Optimal (α, β)-Dense Subgraph Search in Static and Dynamic Bipartite Graphs: an Index-Based Approach

Yalong Zhang
Beijing Institute of Technology
Beijing, China
yalong-zhang@qq.com

Rong-Hua Li
Beijing Institute of Technology
Beijing, China
lironghuabit@126.com

Qi Zhang
University of Science and Technology Beijing
Beijing, China
qizhangcs@ustb.edu.cn

Guoren Wang
Beijing Institute of Technology
Beijing, China
wanggrbit@126.com

ABSTRACT

Dense subgraph search in bipartite graphs is a fundamental problem in graph analysis, with wide-ranging applications in fraud detection, recommendation systems, and social network analysis. The recently proposed (α, β)-dense subgraph model has demonstrated superior capability in capturing the intrinsic density structure of bipartite graphs compared to existing alternatives. However, despite its modeling advantages, the (α, β)-dense subgraph model lacks efficient support for query processing and dynamic updates, limiting its practical utility in large-scale applications. To address these limitations, we propose BD-Index, a novel index that answers (α, β)-dense subgraph queries in optimal time while using only linear space $O(|E|)$, making it well-suited for real-world applications requiring both fast query processing and low memory consumption. We further develop two complementary maintenance strategies for dynamic bipartite graphs to support efficient updates to the BD-Index. The space-efficient strategy updates the index in time complexity of $O(p \cdot |E|^{1.5})$ per edge insertion or deletion, while maintaining a low space cost of $O(|E|)$ (the same as the index itself), where p is typically a small constant in real-world graphs. In contrast, the time-efficient strategy significantly reduces the update time to $O(p \cdot |E|)$ per edge update by maintaining auxiliary orientation structures, at the cost of increased memory usage up to $O(p \cdot |E|)$. These two strategies provide flexible trade-offs between maintenance efficiency and memory usage, enabling BD-Index to adapt to diverse application requirements. Extensive experiments on 10 large-scale real-world datasets demonstrate high efficiency and scalability of our proposed solutions.

1 INTRODUCTION

Bipartite graphs are widely used to represent relationships between two types of entities, such as user-page social networks [5], customer-product networks [12, 39], collaboration networks [24], and gene co-expression networks [11, 20]. Mining dense subgraphs in bipartite graphs has numerous practical applications. For example, in customer-product networks, users within the same dense community typically share similar product preferences, enabling e-commerce platforms to make targeted recommendations. In social networks, fraudsters may hire a large number of bot accounts to boost their visibility through likes or interactions. Such behavior tends to form dense communities around the fraudsters, making it easier for platforms to detect these fraudulent activities.

Motivated by these applications, various cohesive subgraph models have been proposed to identify dense communities in bipartite graphs, including biclique [10, 29, 42, 44], k-biplex [13, 45, 46], k-bitruss [40, 41, 51], (α, β)-core [16, 26, 28], and (α, β)-dense subgraph [50]. However, despite the significant progress made by these models in capturing dense communities, they still suffer from several critical limitations. For example, there are no known polynomial-time algorithms for computing biclique and k-biplex, limiting their applicability in large bipartite networks. The computation of k-bitruss heavily relies on enumerating butterfly structures, which becomes prohibitively expensive in dense graphs [40, 41, 51]. The (α, β)-core model only considers node degrees and often fails to accurately capture the underlying density structure of the graph. In contrast, the (α, β)-dense subgraph is a density-based model that exhibits a desirable "dense-inside, sparse-outside" property (Theorem 1) and has been shown in [50] to capture the density structure of bipartite graphs more effectively than other cohesive subgraph models such as (α, β)-core, bitruss, biplex, and biclique.

Given the favorable properties of the (α, β)-dense subgraph model, it can be applied to a wide range of real-world scenarios where (α, β)-dense subgraph queries are frequently issued. For example: (1) in user–product bipartite networks for recommendation systems, platforms may process millions of recommendation requests per second [25, 47]; by tuning (α, β) values, the system can retrieve product groups with different densities, enabling personalized and diverse recommendations; (2) in fraud detection over financial networks, malicious behaviors may hide in subgraphs with varying density levels, requiring the system to frequently query dense subgraphs with different (α, β) combinations [1, 26]; (3) in social networks, different (α, β) combinations reveal communities with distinct interaction patterns, such as a user–blog bipartite graph where high α and low β (i.e., stronger filtering on the user side) identify highly active users, while low α and high β (i.e., stronger filtering on the blog side) highlight popular blogs [50]. Moreover, these bipartite networks are typically dynamic, with frequent edge insertions and deletions as users interact with items or each other in real time. This makes it essential to support efficient and responsive (α, β)-dense subgraph computation under continuous graph updates.

However, existing state-of-the-art algorithm DSS++ [50] for computing (α, β)-dense subgraphs struggles to meet these demands. The DSS++ algorithm adopts a network-flow-based approach that requires solving a maximum flow problem per query, resulting in a worst-case time complexity of $O(|E|^{1.5})$, where $|E|$ is the number of edges in the bipartite graph. As real-world graphs continue to grow rapidly in size and (α, β)-dense subgraph queries are issued at high frequency, this online approach becomes increasingly inefficient. For instance, as shown in our experiments (Section 6, Exp-1), on the large LI dataset (112.3 million edges), the DSS++ algorithm takes an average of 21.49 seconds to process a single query—far exceeding the latency requirements of many real-time applications, which typically demand a response time below 0.5 seconds [21, 25, 47]. Moreover, real-world bipartite graphs are typically dynamic, with frequent updates such as edge insertions and deletions. Efficient computation of (α, β)-dense subgraphs in such evolving graphs is crucial for numerous applications, including identifying closely-connected communities in social networks over time, generating real-time product recommendations in customer-product networks, and dynamically monitoring dense regions potentially indicative of fraudulent activities in financial networks. Unfortunately, existing methods for handling (α, β)-dense subgraph queries in dynamic graphs remain limited to inefficiently re-executing the DSS++ algorithm whenever the graph is updated.

To address these limitations, we study efficient computation of (α, β)-dense subgraphs on both static and dynamic bipartite graphs. We first propose BD-Index, a novel index structure specifically designed to support the optimal query processing (i.e., the query time complexity is linear to the size of results). Meanwhile, the space complexity of BD-Index is carefully bounded to $O(|E|)$, ensuring scalability to large-scale graphs. Thus, BD-Index serves as a practical solution for real-time and large-scale dense subgraph computation. For dynamic graphs, we investigate the maintence of BD-Index and propose two complementary strategies. The first is a space-efficient approach that preserves the linear space complexity advantage of BD-Index. The second is a time-efficient approach that leverages additional storage for faster updates. Notably, the latter provides a practical trade-off between time and space complexity, enabling efficient maintenance of BD-Index even for graphs with hundreds of millions of edges. In summary, the main contributions of this paper are as follows.

A novel index structure with optimal query time. Our design exploits the hierarchical property of (α, β)-dense subgraphs, i.e., higher-density subgraphs are necessarily contained within lower-density ones. Leveraging this property, we introduce two novel concepts: α-rank and β-rank, which represent the largest (α, β) values for which the node belongs to a dense subgraph. Using these ranks, we elaborately organize nodes into p node lists, where p is the largest integer such that (p, p)-dense subgraph is non-empty. For each (α, β) pair, we maintain a pointer to the corresponding node list, allowing the query to retrieve all result nodes by scanning the list once. This enables an optimal query time of $O(|D_{\alpha,\beta}|)$, where $D_{\alpha,\beta}$ is the result set. Furthermore, by fully exploiting the nested nature of dense subgraphs, our index requires only linear space $O(|E|)$, making it both query-efficient and space-efficient. We also present an index construction algorithm with a time complexity of $O(p \cdot |E|^{1.5} \cdot \log |U \cup V|)$, which can effectively handle graphs with hundreds of millions of edges.

Novel space-efficient index maintenance algorithms. To support efficient updates of the BD-Index, we first establish several update theorems for (α, β)-dense subgraphs, covering both insertion and deletion cases. These theorems reveal that when an edge is updated, the α-rank of nodes in the lower side V and the β-rank of nodes in the upper side $|U|$ may change by at most 1. Leveraging these update properties, we propose two space-efficient maintenance algorithms: BD-Insert-S and BD-Delete-S, which handle edge insertions and deletions, respectively. Both algorithms operate in linear space $O(|E|)$, preserving the space advantage of BD-Index. In terms of time complexity, our BD-Insert-S and BD-Delete-S algorithms achieve an update complexity of $O(p \cdot |E|^{1.5})$, which is significantly lower than the $O(p \cdot |E|^{1.5} \cdot \log |U \cup V|)$ complexity of the baseline that recomputes the entire index from scratch.

