Response Letter for 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

We would like to thank the reviewers for their insightful and invaluable comments. Our paper has been carefully revised according to the comments. We give a summary of the revision below, following the suggestions by the meta-reviewer.

1) We improve the presentation of this paper by providing more intuitions, rewriting the hard-to-read text and conducting a thorough proof-reading to fix the typos.
2) We conduct more experiments to justify the importance and effectiveness of the proposed model and algorithms.
3) We add more discussions of existing studies to better position our work in the literature.

Below is our point-to-point response.

Response to Reviewer #4

Comment 1.1 (W1, D2): *Introduction is not clear. I recommend that authors present their intuitions with respect to the example (especially in Challenges and our approach). The lemma proofs need to be detailed.*
Response: Thanks for your suggestions to improve the presentation of this paper. We rewrite the introduction especially the challenges and the our approach parts. We give more intuitions w.r.t. the example in Figure 2 to illustrate our ideas. We also provide more details in the proofs of lemmas 1, 3, 4, 5 and 6 as suggested. ■

Comment 1.2 (D1): *Discussion of related works in fair clustering literature and paper matching techniques in peer review. It is a slightly different problem statement but worth a mention in the related work and intro.*
Response: Thanks for providing the references. In introduction, we add "*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.*" In related work, we add "*Fair clustering problems [12]-[14] are studied to find 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 [34] 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 we aim to find a community with structure cohesiveness and high significance, which is not modeled as a matching problem.*" ■

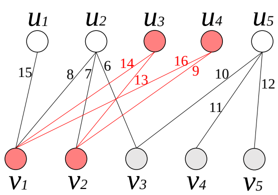
Comment 1.3 (D3): *a) Can the weights be negative or 0? b) Definition 5: G' is not defined.*
Response: a) Thanks for the comment. We make this clear in Section IV: "*The weights can be negative or 0 in our algorithms. We only require that any two edges are comparable given their weights regardless of whether they are positive or not.*" b) Thanks for pointing this out. Actually, G' is not necessary an (α, β)-community since we define the (α, β)-community as the maximal subgraph containing q in the (α, β)-core (Definition 2). Thus, there exists only one (α, β)-community in G and G' should be a subgraph of the (α, β)-community. To make the definition more rigorous, we modify the maximality constraint in Definition 5 and clarify that G' is a subgraph of the (α, β)-community. We also give an example of the (α, β)-community in Example 1. ■

Comment 1.4 (D4): *Section 3 claims that the retrieval is optimal. It is not discussed how it is optimal.*
Response: Thanks for the comment and we appreciate your carefulness. We add Lemma 3 to discuss and prove that the query algorithm is optimal based on the proposed index I^{α}_{bs}. In addition, the optimality of the query algorithms based on indexes I^{β}_{bs} and I_{δ} can be proved similarly as Lemma 3 and we clarify this in Section III. ■

Comment 1.5 (D5): *Why not consider binary search over the weights? It will reduce the complexity to logarithmic.*
Response: Excellent comments. Indeed, using binary search over the weights can achieve a logarithmic number of validations of weights. However, to validate each weight, we still need to run the peeling process which needs $O(m)$ time. In our SCS-Expand algorithm, we start from a small subgraph and double the graph size in each round of validation until a valid community is found. Thus, we can also guarantee that the number of validations is logarithmic (regarding the number of edges). In the revision, we implement the binary search approach and find that its running time is similar to that of SCS-Expand (0.86×-1.08×) on all the 11 datasets. We also add discussions of this binary search approach in Section IV.B in the revised paper. ■

Comment 1.6 (D6): *I think union-find data structure is very popular and intuitive to be discussed directly.*
Response: Thanks for the suggestion. In the revised paper, we update Section IV.B carefully and remove the detailed discussion of the union-find structure. ■

Comment 1.7 (D7): *Experiments: a) The edge weight is chosen randomly. I recommend using similarity measure to get weights. b) Section VB uses a small subset of ML dataset. Any reason why? c) The algorithms Peel and expand are not* *compared with any baseline. d) There is no baseline that justifies the choice of considering this two step procedure to solve the problem. Example baseline can be: Consider the adjacency list of q in decreasing weight and iteratively add by checking for alpha-beta. e) I think authors should at least show the trade-off between varying alpha, beta. f) What is the impact of edge weights on the second component? For example if all edge weights are equal then second procedure is not needed vs. when it is a skewed distribution.*



