Zero-Redundancy Search for Bi-Components in Bipartite Graphs

CHUANYU ZONG, Shenyang Aerospace University, China
LINGBIAO MENG, Shenyang Aerospace University, China
XIAOCHUN YANG, Northeastern University, China
BIN WANG*, Northeastern University, China
TAO QIU*, Shenyang Aerospace University, China
RUI ZHU, Shenyang Aerospace University, China

Efficiently managing and querying large bipartite graphs necessitates powerful indexing structures. A fundamental challenge lies in supporting (α, β)-component (bi-component) search, a task that encompasses retrieving all bi-components (bi-core search) or identifying the specific bi-component containing a query vertex (bi-community search). Existing index-based solutions face a time-space trade-off: single-dimensional approaches incur significant storage redundancy, while holistic approaches suffer from repeated vertex retrievals, hindering their scalability. To overcome these limitations, in this paper, we address this trade-off by introducing a novel, finer-grained cohesive unit, the (α, β, γ)-*cluster* (*bi-cluster*). This unit inherently captures the *nested* and *overlapping* relationships among bi-components, enabling vertex deduplication at the index level. Building on this, we propose the *SGL* (*Summary Graph + Location*) index, which achieves *zero-redundancy* in vertex retrieval during query processing. It organizes bi-components into a compact Summary Graph (SG) of interconnected SNodes. This structure, coupled with a precise Location mapping, allows our search algorithm to retrieve results without redundant vertex accesses, effectively decoupling query cost from the graph's global scale. Furthermore, we provide efficient construction and maintenance algorithms for dynamic bipartite graphs. Extensive experiments on ten real and synthetic large bipartite graphs demonstrate that our method outperforms the state-of-the-art by up to two orders of magnitude in query speed for bi-component search, while maintaining low storage, thus bridging the gap between fast but space-inefficient and compact but slower existing approaches.

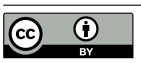
CCS Concepts: • **Theory of computation → Graph algorithms analysis**.

Additional Key Words and Phrases: Bipartite graph, (α, β)-component search, Bi-cluster, γ-group

ACM Reference Format:
Chuanyu Zong, Lingbiao Meng, Xiaochun Yang, Bin Wang, Tao Qiu, and Rui Zhu. 2026. Zero-Redundancy Search for Bi-Components in Bipartite Graphs. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 252 (June 2026), 26 pages. https://doi.org/10.1145/3802129

*Corresponding author

Authors' Contact Information: Chuanyu Zong, Shenyang Aerospace University, Shenyang, Liaoning, China, zongcy@sau.edu.cn; Lingbiao Meng, Shenyang Aerospace University, Shenyang, Liaoning, China, menglingbiao@stu.sau.edu.cn; Xiaochun Yang, Northeastern University, Shenyang, Liaoning, China, yangxc@mail.neu.edu.cn; Bin Wang, Northeastern University, Shenyang, Liaoning, China, binwang@mail.neu.edu.cn; Tao Qiu, Shenyang Aerospace University, Shenyang, Liaoning, China, qiutao@sau.edu.cn; Rui Zhu, Shenyang Aerospace University, Shenyang, Liaoning, China, zhurui@sau.edu.cn.



ACM 2836-6573/2026/6-ART252
https://doi.org/10.1145/3802129

1 Introduction

Bipartite graphs, consisting of two disjoint sets of vertices with edges only connecting vertices from different sets [5, 16, 49], are ubiquitous in modeling real-world relationships. Cohesive subgraph search in bipartite graphs aims to identify subgraphs that satisfy predefined cohesiveness constraints, ensuring that vertices within the subgraph are tightly connected. To this end, a variety of cohesive subgraph models have been proposed, such as the (α, β)-core [6, 9, 17, 20, 24, 35, 41, 42], k-bitruss [29, 40, 52], biclique [49], and butterfly [39]. Given a bipartite graph G, we refer to its two disjoint vertex sets as the upper vertices and the lower vertices. Among existing models, the (α, β)-core (bi-core) [9] is defined as the maximal subgraph in which each upper vertex has a degree of at least α and each lower vertex has a degree of at least β. The maximal connected components within an (α, β)-core are referred to as (α, β)-components (bi-components) [41].

Applications. Bi-components provide a robust structural foundation for various real-world scenarios. In online group recommendation, they function as fault-tolerant subspace clusters [9, 20]. For example, on billion-scale e-commerce platforms like Amazon and Alibaba [18, 37], recommendation engines must handle millions of simultaneous group requests during peak traffic [13, 46]. Meeting these demands requires searching a vast number of bi-components within strict real-time constraints (typically within 0.5 seconds [18]). In social networks such as Facebook and Twitter, users and pages form a bipartite graph where edges represent likes. To promote certain pages, fraudsters use many fake accounts, causing a large number (β) of users to like a small number (α) of pages. Bi-components with small α and large β can help detect such fraudsters [19, 25]. In biological networks, bi-components capture functionally coherent modules of tightly connected genes or proteins [31]. Furthermore, in bipartite networks, bi-components serve as a key step for solving other graph problems, such as biclique search [45] and dense subgraph search [48].

Inspired by these practical requirements, two fundamental problems have been defined: (1) *bi-core search* [23], which identifies all (α, β)-components in a bipartite graph, and (2) *bi-community search* [41], which locates the specific (α, β)-component containing a query vertex q. Traditional solutions predominantly rely on peeling or local search algorithms, while these methods achieve a worst-case time complexity of $O(n + m)$, where n is the number of vertices and m is the number of edges in the bipartite graph. They prove prohibitively expensive for real-time queries on large graphs where m reaches billions. Consequently, researchers have pivoted toward index-based solutions, which can be categorized into *single-dimensional* [20, 41] and *holistic* [23, 24] approaches.

Single-dimensional approaches exploit the nested relationships along a single parameter dimension to organize the index structure. By enumerating all possible values of one parameter (α or β) and constructing an independent index structure for the other parameter under each fixed value, the overall index consists of multiple independent sub-indexes. During query processing, only the sub-indexes corresponding to the query parameters need to be accessed to retrieve the target (α, β)-component, significantly improving query efficiency. Holistic approaches, in contrast, leverage the nested relationships of (α, β)-cores across both the α and β dimensions to construct a unified index, thereby reducing storage redundancy in the constructed index. During query processing, the target (α, β)-component is reconstructed by combining vertices from other bi-components in the index that are nested within the target component.

Motivations. However, by analyzing the aforementioned indexes, we make the following observations. Existing index-based methods face a common and fundamental challenge: redundant vertices lead to storage or query overhead. In single-dimensional approaches, a vertex often belongs to multiple (α, β)-cores or (α, β)-components under different parameter settings. This necessitates it to be stored repeatedly in multiple sub-indexes, resulting in significant storage cost. In holistic

approaches, although storage redundancy is reduced, vertices may still overlap across (α, β)-components, causing repeated vertex retrieval during query processing. When the size of the query result is large, this repeated vertex traversal can lead to significant query overhead. Motivated by these observations, we propose a novel indexing paradigm that simultaneously eliminates both storage and retrieval redundancies.

Our Idea. The persistent vertex redundancy in previous approaches is not accidental; rather, it indicates that current bipartite graph decomposition techniques have not yet reached an atomic level. To resolve this, we explore a novel idea: given a bipartite graph, we first compute all (α, β)-components and further partition each bi-component into atomic units that are neither nested nor overlapping. Each atomic unit captures the intrinsic nested and overlapping relationships among bi-components, providing a finer-grained foundation for constructing a novel index. During (α, β)-component search, the query only needs to access these atomic units to retrieve the target set of vertices, ensuring that each vertex is accessed exactly once, achieving zero-redundancy search for bi-components while maintaining the index compact and efficient. To make this idea practically applicable, the following challenges need to be addressed: (1) *how to formally define and extract these fine-grained atomic structures*; (2) *how to organize these atomic structures into an efficient index to support bi-component searches*; and (3) *how to develop efficient algorithms for index construction and maintenance.*

Contributions. We address the above challenges and make the following contributions:

• **Theoretical foundation.** We perform a systematic structural analysis of nested and overlapping bi-cores. By introducing the structural predecessor as a new dimension (denoted as γ), we propose the concept of (α, β, γ)-cluster (details in Section 3), which provides the theoretical foundation for partitioning bi-components into disjoint, fine-grained atomic units.

• **Novel index.** Based on the bi-component division via (α, β, γ)-clusters, we propose a novel index, *SGL* (*Summary Graph + Location*), and design an efficient algorithm for bi-component search. This architecture ensures that each vertex is accessed exactly once during query processing, achieving zero-redundancy search for bi-components in large bipartite graphs.

• **Effective algorithms.** We propose effective algorithms to construct and dynamically maintain the SGL index for large bipartite graphs. The construction algorithm maximizes information reuse and achieves near-linear time complexity in the worst case, making it practical for massive datasets. We also provide a dominance-based pruning strategy and a threshold-based grouping mechanism to optimize index size and query efficiency.

• **Extensive experiments.** We conduct extensive experiments on ten real-world and synthetic large bipartite graphs. The results demonstrate that our proposed method significantly outperforms state-of-the-art algorithms in both effectiveness and efficiency, achieving a **speedup of two orders of magnitude** in both bi-community search and bi-core search in large bipartite graphs.

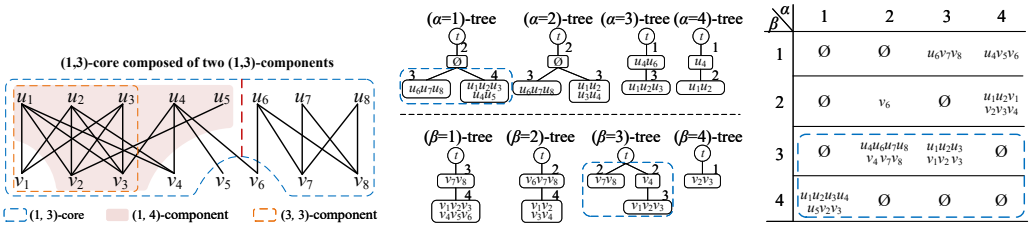
Due to space constraints, *we detail our index for bi-community search, while its broader applicability and effectiveness for bi-core search are also validated experimentally.*

2 Preliminaries

In this section, we first introduce basic concepts and formalize the problem definition. We then analyze the limitation of existing approaches and provide an overview of our proposed framework.

2.1 Problem Statement

We consider a bipartite graph $G = (U, V, E)$, where U and V represent the vertex sets of the upper and lower layers, respectively, and $U \cap V = \emptyset$. The set E denotes the edge set of G, where specifically, $E \subseteq U \times V$ consists of edges between vertices in U and vertices in V. Let $N(u, G)$ denote the set of



β \ α	1	2	3	4
1	Ø	Ø	$u_6v_7v_8$	$u_4v_5v_6$
2	Ø	v_6	Ø	$u_1u_2v_1$ $v_2v_3v_4$
3	Ø	$u_4u_6u_7u_8$ $v_4v_7v_8$	$u_1u_2u_3$ $v_1v_2v_3$	Ø
4	$u_1u_2u_3u_4$ $u_5v_2v_3$	Ø	Ø	Ø

(a) An example of a bipartite graph (b) The α-trees and β-trees of Figure 1(a) (c) (α, β)-cores with redundancies eliminated based on nested relationships

Fig. 1. Examples of existing indexes

neighbors of a vertex u in G. We abbreviate this as $N(u)$ when the context is clear. The degree of a vertex u is denoted as $deg_G(u) = |N(u, G)|$.

Definition 1. **(α, β)-Core (Bi-Core)** [9]. *The (α, β)-core of a bipartite graph $G = (U, V, E)$ is defined as the maximal subgraph C of G with vertex sets $U' \subseteq U$ and $V' \subseteq V$, where $\forall u \in U', deg_C(u) \geq \alpha$ and $\forall v \in V', deg_C(v) \geq \beta$.*

Similar to the classic k-core [12], an (α, β)-core is not necessarily connected. By enforcing a connectivity constraint, an (α, β)-core can be partitioned into several (α, β)-components.

Definition 2. **(α, β)-Component (Bi-Component)** [41] . *Given a subgraph H of an (α, β)-core, H is called an (α, β)-component if it is connected and no other connected subgraph of the (α, β)-core contains it as a subgraph.*

Example 2.1. *As shown in Figure 1(a), the subgraph enclosed by the blue dashed line forms a $(1, 3)$-core, which consists of two distinct $(1, 3)$-components separated by the red dashed line.*

In bipartite graphs, bi-cores and bi-components exhibit both nesting and overlapping properties.