Novel time-efficient index maintenance algorithms. Building on the established update theorems, we further propose time-efficient maintenance algorithms. We introduce a novel concept called *egalitarian orientation*, transforming the maintenance of BD-Index into maintaining a set of egalitarian orientations. Leveraging this transformation, we propose two algorithms, BD-Insert-T and BD-Delete-T, that first maintain the egalitarian orientations and subsequently update BD-Index using only a few breadth-first search operations. These algorithms achieve a significant reduction in time complexity from $O(p \cdot |E|^{1.5})$ required by the space-efficient algorithms to $O(p \cdot |E|)$. While maintaining p additional egalitarian orientations increases the space complexity to $O(p \cdot |E|)$, this remains within acceptable limits. Consequently, BD-Insert-T and BD-Delete-T achieve a favorable time-space trade-off, enabling efficient and scalable maintenance of BD-Index.

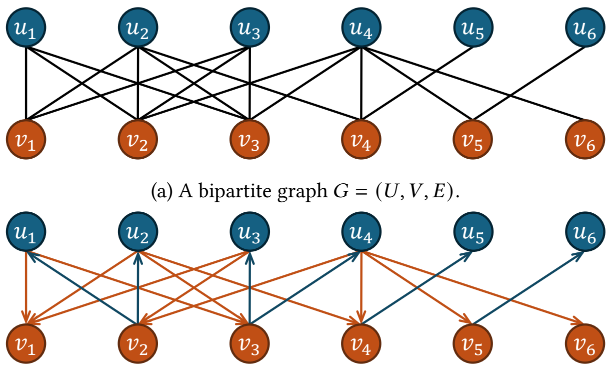
Extensive experiments. We conduct extensive experiments on 10 large real-world datasets, and the results demonstrate the efficiency and scalability of our solutions. First, the index-based query algorithm, Query-BD-Index, outperforms the state-of-the-art online algorithm by 3 to 4 orders of magnitude. On the largest dataset LI (with over 100 million edges), it achieves an average query time of merely 2.74 milliseconds. Second, BD-Index exhibits memory usage comparable to the original graph size, and our index construction algorithm scales effectively to large graphs like LI. Third, our space-efficient maintenance algorithms achieve up to one order of magnitude speedup compared to index recomputation from scratch, while the time-efficient algorithms are 2-4 orders of magnitude faster than the space-efficient approaches. Although requiring approximately one order of magnitude more memory, the time-efficient maintenance algorithms only consumes 51 GB for maintaining BD-Index on the LI dataset, which is easily acceptable in practical applications.

Reproducibility and full version paper. The source code and the full version of this paper can be found at https://github.com/Yalong-Zhang/bd-index.

2 PRELIMINARIES

We consider an undirected and unweighted bipartite graph $G = (U, V, E)$, where U and V are two disjoint node sets, and $E \subseteq U \times V$ represents the set of edges connecting nodes from distinct sets. For each node $x \in U \cup V$, we denote its neighbor nodes set in G as $N_G(x)$, and its degree as $d_x(G) = |N_G(x)|$ (or simply $N(x)$ and d_x when the context is unambiguous). Given a subset of nodes $X \subseteq U \cup V$, we define subsets $X^U = X \cap U$ and $X^V = X \cap V$ corresponding to the respective partitions. The subgraph induced by X is $G(X) = (X^U, X^V, E(X))$, where $E(X)$ includes all edges between nodes in X.

By assigning each edge a specific direction, we can convert the undirected bipartite graph $G = (U, V, E)$ into a directed bipartite graph called an *orientation* of G, denoted $\vec{G} = (U, V, \vec{E})$, where $\vec{E}$ contains the directed edges. In this orientation, the indegree of node x, denoted $\vec{d}_x(\vec{G})$ (or $\vec{d}_x$ for simplicity), counts its incoming edges. A *path* $s \rightsquigarrow t$ in an orientation is a sequence of nodes $x_0, x_1, \ldots, x_l$, where $(x_{i-1}, x_i) \in \vec{E}$ for $i = 1, \ldots, l$, and the length of this path is l. If a path $x \rightsquigarrow y$ exists, we say x can *reach* y. Next, we introduce the definition of the (α, β)-dense subgraph [50].



(a) A bipartite graph $G = (U, V, E)$.

(b) An orientation $\vec{G} = (U, V, \vec{E})$.

Figure 1: An example graph $G = (U, V, E)$ and its orientation.

Definition 1. *[50]* **((α, β)-dense subgraph)** *Given a bipartite graph $G = (U, V, E)$, two non-negative integers α and β, let $\vec{G}$ be an orientation of G, and let $S = \{u \in U | \vec{d}_u < \alpha\} \cup \{v \in V | \vec{d}_v < \beta\}$ and $T = \{u \in U | \vec{d}_u > \alpha\} \cup \{v \in V | \vec{d}_v > \beta\}$. If there is no path $s \rightsquigarrow t$ in $\vec{G}$ with $s \in S$ and $t \in T$, then the (α, β)-dense subgraph $G(D_{\alpha,\beta})$ is induced by $D_{\alpha,\beta} = T \cup \{x | x$ can reach a node in T in $\vec{G}\}$.*

Theorem 1. *[50] The (α, β)-dense subgraph $D_{\alpha,\beta}$ has the following two properties: (1)* ***(inside dense)*** *for any non-empty $X \subseteq D_{\alpha,\beta}$, the removal of X from $D_{\alpha,\beta}$ results in a deletion of more than $\alpha \cdot |X^U| + \beta \cdot |X^V|$ edges; (2)* ***(outside sparse)*** *for any subset $Y \subseteq (U \cup V) \setminus D_{\alpha,\beta}$, the inclusion of Y into $D_{\alpha,\beta}$ leads to an increase of at most $\alpha \cdot |Y^U| + \beta \cdot |Y^V|$ edges.*

The (α, β)-dense subgraph is characterized as being **dense inside** and **sparse outside** [50]. Intuitively, the parameters α and β serve as thresholds that control the density of the two partitions: increasing α directly raises the density requirement for the upper partition U, and increasing β does the same for the lower partition V. As a result, each (α, β) pair corresponds to a subgraph $D_{\alpha,\beta}$ with a distinct density, enabling users to flexibly adjust the density of the resulting subgraph to their specific needs and separately control the density requirements for the two partitions.

Example 1. *As illustrated in Figure 1b, $\vec{G}$ is an orientation of the bipartite G depicted in Figure 1a. Let $\alpha = 1$ and $\beta = 2$. According to Definition 1, we have $S = \{v_5, v_6\}$ and $T = \{v_1\}$. Since no path exists from any node in S to any node in T, we can conclude that $D_{1,2}$ contains T and all nodes that can reach T, specifically $\{u_1, u_2, u_3, u_4, v_1, v_2, v_3\}$. Using the same approach, we have $D_{1,1} = \{u_1, u_2, u_3, u_4, v_1, v_2, v_3, v_4\}$ and $D_{1,3} = \emptyset$. Compared with $D_{1,1}$, $D_{1,2}$ excludes the relatively sparse node v_4 due to the higher β value, resulting in a denser subgraph. Further increasing β to 3 yields no subgraph satisfying the density requirement, thus, $D_{1,3} = \emptyset$.*

In practical applications (such as e-commerce recommendation and fraud detection), it is often necessary to frequently query $D_{\alpha,\beta}$ for different combinations of (α, β) parameters. This necessitates the design of efficient query processing algorithms. Moreover, real-world bipartite graphs are typically dynamic, with frequent edge insertions and deletions. In such scenarios, efficiently computing $D_{\alpha,\beta}$ while keeping up with graph updates is essential for real-time responsiveness. Motivated by these, we formulate the problem studied in this paper as follows.

Problem definition: Given a bipartite graph G, we define a $D_{\alpha,\beta}$-*query* as the computation of $D_{\alpha,\beta}$ for given two non-negative integers α and β. Our problem focuses on efficiently processing $D_{\alpha,\beta}$-queries in both static and dynamic graphs.

3 A NOVEL INDEX: BD-INDEX

In this section, we propose a novel index called BD-Index (short for **B**ipartite **D**ense Subgraph Index), which achieves optimal query processing time of $O(|D_{\alpha,\beta}|)$. Furthermore, BD-Index requires only $O(|E|)$ space, linear to the graph size, making it both time-efficient and space-efficient for query processing.