Response: Thank you for the detailed suggestions. In the revised paper, we improve the experimental evaluations according to each item. (a): As suggested, we apply the random walk with restart model to compute the node relevance and generate the weights accordingly. Note that, in all the datasets we used, only the weights in PA and DT datasets are synthetic. (b): A smaller dataset is used in the original submission for ease of presenting the results. In the revised version, we utilize a large ML dataset with 25M edges and devise a better way to show our case studie in Section V.B. (c, d): As suggested, we design a baseline algorithm (SCS-Baseline) which expands from the edges linked to q (i.e., $q's$ adjacent list). Note that, we cannot only select the weights from the neighbors of q. For example, in the above figure, the significant $(2, 2)$-community $\mathcal{R}$ of u_3 is marked in red color and $f(\mathcal{R})$=9. The resulting weight is not equal to the weights of any edges incident to u_3. Indeed, the weights of the neighbors of q can be used as a starting point of the algorithm (i.e., the community should at least include α or β number of q' neighbors). The main difference between SCS-Baseline and SCS-Expand is that SCS-Baseline iteratively expands the edges (with larger weight value) from the connected component containing q of the whole graph rather than from $C_{\alpha,\beta}(q)$. We evaluate this baseline algorithm in Section V.D and the experimental results show that SCS-Peel and SCS-Expand are significantly faster than SCS-Baseline. (e): We vary α and β separately in the experiments (Figure 9(c)(d) and Figure 13(c)(d)) and update the analysis accordingly. (f): As suggested, we add experiments to evaluate the effect of weight distribution and related analysis in Section V.D. ■

Response to Reviewer #5

Comment 2.1 (W1, D2): *The paper lacks a comparison with several cohesive subgraphs on bipartite graphs such as (α, β)-core, bitruss and biclique.*

Response: Thanks for your insightful suggestion. In the revised paper, we compare 4 different baselines with our significant (α, β)-community in Section V.B. From the structure cohesiveness perspective, we compare our model with the (α, β)-core, bitruss and biclique models as suggested. From the weight perspective, we add a baseline community $C_{4\star}$ which is obtained from the induced subgraph of all the movies with average ratings at least 4. The experimental results reported in Section V.B validate the effectiveness of the significant (α, β)-community model which considers both weight and structure cohesiveness. ■

Comment 2.2 (W2): *It is not necessary to solve the significant (alpha, beta) communities in two steps.*

Response: Thanks for the comment. The intuition of our two-step solution is to find the final result within a smaller subgraph (i.e., $C_{\alpha,\beta}(q)$) rather than the whole graph. To justify our idea, we add a baseline algorithm SCS-Baseline which only involves one step that iteratively expands the edges (with large weights) from the connected component containing q of the whole graph rather than from $C_{\alpha,\beta}(q)$. In the revised paper, we report the experimental results in Figure 12 and observe that SCS-Peel and SCS-Expand are significantly faster than SCS-Baseline especially on large datasets. This is because when we choose relatively large α and β, the search space of SCS-Peel and SCS-Expand is much smaller than that of SCS-Baseline. For example, if we choose $\alpha = \delta$ and $\beta = \delta$, as shown in Table II, the size of $R_{\delta,\delta}$ is much smaller than the original graph size (i.e., $|E|$). Note that $C_{\delta,\delta}(q)$ is always a subgraph of $R_{\delta,\delta}$. We also add more discussions of the two-step framework in both Section I and Section V. ■

Comment 2.3 (D1): *In Algorithm 1, how is alpha_max obtained? Is it provided by users?*

Response: Thanks for the comment. In the revised paper, we add the following clarification in Section III.A: "*α_{max} is the maximal α value such that an $(\alpha, 1)$-core exists in G and it is equal to the maximal vertex degree in $U(G)$*". ■

Comment 2.4 (D3): *Figure 7 is not readable. Please simplify it.*

Response: Thanks for your suggestion. In the revision, we make Figure 7 more clear and show more experimental results in Section V.B to validate the effectiveness of our model. ■

Comment 2.5 (D4): *The authors explain with Fig. 13 that SC-Expand is more efficient than SC-Peel. Regarding Fig. 15(c) and (d), what are the reasons that SCS-Peel is the same or more efficient than SCS-Expand?*

