two datasets DT and ML. From Figure 13(a) and (b), we can see that, when α and β are small, SCS-Expand is more efficient than SCS-Peel. In addition, the running time of SCS-Peel and SCS-Expand decreases as α (or β) increases. Note that the efficiency of these two algorithms largely depends on the size of the (α, β)-community containing q (i.e., size($C_{\alpha,\beta}(q)$), which determines



(a) DT, $\alpha, \beta = c \cdot \delta$ (b) ML, $\alpha, \beta = c \cdot \delta$

(c) DT, $\alpha = c \cdot \delta, \beta = 0.5 \cdot \delta$ (d) ML, $\alpha = 0.5 \cdot \delta, \beta = c \cdot \delta$

Fig. 13: Effect of α and β

the search space) and the size of the final result (i.e., size($\mathcal{R}$), which relates to the actual computation cost). In most cases, when α and β are large, the size of $C_{\alpha,\beta}(q)$ is small and $\mathcal{R}$ is expected to be large since more edges are needed in $\mathcal{R}$ to satisfy the cohesiveness constraints. Thus, the edges need to be peeled are usually few and SCS-Peel is more efficient than SCS-Expand. When α and β are small, the search space (i.e., $C_{\alpha,\beta}(q)$) can be large and $\mathcal{R}$ is expected to be small. Thus, SCS-Expand is usually more efficient than SCS-Peel in these cases. In most cases, we can determine to use SCS-Peel or SCS-Expand according to the choice of α and β.

TABLE III: Running time under different weight distribution

Algorithms	AE	RW	UF	SK
SCS-Baseline	0.03s	3.12s	4.42s	4.31s
SCS-Peel	0.03s	0.34s	0.48s	0.45s
SCS-Expand	0.03s	0.31s	0.41s	0.36s

Evaluate the effect of weight distribution. In Table III, we evaluate the effect of weight distribution on DT dataset. We test four weight distributions: (1) AE: the weights are all equal; (2) RW: the weights are generated using the random walk with restart model [23]; (3) UF: the weights follow uniform distribution; (4) SK: the weights follow skewed normal distribution with skewness = 1.02. When all the edge weights are equal (AE) which can be considered as a special case, all three algorithms can just return $C_{\alpha,\beta}(q)$ after efficiently scanning $C_{\alpha,\beta}(q)$. Note that the performances of these three algorithms are not very sensitive to the other three distributions. This is because both weight and structure cohesiveness are considered in our problem and the impact of RW/SK/UF weight distributions are limited.

VI. Related Work

To the best of our knowledge, this paper is the first to study community search over bipartite graphs. Below we review two closely related areas, community search on unipartite graphs and cohesive subgraph models on bipartite graphs.

Community search on unipartite graphs. On unipartite graphs, community search is conducted based on different cohesiveness models such as k-core [4]–[7], [27]–[35], k-truss [8], [9], [36]–[39], clique [40], [40], [41]. Interested readers can refer to [11] for a recent comprehensive survey.

Based on k-core, [4] and [5] study online algorithms for k-core community search on unipartite graphs. In [6], Barbieri et al. propose a tree-like index structure for the k-core community search. Using k-core, Fang et al. [7] further integrate the attributes of vertices to identify community and the spatial locations of vertices are considered in [27], [28]. For the truss-based community search, [8], [36] study the triangle-connected model and [9] studies the closest model. In [40], the authors study the problem of densest clique percolation community search. However, the edge weights are not considered in any of the above works and their techniques cannot be easily extended to solve our problem. On edge-weighted unipartite graphs, the k-core model is applied to find cohesive subgraphs in [42], [43]. They use a function to associate the edge weights with vertex degrees and the edge weights are not considered as a second factor apart from the graph structure. Thus, these works do not aim to find a cohesive subgraph with both structure cohesiveness and high weight (significance). Under their settings, a subgraph with loose structure can be found in the result. For example, a vertex can be included in the result if it is only incident with one large-weight edge. In [37], the k-truss model is adopted on edge-weighted graphs to find communities. However, the k-truss model is based on the triangle structure which does not exist on bipartite graphs. One may also consider using the graph projection technique [44] to generate a unipartite projection from the original (weighted) bipartite graph. The drawback of this approach is twofold. Firstly, it can cause information loss and edge explosion [19]. Secondly, it is not easy to project a weighted bipartite graph and handle the projected graph using existing methods. This is because we need to consider two kinds of weights (i.e., the original edge weight and the structure weight generated from another layer) on the projected graph.