3.1 Structure of BD-Index

As discussed previously, we can compute all non-empty $D_{\alpha,\beta}$ subgraphs by executing the DSS++ algorithm [50] and storing each $D_{\alpha,\beta}$ individually to construct a basic index structure. Although this naive approach achieves optimal query processing time $O(|D_{\alpha,\beta}|)$, its space complexity becomes impractical for large-scale graphs. Specifically, each $D_{\alpha,\beta}$ requires $O(|V|)$ space, and there exist $O(|V|^2)$ distinct (α, β) pairs that produce non-empty $D_{\alpha,\beta}$ subgraphs [50], leading to an overall storage requirement of $O(|V|^3)$. To address this space limitation, we exploit the hierarchical property of $D_{\alpha,\beta}$ subgraphs, as formalized in the following theorem.

Theorem 2. *[50]* **(Hierarchical property)** *Given a graph G, for $\alpha^+ \geq \alpha$ and $\beta^+ \geq \beta$, we have $D_{\alpha^+,\beta^+} \subseteq D_{\alpha,\beta}$.*

According to the hierarchical property theorem, if a node x belongs to D_{α^+,β^+}, it must necessarily belong to $D_{\alpha,\beta}$. In the straightforward index, x would be redundantly stored in both D_{α^+,β^+} and $D_{\alpha,\beta}$. To optimize storage, we can store x only in D_{α^+,β^+}. When processing queries for $D_{\alpha,\beta}$, we can directly incorporate D_{α^+,β^+} into the result set, thereby eliminating storage redundancy while maintaining query correctness. Based on the above rationale, we define α-rank and β-rank.

Definition 2. **(α-rank and β-rank)** *Given a graph G and a value α, the α-rank of a node $x \in (U \cup V)$, denoted by $r_\alpha(x)$, is the maximum integer k such that $x \in D_{\alpha,k}$ if $x \in D_{\alpha,0}$, and -1 otherwise. Similarly, for a value β, the β-rank of x, denoted by $r_\beta(x)$, is the maximum integer k such that $x \in D_{k,\beta}$ if $x \in D_{0,\beta}$, and -1 otherwise.*

With α-rank and β-rank defined, $D_{\alpha,\beta}$ can be computed using the following theorem.

Theorem 3. *Given a graph G and two non-negative integers α and β, we have: $D_{\alpha,\beta} = \{x | r_\alpha(x) \geq \beta\} = \{x | r_\beta(x) \geq \alpha\}$.*

Proof. By definition, the α-rank $r_\alpha(x)$ implies $x \in D_{\alpha,r_\alpha(x)}$ and $x \notin D_{\alpha,r_\alpha(x)+1}$. Combining this with Theorem 2, if $r_\alpha(x) \geq \beta$, then $x \in D_{\alpha,\beta}$. Conversely, if $r_\alpha(x) < \beta$, it follows that $x \notin D_{\alpha,\beta}$. This establishes the equivalence $r_\alpha(x) \geq \beta \Leftrightarrow x \in D_{\alpha,\beta}$, which yields $D_{\alpha,\beta} = \{x | r_\alpha(x) \geq \beta\}$. Through symmetric reasoning, we further derive $D_{\alpha,\beta} = \{x | r_\beta(x) \geq \alpha\}$. □

According to Theorem 3, for a fixed α, we can construct a node list by sorting all nodes in ascending order of their r_α. Then, given arbitrary β, $D_{\alpha,\beta}$ consists of all nodes from the first node in the

Algorithm 1: Query-BD-Index($\mathbb{I}_{BD}, \alpha, \beta$)

Input: The BD-Index $\mathbb{I}_{BD}$, and the query integers α and β.
Output: $D_{\alpha,\beta}$.

```
1 if α ≤ β then
2     if α ≥ I^U_BD.size or β ≥ I^U_BD[α].size then D_{α,β} ← ∅ ;
3     else D_{α,β} ← nodes from I^U_BD[α][β] to the end of the node list ;
4 else
5     if β ≥ I^V_BD.size or α ≥ I^V_BD[β].size then D_{α,β} ← ∅ ;
6     else D_{α,β} ← nodes from I^V_BD[β][α] to the end of the node list ;
7 return D_{α,β};
```

list with $r_\alpha \geq \beta$ to the end of the node list. Symmetrically, when fixing β and sorting by r_β, we obtain an analogous property for $D_{\alpha,\beta}$. Based on this idea, we propose BD-Index as follows.

Definition 3. **(BD-Index)** *Let p be the maximum integer such that $D_{p,p} \neq \emptyset$. The* BD-Index *comprises two symmetric components:*

(1) $\mathbb{I}^U_{BD}$: For each $\alpha = 0, \ldots, p$, the index stores a node list that contains all nodes with $r_\alpha \geq \alpha$, sorted in ascending order of r_α. For each $\beta = \alpha, \ldots, \max_{x\in(U\cup V)} r_\alpha(x)$, $\mathbb{I}^U_{BD}[\alpha][\beta]$ points to the first node in the node list with $r_\alpha \geq \beta$.

(2) $\mathbb{I}^V_{BD}$: for each $\beta = 0, \ldots, p$, the index stores a node list that contains all nodes with $r_\beta > \beta$, sorted in ascending order of r_β. For each $\alpha = \beta+1, \ldots, \max_{x\in(U\cup V)} r_\beta(x)$, $\mathbb{I}^V_{BD}[\beta][\alpha]$ points to the first node in the node list with $r_\beta \geq \alpha$.

Example 2. *The* BD-Index *of the graph G in Figure 1a is shown in Figure 2. In this* BD-Index, *we have $p = 1$, resulting in both $\mathbb{I}^U_{BD}$ and $\mathbb{I}^V_{BD}$ containing $(p+1) = 2$ node lists. Taking the case of $\alpha = 1$ as an example, we have the following rank values: $r_\alpha(u_5) = r_\alpha(u_6) = -1$, $r_\alpha(v_5) = r_\alpha(v_6) = 0$, $r_\alpha(v_4) = 1$, $r_\alpha(u_1) = r_\alpha(u_2) = r_\alpha(u_3) = r_\alpha(u_4) = r_\alpha(v_1) = r_\alpha(v_2) = r_\alpha(v_3) = 2$. The node list for $\alpha = 1$ includes all nodes with $r_\alpha \geq \alpha$, as illustrated in Figure 2a. Since $\max_{x\in(U\cup V)} r_\alpha(x) = 2$, $\mathbb{I}^U_{BD}[\alpha]$ contains two pointers to the node list: $\mathbb{I}^U_{BD}[\alpha][1]$ and $\mathbb{I}^U_{BD}[\alpha][2]$, which point to v_4 and u_1, respectively.*

3.2 Query processing

Here, we present how BD-Index supports optimal-time query processing. The query processing algorithm, Query-BD-Index, is detailed in Algorithm 1. The index component $\mathbb{I}^U_{BD}$ processes queries where $\alpha \leq \beta$ (lines 1-3), while $\mathbb{I}^V_{BD}$ handles queries where $\alpha > \beta$ (lines 4-6). For $\alpha \leq \beta$, the algorithm first checks whether the queried α and β fall within the valid range to determine if $D_{\alpha,\beta}$ is empty (line 2). If $D_{\alpha,\beta}$ is non-empty, the algorithm retrieves all nodes from $\mathbb{I}^U_{BD}[\alpha][\beta]$ to the end of the node list and adds them into $D_{\alpha,\beta}$ (line 3). For $\alpha > \beta$, the algorithm computes $D_{\alpha,\beta}$ analogously (lines 5-6).

Example 3. *Based on* BD-Index *illustrated in Figure 2, the* Query-BD-Index *processes the $D_{0,2}$ query as follows. Since $\alpha = 0 \leq \beta = 2$, the algorithm employs the index component $\mathbb{I}^U_{BD}$. First, it verifies that $\alpha < \mathbb{I}^U_{BD}.size = 2$ and $\beta < \mathbb{I}^U_{BD}[\alpha].size = 4$, confirming that $D_{\alpha,\beta} \neq \emptyset$. The algorithm then retrieves all nodes from $\mathbb{I}^U_{BD}[\alpha][\beta] = u_5$ to the end of the node list, which includes $\{u_5, v_4, v_1, u_1, v_2, v_3, u_4, u_3, u_2\}$, as the query result $D_{\alpha,\beta}$.*

Next, we prove the correctness of Query-BD-Index and establish its optimal query time complexity.

Theorem 4. Query-BD-Index *can correctly retrieve $D_{\alpha,\beta}$ within optimal query time $O(|D_{\alpha,\beta}|)$.*

Proof. When $\alpha \leq \beta$, according to the definition of BD-Index, the nodes from $\mathbb{I}^U_{BD}[\alpha][\beta]$ to the end of the node list include all nodes with $r_\alpha \geq \beta$. By Theorem 3, $D_{\alpha,\beta}$ can be correctly retrieved. Using a similar proof technique, we can establish the correctness for the case when $\alpha > \beta$.