Response: Thanks for pointing this out. Figure 14(c) and (d) (originally Figure 15(c) and (d)) report the performances of SCS-Peel and SCS-Expand on `DUI` dataset which are similar. This is because on `DUI`, the actual computation costs of SCS-Peel and SCS-Expand (which can be estimated by the number of processed edges as shown in Figure 14(d)) are about the same. However, on most of the datasets, SCS-Expand is on average more efficient than SCS-Peel in Figure 12 (originally Figure 13). This is because the efficiency of SCS-Peel and SCS-Expand depends on both the size of the (α, β)-community (which determines the search space) and the size of the significant (α, β)-community (which relates to the actual computation cost). Usually, SCS-Peel and SCS-Expand both need more time to handle the cases when the search space is large and SCS-Expand is faster than SCS-Peel in these cases. In the revised paper, we make this more clear and add more discussions about the comparison of two algorithms in Section V.D. ■

RESPONSE TO REVIEWER #6

Comment 3.1 (D1): *The paper assumes that all edges in the input graph have different edge weights and claims that this assumption comes at no loss to generality.*

Response: Thanks for the comments and we appreciate your carefulness. In the revised paper, we remove this assumption and update our algorithms to correctly handle the cases where the edges can have the same weight. Please refer to the highlighted parts in Algorithms 4 and Algorithm 5 for the details. Now in our algorithms, the edges with the same weight are processed in the same iteration. In addition, we preserve all the valid vertices which satisfy the degree constraints and the corresponding valid edges in the final result to guarantee the maximality constraint is still satisfied. ■

Comment 3.1 (D2): *Based on the aforementioned assumption, the paper assumes that when two edge weights are equal, the edge ids can be used to break ties.*

Response: Thanks for the comment. We remove the assumption and update our algorithms accordingly. Now our proposed algorithms can always produce the same result according to the definition of the significant (α, β)-community. ■

Comment 3.2 (D3): *The effectiveness evaluation is not based on a meaningful metric related to recommendation quality. One approach to quality evaluation might be comparing the community with one generated by a naive baseline, such as retrieving all comedy movies with at least 5-star ratings.*

Response: Thanks for the insightful suggestion and inspiration. We find that only 94 comedy movies with at least 5-star ratings exist in the dataset and many of them are rated only once or twice. To make the baseline more comparable, in the revised paper, we generate the community $C_{4\star}$ which is the induced subgraph of all comedy movies with at least 4-star ratings. In Figure 6, we show the bipartite graph density, average rating and percentage of dislike users of all the models including $C_{4\star}$. We can see that, the density of $C_{4\star}$ is lower than the significant (α, β)-community (SC). This is because the structure cohesiveness is not considered in $C_{4\star}$ and the users in $C_{4\star}$ are loosely connected with the query user q. In Table II, we also show the statistics of the results for a specific query. We can observe that, M_{avg} (i.e., the average number of movies a user watched in the community) of $C_{4\star}$ is only 2.39 which indicates that $C_{4\star}$ contains many users who only watched few high rating movies. In comparison, the difference between SC and $C_{4\star}$ is that SC aims to find a group of people who give common high ratings with q on a collection of movies. In addition, the movies in SC are highly rated by these users who are more likely to have common interests with q. On the other hand, $C_{4\star}$ can only find a group of people who watched these movies with high ratings and the movies with globally high ratings. Thus, the results of these two communities are not similar as evaluated and SC returns more relevant movies and users to q considering both weight and structure cohesiveness. ■

Comment 3.3 (D4): *The paper needs to discuss how the proposed index can be maintained when the graph changes.*

Response: Excellent comments. In the revised paper, we add discussions of the incremental index maintenance algorithms when the graph changes in Section III.B. Due to the space limitation, we introduce the key observations and the detailed techniques will be a valuable future research direction. We also put this as a future work in the conclusion. ■

Comment 3.4 (D5): *The intuition behind choosing to maximize the edge weight needs further explanation.*

Response: Thanks for the comment. In the revised paper, we add the following explanations in the introduction "*A community with a high weight value indicates that every edge in the community represents a highly significant interaction. ... 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.*" We also conduct more experiments in Section V.B to validate the effectiveness of the proposed model. ■

Comment 3.5 (D6): *The paper proposes the peeling and the expansion approach. It would be helpful to also provide a comparison between this two approaches.*