Finding cohesive subgraphs on bipartite graphs. On bipartite graphs, several existing works [15], [16], [45], [46] extend the k-core model on unipartite graph to the (α, β)-core model. Based on the butterfly structure [47], [17]–[19] study the bi-truss model in bipartite graphs which is the maximal subgraph where each edge is contained in at least k butterflies. [20] studies the biclique enumeration problem. However, the above works only consider the structure cohesiveness and ignore the edge weights which are important as validated in the experiments. In the literature, fair clustering problems [12]–[14] are studied to find communities (i.e., clusters) under fairness constraints on bipartite graphs. The problem is inherently different and the techniques are not applicable to the problem studied in this paper. An interesting work in [48] studies the paper matching problem in peer-review process which also finds dense subgraphs on bipartite graphs. However, their flow-based techniques are often used to solve a matching problem while our problem is not modeled as a matching problem.

VII. Conclusion

In this paper, we study the significant (α, β)-community search problem. To solve this problem efficiently, we follow a two-step framework which first retrieves the (α,

β)-community, and then identifies the significant (α, β)-community from the (α, β)-community. We develop a novel index I_δ to retrieve the (α, β)-community in optimal time. In addition, we propose efficient peeling and expansion algorithms to obtain the significant (α, β)-community. We conduct extensive experiments on real-world graphs, and the results demonstrate the effectiveness of the significant (α, β)-community model and the proposed techniques.

VIII. Acknowledgment

Xuemin Lin is supported by NSFC61232006, 2018YFB1003504, ARC DP200101338, ARC DP180103096 and ARC DP170101628. Lu Qin is supported by ARC FT200100787. Wenjie Zhang is supported by ARC DP180103096 and ARC DP200101116. Ying Zhang is supported by FT170100128 and ARC DP180103096. We would like to thank Yizhang He for his proofreading.

References

[1] J. Wang, A. P. De Vries, and M. J. Reinders, "Unifying user-based and item-based collaborative filtering approaches by similarity fusion," in *SIGIR*. ACM, 2006, pp. 501–508.

[2] A. Beutel, W. Xu, V. Guruswami, C. Palow, and C. Faloutsos, "Copycatch: stopping group attacks by spotting lockstep behavior in social networks," in *WWW*. ACM, 2013, pp. 119–130.

[3] M. Ley, "The DBLP computer science bibliography: Evolution, research issues, perspectives," in *Proc. Int. Symposium on String Processing and Information Retrieval*, 2002, pp. 1–10.

[4] W. Cui, Y. Xiao, H. Wang, and W. Wang, "Local serach of communities in large graphs," in *SIGMOD*, 2014, pp. 991–1002.

[5] M. Sozio and A. Gionis, "The community-search problem and how to plan a succesful cocktail party," in *SIGKDD*, 2010, pp. 939–948.

[6] N. Barbieri, F. Bonchi, E. Galimberti, and F. Gullo, "Efficient and effective community search," *Data mining and knowledge discovery*, vol. 29, no. 5, pp. 1406–1433, 2015.

[7] Y. Fang, R. Cheng, S. Luo, and J. Hu, "Effective community search for large attributed graphs," *PVLDB*, vol. 9, no. 12, pp. 1233–1244, 2016.