Regarding the query complexity, lines 1-2 and lines 4-5 of the algorithm execute in constant time. In lines 3 and 6, the algorithm perform a traversal of $D_{\alpha,\beta}$, requiring $O(|D_{\alpha,\beta}|)$ time. Therefore, the overall query complexity is $O(|D_{\alpha,\beta}|)$. □

The optimal query time $O(|D_{\alpha,\beta}|)$ of BD-Index guarantees efficient processing with minimal overhead, enabling result retrieval independent of the graph size.

3.3 Space complexity of BD-Index

In this subsection, we analyze the space complexity of BD-Index. We begin by introducing the following lemma.

Lemma 1. *Given a bipartite graph G and $D_{\alpha,\beta}$, for any node $u \in D^U_{\alpha,\beta}$, we have $d_u(G) > \alpha$, and for any node $v \in D^V_{\alpha,\beta}$, we have $d_v(G) > \beta$.*

The correctness of this lemma can be directly derived by definition. Below, we prove that BD-Index achieves $O(|E|)$ space complexity.

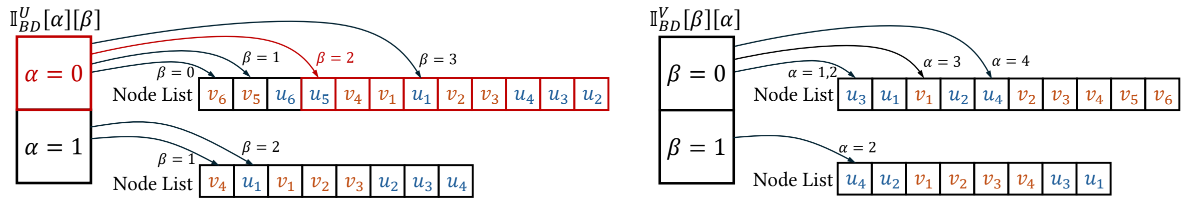
Theorem 5. *Given a graph G, the space complexity of its* BD-Index *is $O(|E|)$.*

Proof. We first prove that the space complexity of $\mathbb{I}^U_{BD}$ is $O(|E|)$. To begin, we show that the $(p+1)$ node lists in $\mathbb{I}^U_{BD}$ occupy $O(|E|)$ space. For a fixed α, by definition, the corresponding node list contains the nodes in $D_{\alpha,\alpha}$. The total number of nodes across all node lists in $\mathbb{I}^U_{BD}$ is $\sum_{\alpha=0}^{p} |D_{\alpha,\alpha}| \leq \sum_{\alpha=0}^{p} |\{x | d_x(G) > \alpha\}| \leq \sum_{x\in U\cup V} d_x(G) = 2|E|$, where the first inequality follows from Lemma 1. Therefore, the space occupied by the $(p+1)$ node lists in $\mathbb{I}^U_{BD}$ is $O(|E|)$.

Next, we prove that the space occupied by all pointers $\mathbb{I}^U_{BD}[\cdot][\cdot]$ is also $O(|E|)$. Let $r^{\max}_\alpha = \max_{x\in U\cup V} r_\alpha(x)$. Fixing a specific α, Lemma 1 implies that the nodes in $D^V_{\alpha,r^{\max}_\alpha}$ have degrees greater than $r^{\max}_\alpha$, which leads to $|D^U_{\alpha,r^{\max}_\alpha}| > r^{\max}_\alpha$. Thus, by Lemma 1, we obtain $r^{\max}_\alpha < |D^U_{\alpha,r^{\max}_\alpha}| \leq |\{u \in U | d_u > \alpha\}|$. By the definition of $\mathbb{I}^U_{BD}$, the number of pointers in $\mathbb{I}^U_{BD}$ is $\sum_{\alpha=0}^{p}(r^{\max}_\alpha - \alpha + 1) \leq \sum_{\alpha=0}^{p}(r^{\max}_\alpha + 1) \leq \sum_{\alpha=0}^{p} |\{u \in U | d_u > \alpha\}| \leq \sum_{u\in U} d_u = |E|$. Thus, the space occupied by the pointers in $\mathbb{I}^U_{BD}$ is also $O(|E|)$. Consequently, the total space complexity of $\mathbb{I}^U_{BD}$ is $O(|E|)$.

Using a similar approach, we can prove that $\mathbb{I}^V_{BD}$ also occupies $O(|E|)$ space. Therefore, the total space complexity of $\mathbb{I}_{BD}$ is $O(|E|)$. □

The linear space complexity $O(|E|)$ of BD-Index indicates that its space usage grows asymptotically proportional to the number of edges, ensuring highly stable and predictable memory consumption. Our experiments further show that the actual space usage is almost equal to $8|E|$ bytes in practice. This property enables BD-Index to efficiently handle large bipartite graphs while maintaining fast query performance.



(a) $\mathbb{I}^U_{BD}$. (b) $\mathbb{I}^V_{BD}$.

Figure 2: Example of BD-Index **and querying** $D_{0,2}$**.**

3.4 Index construction

In this subsection, we introduce the construction algorithm for BD-Index. The construction relies on three existing algorithms for computing (α, β)-dense subgraphs: DSS++, Divide-a, and Divide-b [50]. The DSS++ algorithm computes a single $D_{\alpha,\beta}$ subgraph. Given a bipartite graph G and parameters α and β, it constructs a network flow model and performs a minimum cut computation to obtain $D_{\alpha,\beta}$, with the worst-case time complexity of $O(|E|^{1.5})$. The Divide-a algorithm computes all non-empty $D_{\alpha,\beta}$ for a fixed α across all possible β values. It utilizes DSS++ along with a divide-and-conquer strategy for pruning, achieving a time complexity of $O(|E|^{1.5} \log |U \cup V|)$. Its symmetric counterpart, Divide-b, computes all non-empty $D_{\alpha,\beta}$ subgraphs for a fixed β across all α values.

With the three algorithms DSS++, Divide-a, and Divide-b, we propose the Build-BD-Index algorithm for constructing BD-Index, as shown in Algorithm 2. The algorithm proceeds as follows. First, it computes p as the maximum integer satisfying $D_{p,p} \neq \emptyset$ (line 1), which can be achieved by performing a binary search with invocations to DSS++ [50]. Then, for each α, the algorithm first invokes Divide-a to identify all non-empty $D_{\alpha,\beta}$ subgraphs across all β values (line 3). Based on these results, it determines the r_α value for each node (lines 4-5), sorts the nodes according to r_α, and constructs the corresponding node list (line 6). The algorithm subsequently sets the pointers $\mathbb{I}^U_{BD}[\alpha][\cdot]$ in $\mathbb{I}^U_{BD}$ using this node list (lines 7-8). Once $\mathbb{I}^U_{BD}[\alpha][\cdot]$ is constructed for all α, the algorithm performs a symmetric procedure for each β (lines 9-15). Finally, it returns $\mathbb{I}_{BD} = \mathbb{I}^U_{BD} \cup \mathbb{I}^V_{BD}$ (line 16).

The correctness of Build-BD-Index follows directly from the correctness of its constituent algorithms Divide-a and Divide-b. For the time complexity of the Build-BD-Index algorithm, the most time-consuming part is the invocation of the Divide-a and Divide-b algorithms. Since each invocation takes $O(|E|^{1.5} \log |U \cup V|)$ time and there are $O(p)$ invocations in total, the overall time complexity is $O(p \cdot |E|^{1.5} \cdot \log |U \cup V|)$. As established in [48, 50], the value of p is typically a small constant in real-world graphs. While its worst-case value is $p \leq \sqrt{|E|}/2$, this bound is only achieved in complete bipartite graphs, which are uncommon in practice. A similar parameter-dependent complexity also appears in bipartite core decomposition, where the time complexity is $O(\delta \cdot |E|)$ and δ is the largest integer such that the (δ, δ)-core is non-empty [26]. Since p tends to be small in real-world graphs, efficient index construction on large-scale bipartite networks is ensured.