Property 2.1. *Given an (α, β)-core C_i and an (α^+, β^+)-core C_j, if $\alpha^+ \geq \alpha$ and $\beta^+ \geq \beta$ (with at least one inequality being strict), then $C_j \subseteq C_i$ (C_j is **nested** within C_i). Similarly, for two bi-components H_i and H_j with corresponding parameters, if H_j is connected to H_i and their parameters satisfy the same condition, then H_j is nested within H_i (i.e., $H_j \subseteq H_i$).*

Property 2.2. *Two bi-cores C_i and C_j are **overlapping** if they are non-nested and $C_i \cap C_j \neq \emptyset$. Similarly, two bi-components H_i and H_j are **overlapping** if they are non-nested and $H_i \cap H_j \neq \emptyset$.*

Example 2.2. *Returning to Figure 1(a), the pink module represents an $(1, 4)$-component, which is nested within the left $(1, 3)$-component. The subgraph enclosed by the orange dashed line denotes an $(3, 3)$-component, which is also nested within the $(1, 3)$-component but non-nested with the $(1, 4)$-component. Since $(3, 3)$-component $\cap$ $(1, 4)$-component $= \{u_1, u_2, u_3, v_2, v_3\} \neq \emptyset$, they are overlapping.*

Definition 3. **(α, β)-Community (Bi-Community)** [43] . *Given a query vertex q, the (α, β)-component containing q is defined as the (α, β)-community.*

Based on the above definitions, we define the bi-community search problem as follows.

Problem 1. ***Bi-Community Search***. *Given a bipartite graph $G = (U, V, E)$, two parameters α and β, and a query vertex q, the problem of bi-community search is to identify the (α, β)-community containing q in G.*

Table 1. The number of vertices accessed during storage or search for different indexes. Here, n represents the number of vertices in G; $|H|$ denotes the number of vertices in the target bi-component H (where H belongs to the bi-core C); $E(H)$ denotes the set of edges in H; μ denotes the average number of times each vertex is stored in indexes; and $\bar{d}$ denotes the average degree of vertices ($\bar{d} \geq \mu \geq 1$)

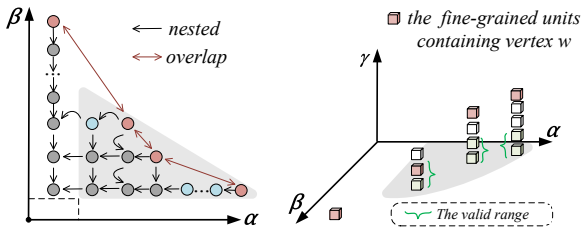
Index	*BI*	*BN*	*MCR*	*BH*	*Our*
Storage	$O(\bar{d}n)$	$O(\mu n)$	$O(\mu n)$	$O(\bar{d}n)$	$O(\mu n)$
Search	$O(\lvert C\rvert+\lvert E(H)\rvert)$	$O(\lvert H\rvert \log \mu+\lvert E(H)\rvert)$	$O(\mu\lvert H\rvert)$	$O(\mu\lvert H\rvert)$	$O(\lvert H\rvert)$

2.2 Limitations of Existing Approaches

For the bi-community search problem, index-based approaches [20, 23, 24, 41] can be divided into the following two categories:

2.2.1 Single-Dimensional Approaches. Liu et al. [20] proposed the Bipartite Index (*BI*), which first computes all the non-empty (α, β)-cores. For each α, *BI* builds a one-dimensional table storing the upper vertices across all β values, and for each β, another table storing the lower vertices across all α values. Wang et al. [41] introduced the Bipartite Hierarchy (*BH*) index, which constructs a tree for each α to store the upper vertices across all β values, and a tree for each β to store the lower vertices across all α values. In each tree, nodes store vertices, and the subtree rooted at a node represents a specific (α, β)-component. In Figure 1(a), the α and β values of the bi-components range from 1 to 4. Fixing $\alpha = 1$, the $(1, \beta)$-component containing vertices u_1–u_5 attains a maximum β of 4, while u_6–u_8 belong to a $(1, 3)$-component. Notably, the $(1, 3)$-component (containing u_6–u_8) and the $(1, 4)$-component (containing u_1–u_5) are disconnected from each other, yet both are part of the same $(1, 2)$-component. As shown in Figure 1(b), the $\alpha = 1$ tree stores the upper vertices of the $(1, 3)$-component. Similarly, fixing $\beta = 3$, the $\beta = 3$ tree stores the lower vertices of the same $(1, 3)$-component. However, *BI* fails to account for the connectivity of (α, β)-components, necessitating additional online computations during searches. Both *BI* and *BH* require storing $O(\bar{d}n)$ vertices, where $\bar{d}$ denotes the average degree of vertices in the bipartite graph. In Figure 1(b), vertices u_1, u_2, and u_4 are stored in all α-trees, while v_2 and v_4 are stored in all β-trees.

2.2.2 Holistic-based Approaches. Holistic approaches construct indexes by leveraging the nesting property of (α, β)-cores across both the α and β dimensions to reduce space redundancy. Luo et al. [24] proposed the Bi-core Number (*BN*) index, which computes all (α, β)-cores and eliminates redundancy through nested relationships between them. Figure 1(c) illustrates the (α, β)-cores from Figure 1(a) after redundancy elimination. For each remaining non-empty (α, β)-core, its parameter pair (α, β) is assigned as the bi-core number of its vertices, supporting bi-community search. Building upon this, Luo et al. [23] proposed the MCR-tree (*MCR*), which organizes the non-redundant (α, β)-cores as leaf nodes of an R-tree. Vertices within leaf nodes are bucketed according to connectivity. During query processing, the R-tree and connectivity buckets are combined to retrieve bi-components. However, *BN* fails to account for connectivity, requiring additional online computations. *MCR* can reduce space cost, but vertex redundancy during queries limits its performance. For example, in Figure 1(c), when searching for the $(1, 3)$-core, the vertex set enclosed by the blue dashed box contains vertices u_1, u_2, u_3, u_4, v_2, and v_3, and several of these vertices are encountered multiple times due to duplication across stored cores. Let μ denote the average number of times a vertex is stored. Then, *MCR* incurs a vertex traversal cost of $O(\mu|H|)$, whereas *BH* requires only $O(|H|)$, where H is the vertex set of the target bi-component. In real bipartite graphs, μ is non-negligible; for example, $\mu = 37$ in OG (see Table 3 for more data). Thus, *MCR* speeds up only for very small H and is usually slower than *BH*.



(a) Idea of existing methods (b) Core insights of our method

Fig. 2. Core insights

Discussion. In summary, existing methods suffer from vertex redundancy during storage or search, which increases space and query costs. Table 1 summarizes the number of vertices involved in storage or accessed during search for different indexes. Since *BI* and *BN* rely on online search, their edge traversal costs must also be considered. Note that while this table only reflects vertex storage and traversal, it does not directly represent the overall space and time complexity due to the unique structural characteristics of each index. The actual space and time costs are detailed in the experimental results (Figures 9 and 10 in Section 7).

2.3 Our Solution

Building on the discussion of existing methods in Section 2.2, we illustrate the core idea of our approach in Figure 2. Existing methods typically organize bi-components using either one-dimensional or two-dimensional nested relationships. Figure 2(a) depicts all bi-components containing vertex w. Using one-dimensional nesting (e.g., the vertical dimension), w must be stored redundantly in multiple bi-components (e.g., both red and blue nodes). Two-dimensional nesting reduces storage overhead by retaining w only in the red node, but vertices may still be accessed multiple times within the query region (gray area). This residual redundancy stems from overlapping non-nested bi-cores—a key limitation of prior approaches *BN* and *MCR*, which eliminates storage redundancy but not retrieval redundancy.

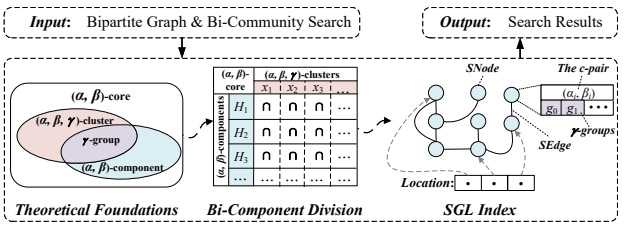


Fig. 3. Overview of the solution framework

To address this, we introduce a new dimension γ to partition each vertex's presence across cores into fine-grained units, called (α, β, γ)-*clusters* (Section 3). In Figure 2(b), red squares represent units containing vertex w, exhibiting monotonicity along the γ dimension. During queries, γ-based constraints prune all redundant vertices. Unlike existing methods that only capture dominance relationships, our fine-grained partitioning captures both nested and overlapping relationships at the minimal unit level while preserving hierarchical structure, achieving low storage redundancy and zero query redundancy. Furthermore, we extend this concept to bi-components by decomposing them into atomic units, called γ-*groups* (Section 4.1). Based on these atomic units, we design a novel index structure *SGL* (*Summary Graph + Location*) to support fast queries and optimized storage (Section 4.2). Figure 3 presents an overview of the proposed solution framework.

3 Theoretical Foundations of (α, β, γ)-Cluster

In this section, we first analyze the nesting and overlapping properties of bi-cores, and then introduce the (α, β, γ)-cluster.

For convenience, given an (α, β)-core C, we denote its associated parameter pair (α, β) as the c-pair of C, referred to as $Cpair(C) = (\alpha, \beta)$. If (α^+, β^+) satisfies $\alpha^+ \geq \alpha$ and $\beta^+ \geq \beta$ with at least one inequality being strict, we say that (α^+, β^+) dominates (α, β), denoted by $(\alpha^+, \beta^+) \succ (\alpha, \beta)$ [24]. Moreover, if either α or β in $Cpair(C)$ equals 0, this implies that some (α, β)-components in C are isolated vertices (i.e., vertices with no neighbors), which are not meaningful in practice. Therefore, in this paper, we only consider (α, β)-cores with $\alpha > 0$ and $\beta > 0$.

Property 3.1. *Given two bi-cores C_i and C_j, if $C_i \subseteq C_j$ and there exists no bi-core C_k such that $Cpair(C_i) \succ Cpair(C_k) \succ Cpair(C_j)$, then C_i and C_j are direct-nested.*

The direct-nested relationship in Property 3.1 enables the elimination of duplicate vertices in (α, β)-cores. Given an (α, β)-core, there are at most four different bi-cores that are direct-nested with it: the $(\alpha + 1, \beta)$-core, the $(\alpha, \beta + 1)$-core, the $(\alpha - 1, \beta)$-core, and the $(\alpha, \beta - 1)$-core. Thus, it suffices for each (α, β)-core to remove vertices that also appear in the $(\alpha{+}1, \beta)$- and $(\alpha, \beta{+}1)$-cores.

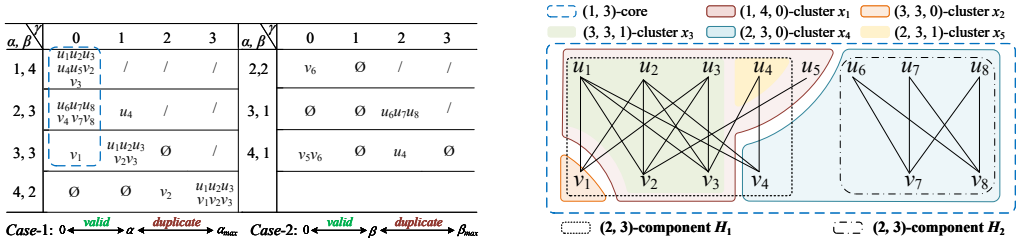
After removing duplicate vertices from (α, β)-cores based on nested relationships, a vertex may still belong to multiple mutually non-nested bi-cores. For example, in Figure 1(c), vertex u_1 belongs to three mutually non-nested bi-cores: the $(1, 4)$-core, $(3, 3)$-core, and $(4, 2)$-core. The corresponding α and β values are not consecutive, and because these bi-cores share the same vertex, they exhibit overlapping relationships as stated in Property 2.2. Such overlapping relationships are inherently complex, as each bi-core overlaps with multiple others. These overlaps neither follow a fixed structural pattern nor can they be directly expressed through parameter relations. To better describe this characteristic, we introduce the concept of the *predecessor value*, which quantifies the overlapping relationships among different bi-cores from the perspective of individual vertices.