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

[9] X. Huang, L. V. S. Lakshmanan, J. X. Yu, and H. Cheng, "Approximate closest community search in networks," *PVLDB*, vol. 9, no. 4, pp. 276–287, 2015.

[10] X. Huang and L. V. Lakshmanan, "Attribute-driven community search," *PVLDB*, vol. 10, no. 9, pp. 949–960, 2017.

[11] Y. Fang, X. Huang, L. Qin, Y. Zhang, R. Cheng, and X. Lin, "A survey of community search over big graphs," *VLDB J.*, vol. 29, no. 1, pp. 353–392, 2020.

[12] F. Chierichetti, R. Kumar, S. Lattanzi, and S. Vassilvitskii, "Fair clustering through fairlets," in *Advances in Neural Information Processing Systems*, 2017, pp. 5029–5037.

[13] S. Ahmadi, S. Galhotra, B. Saha, and R. Schwartz, "Fair correlation clustering," *arXiv preprint arXiv:2002.03508*, 2020.

[14] S. Ahmadian, A. Epasto, M. Knittel, R. Kumar, M. Mahdian, B. Moseley, P. Pham, S. Vassilvtiskii, and Y. Wang, "Fair hierarchical clustering," *arXiv preprint arXiv:2006.10221*, 2020.

[15] B. Liu, L. Yuan, X. Lin, L. Qin, W. Zhang, and J. Zhou, "Efficient (α, β)-core computation: An index-based approach," in *WWW*. ACM, 2019, pp. 1130–1141.

[16] D. Ding, H. Li, Z. Huang, and N. Mamoulis, "Efficient fault-tolerant group recommendation using alpha-beta-core," in *CIKM*, 2017, pp. 2047–2050.

[17] K. Wang, X. Lin, L. Qin, W. Zhang, and Y. Zhang, "Efficient bitruss decomposition for large-scale bipartite graphs," in *ICDE*. IEEE, 2020, pp. 661–672.

[18] Z. Zou, "Bitruss decomposition of bipartite graphs," in *DASFAA*. Springer, 2016, pp. 218–233.

[19] A. E. Sarıyüce and A. Pinar, "Peeling bipartite networks for dense subgraph discovery," in *WSDM*. ACM, 2018, pp. 504–512.

[20] Y. Zhang, C. A. Phillips, G. L. Rogers, E. J. Baker, E. J. Chesler, and M. A. Langston, "On finding bicliques in bipartite graphs: a novel algorithm and its application to the integration of diverse biological data types," *BMC bioinformatics*, vol. 15, no. 1, p. 110, 2014.

[21] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo, "K-core decomposition of large networks on a single pc," *PVLDB*, vol. 9, no. 1, pp. 13–23, 2015.

[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[23] H. Tong, C. Faloutsos, and J.-Y. Pan, "Fast random walk with restart and its applications," in *ICDM*. IEEE, 2006, pp. 613–622.

[24] X. Chen, L. Lai, L. Qin, and X. Lin, "Structsim: Querying structural node similarity at billion scale," in *ICDE*. IEEE, 2020, pp. 1950–1953.

[25] G. Jeh and J. Widom, "Simrank: a measure of structural-context similarity," in *SIGKDD*, 2002, pp. 538–543.

[26] R. Kannan and V. Vinay, *Analyzing the structure of large graphs*. Rheinische Friedrich-Wilhelms-Universität Bonn Bonn, 1999.

[27] Y. Fang, R. Cheng, X. Li, S. Luo, and J. Hu, "Effective community search over large spatial graphs," *PVLDB*, vol. 10, no. 6, pp. 709–720, 2017.

[28] K. Wang, X. Cao, X. Lin, W. Zhang, and L. Qin, "Efficient computing of radius-bounded k-cores," in *ICDE*. IEEE, 2018, pp. 233–244.

[29] M. Ghafouri, K. Wang, F. Zhang, Y. Zhang, and X. Lin, "Efficient graph hierarchical decomposition with user engagement and tie strength," in *DASFAA*. Springer, 2020, pp. 448–465.