Response: Thanks for your suggestion. In the revision, the following discussion is added in Section V. "*The efficiency of these two algorithms largely depends on the size of the (α, β)-community containing q (i.e.,* $\mathsf{size}(C_{\alpha,\beta}(q))$*, which determines the search space) and the size of the final result (i.e.,* $\mathsf{size}(\mathcal{R})$*, 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.*" ■

Comment 3.6 (D7): *The description of the union-find connected components algorithm in Section IV-B is well-known and you can probably skip it.*

Response: Thanks for the comment. As suggested, we carefully update this part and we skip the description of the union-find structure in Section IV.B. ■

Comment 3.7 (D8): *Averaging might hide the tail latency, so I would suggest either plotting a CDF or reporting the standard deviation alongside Figure 13.*

Response: Thanks for your suggestion. In the revised paper, we report the standard deviation in Figure 12 (orginally Figure 13) and add related analysis. ■

Comment 3.8 (D9): *It needs to better position this work among existing community search approaches.*

Response: Thanks for the insightful comments. Our work is the first to find a cohesive subgraph with both structure cohesiveness and high weight (significance) on bipartite graphs. To better position our work in the literature, we add more discussions about the difference between our work and the existing works (especially these works on edge-weighted unipartite graphs and projection approaches) in Section VI. ■

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 the 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 much smaller size than the orginal 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 search 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

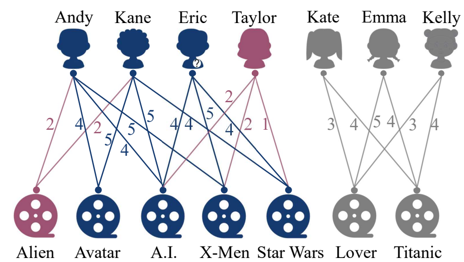


Fig. 1: A user-movie network

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).

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.

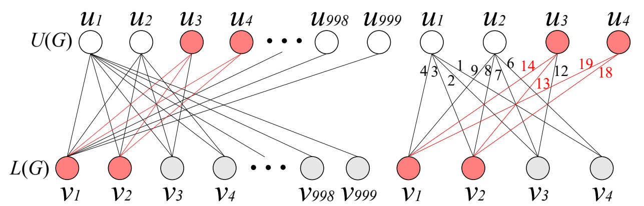
• *Personalized Recommendation.* In user-item networks, users leave reviews for items with ratings. Examples include viewer-movie network in `IMDB` (https://www.imdb.com), reader-book network in `goodreads` (https://www.goodreads.com), etc. The platforms can utilize the significant (α, β)-community model 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. 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 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 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.

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.
- 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 $\frac{|\mathsf{size}(C_{\alpha,\beta}(q))|}{\sum_{v \in V(C_{\alpha,\beta}(q))} deg(v,G)}$ 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.*

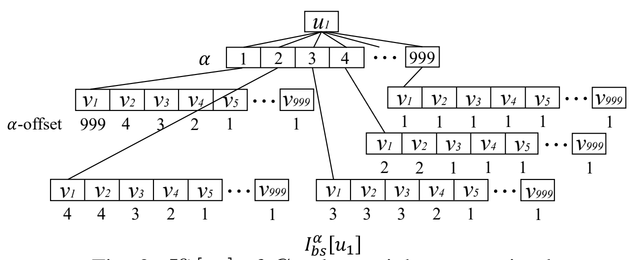
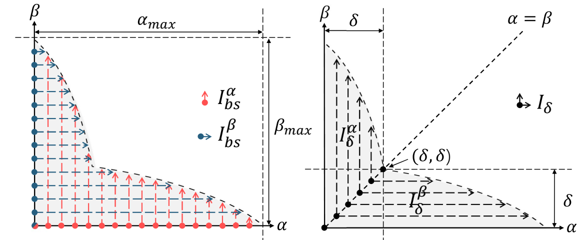


Fig. 3: $I^{\alpha}_{bs}[u_1]$ of G, edge weights are omitted

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

Input: G
Output: I^{α}_{bs}
$\alpha \leftarrow 1$;
$\alpha_{max} \leftarrow$ the maximal vertex degree in $U(G)$;
while $\alpha \leq \alpha_{max}$ **do**
  compute $s_a(u,\alpha)$ for each vertex $u \in V(G)$;
  foreach $u \in (\alpha,1)$-*core* **do**
    foreach $v \in N(u,G)$ **do**
      if $s_a(v,\alpha) \geq 1$ **then**
        $I^{\alpha}_{bs}[u][\alpha] \leftarrow \{v, w(u,v), s_a(v,\alpha)\}$;
    sort $I^{\alpha}_{bs}[u][\alpha]$ in decreasing order of their α-offsets;
  $\alpha \leftarrow \alpha + 1$;
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 it is equal to the maximal vertex degree in $U(G)$. We then



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

Fig. 4: Illustrating the ideas of indexing techniques

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 $\geq$ β. 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 $O(\mathsf{size}(C_{\alpha,\beta}(q)))$ 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. $\square$

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.