Definition 4. **Predecessor Value.** *Given a vertex w in a graph G, let $(\alpha_1, \beta_1), (\alpha_2, \beta_2), \ldots, (\alpha_k, \beta_k)$ be the c-pairs of all (α, β)-cores in G that contain the vertex w and are mutually non-nested, sorted by ascending α and descending β. The predecessor value of w with respect to α_i is defined as follows: for the first c-pair (α_1, β_1), $P(w, \alpha_1) = 0$, and for $2 \leq i \leq k$, $P(w, \alpha_i) = \alpha_{i-1}$.*

Table 2. Predecessor values of all vertices in Figure 1(a)

Vertex	c-pairs Ordered	Predecessor Values
u_1, u_2, v_2, v_3	(1,4) (3,3) (4,2)	$P(\cdot, 1) = 0$ $P(\cdot, 3) = 1$ $P(\cdot, 4) = 3$
u_3	(1,4) (3,3)	$P(\cdot, 1) = 0$ $P(\cdot, 3) = 1$
u_4	(1,4) (2,3) (4,1)	$P(\cdot, 1) = 0$ $P(\cdot, 2) = 1$ $P(\cdot, 4) = 2$
u_5	(1,4)	$P(\cdot, 1) = 0$
u_6, v_7, v_8	(2,3) (3,1)	$P(\cdot, 2) = 0$ $P(\cdot, 3) = 2$
u_7, u_8	(2,3)	$P(\cdot, 2) = 0$
v_1	(3,3) (4,2)	$P(\cdot, 3) = 0$ $P(\cdot, 4) = 3$
v_4	(2,3) (4,2)	$P(\cdot, 2) = 0$ $P(\cdot, 4) = 2$
v_5	(4,1)	$P(\cdot, 4) = 0$
v_6	(2,2) (4,1)	$P(\cdot, 2) = 0$ $P(\cdot, 4) = 2$

Example 3.1. *As shown in Table 2, u_1 belongs to three mutually non-nested bi-cores, whose c-pairs are sorted in ascending order of α as $(1, 4)$, $(3, 3)$, and $(4, 2)$. Therefore, the corresponding predecessor values of u_1 are $P(u_1, 1) = 0$, $P(u_1, 3) = 1$, and $P(u_1, 4) = 3$.*



(a) An example for bi-cluster and Theorem 1 (b) The bi-clusters of (1,3)-core

Fig. 4. Examples of bi-clusters

It is evident that the predecessor value of a vertex is monotonic, providing a new dimension to quantify and distinguish bi-cores.

Definition 5. **(α, β, γ)-Cluster (Bi-Cluster).** *Given a subgraph x of G, x is an (α, β, γ)-cluster if it satisfies the following conditions: (1) $x \subseteq (\alpha, \beta)$-core $\wedge x \nsubseteq (\alpha^+, \beta^+)$-core, where $(\alpha^+, \beta^+) \succ (\alpha, \beta)$; (2) $\forall w \in x, P(w, \alpha) = \gamma$; (3) No other subgraph x' satisfies conditions (1) and (2), and contains x (i.e., $x \subset x'$).*

Example 3.2. *The sorted c-pairs of u_1 are $(1, 4)$, $(3, 3)$, and $(4, 2)$, with the corresponding predecessor values $P(u_1, 1) = 0$, $P(u_1, 3) = 1$, and $P(u_1, 4) = 3$. Therefore, u_1 belongs to the $(1, 4, 0)$-cluster, $(3, 3, 1)$-cluster, and $(4, 2, 3)$-cluster. Figure 4(a) illustrates all the bi-clusters in Figure 1(a) and the vertices contained in each of them.*

Lemma 1. *Given an $(\alpha_i, \beta_i, \gamma_i)$-cluster $x_i \subset (\alpha, \beta)$-core with $\gamma_i \geq \alpha$, for any vertex $w \in x_i$, there exists at least one $(\alpha_j, \beta_j, \gamma_j)$-cluster $x_j \subset (\alpha, \beta)$-core with $\gamma_j < \alpha$ such that $w \in x_j$.*

Proof. Let $w \in x_i$ and suppose that $P(w, \alpha_i) = \gamma_i \geq \alpha$. The predecessor values $P(w, \cdot)$ are defined over the c-pairs of (α, β)-cores containing w, ordered by ascending α and descending β. By Definition 4, $P(w, \alpha_1) = 0 < \alpha$, and $P(w, \alpha_i)$ increases monotonically with α_i. Hence, there must exist some α_j such that $\alpha_i > \alpha_j \geq \alpha$ and $P(w, \alpha_j) < \alpha$; otherwise, $P(w, \alpha_1) \geq \alpha$ would hold, leading to a contradiction. Since the c-pairs are sorted by ascending α and descending β, it follows that $\beta_j > \beta_i$, and thus $(\alpha_j, \beta_j) \succ (\alpha, \beta)$. Let $\gamma_j = P(w, \alpha_j) < \alpha$. Therefore, there exists an $(\alpha_j, \beta_j, \gamma_j)$-cluster $x_j \subset (\alpha, \beta)$-core such that $w \in x_j$. □

Theorem 1. *Let C be an (α, β)-core, and let $\{x_1, x_2, \ldots, x_k\}$ denote the set of all $(\alpha^+, \beta^+, \gamma)$-clusters in G such that $(\alpha^+, \beta^+) \succeq (\alpha, \beta)$ and $\gamma < \alpha$. Then, (1) $C = \bigcup_{i=1}^{k} x_i$ and (2) $|C| = \sum_{i=1}^{k} |x_i|$, where $|C|$ and $|x_i|$ represent the number of vertices in C and x_i, respectively.*

Proof. (1) Let $X_<$ and $X_\geq$ denote the sets of bi-clusters satisfying $(\alpha^+, \beta^+) \succeq (\alpha, \beta)$ with $\gamma < \alpha$ and $\gamma \geq \alpha$, respectively. Since bi-clusters are distinguished by different γ values within a given (α, β)-core, $X_< \cup X_\geq = C$. Moreover, by Lemma 1, the vertices in $X_\geq$ must also belong to $X_<$, i.e., $X_\geq \subset X_<$. It follows that $X_< = C$. (2) We prove this by contradiction. Assume there exist clusters x_i and x_j such that $x_i.\gamma < \alpha \wedge x_j.\gamma < \alpha \wedge x_i \cap x_j \neq \emptyset$. Without loss of generality, let vertex $w \in x_i \cap x_j$ and $\alpha < x_i.\alpha < x_j.\alpha$. Based on Definition 4, we have $P(w, x_j.\alpha) \geq x_i.\alpha$, which contradicts the assumption that $x_j.\gamma = P(w, x_j.\alpha) < \alpha$. □

Example 3.3. *In Figure 4(a), when searching for an (α, β)-core, the bi-clusters within the query region that satisfy $\gamma < \alpha$ are valid, while the others are duplicates (Case-1). For example, the $(1, 3)$-core consists of disjoint bi-clusters satisfying $(\alpha, \beta) \succeq (1, 3)$ and $\gamma < 1$, which are enclosed by blue dashed boxes in Figure 4(a). A more intuitive visualization is provided in Figure 4(b). The figure represents*

the $(1,3)$*-core, which is divided into three disjoint modules: the pink module* $(1,4,0)$*-cluster, the blue module* $(2,3,0)$*-cluster, and the orange module* $(3,3,0)$*-cluster. All these bi-clusters have* γ *values less than 1. In addition, there are two bi-clusters with* $\gamma \geq 1$*, namely the green module* $(3,3,1)$*-cluster and the yellow module* $(2,3,1)$*-cluster, both of which overlap with the pink module and contain duplicate vertices.*

4 SGL Index

In this section, based on the above insights, we first combine bi-clusters with bi-components to obtain (α,β,γ)-groups. We then design an SGL (Summary Graph + Location) index structure for bi-component search and present the SGL-based search algorithm.

4.1 Bi-Component Division

Properties 2.1 and 2.2 also characterize the nested and overlapping relationships among bi-cores' substructures, the bi-components. In Theorem 1, we divide bi-cores using disjoint (α,β,γ)-clusters. Following the same principle, bi-components can also be divided according to the corresponding substructures of (α,β,γ)-clusters. We formalize this as follows:

Definition 6. **γ-Group.** *Given a subgraph g of (α,β)-component H, g is an γ-group if it satisfies the following conditions: (1) $g \subseteq (\alpha,\beta,\gamma)$-cluster; (2) No other subgraph $g' \subseteq H$ satisfies conditions (1), and contains g (i.e., $g \subset g'$).*

In summary, a γ-group g is defined as the intersection of an (α,β)-component H and an (α,β,γ)-cluster x, that is, $g = H \cap x$. As depicted in Figure 4(b), the two $(2,3)$-components H_1 and H_2 are enclosed in black dashed boxes. The blue and yellow modules represent the $(2,3,0)$-cluster x_4 and the $(2,3,1)$-cluster x_5, respectively. Using these clusters, H_1 is divided into two γ-groups: the 0-group $g_0 = H_1 \cap x_4 = \{v_4\}$, and the 1-group $g_1 = H_1 \cap x_5 = \{u_4\}$. In contrast, H_2 contains only one 0-group, where $g_2 = H_2 \cap x_4 = \{u_6, u_7, u_8, v_7, v_8\}$.

Lemma 2. *Given the (α,β,γ)-cluster x, and let $\mathcal{G}_{\alpha,\beta,\gamma}$ denote the set of all γ-groups contained in the (α,β)-components of the (α,β)-core. Then $x = \bigcup_{g\in\mathcal{G}_{\alpha,\beta,\gamma}} g$, and $|x| = \sum_{g\in\mathcal{G}_{\alpha,\beta,\gamma}} |g|$.*

Proof. Since $x \subseteq (\alpha,\beta)$-core C, and C can be decomposed as the union of its (α,β)-components, i.e., $C = \bigcup H_i$ where each H_i is an (α,β)-component, we have $x = x \cap C = \bigcup (x \cap H_i) = \bigcup_{g\in\mathcal{G}_{\alpha,\beta,\gamma}} g$. Since (α,β)-components are maximal and disjoint, the γ-groups they contain are also disjoint, so $|x| = \sum_{g\in\mathcal{G}_{\alpha,\beta,\gamma}} |g|$. □

Theorem 2. *Let H be an (α,β)-component in G. Let $\{\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_k\}$ denote the sets of γ-groups contained in all (α^+,β^+)-components in H under different parameter values, with $(\alpha^+,\beta^+) \succeq (\alpha,\beta)$ and $\gamma < \alpha$. Then (1) $H = \bigcup_{i=1}^{k} \bigcup_{g\in\mathcal{G}_i} g$, and (2) $|H| = \sum_{i=1}^{k} \sum_{g\in\mathcal{G}_i} |g|$.*

Proof. Let C' be an (α,β)-core that contains only H, i.e., $C' = H$. Then, according to Lemma 2, the set $\{\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_k\}$ is equivalent to the bi-clusters in C' satisfying $(\alpha^+,\beta^+) \succeq (\alpha,\beta)$ and $\gamma < \alpha$. Substituting this into Theorem 1, then Theorem 2 holds. □

Next, we introduce our index structure based on γ-groups.

4.2 The Structure of SGL

Figure 5 shows the proposed SGL index, which consists of three key components: **SNodes**, **SEdges**, and **Location** as described below.

SNode. An SNode corresponds to a bi-component in the bipartite graph. Each SNode s consists of a core pair and multiple γ-groups. All γ-groups within the SNode s belong to the same (α,β)-component, where $(\alpha,\beta) = Cpair(s)$. Figure 7(a) shows the SNodes obtained from Figure 1(a).

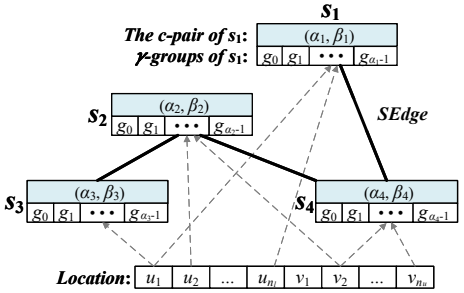


Fig. 5. The structure of SGL

SEdge. An SEdge models nested relationships among bi-components. Each SEdge connects two SNodes, indicating that their corresponding bi-components are either direct-nested or co-nested.