Discussion: index construction and practical efficiency. While building the BD-Index incurs construction cost, this overhead is quickly offset in practical scenarios for several reasons:
(1) **Fast construction and cost amortization.** Our experiments Exp-1 and Exp-4 indicate that the index construction time approximates processing 1,000 online queries. However, once built, the index can efficiently support arbitrary (α, β) queries. The number of possible queries is much more than the initial cost required to build the index (e.g., for the HE dataset in our experiments, there are 549,464 non-empty $D_{\alpha,\beta}$ subgraphs), demonstrating that the index covers a query space far exceeding the number of queries needed to amortize its construction. In addition, in practical applications, query workloads can be very frequent (≈100,000 queries per second [47]), and the total number of queries soon surpasses this initial cost (≈1,000 online queries), making the construction overhead negligible in the long run.
(2) **Significant query efficiency.** The online algorithm exhibits slow query response times (up to 60 seconds per query, as shown in Exp-1), whereas our index achieves a maximum response time of only 0.018 seconds. In many real-world applications, the required response time must be returned in real time (e.g., within 0.5 seconds) [21, 25, 47], highlighting that the online approach is far too slow to be practical. In contrast, our index not only satisfies these stringent latency requirements but also achieves optimal $O(|R|)$ query complexity, allowing it to directly output the nodes contained in the identified community.

Algorithm 2: Build-BD-Index(G)

Input: A bipartite graph $G = (U, V, E)$.
Output: BD-Index.

1 $p \leftarrow$ the maximum integral such that $D_{p,p} \neq \emptyset$;
2 **for** $\alpha = 0, 1, 2, \ldots, p$ **do**
3   Invoke algorithm Divide-a in [50] to compute the non-empty $D_{\alpha,\beta}$ for all β;
4   **foreach** $x \in U \cup V$ **do**
5       $r_\alpha(x) \leftarrow$ the maximum β such that $x \in D_{\alpha,\beta}$;
6   Create a node list containing all nodes with $r_\alpha \geq \alpha$, and sort them in ascending order of r_α;
7   **for** $\beta = \alpha, \alpha + 1, \ldots, \max_{x \in (U \cup V)} r_\alpha(x)$ **do**
8       $\mathbb{I}^U_{BD}[\alpha][\beta] \leftarrow$ the first node in the node list with $r_\alpha \geq \beta$;
9 **for** $\beta = 0, 1, 2, \ldots, p$ **do**
10   Invoke algorithm Divide-b in [50] to compute the non-empty $D_{\alpha,\beta}$ for all α;
11   **foreach** $x \in U \cup V$ **do**
12       $r_\beta(x) \leftarrow$ the maximum α such that $x \in D_{\alpha,\beta}$;
13   Create a node list containing all nodes with $r_\beta > \beta$, and sort them in ascending order of r_β;
14   **for** $\alpha = \beta + 1, \beta + 2, \ldots, \max_{x \in (U \cup V)} r_\beta(x)$ **do**
15       $\mathbb{I}^V_{BD}[\beta][\alpha] \leftarrow$ the first node in the node list with $r_\beta \geq \alpha$;
16 **return** $\mathbb{I}_{BD} = \mathbb{I}^U_{BD} \cup \mathbb{I}^V_{BD}$;

(3) **Efficient maintenance for dynamic graphs.** Our proposed maintenance algorithms in subsequent sections ensure that BD-Index is incrementally updatable as the graph evolves through edge insertions and deletions. This design transforms the construction cost into a one-time investment, since ongoing maintenance can be performed efficiently without the need for complete reconstruction.

4 SPACE-EFFICIENT INDEX MAINTENANCE

This section aims to develop space-efficient maintenance algorithms for BD-Index under dynamic graph updates, preserving its advantage of linear space complexity $O(|E|)$. A straightforward method is to reconstruct BD-Index from scratch after each edge insertion or deletion. However, such recomputation incurs $O(p \cdot |E|^{1.5} \cdot \log |U \cup V|)$ time, which is prohibitive for real-world graphs with frequent edge updates. To address this bottleneck, we first formalize the update theorems, followed by proposing two incremental maintenance algorithms: BD-Insert-S and BD-Delete-S.

4.1 Update theorems for α-rank and β-rank

Since $\mathbb{I}_{BD}^{U}$ and $\mathbb{I}_{BD}^{V}$ are two symmetric structures, we focus on maintaining $\mathbb{I}_{BD}^{U}$ for brevity, and the maintenance of $\mathbb{I}_{BD}^{V}$ can be implemented symmetrically. In essence, maintaining $\mathbb{I}_{BD}^{U}$ requires updating the r_α values of nodes, with emphasis on how r_α evolve after edge insertions or deletions. To formalize this, we present the following update theorems for r_α that addresses edge insertion and deletion.

Theorem 6. **(Insertion Update Theorem)** *Given a graph G and an integer α, for an insertion edge (u, v), let r^N be the r_α value of the $(\alpha+1)$-th highest-ranked node in $N(u) \cup \{v\}$ (specifically, if $|N(u)| < \alpha$, then set $r^N = -1$), and let $\beta = \min\{r^N, r_\alpha(v)\}$. Then, for all nodes $x \in V$:*

1. *If $r_\alpha(x) = \beta$, then $r_\alpha(x)$ may be updated to $\beta + 1$.*
2. *If $r_\alpha(x) \neq \beta$, then $r_\alpha(x)$ remains unchanged.*

Proof. We divide the proof into three cases.

Case 1: When $|N(u)| < \alpha$, both r_N and β equal -1. Let $\vec{E}$ be an egalitarian orientation (defined in Definition 4, which we will introduce later) of the given graph G with respect to α. We directly insert the edge (u, v) into $\vec{E}$ with direction (v, u). According to the definition of egalitarian orientation, the resulting $\vec{E}$ remains egalitarian. Since the reachability of all nodes remains unchanged, the α-rank of each node also remains unchanged, which proves the case.

Case 2: When $|N(u)| \geq \alpha$ and $r_\alpha(u) > r_\alpha(v)$, we have $r^N = r_\alpha(u)$ and $\beta = r_\alpha(v)$ by Theorem 8 (we will prove later). For any $\beta' \in [-1, \beta] \cup [\beta+2, +\infty)$, let $\vec{E}$ be an orientation that satisfies the condition in Definition 1 with alpha value as α and beta value as β'. We insert the edge (u, v) into $\vec{E}$ with direction towards v. It is easy to verify that the updated $\vec{E}$ still satisfies the condition in Definition 1, so $D_{\alpha,\beta'}$ remains unchanged after insertion. Therefore, only $D_{\alpha,\beta+1}$ may change after insertion. Since α-rank values do not decrease, it follows that only the nodes with $r_\alpha = \beta$ may have their α-rank increased to $\beta + 1$, which proves the case.

Case 3: When $|N(u)| \geq \alpha$ and $r_\alpha(u) \leq r_\alpha(v)$, we have $r^N \geq r_\alpha(u)$ and $\beta = r^N$. Let $\vec{E}$ be an egalitarian orientation of the given graph G with respect to α. We insert the edge (u, v) into $\vec{E}$ with direction toward u, so u now has $(\alpha+1)$ in-neighbors. The value r^N represents the lowest α-rank among these $(\alpha+1)$ neighbors; let v_N be the corresponding neighbor. We then reverse the edge (v_N, u) in $\vec{E}$ to become (u, v_N). For any $\beta' \in [-1, \beta] \cup [\beta+2, +\infty)$, the updated $\vec{E}$ still satisfies the condition in Definition 1 with alpha value as α and beta value as β'. Therefore, only the nodes in V with $r_\alpha = \beta$ may have their α-rank increased to $\beta + 1$, which completes the proof. □

Theorem 7. **(Deletion Update Theorem)** *Given a graph G and an integer α, for an deletion edge (u, v), let $\beta = \min\{r_\alpha(u), r_\alpha(v)\}$. Then, for all nodes $x \in V$:*

1. *If $r_\alpha(x) = \beta$, then $r_\alpha(x)$ may be updated to $\beta - 1$.*
2. *If $r_\alpha(x) \neq \beta$, then $r_\alpha(x)$ remains unchanged.*

Proof. Let $\vec{E}$ be an egalitarian orientation of the given graph G and alpha α. We divide the proof into three cases.

Case 1: When $|N(u)| \leq \alpha$, it follows from Theorem 1 that $r_\alpha(u) = -1$. According to the definition of egalitarian orientation, all neighbors of u are directed toward u in $\vec{E}$, so the directed edge (v, u) can be directly removed. It is clear that the orientation remains egalitarian, and all nodes' r_α values remain unchanged, which completes the proof.

Case 2: When $|N(u)| > \alpha$ and the edge (u, v) is directed toward v in $\vec{E}$. According to Lemma 2 (we will prove later), we have $r_\alpha(u) \geq r_\alpha(v)$, and thus $\beta = r_\alpha(v)$. We directly remove the edge (u, v) from $\vec{E}$. For any beta value $\beta' \in [-1, \beta-1] \cup [\beta+1, +\infty)$, it is easy to verify that $\vec{E}$ still satisfies the condition in Definition 1 with alpha value α and beta value β'. This implies that only the subgraph $D_{\alpha,\beta}$ is affected by the edge deletion. Since deleting an edge cannot increase any node's α-rank, it follows that only the nodes with $r_\alpha = \beta$ may have their rank reduced to $\beta - 1$, which proves the case.