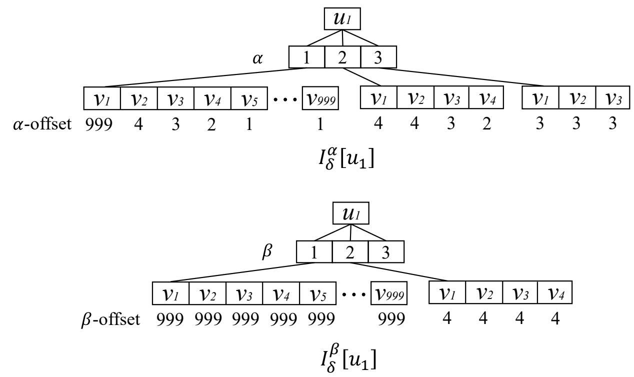


Fig. 5: $I_\delta[u_1]$ of G, edge weights are omitted

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
$\tau \leftarrow 1$;
compute δ using the k-core decomposition algorithm;
while $\tau \le \delta$ **do**
  compute α-offset $s_a(u, \tau)$ and β-offset $s_b(u, \tau)$ for each vertex $u \in V(G)$;
  foreach $u \in (\tau, \tau)$-*core* **do**
    foreach $v \in N(u, G)$ **do**
      if $s_a(v, \tau) \ge \tau$ **then**
        $I_\delta^\alpha[u][\tau] \leftarrow \{v, w(u, v), s_a(v, \tau)\}$;
      if $s_b(v, \tau) > \tau$ **then**
        $I_\delta^\beta[u][\tau] \leftarrow \{v, w(u, v), s_b(v, \tau)\}$;
    sort $I_\delta^\alpha[u][\tau]$ in decreasing order of their α-offsets;
    sort $I_\delta^\beta[u][\tau]$ in decreasing order of their β-offsets;
  $\tau \leftarrow \tau + 1$;
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 [21] since δ is equal to the maximum core number in G. Then, for each vertex u, we compute its α-offset for each $\alpha \le \delta$ and its β-offset for each $\beta \le \delta$. 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 $\alpha \le \delta$, we first add $u(v)$ into $I_\delta^\alpha[v][\alpha]$ ($I_\delta^\alpha[u][\alpha]$) if $s_a(u, \alpha) \ge \alpha$ ($s_a(v, \alpha) \ge \alpha$). Then, for each $\alpha \le \delta$, 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 $\beta \le \delta$, 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 $\alpha \le \delta$, we first remove $u(v)$ from $I_\delta^\alpha[v][\alpha]$ ($I_\delta^\alpha[u][\alpha]$) if $s_a(u, \alpha) \ge \alpha$ ($s_a(v, \alpha) \ge \alpha$). 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. In addition, we introduce the space-aware index in the technical report [22] to handle the scenarios when the space budget is given.

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: $\mathcal{R}$
get $C_{\alpha,\beta}(q)$ from the index;
$S \leftarrow \emptyset$; $Q \leftarrow \emptyset$;
sort edges of $C_{\alpha,\beta}(q)$ in non-decreasing order by weights;
while $C_{\alpha,\beta}(q)$ *is not empty* **do**
  $\mathsf{w_{min}} \leftarrow$ the minimal edge weight in $C_{\alpha,\beta}(q)$
  foreach $(u, v) \in C_{\alpha,\beta}(q)$ *with* $w(u, v) = \mathsf{w_{min}}$ **do**
    remove (u, v) from $C_{\alpha,\beta}(q)$;
    $S.add((u, v))$;
    if $deg(u, C_{\alpha,\beta}(q)) < \alpha \land u \notin Q$ **then**
      $Q.push(u)$;
    if $deg(v, C_{\alpha,\beta}(q)) < \beta \land v \notin Q$ **then**
      $Q.push(v)$;
  while Q *is not empty* **do**
    $u' \leftarrow Q.pop()$;
    foreach $v' \in N(u', C_{\alpha,\beta}(q))$ **do**
      remove (u', v') from $C_{\alpha,\beta}(q)$;
      $S.add((u', v'))$;
      if v' *does not have enough degree* **then**
        $Q.push(v')$;
        if v'=q **then**
          $G' \leftarrow S \cup C_{\alpha,\beta}(q)$;
          Obtain $\mathcal{R}$ from G'
          return $\mathcal{R}$;
  $S = \emptyset$;