Direct-nested: Given two bi-components H_i and H_j, if $H_i \subseteq H_j$ and no non-empty bi-component H_k exists such that $Cpair(H_i) \succ Cpair(H_k) \succ Cpair(H_j)$, then H_i and H_j are direct-nested based on Property 3.1.

Co-nested: Given two bi-components H_i and H_j, if H_i and H_j are non-nested, but there exists another bi-component H_k such that $H_i \subset H_k$ and $H_j \subset H_k$, they are co-nested.

Example 4.1. *In Figure 1(a), the* $(1,3)$*-core contains two* $(1,3)$*-components, separated by the red dashed line. The left* $(1,3)$*-component contains one* $(1,4)$*-component and one* $(3,3)$*-component. The* $(1,4)$*-component is direct-nested within the* $(1,3)$*-component, whereas the* $(3,3)$*-component is not, because there exists a* $(2,3)$*-component between them. The* $(1,4)$*-component and the* $(3,3)$*-component are non-nested, and since both belong to the same* $(1,3)$*-component, they are classified as co-nested.*

We explicitly model these two types of relationships to establish the relational structure among bi-components, capturing their direct-nested and co-nested relationships. It is important to note that, after redundancy elimination, some bi-components may be empty; therefore, only non-empty bi-components are considered. If $Cpair(s_1)$ dominates $Cpair(s_2)$ (or vice versa), the relationship is direct-nested; otherwise, it is co-nested. Note that $Cpair(s_1) = Cpair(s_2)$ never occurs.

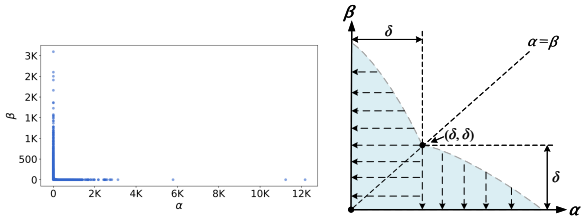
Location. The Location is a set of vertex mappings used to locate the SNode in which each vertex is contained.

These three structures model the bi-components as a connected Summary Graph (SG), where each SNode represents a deduplicated bi-component and uses γ-groups to handle overlaps among different bi-components. SEdges explicitly capture the nested or co-nested relationships between nodes, while Location provides a fast mapping from vertices to their corresponding SNodes.

In the SGL, SNodes are divided into γ-groups by their α values; thus, each SNode contains at most $\alpha_{\max}$ γ-groups, giving an upper bound of $O(\alpha_{\max} \cdot |S|)$ for all SNodes. SNodes are connected via SEdges, and in the worst case, they may form a dense graph, leading to $O(|S|^2)$ SEdges. Let μ denote the average number of SNodes each vertex belongs to, so the total number of vertices in all SNodes is $O(\mu \cdot n)$. Therefore, given a bipartite graph G, the space required to store the SGL index of G is $O(\alpha_{max} \cdot |S| + |S|^2 + \mu \cdot n)$, where $\alpha_{\max}$ denotes the maximum α value among all (α, β)-cores in G, and $|S|$ is the number of SNodes.

4.3 Dual-Driven Spatial Optimization

To reduce the index size, we propose two optimization strategies: (1) threshold-based grouping, and (2) dominance-based ordered pruning of SEdges.



(a) (α, β)-cores of AR (b) δ-based grouping

Fig. 6. Threshold-based grouping

Strategy I: Threshold-based Grouping. Our goal is to reduce the number of γ-groups. Since the predecessor value is computed from the α value, each SNode s contains $s.\alpha$ γ-groups. In real-world bipartite graphs, the distribution of (α, β)-cores is often highly skewed; for example, in the AR graph (Figure 6(a)), the maximum α is 12,180, the maximum β is 3,096, while the maximum δ is only 26. This motivates partitioning γ-groups based on δ: for bi-clusters with $\alpha \leq \delta$, the computation of γ (i.e., predecessor value) remains unchanged, whereas for vertices in bi-clusters with $\alpha > \delta$, the c-pairs of their non-nested (α, β)-cores are sorted in ascending order of β, with the predecessor value set as $P(w, \beta_1) = 0$ for the first c-pair (α_1, β_1) and $P(w, \beta_i) = \beta_{i-1}$ for the remaining c-pair (α_i, β_i). As shown in **Case-2** of Figure 4(a), we determine whether a γ-group is redundant by checking if its γ value exceeds the corresponding β value. In summary, for a vertex w in an (α, β, γ)-cluster, the value of γ is formally defined as:

$$\gamma = \begin{cases} P(w, \alpha), & \text{if } \alpha \leq \delta, \\ P(w, \beta), & \text{if } \alpha > \delta. \end{cases} \tag{1}$$

Lemma 3. *Given an $(\alpha_i, \beta_i, \gamma_i)$-cluster $x_i \subset (\alpha, \beta)$-core with $\alpha_i > \delta \wedge \gamma_i > \beta$, for any vertex $w \in x_i$, there exists at least one $(\alpha_j, \beta_j, \gamma_j)$-cluster $x_j \subset (\alpha, \beta)$-core with $\alpha_j > \delta \wedge \gamma_j < \beta$ such that $w \in x_j$.*

Proof. The proof is similar to that of Lemma 1. □

As shown in Figure 6(b), the dashed arrows represent the directions in which the predecessor values of vertices in different bi-cores are computed. After partitioning based on δ, the number of γ-groups in each SNode does not exceed δ, and it further decreases as either α or β increases.

Strategy II: Dominance-based Ordered Pruning of SEdges. A minimal spanning tree is often used to preserve graph connectivity while reducing the number of edges. However, since each bi-community search focuses only on a local subgraph, applying a global minimal spanning tree is not feasible. To address this, we introduce the concept of *maximum query pair* to define the effective query bounds of each SEdge.

Definition 7. **Maximum Query Pair**. *Given a SEdge $e = (s_i, s_j)$, its maximum query pair is uniquely defined as $MQpair(e) = (\min(s_i.\alpha, s_j.\alpha),\ \min(s_i.\beta, s_j.\beta))$.*

Lemma 4. *Given an SEdge $e = (s_i, s_j)$, if there exists a path P in the SG that connects s_i and s_j that bypasses e and all SNodes $s \in P$ satisfy $Cpair(s) \succeq MQpair(e)$, then e is redundant.*

Proof. Since $MQpair(e)$ is dominated by the c-pairs of all SNodes in P, these SNodes are all contained within the corresponding bi-components that include e. Therefore, when querying these bi-components, e can be replaced by path P. Thus, e is redundant. □

Example 4.2. *In Figure 7(b), the left side shows the original SG obtained from Figure 1(a), and the right side shows the pruned SG. Our strategy is to first compute the maximum query pairs for each*

SEdge in the left figure, and then group SEdges into buckets according to their maximum query pairs. Edges in buckets with stronger dominance are inserted first into the SNodes of the right figure. If an SEdge is inserted and its two endpoints are already connected, then by Lemma 4 it is redundant and therefore skipped.

Complexity analysis. After applying Strategies I and II, the space complexity of the SGL is reduced to $O(\delta \cdot |S| + \mu \cdot n)$. By Strategy I, each SNode contains at most δ γ-groups, giving $O(\delta \cdot |S|)$ in total. According to Lemma 4, any SEdge forming a cycle with dominating SNodes is redundant and will be pruned. Therefore, no cycles composed solely of SEdges remain, and only $O(|S|)$ essential edges are preserved to maintain connectivity. Therefore, the overall space complexity of the SGL is $O(\delta \cdot |S| + \mu n)$.

4.4 SGL-based Bi-Community Search

In this section, based on Theorem 2, we design a bi-community search algorithm using the SGL index, as outlined in Algorithm 1.

Algorithm 1: SGL-Based Bi-Community Search

Input: The SGL index S, Location L, query pair (α, β), and query vertex q.
Output: The (α, β)-community H

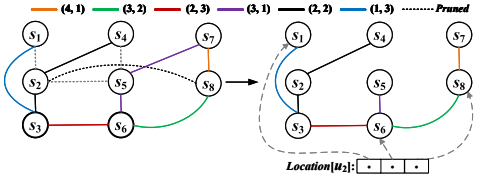
```
1  Q ← ∅ ; F ← ∅ ; H ← ∅;
2  foreach s ∈ L_q do
3      if Cpair(s) ⪰ (α, β) then
4          add s to Q ; add s to F;
5  while Q ≠ ∅ do
6      s ← Q.pop();
7      foreach w ∈ N(s) do
8          if w ∉ F ∧ Cpair(w) ⪰ (α, β) then
9              add w to Q ;
10         add w to F;
11     foreach g ∈ groups(s) do
12         if s.α ≤ δ ∧ g.γ < α or s.α > δ ∧ g.γ < β then
13             H ← H ∪ g;
14 return H;
```

To obtain the (α, β)-component containing the query vertex q, we first examine the mapping L_q to locate the first SNode s that contains q and whose c-pair $Cpair(s)$ dominates the query pair (α, β) (Lines 2–4). From s, we perform a breadth-first traversal on the SGL (Lines 5–10). Next, we collect vertices from SNodes as follows (Lines 11–13): (1) for SNodes s with $s.\alpha \leq \delta$, we retrieve vertices from γ-groups satisfying $g.\gamma < \alpha$; (2) for SNodes s with $s.\alpha > \delta$, we retrieve vertices from γ-groups with $g.\gamma < \beta$.

Example 4.3. *In Figure 7(b), we search for the $(1, 3)$-community containing u_2. First, u_2 is mapped to s_1, and since $Cpair(s_1) = (1, 4) \succ (1, 3)$, a breadth-first traversal on the SG is started from s_1. The SNodes s_3 and s_6 satisfy $Cpair(\cdot) \succeq (1, 3)$ while all other SNodes do not, so the traversal terminates. We then collect the γ-groups with $\gamma < 1$ from s_1, s_3, and s_6 (i.e., the g_0 groups of the three SNodes in Figure 7(a)), yielding the bi-community $H = \{u_1, u_2, u_3, u_4, u_5, v_2, v_3\} \cup \{v_4\} \cup \{v_1\}$.*

SNode	γ-groups	SNode	γ-groups
s_1: (1, 4)	g_0: $\{u_1, u_2, u_3, u_4, u_5, v_2, v_3\}$	s_5: (3, 1)	g_2: $\{u_6, v_7, v_8\}$
s_2: (2, 2)	g_0: $\{v_6\}$	s_6: (3, 3)	g_0: $\{v_1\}$ g_1: $\{u_1, u_2, u_3, v_2, v_3\}$
s_3: (2, 3)	g_0: $\{v_4\}$ g_1: $\{u_4\}$	s_7: (4, 1)	g_0: $\{v_5, v_6\}$ g_2: $\{u_4\}$
s_4: (2, 3)	g_0: $\{u_6, u_7, u_8, v_7, v_8\}$	s_8: (4, 2)	g_2: $\{v_4\}$ g_3: $\{u_1, u_2, v_1, v_2, v_3\}$

(a) The SNodes of Figure 1(a)



(b) The SGL and pruning of SEdges

Fig. 7. Illustration of SNodes and the SGL with SEdge pruning

Complexity analysis. In the worst case, Algorithm 1 needs to access all γ-groups within the searched SNodes, which takes $O(\delta \cdot s)$ time. Once the qualified SNodes are identified, retrieving their vertices takes $O(|H|)$ time, where $|H|$ is the number of vertices in the (α, β)-community. Therefore, the total time complexity is $O(\delta \cdot s + |H|)$. Since $\delta \cdot s < |H|$ in practice, the time complexity of Algorithm 1 is $O(|H|)$.

5 SGL Construction

Based on the above introduction, we present the construction of the SGL index, which consists of three phases. The construction process of the SGL is detailed in Algorithm 2. We use S, SE, and L to denote the set of SNodes, the set of SEdges, and Location, respectively.

5.1 The Overview of SGL Construction

The construction process of the SGL consists of three phases.

Phase I: Decomposition of (α, β)-Components. In the first phase, we decompose the bipartite graph $G = (U, V, E)$ into all (α, β)-components by performing Algorithm 3 (Line 1). We first compute the δ value, and then, for each α value smaller than δ, we iteratively decompose the corresponding (α, β)-components. Similarly, for each β value smaller than δ, we decompose the corresponding (α, β)-components in order. For each newly obtained bi-component, we create an empty SNode and update the location information of all vertices in this bi-component. In addition, each SEdge represents a nested relationship between bi-components, which can be recorded directly during the decomposition process. The detailed decomposition algorithm is presented in Section 5.2.