[30] C. Zhang, F. Zhang, W. Zhang, B. Liu, Y. Zhang, L. Qin, and X. Lin, "Exploring finer granularity within the cores: Efficient (k, p)-core computation," in *ICDE*. IEEE, 2020, pp. 181–192.

[31] B. Liu, F. Zhang, C. Zhang, W. Zhang, and X. Lin, "Corecube: Core decomposition in multilayer graphs," in *WISE*. Springer, 2019, pp. 694–710.

[32] Y. Fang, Z. Wang, R. Cheng, H. Wang, and J. Hu, "Effective and efficient community search over large directed graphs," *TKDE*, vol. 31, no. 11, pp. 2093–2107, 2018.

[33] Y. Fang, Z. Wang, R. Cheng, X. Li, S. Luo, J. Hu, and X. Chen, "On spatial-aware community search," *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 4, pp. 783–798, 2018.

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

[35] F. Zhang, W. Zhang, Y. Zhang, L. Qin, and X. Lin, "Olak: an efficient algorithm to prevent unraveling in social networks," *Proceedings of the VLDB Endowment*, vol. 10, no. 6, pp. 649–660, 2017.

[36] E. Akbas and P. Zhao, "Truss-based community search: a truss-equivalence based indexing approach," *PVLDB*, vol. 10, no. 11, pp. 1298–1309, 2017.

[37] Z. Zheng, F. Ye, R.-H. Li, G. Ling, and T. Jin, "Finding weighted k-truss communities in large networks," *Information Sciences*, vol. 417, pp. 344–360, 2017.

[38] B. Liu, F. Zhang, W. Zhang, X. Lin, and Y. Zhang, "Efficient community search with size constraint," in *ICDE*. IEEE, 2021.

[39] F. Zhang, C. Li, Y. Zhang, L. Qin, and W. Zhang, "Finding critical users in social communities: The collapsed core and truss problems," *TKDE*, 2018.

[40] L. Yuan, L. Qin, W. Zhang, L. Chang, and J. Yang, "Index-based densest clique percolation community search in networks," *TKDE*, vol. 30, no. 5, pp. 922–935, 2017.

[41] Y. Fang, K. Yu, R. Cheng, L. V. S. Lakshmanan, and X. Lin, "Efficient algorithms for densest subgraph discovery," *PVLDB*, vol. 12, no. 11, pp. 1719–1732, Jul. 2019.

[42] A. Garas, F. Schweitzer, and S. Havlin, "A k-shell decomposition method for weighted networks," *New Journal of Physics*, vol. 14, no. 8, p. 083030, 2012.

[43] M. Eidsaa and E. Almaas, "S-core network decomposition: A generalization of k-core analysis to weighted networks," *Physical Review E*, vol. 88, no. 6, 2013.

[44] M. E. Newman, "Scientific collaboration networks. i. network construction and fundamental results," *Physical review E*, vol. 64, no. 1, p. 016131, 2001.

[45] Y. He, K. Wang, W. Zhang, X. Lin, and Y. Zhang, "Exploring cohesive subgraphs with vertex engagement and tie strength in bipartite graphs," *arXiv preprint arXiv:2008.04054*, 2020.

[46] B. Liu, L. Yuan, X. Lin, L. Qin, W. Zhang, and J. Zhou, "Efficient (α, β)-core computation in bipartite graphs," *VLDB J.*, pp. 1–25, 2020.

[47] K. Wang, X. Lin, L. Qin, W. Zhang, and Y. Zhang, "Vertex priority based butterfly counting for large-scale bipartite networks," *PVLDB*, vol. 12, no. 10, pp. 1139–1152, 2019.

[48] A. Kobren, B. Saha, and A. McCallum, "Paper matching with local fairness constraints," in *SIGKDD*, 2019, pp. 1247–1257.