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 (size($C_{\alpha,\beta}(q)$)) time. Then, sorting the edges in $C_{\alpha,\beta}(q)$ needs sort($C_{\alpha,\beta}(q)$) time which will be $O(\text{size}(C_{\alpha,\beta}(q))\cdot(log(\text{size}(C_{\alpha,\beta}(q)))))$ 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 $O(\text{size}(C_{\alpha,\beta}(q))$ time. In total, the time complexity of SCS-Peel is $O(\text{sort}(C_{\alpha,\beta}(q)) + \text{size}(C_{\alpha,\beta}(q)))$.
Space complexity. In the SCS-Peel algorithm, we need only $O(\text{size}(C_{\alpha,\beta}(q)))$ 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 size($\mathcal{R}$) is much smaller than 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 [23], 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 $V(C^*)\setminus V(\mathcal{R})$ 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 $O(\text{size}(C^*))$ 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, size(C_i^*) is exactly ϵ times of size(C_{i-1}^*). Since we can find $\mathcal{R}$ in the final check, we have size(C_d^*) $< \epsilon(\text{size}(\mathcal{R}))$. The time complexity of using the peeling algorithm to check all these connected subgraphs is $O(\sigma_{i=1}^{d}\text{size}(C_i^*))$, and $\sigma_{i=1}^{d}\text{size}(C_i^*)$ = size(C^d) + $\frac{1}{\epsilon}$size(C^d) + $\frac{1}{\epsilon^2}$size(C^d) + ... + $\frac{1}{\epsilon^d}$size(C^d), we can have $O(\sigma_{i=1}^{d}\text{size}(C_i^*)) = O(\epsilon(\frac{1}{\epsilon-1})\text{size}(\mathcal{R}))$. 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 first 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: $\mathcal{R}$
1 $G^* \leftarrow \emptyset$; $C^* \leftarrow \emptyset$; pre_size $= 0$;
2 get $C_{\alpha,\beta}(q)$ from the index;
3 sort edges of $C_{\alpha,\beta}(q)$ in non-increasing order by weights;
4 **while** *$C_{\alpha,\beta}(q)$ is not empty* **do**
5 $\quad$ $\mathsf{w_{max}} \leftarrow$ the maximal edge weight in $C_{\alpha,\beta}(q)$
6 $\quad$ **foreach** *$(u, v) \in C_{\alpha,\beta}(q)$ with $w(u, v) = \mathsf{w_{max}}$* **do**
7 $\quad\quad$ remove (u, v) from $C_{\alpha,\beta}(q)$;
8 $\quad\quad$ *$G^*.add((u, v))$*;
9 $\quad\quad$ maintain the connected subgraphs in G^*;
10 $\quad$ **if** *C^* is not changed or violates constraints in Lemma 7 and Lemma 8* **then**
11 $\quad\quad$ **continue**;
12 $\quad$ **if** size$(C^*) \geq$ pre_size $\cdot\, \epsilon$ **then**
13 $\quad\quad$ pre_size $\leftarrow$ size(C^*);
14 $\quad$ **else**
15 $\quad\quad$ **continue**;
16 $\quad$ Remove the vertices without enough degree from C^*;
17 $\quad$ **if** *$q \in C^*$* **then**
18 $\quad\quad$ run Algorithm 4 lines 3 - 23, replace $C_{\alpha,\beta}(q)$ 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. $\square$

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. Note that the weights can be negative or 0 in our algorithms. We only require that any two edges are comparable given their weights regardless of whether they are positive or not. In addition, 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, 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.

TABLE I: Summary of Datasets

Dataset	$\lvert E\rvert$	$\lvert U\rvert$	$\lvert L\rvert$	δ	α_{max}	β_{max}	$\lvert R_{\delta,\delta}\rvert$
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 $C_{\alpha,\beta}(q)$.