Phase II: Computation of Predecessor Values and Formation of γ-Groups. In this phase, we compute the predecessor value for each vertex $w \in U \cup V$ based on its mapping L_w, which records the series of SNodes to which w belongs during the decomposition process. Since decomposition is performed in an ordered manner, L_w is also ordered, with α values arranged in ascending order and β values in descending order. For each vertex w, we traverse its location sequence L_w to compute the predecessor value γ. For each $l \in L_w$, let l^- and l^+ denote the previous and next elements of l in L_w, respectively. We initialize γ as 0. If the α value of the SNode s_l is smaller than δ and l^- exists, then γ is set to the α value of s_{l^-}. If the α value of s_l is greater than δ and l^+ exists, then γ is set to the β value of s_{l^+}. Finally, the vertex w is inserted into the corresponding γ-group of s_l according to the computed γ value (Lines 2-9).

Phase III: Dominance-based Pruning of SEdges. In the final phase, We perform the pruning algorithm on all candidate SEdges in SE (Line 11). First, we compute the $MQpair(e)$ for each SEdge e and insert the SEdges into the SG in descending order of these maximum query pairs. For each SEdge, we check whether it satisfies the pruning condition described in Lemma 4. The detailed pruning procedure is presented in Section 5.3.

Complexity analysis. The decomposition of (α, β)-components requires $O(\delta m \cdot \psi(n))$ time (Section 5.2). The computation of predecessor values for all vertices needs to access their location

Algorithm 2: Overview of SGL Construction Algorithm

```
Input: A bipartite graph G = (U, V, E)
Output: The SGL of G
// Phase I
1  S, SE, L ← decompose the bi-components on G ;                    // Algorithm 3
// Phase II
2  foreach w ∈ U ∪ V do
3      foreach l ∈ L_w do
4          γ ← 0;
5          if s_l.α ≤ δ ∧ ∃l⁻ then
6              γ ← the α value of s_{l⁻} ;
7          else if s_l.α > δ ∧ ∃l⁺ then
8              γ ← the β value of s_{l⁺} ;
9          add w to the γ-group in s_l;
// Phase III
10 Perform dominance-based pruning on SE ;                          // Algorithm 4
```

sequences, which takes at most $O(m)$ time. Finally, during the pruning of SEdges, the connectivity is checked using an Union–Find data structure, which takes $O(|SE| \cdot \psi(|S|))$ time (Section 5.3). Since $\delta \leq \sqrt{m}$ [20], the worst-case complexity is $O(m^{3/2} \cdot \psi(n) + |SE| \cdot \psi(|S|))$. Therefore, the time complexity of Algorithm 2 is $O(\delta m \cdot \psi(n) + |SE| \cdot \psi(|S|))$, where $\psi(\cdot)$ denotes the inverse Ackermann function [33].

5.2 Decomposition of (α, β)-Components

In this section, we present the bi-component decomposition algorithm, which consists of three parts: **(1) Bi-core Decomposition.** For each α value smaller than δ, decomposing its corresponding series of (α, β)-cores (Lines 4–11). **(2) Bi-component Decomposition.** For the obtained (α, β)-cores, incrementally dividing them into connected components in descending order of β to obtain bi-components. Then, constructing SNodes and updating their locations based on these bi-components (Lines 12–20). **(3) SEdges Recording.** Recording the SEdges by capturing the nested relationships among the bi-components obtained during the decomposition process (Lines 18 and 20). After these three parts, the above procedure is repeated for each β value smaller than δ (Line 21).

Part (1): Bi-core Decomposition. We first compute the δ value of the maximum (δ, δ)-core in graph G. This is done by iteratively removing vertices with degrees less than 1 up to δ. Then, we iterate α from 1 to δ, and for each fixed α, we decompose the corresponding (α, β)-cores (Lines 3–11). The process begins by initializing G^- and D, where D is a hierarchical directed acyclic graph (DAG) with distinct layers, used to store vertices from different (α, β)-cores according to their β values. We iteratively increase β, adding lower-layer vertices w with degrees less than β, or upper-layer vertices w with degrees less than α, along with their neighbors $N(w, G^-)$ in G^-, to the $(\beta - 1)$-th layer of D. Subsequently, w and its connected edges are removed from G^-. In this way, D stores a series of (α, β)-cores for a specific α with different β values, where the neighbors of each vertex are all vertices located in higher β layers than itself.

Part (2): Bi-component Decomposition. We partition the components in the DAG using an Union–Find data structure (Lines 12–18). We traverse the layer $D[\beta]$ of the DAG in reverse order, and the vertices within $D[\beta]$ are also processed in reverse. For each vertex w, we first check whether there exists a neighbor w' such that $w' \in N(w, D)$ and $w' \in D[\beta]$. If such a neighbor exists, w is

Algorithm 3: Bi-Component Decomposition

Input: A bipartite graph $G = (U, V, E)$
Output: The set B of bi-component numbers of all vertices in G

1 initialize $S \leftarrow \emptyset$, $SE \leftarrow \emptyset$, $L \leftarrow \emptyset$;
2 $\delta \leftarrow$ the maximum value s.t. (δ, δ)-core $\neq \emptyset$;
3 **for** α *from 1 to* δ **do**
  `// Part (1)`
4 $G^- = (U^-, V^-, E^-) \leftarrow G$, $D[][] \leftarrow \emptyset$;
5 $\beta \leftarrow 1$;
6 **while** $G' \neq \emptyset$ **do**
7 **foreach** $w \in U^- \cup V^-$ **do**
8 **if** $|N(w, G^-)| < \alpha \wedge w \in U^-$ *or* $|N(w, G^-)| < \beta \wedge w \in V^-$ **then**
9 add $(w, N(w, G^-))$ to $D[\beta\text{-}1]$;
10 remove w and its incident edges from G^-;
11 $\beta \leftarrow \beta + 1$;
  `// Part (2)`
12 **for** β *from len(D) to 1* **do**
13 **foreach** $w \in$ *reverse*$(D[\beta])$ **do**
14 **if** $\exists w' \in N(w, D) \wedge w' \in D[\beta]$ **then**
15 $l_w \leftarrow l_{w'}$; $w.father \leftarrow Find(w')$;
16 **else**
17 add a new node s with $Cpair(s) = (\alpha, \beta)$ to S and $l_w \leftarrow$ the position of s in S;
18 $Union(w, \{w'.\text{root} \mid w' \in N(w)\})$;/*__Record-1__*/
19 **foreach** $w \in D[\beta]$ **do**
20 $updateLocation(L_w, l_w)$; /*__Record-2__*/
21 run Lines 3-20 by swapping α with β, but modify Line 5 to be $\alpha \leftarrow \delta$;

added to the same connected component as w', and the parent of w is updated to the root of w' in the Union–Find, while inheriting $l_{w'}$ (i.e., the position of SNode to which w' belongs in S). Otherwise, w is merged with the roots of all its neighbors in the Union–Find, $Union(w, \{w'.\text{root} \mid w' \in N(w)\})$. A new node s is then added to S with $Cpir(s) = (\alpha, \beta)$, and l_w is set to the position of s in S. Finally, we update the set L_w by adding the current l_w to L_w; if any other element in L_w points to an SNode whose c-pair is dominated by $Cpair(S_{l_w})$, that element is removed.

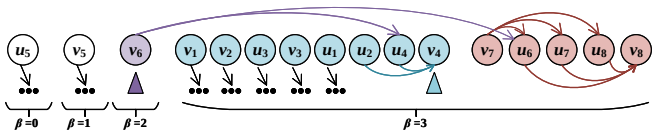


Fig. 8. The DAG of graph in Figure 1(a) when $\alpha = 2$

Example 5.1. *Figure 8 shows the DAG resulting from decomposing the bipartite graph in Figure 1(a) with $\alpha = 2$. The DAG is divided into four regions based on the β values, with vertices ordered by their removal sequence. We traverse the vertices from right to left and compute the bi-components. Upon reaching v_4 (blue triangle), a (2, 3)-component H_1 (red vertices) is identified. Since v_4 has no*

outgoing edges, we create a new component and add it. After processing vertices with β=3, the second (2, 3)-component H_2 (blue vertices) is obtained. When reaching v_6 (purple triangle), whose edges point to u_6 in H_1 and u_4 in H_2, we merge H_1 and H_2 into a (2, 2)-component H_3, and update L_{v_6}. Finally, we process vertex v_5.

Part (3): SEdges Recording. In this part, we introduce how to obtain SEdges by using Record-1 and Record-2 to record the nested relationships between bi-components during the decomposition process. During the decomposition for a specific α (or β) value, Record-1 captures SEdges for direct-nested bi-components sharing the same α (or β), while Record-2 captures SEdges for direct-nested or co-nested bi-components with different α (or β) values.

Record-1. We decompose bi-components in order of their α or β values. Consequently, when a bi-component is merged with others to form a new bi-component, the merged bi-components are necessarily direct-nested. Consider Line 18 of Algorithm 3, where the bi-component containing a neighbor w' of w is merged into the bi-component containing w, and we record $(S[l_w], S[l_{w'}])$. For example, in Figure 8, the $(2,3)$-core contains two bi-components (red and blue): $H_1 = \{v_7, u_6, u_7, u_8, v_8\}$ and $H_2 = \{v_1, v_2, u_3, v_3, u_1, u_2, u_4, v_4\}$. Processing in reverse order, when reaching vertex v_6, we merge H_1, and H_2 into a $(2,2)$-component H_3, and consequently H_1 and H_3, as well as H_2 and H_3, are direct-nested.

Record-2. We record nested relationships when a vertex's location is updated. Consider Line 20 of Algorithm 3: during decomposition, if a vertex w belongs to two bi-components with different α (or β) values, these bi-components are either direct-nested or co-nested. Therefore, each time a new SNode position l_w for w is obtained, we can generate an SEdge $(S[l_w], S[l_w^-])$, where $S[l_w^-]$ is the last obtained position. For example, in Table 2, we already know that u_1 belongs to three mutually non-nested bi-cores, whose c-pairs are $(1,4)$, $(3,3)$, $(4,2)$. Therefore, under the decomposition with $\alpha = 1$, $u_1 \in (1,4)$-component H_1. Under the decomposition with $\alpha = 3$, $u_1 \in (3,3)$-component H_3. In Figure 8, under the decomposition with $\alpha = 2$, $u_1 \in (2,3)$-component H_2. Since $(1,4) \not\succeq (2,3)$, H_2 and H_1 are co-nested; since $(3,3) \not\succeq (2,3)$, H_2 and H_3 are direct-nested.

Complexity analysis. The Algorithm 3 processes each vertex and edge at most once per (α, β)-core, with connectivity maintained via Union-Find with path compression. Across all $\alpha \leq \delta$ and $\beta < \delta$, the total worst-case time complexity is $O(\delta m \cdot \psi(n))$, with the Union-Find operations dominating the cost.

5.3 Dominance-based Pruning of SEdges

Next, we perform pruning as described in Example 4.2. Let rec_1 and rec_2 denote the sets of SEdges obtained from Record-1 and Record-2, respectively, such that $SE = rec_1 \cup rec_2$. Since the SEdges in rec_1 are generated via Union-Find [33] and are mutually non-redundant, pruning is applied only to the SEdges in rec_2.

Algorithm 4 first computes the maximum query pair $MQpair(e)$ for each edge e, and groups all edges into the corresponding buckets $B[MQpair(e)]$ based on their maximum query pair (Lines 2–4), where B denotes the set of all buckets, and each bucket $b \in B$ corresponds to a unique maximum query pair. Then, the algorithm traverses these buckets in descending order of dominance hierarchy, i.e., from stronger to weaker dominance (Line 5). For each edge $e = (s_i, s_j)$ in bucket b, if e belongs to rec_1, the algorithm directly merges s_i and s_j in the Union–Find, and adds them as mutual neighbors (Lines 6–7); if e belongs to rec_2, the merge and neighbor connection are executed only when s_i and s_j are not yet connected, thereby eliminating dominated or redundant edges (Lines 8–10).