Case 3: When $|N(u)| > \alpha$ and the edge (u, v) is directed toward u in $\vec{E}$. According to Lemma 2, we have $r_\alpha(u) \leq r_\alpha(v)$, and thus $\beta = r_\alpha(u)$. From the proof of Theorem 8, there exists a neighbor $v_2 \in N(u)$ such that (u, v_2) is in $\vec{E}$ and $r_\alpha(v_2) = r_\alpha(u)$. We first remove the edge (v, u) from $\vec{E}$, and then reverse the edge (u, v_2) to become (v_2, u). For any beta value $\beta' \in [-1, \beta-1] \cup [\beta+1, +\infty)$, it is easy to verify that the updated $\vec{E}$ still satisfies the condition in Definition 1 with alpha value α and beta value β'. This shows that only the nodes in V with $r_\alpha = \beta$ may have their rank decreased to $\beta - 1$, which completes the proof. □

The above two update theorems indicate that when an edge is inserted or deleted, there exists a value β such that in V, only nodes with $r_\alpha = \beta$ require updates, and their r_α values change by at most 1 (note that this property does not hold for nodes in U). In the next theorem, we describe how to compute the r_α values of nodes in U given the r_α values of nodes in V. Based on this theorem, we can first update the r_α values of nodes in V and then use these updated values to update the r_α values of nodes in U, forming the foundation of our maintenance algorithms.

Theorem 8. *Given a graph G and a value α, for any $u \in U$, we have:*

1. *If $|N(u)| \leq \alpha$, then $r_\alpha(u) = -1$.*
2. *If $|N(u)| > \alpha$, then $r_\alpha(u)$ is equal to the r_α value of the $(\alpha+1)$-th highest-ranked node in $N(u)$.*

Proof. When $|N(u)| \leq \alpha \Rightarrow d_u \leq \alpha$, it follows from Lemma 1 that $u \notin D_{\alpha,0}$, and thus $r_\alpha(u) = -1$. Otherwise, if $|N(u)| > \alpha$, let $\vec{E}$ be an egalitarian orientation of the given graph G with respect to α.

Algorithm 3: BD-Insert-S$(G, \mathbb{I}_{BD}, u, v)$

Input: A bipartite graph G and its BD-Index $\mathbb{I}_{BD}$, the edge (u, v) to be inserted.
Output: The updated G and $\mathbb{I}_{BD}$.
1 **for** $\alpha = 0, 1, \ldots, p$ **do**
2 Compute β according to Theorem 6;
3 Invoke DSS++ algorithm in [50] to compute the $D_{\alpha,\beta+1}$ of $G \cup \{(u, v)\}$;
4 **foreach** $x \in V \cap D_{\alpha,\beta+1}$ *and* $r_\alpha(x) = \beta$ **do**
5 $r_\alpha(x) \leftarrow \beta + 1$, update $\mathbb{I}_{BD}^U$ accordingly;
6 **foreach** $x \in U$ **do**
7 Compute $r_\alpha(x)$ according to Theorem 8, update $\mathbb{I}_{BD}^U$ accordingly;
8 Update $\mathbb{I}_{BD}^V$ similarly;
9 $G \leftarrow G \cup \{(u, v)\}$;
10 **return** $G, \mathbb{I}_{BD}$;

Algorithm 4: BD-Delete-S$(G, \mathbb{I}_{BD}, u, v)$

Input: A bipartite graph G and its BD-Index $\mathbb{I}_{BD}$, the edge (u, v) to be deleted.
Output: The updated G and $\mathbb{I}_{BD}$.
1 **for** $\alpha = 0, 1, \ldots, p$ **do**
2 Compute β according to Theorem 7;
3 Invoke DSS++ algorithm in [50] to compute the $D_{\alpha,\beta}$ of $G \setminus \{(u, v)\}$;
4 **foreach** $x \in V \setminus D_{\alpha,\beta}$ *and* $r_\alpha(x) = \beta$ **do**
5 $r_\alpha(x) \leftarrow \beta - 1$, update $\mathbb{I}_{BD}^U$ accordingly;
6 **foreach** $x \in U$ **do**
7 Compute $r_\alpha(x)$ according to Theorem 8, update $\mathbb{I}_{BD}^U$ accordingly;
8 Update $\mathbb{I}_{BD}^V$ similarly;
9 $G \leftarrow G \setminus \{(u, v)\}$;
10 **return** $G, \mathbb{I}_{BD}$;

According to the definition of egalitarian orientation, the indegree of u is exactly α, so there are α neighbors in $N(u)$ with directed edges pointing to u. By Lemma 2, these neighbors must have α-rank values greater than or equal to $r_\alpha(u)$.

In addition, since $r_\alpha(u)$ is the α-rank of u, u must be able to reach a node $v_1 \in V$ whose indegree is greater than $r_\alpha(u)$. Let (u, v_2) be the first edge on the path $u \rightsquigarrow v_1$. It follows that $r_\alpha(v_2) = r_\alpha(u)$, and by contradiction, we can show that u cannot reach any node with an α-rank greater than $r_\alpha(u)$.

In summary, $N(u)$ contains α neighbors pointing to u, all of whose α-ranks are greater than or equal to $r_\alpha(u)$. On the other hand, among the neighbors pointed to by u, the one with the highest α-rank is v_2, whose rank is exactly $r_\alpha(u)$. Therefore, the $(\alpha + 1)$-th largest α-rank among the neighbors of u is $r_\alpha(u)$, which completes the proof. □

Example 4. *For the insertion case, consider the scenario where $\alpha = 0$ and we insert (u_6, v_3) into G in Figure 1a. By Theorem 6, we first examine $N(u_6) \cup \{v_3\} = \{v_3, v_5\}$. From the* BD-Index *in Figure 2, we have $r_\alpha(v_3) = 3$ and $r_\alpha(v_5) = 1$, thus obtaining $r^N = r_\alpha(v_3) = 3$. This results in $\beta = 3$, indicating that nodes in V with $r_\alpha = 3$, namely v_2 and v_3, may have their r_α values updated to 4, while the r_α values of all other nodes in V remain unchanged. In fact, after inserting (u_6, v_3), only $r_\alpha(v_3)$ increases to 4. Next, we update the r_α values of nodes in U according to Theorem 8. As a result, the r_α values of $\{u_1, u_2, u_3, u_4, u_6\}$ are updated to 4. Thus, we obtain the updated r_α values for all nodes.*

For the deletion case, suppose that $\alpha = 0$ and we delete (u_5, v_4) from G. By Theorem 7, we compute $\beta = \min\{2, 2\} = 2$. This means that nodes in V with $r_\alpha = 2$, namely v_1 and v_4, may have their r_α values updated to 1. In fact, after this deletion, only v_4 undergoes an update, with $r_\alpha(v_4)$ decreasing to 1, while the r_α values of all other nodes in V remain unchanged. Next, according to Theorem 8, the r_α value of u_5 is updated to -1.

4.2 The incremental maintenance algorithms

Building on Theorem 6, Theorem 7, and Theorem 8, we present the incremental maintenance algorithms, BD-Insert-S and BD-Insert-D, to handle edge insertion and deletion, as depicted in Algorithm 3 and Algorithm 4.

The BD-Insert-S algorithm for edge insertion. In Algorithm 3, BD-Insert-S processes each α iteration independently (line 1). First, it determines the value β according to Theorem 6 (line 2). Since nodes with $r_\alpha = \beta$ may increment their r_α to $\beta + 1$, the algorithm computes $D_{\alpha,\beta+1}$ to identify these affected nodes (line 3). It then iterates over all $x \in V$: if $x \in D_{\alpha,\beta+1}$ and $r_\alpha(x) = \beta$, it updates $r_\alpha(x)$ to $\beta + 1$ and maintains $\mathbb{I}_{BD}^U$ accordingly (lines 4-5). Once all nodes in V are processed, the algorithm applies Theorem 8 to update the nodes in U (lines 6-7). After processing all α values, the algorithm completes updating $\mathbb{I}_{BD}^U$ and symmetrically maintains $\mathbb{I}_{BD}^V$ (line 8). Finally, the updated G and BD-Index are returned (line 10).