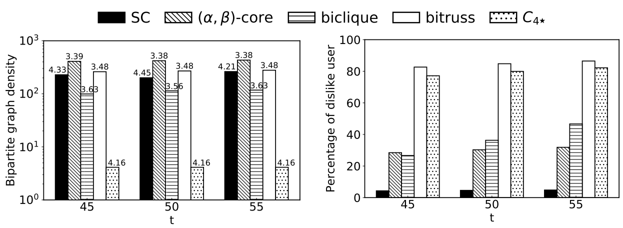
The algorithms are implemented in C++ and the experiments are run on a Linux server with Intel Xeon 2650 v3 2.3GHz processor and 768GB main memory. *We terminate an algorithm if the running time is more than 10^4 seconds.*
Datasets. We use 11 real datasets in our experiments which are `Bookcrossing` (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 use the random walk with restart model [24] to compute the node relevance and generate the weights.

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 $d(G) = |E(G)|/\sqrt{|U(G)||L(G)|}$ [25]. 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.

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

Models	$\lvert U\rvert$	$\lvert M\rvert$	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 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.

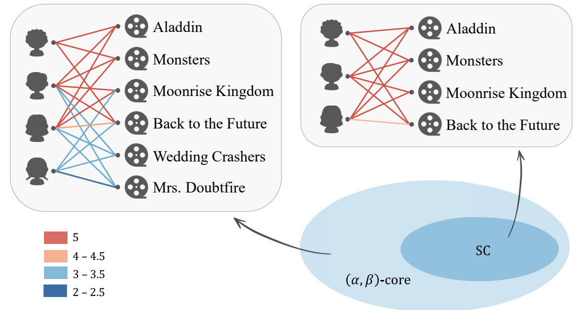


Fig. 7: Representative components of real-life communities

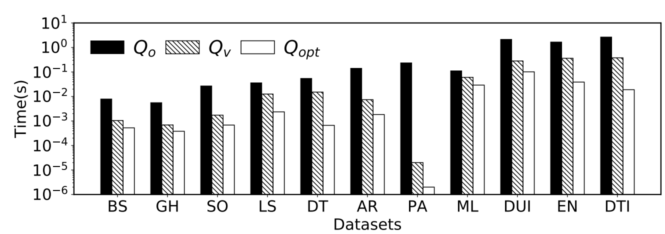
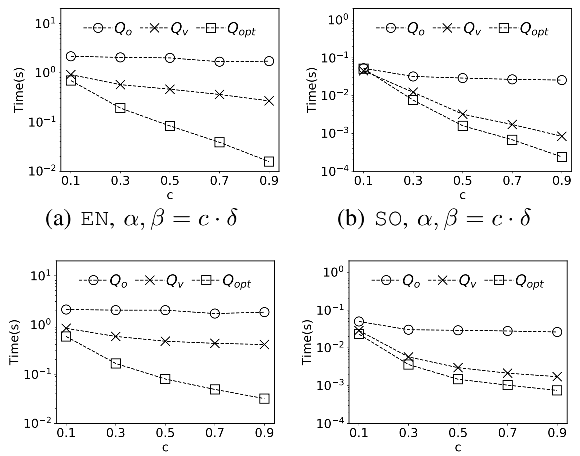


Fig. 8: Retrieving the (α, β)-communities

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

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$\times$ 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



(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

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.

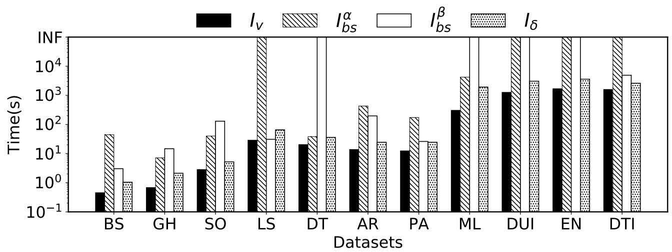


Fig. 10: Index construction time

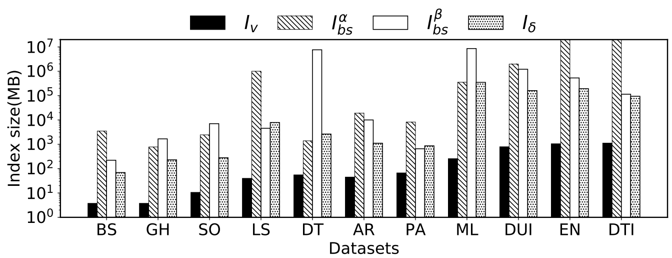


Fig. 11: Index size

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.