Algorithm 4: Dominance-based Pruning

Input: The SNode set S and SEdge set $SE = rec_1 \cup rec_2$.
Output: The pruned SE

```
1  B ← ∅;
2  foreach e = (s_i, s_j) ∈ SE do
3      MQpair(e) ← ( min(s_i.α, s_j.α), min(s_i.β, s_j.β));
4      add e to B[MQpair(e)];
5  foreach b ∈ B in descending order do
6      foreach e = (s_i, s_j) ∈ b ∧ e ∈ rec_1 do
7          Union(s_i, s_j); add s_i to N(s_j) and add s_j to N(s_i) ;
8      foreach e = (i, j) ∈ b ∧ e ∈ rec_2 do
9          if Find(s_i) ≠ Find(s_j) then
10             Union(s_i, s_j); add s_i to N(s_j) and add s_j to N(s_i) ;
```

Complexity analysis. Each edge in SE is processed at most once, and checking and merging of connectivity are performed using Union-Find with path compression, whose amortized cost per operation is $O(\psi(|S|))$[33]. Therefore, the time complexity of Algorithm 4 is $O(|SE| \cdot \psi(|S|))$.

6 SGL Maintenance

In this section, we present the SGL maintenance algorithm. Since vertex insertion/removal can be simulated by a series of edge insertions/removals [24], we primarily focus on edge-based updates. When an edge (u, v) is inserted or deleted in the bipartite graph, the bi-components containing u, v, and some of their connected vertices may change. When such changes occur, we need to update the *Location* of these vertices and add or remove them from the corresponding SNodes. The change in the number of vertices within an SNode affects the SEdges, and we update them accordingly. Therefore, the maintenance process mainly consists of the following parts:

Updating the Location and recording the changes. When a new edge is inserted or deleted in the bipartite graph, the bi-components containing certain vertices may change, and it is necessary to compute the updated bi-components to which these vertices belong. Prior work [24] has proposed efficient and effective frameworks for maintaining this information, which can be easily extended to support the Location structure in our framework. Therefore, we update the Location based on these existing algorithms and record the changes for subsequent maintenance.

Recomputing predecessor values and updating SNodes. We recompute the predecessor values of the affected vertices and add or remove them from the γ-groups of the corresponding SNodes. We analyze the impact of these changes on the SNodes, and three possible scenarios may arise: (1) The existing SNodes remain unchanged, but some vertices need to be either added to or removed from these SNodes; (2) Some SNodes become empty due to vertex departures, necessitating their removal from the SGL; (3) A new bi-component is generated, requiring the creation of a new SNode to accommodate the affected vertices.

Updating the SEdges of the affected SNodes. SEdges reflect the nesting relationships among bi-components. When the vertex set within an SNode changes, these updates include the following two cases: (1) **Vertex transfer among SNodes.** When a vertex w is updated from an SNode s_i to a newly created or existing SNode s_j, we consider adding an SEdge between s_i and s_j. Similar to Record-2, the new SEdges can be identified by tracking the changes of the affected vertices. (2) **Deleting an SNode.** When an SNode becomes empty after all its vertices are removed, that SNode and all its associated SEdges should be deleted. If the deleted SNode is located in the

Algorithm 5: SGL-Insertion/Deletion

Input: the set of edges to be inserted/deleted, denoted as E'
Output: the updated index SGL

```
1  U' ← ∅ ; V' ← ∅ ; rec ← ∅ ; S_new ← ∅;
2  compute U', V', rec and S_new by Algorithm 1/ Algorithm 5 in [24];
3  S ← S ∪ S_new;
4  foreach w ∈ U' ∪ V' do
5      foreach (s_i, s_j) ∈ rec do
6          γ ← perform Lines 4-8 of Algorithm 2, replacing s_l with s_j;
7          remove w from s_i and add it to the γ-group in s_j;
8          if s_i ∉ N(s_j) then
9              add s_i to N(s_j) and add s_j to N(s_i) ;
10         if s_i = ∅ then
11             remove s_i and add the corresponding SEdges between its neighboring SNodes;
12 perform Algorithm 4 to prune the SEdges;
```

middle of a nesting hierarchy, new SEdges can be established between its adjacent levels. For instance, if an SNode s_i is nested between s_j and s_k with $Cpair(s_j) \succ Cpair(s_i) \succ Cpair(s_k)$, removing s_i requires directly connecting s_j and s_k. Moreover, if s_i connects two SNodes s_j and s_k with $Cpair(s_i) \succ Cpair(s_j)$ and $Cpair(s_i) \succ Cpair(s_k)$, then after removing s_i, s_j and s_k become co-nested, and a new SEdge should be added between them.

Algorithm 5 describes the dynamic maintenance of the SGL index under edge insertions and deletions. First, the affected vertex sets U' and V', the record set rec, and the set of newly created SNodes S_{new} are initialized as empty (Line 1). Here, rec records, for each affected vertex, the changes in its membership from one SNode to another. The sets U', V', rec, and S_{new} are then computed by invoking Algorithm 1 or Algorithm 5 in [24] (Line 2). Next, the new SNodes S_{new} are added to the index S. For each affected vertex $w \in U' \cup V'$ and each pair $(s_i, s_j) \in rec$, the algorithm performs Lines 4-8 of Algorithm 2 in paper with s_l replaced by s_j, removing w from s_i and adding it to the corresponding γ-group in s_j. If s_i is not a neighbor of s_j, then sets s_i and s_j as neighbors of each other. If s_i becomes empty, it is removed from the index, and the corresponding SEdges are added between its neighboring SNodes. Finally, Algorithm 4 in paper is performed to prune the SEdges, completing the dynamic maintenance.

Complexity analysis. Updating the Location and recording changes takes $O(m)$ time in the worst case[24]. Updating SNodes and SEdges requires considering the nodes involved in the record set rec, with time complexity $\sum_{s \in rec} N(s)$. Pruning newly added SEdges only needs to run Algorithm 4 up to the minimum $MQpair$ of these SEdges, with a worst-case complexity of $O(|SE| \cdot \psi(|S|))$. Therefore, the overall time complexity is $O\big(m + \sum_{s \in rec} N(s) + |SE| \cdot \psi(|S|)\big)$.

7 Experiments

In this section, we present the experimental results.

7.1 Experimental Setup

Datasets. We conduct extensive experiments using nine real-world bipartite graphs, all sourced from KONECT[1], and one synthetic bipartite graph [24]. Table 3 presents the statistics for each

[1]http://konect.cc/networks/

Table 3. Statistics of used bipartite graphs

Graphs	$\|U\|$	$\|V\|$	$\|E\|$	δ	μ
IMDB (IMDB)	303,617	896,302	3,782,463	23	4.3
Wikipedia-en (WC)	1,853,493	182,947	3,795,796	18	2.1
amazon-ratings (AR)	2,146,057	1,230,915	5,743,258	26	2.3
dblp (DBLP)	1,953,085	5,624,219	12,282,059	14	2.3
Wikipedia-edits-de (DE)	1,025,084	5,910,432	55,231,903	212	8.9
LiveJournal (LJ)	3,201,203	7,489,073	112,307,385	108	12.3
Delicious tag–item(DTI)	4,511,972	33,777,768	137,240,382	180	3.9
Web trackers (WT)	27,665,730	12,756,244	140,613,762	437	3.8
Orkut (OG)	2,783,196	8,730,857	327,037,487	466	36.9
Power law (PL)	5,000,000	5,000,000	1,000,000,000	169	88.9

bipartite graph, where $|U|$ and $|V|$ denote the number of vertices on the two sides of the graph, $|E|$ denotes the number of edges, δ denotes the largest value such that the (δ, δ)-core $\neq \emptyset$, and μ denotes the average number of times a vertex is stored.

Algorithms. We evaluate the construction efficiency and query efficiency of the following algorithms:

- ***BI*:** the algorithm based on the bi-core index in [20].
- ***BH*:** the algorithm based on the hierarchy index in [41].
- ***BN*:** the algorithm based on the bi-core number in [24].
- ***MCR*:** the algorithm based on the MCR-Tree index in [23].
- ***SGL*:** our algorithm based on the SGL index.

Experimental settings. For the above algorithms, we utilize the source code provided by the authors. All algorithms are implemented in C++, and the experiments are conducted on a Windows Server equipped with a 2.10 GHz 16-core CPU and 256 GB of memory. The reported running time represents the average time over multiple query vertices. If the running time of an algorithm exceeds 72 hours, its execution is terminated.

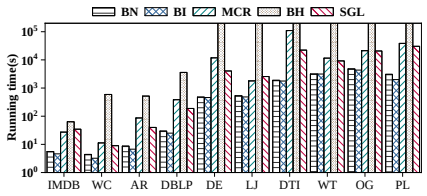


Fig. 9. Construction time of indexes

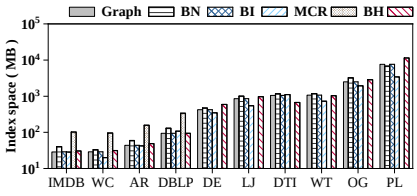


Fig. 10. Size of indexes

7.2 Overall Performance of Indexes

We evaluate the performance of the indexes by comparing their construction time and space consumption across various datasets.

Exp-1: Construction time over different datasets. Figure 9 compares the construction efficiency of five indexing methods across multiple datasets. The results show that *BI* and *BN* exhibit relatively fast construction times, as they only process the structurally simple (α, β)-core. In contrast, *BH* incurs significantly higher construction time due to the necessity of building numerous α-trees and β-trees for various values of α and β. On large datasets such as DE, OG and PL, *BH* often fails to complete index construction due to timeouts (e.g., exceeding 72 hours on DE and LJ) or excessive memory consumption (exceeding the 256GB memory limit on DTI, WT, OG and PL), which justifies its absence from the corresponding figures. In contrast, *SGL* significantly reduces redundant computations, thereby outperforming *BH* in construction efficiency. For example, on the DBLP dataset, *SGL* completes construction in 187.86s, while *BH* requires 3575.37s, and *MCR* takes

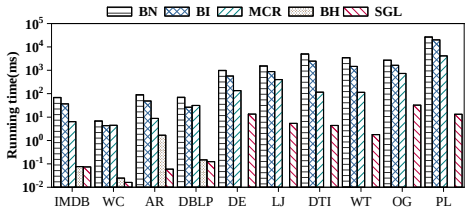


Fig. 11. Performance of indexes for bi-community search

385.07s. Overall, *SGL* achieves superior construction efficiency and scalability while maintaining index expressiveness, making it suitable for large bipartite graphs.

Exp-2: Index size over different datasets. Figure 10 illustrates the space consumption of various indexes across different datasets. As previously discussed, *BH* incurs significantly higher space overhead due to the construction of numerous α-trees and β-trees at each hierarchy. In contrast, the space consumption of *BI*, *BN*, *MCR*, and *SGL* remains relatively stable and low. Notably, the space complexity of *SGL* grows linearly with the graph size, highlighting its strong scalability and practical applicability. However, for some datasets, our index incurs higher overhead than the baselines, because it increases the number of storage elements required, and the vertex mapping structures also contribute to the overhead.

Exp-3: Performance of indexes for bi-community search over all datasets. Figure 11 shows that *SGL* consistently achieves the best query efficiency across all datasets. In small datasets, although *BH* exhibits impressive query speeds, *SGL* still maintains a clear advantage. For example, on AR, *SGL* queries in 0.058 ms compared to 1.6 ms for *BH*. More importantly, compared to the high construction time and memory overhead of *BH*, *SGL* not only maintains high query efficiency but also features a smaller index size and stronger scalability, making it more suitable for large datasets processing. On large datasets, *SGL* shows an order-of-magnitude improvement over the current state-of-the-art algorithms. For example, on the WT, *SGL* completes queries in 5.4 ms, while *MCR* requires 401.138 ms, representing a speedup of over 70 times.

7.3 Performance for Bi-Community Search

In this section, we first fix the values of α and β, which are typically set to half of the maximum α and β values among all (α, β)-cores in the bipartite graph. Then, we randomly generate 100 query vertices to evaluate the performance of each algorithm. Finally, we assess the impact of varying α and β on query performance.

Exp-4: Varying α. Figures 12(a)-12(d) evaluate the query performance by varying the value of α, for the AR dataset, we fix $\beta = 10$ and test $\alpha = 2, 4, 6, 8, 10$; for the other datasets, $\beta = 100$ and $\alpha = 20, 40, 60, 80, 100$. The results demonstrate that as the value of α increases, the running time of all algorithms decreases, with *SGL* showing a more significant reduction. This is mainly because in bipartite graphs, the number of bi-clusters with higher α values is typically smaller, and these bi-clusters tend to contain more vertices, enabling *SGL* to retrieve query results more efficiently.