The BD-Delete-S algorithm for edge deletion. The workflow of BD-Delete-S mirrors BD-Insert-S. The algorithm first iterates over each α (line 1) and derives β via Theorem 7 (line 2). To identify nodes with $r_\alpha = \beta$ that require updating to $\beta - 1$, it computes $D_{\alpha,\beta}$ after the edge deletion (line 3). For each node $x \in V$, if $r_\alpha(x) = \beta$ and $x \notin D_{\alpha,\beta}$, the algorithm reduces $r_\alpha(x)$ to $\beta - 1$ and updates $\mathbb{I}_{BD}^U$ (lines 4-5). Subsequently, it updates the r_α values of nodes in U following Theorem 8 (lines 6-7). After processing all α, the algorithm finalizes updates to $\mathbb{I}_{BD}^U$ and symmetrically maintains $\mathbb{I}_{BD}^V$ (line 8). Finally, the updated G and $\mathbb{I}_{BD}$ are returned (line 10).

The correctness of BD-Insert-S and BD-Delete-S is directly established by Theorem 6 and Theorem 7. Next, we analyze their time and space complexities.

Theorem 9. *The* BD-Insert-S *and* BD-Delete-S *algorithms have a time complexity of $O(p \cdot |E|^{1.5})$ and a space complexity of $O(|E|)$.*

Proof. In the BD-Insert-S algorithm, the dominant computational cost lies in invoking DSS++, which has a worst-case time complexity of $O(|E|^{1.5})$ per call [50]. As BD-Insert-S executes DSS++ $O(p)$ times, the algorithm derives its overall time complexity of $O(p \cdot |E|^{1.5})$. Regarding space complexity, the most memory-intensive operation is DSS++, which requires $O(|E|)$ space [50]. Therefore, the space complexity of BD-Insert-S is $O(|E|)$. The BD-Delete-S algorithm operates analogously to BD-Insert-S; applying the same analysis, its time complexity is $O(p \cdot |E|^{1.5})$, and its space complexity is $O(|E|)$. □

Compared to the baseline method of recomputing from scratch, BD-Insert-S and BD-Delete-S retain the space-efficient advantage by retaining $O(|E|)$ space. Meanwhile, they reduce the time complexity of updating a single edge from $O(p \cdot |E|^{1.5} \cdot \log |U \cup V|)$

Algorithm 5: OrientationToRank($\alpha, \vec{E}$)

Input: An alpha value α and an egalitarian orientation.
Output: The α-rank of all nodes.

1 $\vec{d}_{\max} \leftarrow \max_{v \in V} \vec{d}_v(\vec{E})$, $vis \leftarrow \emptyset$;
2 **foreach** $k = \vec{d}_{\max} - 1, \vec{d}_{\max} - 2, \ldots, 0$ **do**
3 $\quad T \leftarrow \{v \in V \setminus vis \mid \vec{d}_v(\vec{E}) = k + 1\}$;
4 $\quad$ **forall** $x \in (U \cup V) \setminus vis$, $x \in T$ *or* x *can reach a node in* T **do**
5 $\quad\quad r_\alpha(x) \leftarrow k$, $vis \leftarrow vis \cup \{x\}$;
6 **forall** $x \in (U \cup V) \setminus vis$ **do** $r_\alpha(x) \leftarrow -1$;
7 **return** r_α;

to $O(p \cdot |E|^{1.5})$. Unlike the baseline, which recomputes all layers, BD-Insert-S and BD-Delete-S update only a single node layer in V, resulting in superior performance. As shown by our experiments, they achieve approximately one order of magnitude of speedup compared to the baseline approach.

5 TIME-EFFICIENT INDEX MAINTENANCE

In this section, we focus on developing time-efficient algorithms for maintaining BD-Index. Recall that if an orientation satisfies the structural constraints in Definition 1, the corresponding dense subgraph can be derived dirctly. This insight motivates our idea: by maintaining the orientation, we can efficiently compute the dense subgraph (i.e., node ranks), thereby enabling efficient BD-Index maintenance. Building on this, we first define the concept of egalitarian orientation (Definition 4), and then show how to efficiently compute the BD-Index from egalitarian orientation (Algorithm 5), thereby transforming the task of maintaining the BD-Index into maintaining the egalitarian orientation. We then present the algorithms BD-Insert-T and BD-Delete-T, which are designed to maintain the egalitarian orientations efficiently.

5.1 A novel concept: egalitarian orientation

In this subsection, we introduce the concept of egalitarian orientation based on α for maintaining $\mathbb{I}^U_{BD}$. The maintenance of $\mathbb{I}^V_{BD}$ follows symmetrically by replacing α with β and swapping the constraints between U and V in Definition 4.

Definition 4. **(Egalitarian orientation)** *Given a bipartite graph G and an alpha value α, an orientation $\vec{E}$ is called an* egalitarian orientation *if it satisfies the following conditions:*

(1) *For any node $u \in U$, if $d_u(G) > \alpha$, then $\vec{d}_u(\vec{E}) = \alpha$; otherwise, if $d_u(G) \leq \alpha$, then $\vec{d}_u(\vec{E}) = d_u(G)$.*
(2) *There exists no path $v_s \rightsquigarrow v_t$ in $\vec{E}$ such that $v_s, v_t \in V$ and $\vec{d}_{v_t}(\vec{E}) - \vec{d}_{v_s}(\vec{E}) \geq 2$.*

For example, the orientation shown in Figure 1b is an egalitarian orientation given $\alpha = 1$. Each node in U has an indegree of exactly $\alpha = 1$, and no path $v_s \rightsquigarrow v_t$ exists among nodes in V that violates the condition of Definition 4.

Intuition behind egalitarian orientation. The term "egalitarian" in egalitarian orientation refers to the balancing of indegrees among nodes in V. Suppose there exists a path $v_s \rightsquigarrow v_t$ in an orientation $\vec{E}$ such that $v_s, v_t \in V$ and $\vec{d}_{v_t}(\vec{E}) - \vec{d}_{v_s}(\vec{E}) \geq 2$. By reversing this path (i.e., reversing the direction of all edges along the path), the indegree of v_t decreases by one, the indegree of v_s increases by one, and the indegrees of all other nodes remain unchanged. This reversing operation balances the indegrees of v_t and v_s, leading to a more even distribution of indegrees among nodes in V. An egalitarian orientation guarantees that no such path exists, intuitively meaning the indegrees of nodes in V are already as "egalitarian" as possible.

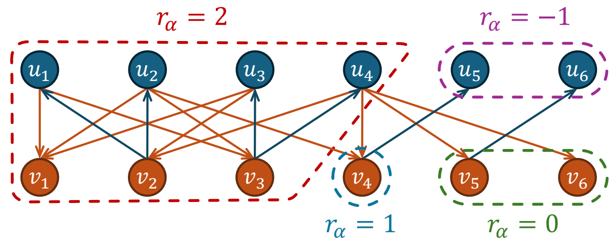


Figure 3: Example of computing r_α from an egalitarian orientation.

Next, we introduce how to compute the rank of nodes based on the egalitarian orientation. The algorithm OrientationToRank for computing α-rank from an egalitarian orientation is shown in Algorithm 5. The algorithm first computes the maximum indegree $\vec{d}_{\max}$ among all nodes in V, and initializes the set vis to record visited nodes (line 1). In each round of the "foreach" loop (line 2), the algorithm identifies the nodes whose r_α is equal to k. Specifically, it first collects nodes in $V \setminus vis$ with indegree equal to $k+1$ into a set T (line 3). Then, for each node in T and those that can reach nodes in T, their r_α values are set to k, and they are added to the visited set vis (lines 4–5). Finally, all unvisited nodes are assigned $r_\alpha = -1$ (line 6), and the algorithm returns the r_α values for all nodes (line 7). Below is an example of running algorithm OrientationToRank.

Example 5. *We set $\alpha = 1$ and use the egalitarian orientation $\vec{E}$ in Figure 1b as the input to run the* OrientationToRank *algorithm. The procedure is illustrated in Figure 3. The algorithm first determines that $\vec{d}_{\max} = \vec{d}_{v_1}(\vec{E}) = 3$. Then, it enters the loop with $k = 2$ and identifies all nodes in $V \setminus vis$ with indegree 3, i.e., $T = \{v_1\}$. Next, the algorithm computes the set of nodes that can reach v_1, which is $\{v_2, v_3, u_1, u_2, u_3, u_4\}$. As a result, all nodes in $\{v_1, v_2, v_3, u_1, u_2, u_3, u_4\}$ are assigned $r_\alpha = 2$ and added to the set vis. In the next iteration with $k = 1$, the algorithm considers the nodes outside of vis and computes $T = \{v_4\}$. Since no nodes can reach v_4, the only node with $r_\alpha = 1$ is v_4. In the following iteration with $k = 0$, we have $T = \{v_5, v_6\}$. Again, there are no nodes that can reach v_5 or v_6, so these two nodes are the only ones with $r_\alpha = 0$. Finally, the nodes that have not been visited, i.e., $\{u_5, u_6\}$, are assigned $r_\alpha = -1$.*

Next, we prove the correctness and analyze the time complexity of the OrientationToRank algorithm.