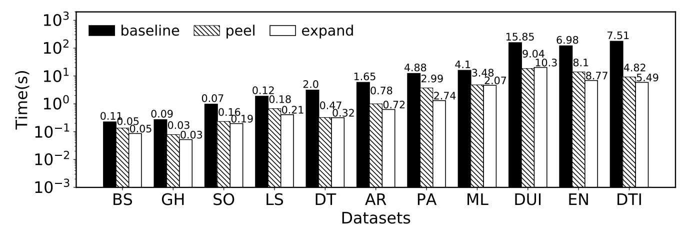
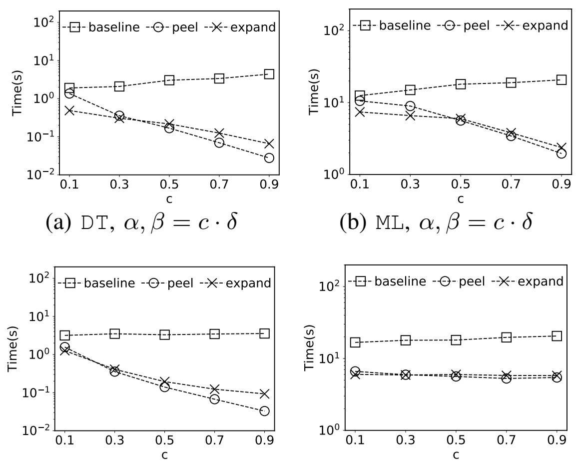


Fig. 12: Query performance on different datasets

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 in these cases.

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 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,

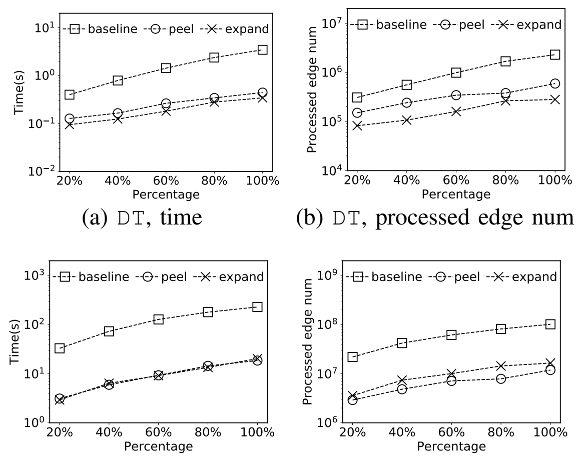


(a) DT, $\alpha, \beta = c \cdot \delta$ (b) ML, $\alpha, \beta = c \cdot \delta$

(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

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.



(a) DT, time (b) DT, processed edge num

(c) DUI, time (d) DUI, processed edge num

Fig. 14: Scalability of query algorithms

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 [24]; (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. In addition, SCS-Peel and SCS-Expand are more efficient than SCS-Baseline under non-trivial distributions excluding AE as expected.

Scalability of query algorithms. In Figure 14, we evaluate the scalability of SCS-Baseline, SCS-Peel, and SCS-Expand by varying the graph size m. When varying m, we randomly sample 20% to 100% edges of the original graphs. We can observe that the algorithms are scalable. Note that on DUI, the running time of SCS-Peel and SCS-Expand is similar. This is because the actual computation cost of SCS-Peel and SCS-Expand (which can be estimated by the number of processed edges) is about the same. As expected, SCS-Peel and SCS-Expand are more efficient than SCS-Baseline.

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], [26], [27], k-truss [8]–[10], [28], [29], k-clique [30]. 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 [26], [27]. For the truss-based community search, [8], [28] 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 [30], 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 [31], [32]. 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 [29], 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 [33] 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] extend the k-core model on unipartite graph to the (α, β)-core model. [17]–[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]–[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 [34] 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 we find a community with structure cohesiveness and high significance, which 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. In the future, we will investigate the detailed index maintenance 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] 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.

[22] Tech. Rep., https://www.dropbox.com/sh/jj3t9q4c5bt95rw/AADsAkHSc3OXVdjMV7nBK7Jfa?dl=0.

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

[24] 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.

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

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

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