Exp-5: Varying β. We evaluate performance by varying the value of β. In Figures 12(e)-12(h), for the AR dataset, we fix $\alpha = 10$ and test $\beta = 2, 4, 6, 8, 10$; for other datasets, $\alpha = 100$ and $\beta = 20, 40, 60, 80, 100$. The results indicate that query time also decreases as β increases. For the AR dataset, we fix α=10, and for the other datasets, we fix α=100. We then evaluate the performance of all algorithms. The results indicate that query time decreases as β increases for reasons similar to those observed for varying α.

Exp-6: Varying graph size. In Figures 12(i)-12(l), we evaluate the performance of all algorithms as the number of edges $|E|$ increases. The results demonstrate that as the number of edges increases,

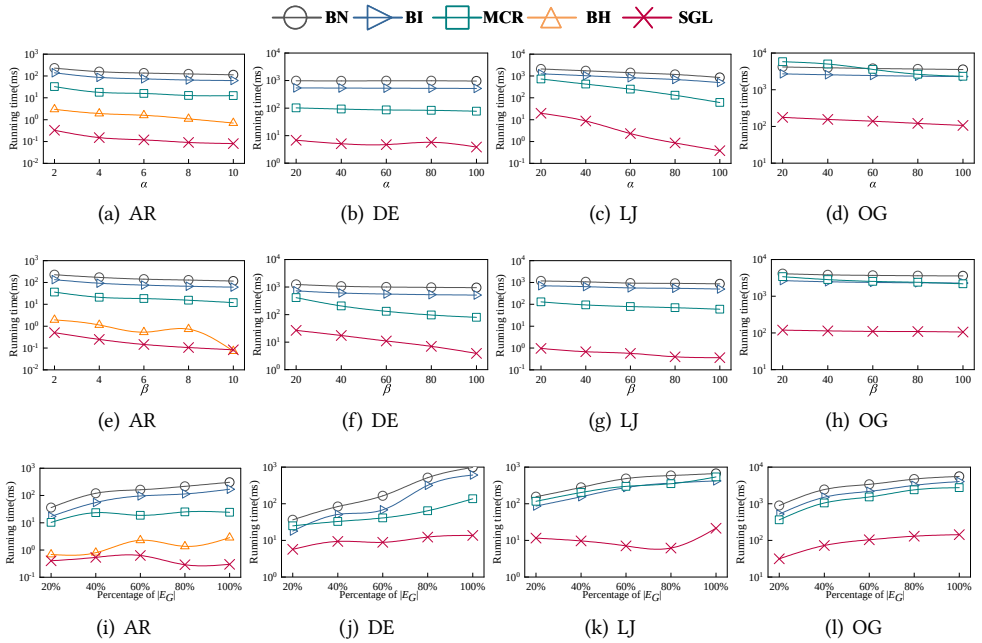


Fig. 12. Effect of α, β, and $|E|$ on running time for bi-community search

the running time of all algorithms increases accordingly, indicating the direct impact of edge quantity on performance. However, a slight decrease in running time can be observed in certain cases, as adding edges increases graph cohesiveness and causes SNodes to merge, thereby reducing the number of γ-groups that need to be scanned. Since scanning groups via pointer-based access incurs higher overhead than reading contiguous vertices, this reduction in group count can outweigh the cost of a larger result size $|H|$, leading to an overall decrease in query time. Notably, *SGL* consistently achieves significantly lower running time.

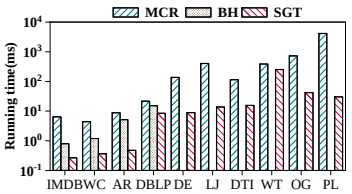
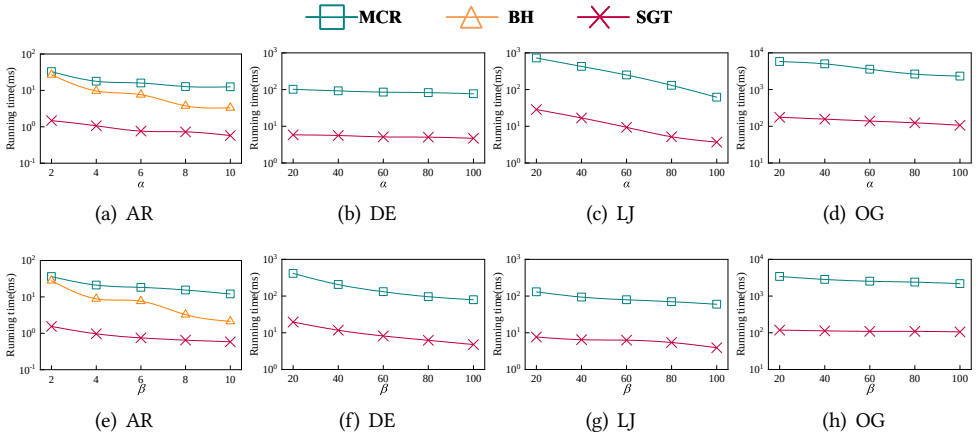


Fig. 13. Performance of indexes for bi-core search

7.4 Performance for Bi-Core Search

For the bi-core search problem, we first select *SNodes* that are not dominated by their neighbors to form an initial set F. Then, the nodes in F are mapped onto a two-dimensional coordinate space and organized using a spatial quadtree T. Based on the SG and quadtree T (denoted as *SGT*), we search for all (α, β)-components.

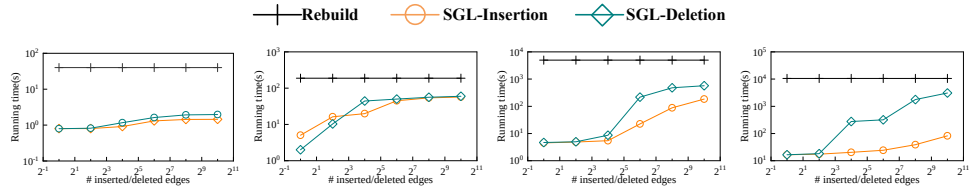
Exp-7: Performance of indexes for bi-core search over all datasets. Similar to Exp-3, we perform bi-core search on each dataset with fixed values of α and β. Figure 13 presents a comparison of query efficiency between *SGT* and the current state-of-the-art method *MCR* across multiple datasets. Overall, *SGT* consistently achieves the best query performance. For example, on the AR



(a) AR (b) DE (c) LJ (d) OG

(e) AR (f) DE (g) LJ (h) OG

Fig. 14. Effect of α and β on running time for bi-core search



(a) AR (b) DBLP (c) DE (d) WT

Fig. 15. Maintenance performance on four datasets

dataset, *SGL* completes queries in only 0.476 milliseconds, while *MCR* requires 8.784 milliseconds, achieving nearly an 18-fold speedup. On the PL dataset, *SGT* takes 30.27 ms, while *MCR* requires 4139.33 ms, achieving a 136× speedup. Therefore, *SGT* not only has an advantage in query efficiency but also demonstrates strong effectiveness and stability in processing large datasets.

Exp-8: Varying α for bi-core search. Figures 14(a)-14(d) shows the query performance of *SGT* on various datasets. For AR, we fix $\beta = 10$ and test $\alpha = 2, 4, 6, 8, 10$; for others, $\beta = 100$ and $\alpha = 20, 40, 60, 80, 100$. The results demonstrate that as α increases, the query time of all algorithms decreases, with *SGT* consistently achieving lower query times. This improvement is mainly because higher α values lead to smaller bi-components. Furthermore, *SGT* avoids redundant access to the same vertices, which significantly improves the efficiency of bi-core search.

Exp-9: Varying β for bi-core search. Figures 14(e)-14(h) illustrates the performance of *SGT* for different values of β. Similar to Exp-5, for the AR dataset, we fix $\alpha = 10$, while for the other datasets, $\alpha = 100$. We evaluate the query time across varying β. The results demonstrate that *SGT* consistently achieves better performance than *MCR* across all datasets. The reason is also attributed to the filtering capability of the *SGT*, which significantly reduces unnecessary traversal overhead during the query process.

7.5 Index Maintenance

To evaluate the efficiency of the proposed maintenance algorithm, we randomly perform 2^0 to 2^{10} edge insertions and deletions on the datasets and report the total execution time.

Exp-10: SGL maintenance. As illustrated in Figure 15, we compare the performance of the *Insert* and *Delete* operations against the baseline *Rebuild*. Since *Insert* and *Delete* update the index by inspecting only a subset of nodes, they consistently outperform full reconstruction on all datasets.

We also observe that *Delete* generally takes longer than *Insert*, especially on the DE and WT datasets, because *Delete* may cause node splitting, which requires more computation than node merging during *Insert*.

7.6 Performance for Other Model.

Many subgraph models involve two constraints, such as the (θ, k)-core [32] in undirected graphs, as well as the (k, l)-core [14] and the (k_c, k_f)-truss [22] in directed graphs. They exhibit nesting and overlapping properties similar to those of the (α, β)-core. Our method can be extended to these model-related community search problems; for example, we apply it to the (k_c, k_f)-truss search problem [1]. Specifically, we can derive cohesive units, (k_c, k_f, γ)-clusters, from the (k_c, k_f)-truss, and build an *SGL* index structure based on these units to support community search.

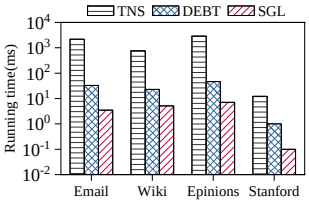
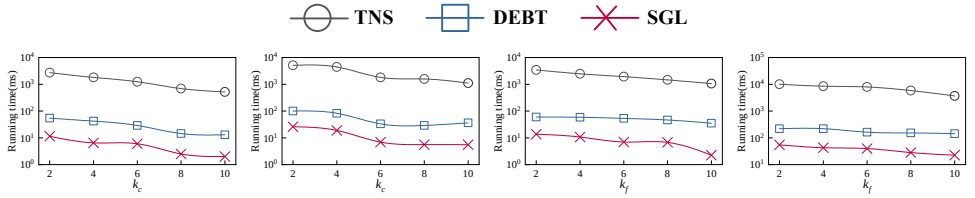


Fig. 16. Performance of indexes for D-truss search

Table 4. Statistics of used directed graphs

Graphs	Email	Wiki	Epinions	Stanford
Number of vertices/edges	265.2K/420K	10K/128.2K	75.9K/0.51M	281.9K/2.3M

Exp-11: Performance of indexes for (k_c, k_f)-truss search over directed graphs. We evaluate the performance of different indexing methods for the (k_c, k_f)-truss search task on four real-world directed graph datasets: Email, Wiki, Epinions, and Stanford[2]. The statistics of these datasets are summarized in Table 4. Figure 16 compares the query running times of three methods: trussness [22] (*TNS*) , EquiDtruss [1] (*DEBT*), and *SGL*, across all datasets. *TNS* is an indexing method similar to *BN*, which eliminates redundancy by leveraging the nested relationships among different (k_c, k_f)-trusses; *DEBT* further constructs a more compact index structure based on *TNS*, thereby reducing the query overhead. The experimental results demonstrate that *SGL* consistently achieves the best query performance. Its execution times are significantly lower than those of both *TNS* and *DEBT*, validating the good scalability of the proposed method for this search task.



(a) Email (b) Epinions (c) Email (d) Epinions

Fig. 17. Effect of k_c and k_f on running time for D-truss search

Exp-12: Varying k_c and k_f. Figures 17(a)-17(d) report the query performance when varying k_c with k_f fixed at 10 or varying k_f with k_c fixed at 10. On both Email and Epinions datasets, we test

[2]https://snap.stanford.edu/data/

$k_c, k_f = 2, 4, 6, 8, 10$. The results show that as k_c or k_f increases, the running time of all algorithms decreases. Among the three methods, *SGL* consistently achieves the lowest query time, followed by *DEBT* and then *TNS*, since higher (k_c, k_f) values correspond to denser subgraphs, reducing the number of candidate subgraphs to check and allowing SGL to retrieve results more efficiently.

8 Related Work

Community search on unipartite graphs. Community search on unipartite graphs is typically based on various cohesion models, such as k-core [4, 27, 47], k-truss [29, 40, 52], k-clique [49], k-ECC [7, 51], and the densest subgraph [26, 28, 34, 36, 44]. Based on the k-core model, [8] and [30] investigated online algorithms for k-core community search on unipartite graphs. Barbieri et al. [3] proposed a tree-like index structure for k-core community search. Fang et al. [10] further integrated vertex attributes to identify communities and considered the spatial locations of vertices in [11] and [38]. For truss-based community search, [15] and [2] explored the triangle-connected model. In [45], the authors studied the problem of densest clique percolation community search.

Community search on bipartite graphs. Ding et al. [9] proposed GraphRec, a graph search algorithm based on the (α, β)-core. Liu et al. [21] investigated the (α, β)-core decomposition problem in distributed environments and developed novel algorithms to support (α, β)-core decomposition in such settings. Zhang et al. [50] introduced a new community model called SABC (Size-bounded (α, β)-community). In [20], Boge Liu presented an efficient (α, β)-core decomposition algorithm leveraging a novel indexing structure. Wang et al. [41] proposed an index based on the bipartite graph hierarchy. Luo et al. [24] introduced the concept of bi-core number and analyzed the impact of edge insertions and deletions on its changes. In [23], a new index structure named MCR-Tree was introduced, designed for efficient multidimensional core searches, such as k-core, (α, β)-core, and (k, l)-core.

9 Conclusion

In this paper, we address the time-space trade-off inherent in existing indexing methods for bi-components search in large bipartite graphs by introducing a novel cohesive unit: the (α, β, γ)-*cluster*. This unit precisely characterizes vertex membership across non-nested bi-components through the predecessor value γ, enabling vertex-level deduplication. Building on this theoretical foundation, we propose the *SGL* (*Summary Graph + Location*) index. By decomposing bi-components into disjoint γ-groups, SGL organizes these atomic units into a compact summary graph of interconnected *SNodes* with precise *Location* mapping, achieving *zero-redundancy* in vertex retrieval. Then, we propose an effective algorithm for identifying bi-communities based on SGL. Next, we present efficient construction and maintenance algorithms for the SGL. Finally, extensive experiments demonstrate the effectiveness and efficiency of our approach, achieving performance improvements of up to two orders of magnitude.

Acknowledgments

The work is supported by the National Key R&D Program of China (No. 2024YFF0617702), the NSFC (Nos. U22A2025, 62232007, U23A20309, 62472293), the Key R&D Program of Liaoning (No. 2025JH2/101800116), the 111 Project (No. B16009), and the NSF of Liaoning (No. 2025-MS-125).

References

[1] Wei Ai, Canhao Xie, Tao Meng, Jayi Du, and Keqin Li. 2024. A D-Truss-Equivalence Based Index for Community Search Over Large Directed Graphs. *IEEE Transactions on Knowledge and Data Engineering* 36, 11 (2024), 5482–5494.

[2] Esra Akbas and Peixiang Zhao. 2017. Truss-based community search: a truss-equivalence based indexing approach. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1298–1309.

[3] Nicola Barbieri, Francesco Bonchi, Edoardo Galimberti, and Francesco Gullo. 2015. Efficient and effective community search. *Data mining and knowledge discovery* 29 (2015), 1406–1433.

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

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

[6] Monika Cerinsek and Vladimir Batagelj. 2015. Generalized two-mode cores. *Soc. Networks* 42 (2015), 80–87.

[7] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Xuemin Lin, Chengfei Liu, and Weifa Liang. 2013. Efficiently computing k-edge connected components via graph decomposition. In *Proceedings of the 2013 ACM SIGMOD international conference on management of data*. 205–216.

[8] Wanyun Cui, Yanghua Xiao, Haixun Wang, and Wei Wang. 2014. Local search of communities in large graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 991–1002.

[9] Danhao Ding, Hui Li, Zhipeng Huang, and Nikos Mamoulis. 2017. Efficient fault-tolerant group recommendation using alpha-beta-core. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. 2047–2050.

[10] Yixiang Fang, CK Cheng, Siqiang Luo, and Jiafeng Hu. 2016. Effective community search for large attributed graphs. *Proceedings of the VLDB Endowment* (2016).

[11] Yixiang Fang, CK Cheng, S Luo, J Hu, and X Li. 2017. Effective community search over large spatial graphs. *Proceedings of the VLDB Endowment (PVLDB)* (2017).

[12] Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. 2020. A survey of community search over big graphs. *VLDB J.* 29, 1 (2020), 353–392.

[13] Mike Gartrell, Xinyu Xing, Qin Lv, Aaron Beach, Richard Han, Shivakant Mishra, and Karim Seada. 2010. Enhancing group recommendation by incorporating social relationship interactions. In *Proceedings of the 2010 ACM International Conference on Supporting Group Work*. 97–106.

[14] Christos Giatsidis, Dimitrios M Thilikos, and Michalis Vazirgiannis. 2013. D-cores: measuring collaboration of directed graphs based on degeneracy. *Knowledge and information systems* 35, 2 (2013), 311–343.

[15] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1311–1322.

[16] Michael Ley. 2002. The DBLP computer science bibliography: Evolution, research issues, perspectives. In *International symposium on string processing and information retrieval*. Springer, 1–10.

[17] Shunyang Li, Kai Wang, Xuemin Lin, Wenjie Zhang, Yizhang He, and Long Yuan. 2024. Querying Historical Cohesive Subgraphs over Temporal Bipartite Graphs. In *IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 2503–2516.

[18] Greg Linden, Brent Smith, and Jeremy York. 2003. Amazon. com recommendations: Item-to-item collaborative filtering. *IEEE Internet computing* 7, 1 (2003), 76–80.

[19] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. 2019. Efficient (α, β)-core computation: An index-based approach. In *The World Wide Web Conference*. 1130–1141.

[20] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. 2020. Efficient (α, β)-core computation in bipartite graphs. *The VLDB Journal* 29, 5 (2020), 1075–1099.

[21] Qing Liu, Xuankun Liao, Xin Huang, Jianliang Xu, and Yunjun Gao. 2023. Distributed (α, β)-core decomposition over bipartite graphs. In *IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 909–921.

[22] Qing Liu, Minjun Zhao, Xin Huang, Jianliang Xu, and Yunjun Gao. 2020. Truss-based community search over large directed graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2183–2197.

[23] Chengyang Luo, Yifan Zhu, Qing Liu, Yunjun Gao, Lu Chen, and Jianliang Xu. 2024. MCR-Tree: An Efficient Index for Multi-dimensional Core Search. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–25.

[24] Wensheng Luo, Qiaoyuan Yang, Yixiang Fang, and Xu Zhou. 2023. Efficient Core Maintenance in Large Bipartite Graphs. *Proceedings of the ACM on Management of Data* 1, 3 (2023), 1–26.

[25] Bingqing Lyu, Lu Qin, Xuemin Lin, Ying Zhang, Zhengping Qian, and Jingren Zhou. 2020. Maximum biclique search at billion scale. *Proceedings of the VLDB Endowment* (2020).

[26] Chenhao Ma, Reynold Cheng, Laks VS Lakshmanan, and Xiaolin Han. 2022. Finding locally densest subgraphs: a convex programming approach. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2719–2732.

[27] David W Matula and Leland L Beck. 1983. Smallest-last ordering and clustering and graph coloring algorithms. *Journal of the ACM (JACM)* 30, 3 (1983), 417–427.

[28] Lu Qin, Rong-Hua Li, Lijun Chang, and Chengqi Zhang. 2015. Locally densest subgraph discovery. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 965–974.

[29] Ahmet Erdem Sarıyüce and Ali Pinar. 2018. Peeling bipartite networks for dense subgraph discovery. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. 504–512.

[30] Mauro Sozio and Aristides Gionis. 2010. The community-search problem and how to plan a successful cocktail party. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. 939–948.
[31] Amos Tanay, Roded Sharan, Martin Kupiec, and Ron Shamir. 2004. Revealing modularity and organization in the yeast molecular network by integrated analysis of highly heterogeneous genomewide data. *Proceedings of the National Academy of Sciences* 101, 9 (2004), 2981–2986.
[32] Yifu Tang, Jianxin Li, Nur Al Hasan Haldar, Ziyu Guan, Jiajie Xu, and Chengfei Liu. 2022. Reliable Community Search in Dynamic Networks. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2826–2838.
[33] Robert E Tarjan and Jan Van Leeuwen. 1984. Worst-case analysis of set union algorithms. *Journal of the ACM (JACM)* 31, 2 (1984), 245–281.
[34] Nikolaj Tatti and Aristides Gionis. 2015. Density-friendly graph decomposition. In *Proceedings of the 24th International Conference on World Wide Web*. 1089–1099.
[35] Anxin Tian, Alexander Zhou, Yue Wang, Xun Jian, and Lei Chen. 2024. Efficient Index for Temporal Core Queries over Bipartite Graphs. *Proceedings of the VLDB Endowment* 17, 11 (2024), 2813–2825.
[36] Tran Ba Trung, Lijun Chang, Nguyen Tien Long, Kai Yao, and Huynh Thi Thanh Binh. 2023. Verification-free approaches to efficient locally densest subgraph discovery. In *IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 1–13.
[37] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. 2018. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 839–848.
[38] Kai Wang, Xin Cao, Xuemin Lin, Wenjie Zhang, and Lu Qin. 2018. Efficient computing of radius-bounded k-cores. In *IEEE 34th international conference on data engineering (ICDE)*. IEEE, 233–244.
[39] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2019. Vertex Priority Based Butterfly Counting for Large-scale Bipartite Networks. *PVLDB* (2019).
[40] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2020. Efficient bitruss decomposition for large-scale bipartite graphs. In *IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 661–672.
[41] Kai Wang, Wenjie Zhang, Xuemin Lin, Ying Zhang, and Shunyang Li. 2022. Discovering hierarchy of bipartite graphs with cohesive subgraphs. In *IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2291–2305.
[42] Kai Wang, Wenjie Zhang, Xuemin Lin, Ying Zhang, Lu Qin, and Yuting Zhang. 2021. Efficient and effective community search on large-scale bipartite graphs. In *IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 85–96.
[43] Kai Wang, Wenjie Zhang, Ying Zhang, Lu Qin, and Yuting Zhang. 2021. Discovering significant communities on bipartite graphs: An index-based approach. *IEEE Transactions on Knowledge and Data Engineering* 35, 3 (2021), 2471–2485.
[44] Yichen Xu, Chenhao Ma, Yixiang Fang, and Zhifeng Bao. 2023. Efficient and effective algorithms for generalized densest subgraph discovery. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.
[45] L. Yuan, L. Qin, W. Zhang, and et al. 2017. Index-based densest clique percolation community search in networks. *IEEE Transactions on Knowledge and Data Engineering* 30, 5 (2017), 922–935.
[46] Quan Yuan, Gao Cong, and Chin-Yew Lin. 2014. COM: a generative model for group recommendation. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 163–172.
[47] Fan Zhang, Qingyuan Linghu, Jiadong Xie, Kai Wang, Xuemin Lin, and Wenjie Zhang. 2023. Quantifying Node Importance over Network Structural Stability. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 3217–3228.
[48] Yalong Zhang, Rong-Hua Li, Qi Zhang, Hongchao Qin, Lu Qin, and Guoren Wang. 2025. Density Decomposition of Bipartite Graphs. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–25.
[49] Yun Zhang, Charles A Phillips, Gary L Rogers, Erich J Baker, Elissa J Chesler, and Michael A Langston. 2014. On finding bicliques in bipartite graphs: a novel algorithm and its application to the integration of diverse biological data types. *BMC bioinformatics* 15 (2014), 1–18.
[50] Yuting Zhang, Kai Wang, Wenjie Zhang, Wei Ni, and Xuemin Lin. 2024. Size-bounded Community Search over Large Bipartite Graphs.. In *EDBT*. 320–331.
[51] Rui Zhou, Chengfei Liu, Jeffrey Xu Yu, Weifa Liang, Baichen Chen, and Jianxin Li. 2012. Finding maximal k-edge-connected subgraphs from a large graph. In *Proceedings of the 15th international conference on extending database technology*. 480–491.
[52] Zhaonian Zou. 2016. Bitruss decomposition of bipartite graphs. In *International conference on database systems for advanced applications*. Springer, 218–233.

Received October 2025; revised January 2026; accepted February 2026