Theorem 10. *The* OrientationToRank *algorithm can correctly return α-rank within $O(|E|)$ time.*

Proof. We first prove the correctness of the algorithm. We begin by showing that in the egalitarian orientation $\vec{E}$, the path $s \rightsquigarrow t$ described in Definition 1 does not exist. Assume for contradiction that such a path $s \rightsquigarrow t$ exists. First, consider any node $u \in U$ with $d_u(G) \leq \alpha$. In the egalitarian orientation, all its incident edges are directed toward u, and its indegree cannot exceed α. Hence, u cannot be either s or t in the path. Next, for any node $u \in U$ with $d_u(G) > \alpha$, its indegree is exactly α, so again u cannot be s or t. Therefore, both s and t must be in V. Since $\vec{d}_s < \beta$ and $\vec{d}_t > \beta$, it follows that $\vec{d}_t - \vec{d}_s \geq 2$, which contradicts the definition of an egalitarian orientation. Thus, such a path $s \rightsquigarrow t$ cannot exist.

Algorithm 6: BD-Insert-T$(G, \mathbb{I}_{BD}, \vec{\mathbb{E}}, u, v)$

Input: A bipartite graph G and its BD-Index $\mathbb{I}_{BD}$, the egalitarian orientations, the edge (u, v) to be inserted.
Output: The updated G, $\mathbb{I}_{BD}$, and egalitarian orientations.

```
1  for α = 0, 1, ..., p do
2      Let E⃗ ∈ 𝔼⃗ be the egalitarian orientation for the current α;
3      E⃗ ← E⃗ ∪ (v, u);
4      if d_u(G) < α then continue to the next α value ;
5      Let v_min ← argmin_{v′∈V, v′ can reach u} d⃗_{v′}(E⃗);
6      Reverse the path v_min ⇝ u;
7      r_α(·) ← OrientationToRank(α, E⃗), and update 𝕀^U_BD
         accordingly;
8  Update 𝕀^V_BD similarly;
9  G ← G ∪ {(u, v)};
10 return G, 𝕀_BD, 𝔼⃗;
```

It follows that in an egalitarian orientation, $D_{\alpha,\beta} = T \cup \{x \mid x \text{ can reach a node in } T \text{ in } \vec{E}\}$, where $T = \{v \in V \mid \vec{d}_v(\vec{E}) > \beta\}$. Next, we prove that if a node $x \in D_{\alpha,\beta} \setminus D_{\alpha,\beta+1}$, then the algorithm OrientationToRank correctly computes $r_\alpha(x) = \beta$. Let $T_1 = \{v \in V \mid \vec{d}_v(\vec{E}) > \beta + 1\}$ and $T_2 = \{v \in V \mid \vec{d}_v(\vec{E}) > \beta\}$. Since $x \in D_{\alpha,\beta} \setminus D_{\alpha,\beta+1}$, we know that $x \notin T_1$ and cannot reach any node in T_1. Therefore, in the "foreach" loop of the algorithm, x will not be visited in any iteration where $k \geq \beta + 1$. On the other hand, $x \in T_2$ or x can reach a node in T_2, so in the iteration where $k = \beta$, x will be visited and assigned $r_\alpha(x) = \beta$. According to the definition of α-rank, this confirms that OrientationToRank computes the correct α-rank.

Next, we analyze the time complexity. In each round of the "foreach" loop, computing the nodes that can reach T can be done via a breadth-first search. Once a node is visited, it is added to the set vis, and no node is revisited. Therefore, the total cost of all breadth-first searches is bounded by $O(|E|)$, which shows that the overall time complexity of OrientationToRank is $O(|E|)$. □

By using the OrientationToRank algorithm, once we have an egalitarian orientation, the r_α values of all nodes can be computed efficiently in linear time. Therefore, we transform the task of maintaining r_α and the BD-Index into the task of maintaining egalitarian orientation. When an edge is inserted or deleted, we can first update the egalitarian orientation, from which we can then compute the updated r_α values and correspondingly update the BD-Index. Compared to directly maintaining the BD-Index (as done in BD-Insert-S and BD-Delete-S), maintaining the egalitarian orientation is significantly more efficient. Next, we present the algorithms BD-Insert-T and BD-Delete-T, which provide efficient methods for maintaining egalitarian orientations.

5.2 The insertion algorithm BD-Insert-T

According to Definition 4 and Theorem 10, each egalitarian orientation corresponds to a unique α value. The index $\mathbb{I}_{BD}^U$ maintains $(p + 1)$ distinct α values, each associated with a node list, thus requiring $(p + 1)$ corresponding egalitarian orientations. Similarly, $\mathbb{I}_{BD}^V$ requires another $(p + 1)$ orientations. Consequently, the total number of required egalitarian orientations is $(2p+2)$. We represent the set of all such orientations as $\vec{\mathbb{E}}$.

Given all egalitarian orientations $\vec{\mathbb{E}}$, we propose the BD-Insert-T algorithm for edge insertion in Algorithm 6. Similar to BD-Insert-S (Algorithm 3), BD-Insert-T processes each α in $\mathbb{I}_{BD}^U$ individually (line 1). For each α, it first retrieves the corresponding egalitarian orientation $\vec{E}$, and inserts the directed edge (v, u) into $\vec{E}$ (lines 2-3). If $d_u(G) < \alpha$, the orientation $\vec{E}$ remains egalitarian with unchanged r_α values, and BD-Insert-T directly proceeds to the next α (line 4). Otherwise, if $d_u(G) \geq \alpha$, $\vec{E}$ may violate the egalitarian conditions, requiring the following adjustments. First, let $v_{\min}$ be the node in V with the minimum indegree among all nodes that can reach u (line 5). By definition of reachability, there is a path $v_{\min} \rightsquigarrow u$, and the algorithm reverses this path in $\vec{E}$ by reversing all edge directions along this path (line 6). This reversal restores $\vec{E}$ to an egalitarian orientation. Thus, the algorithm updates the r_α values of all nodes using algorithm OrientationToRank and correspondingly maintains $\mathbb{I}_{BD}^U$ (line 7). After processing all values of α (for $\mathbb{I}_{BD}^U$) and β (for $\mathbb{I}_{BD}^V$) using the above method, BD-Insert-T returns the updated graph G, BD-Index, and egalitarian orientations $\vec{\mathbb{E}}$ (line 10).

Example 6. *Consider the orientation shown in Figure 1b, which by definition forms an egalitarian orientation for $\alpha = 1$. We analyze the execution of the* BD-Insert-T *algorithm when inserting edge (u_6, v_3). After processing the $\alpha = 0$ case, the algorithm proceeds to $\alpha = 1$. First, it inserts the directed edge (v_3, u_6) into the orientation. Since $d_{u_6}(G) = 1 = \alpha$,* BD-Insert-T *cannot directly continue to the next α. It identifies all nodes in V that can reach u_6, i.e., v_2, v_3, v_5. Among these nodes, v_5 has the lowest indegree, so we have $v_{\min} = v_5$. Then, the algorithm reverses the path $v_5 \rightsquigarrow u_6$, which contains a single directed edge (v_5, u_6), changing its direction to (u_6, v_5). After this reversal, the orientation once again becomes egalitarian. Using algorithm* OrientationToRank, BD-Insert-T *then computes the updated r_α values for all nodes and determines that only nodes u_6 and v_5 have changed their r_α values, from 0 to 1. The index $\mathbb{I}_{BD}^U$ is maintained accordingly, completing the $\alpha = 1$ case processing for this edge insertion.*

Next, we propose the following lemma, which serves as the foundation for analyzing the correctness and complexity of the BD-Insert-T algorithm.

Lemma 2. *Given a graph G, an alpha value α, and an egalitarian orientation $\vec{E}$, we have the following properties: (1) For any node $v \in V$, it holds that $\vec{d}_v(\vec{E}) \in \{r_\alpha(v), r_\alpha(v) + 1\}$; (2) For any nodes $x, y \in (U \cup V)$, if $r_\alpha(x) > r_\alpha(y)$, then the directed edge (x, y) in $\vec{E}$ must point to y.*

Proof. We first prove property (1). According to the definition of egalitarian orientation and Theorem 10, we know that the indegree of v $\vec{d}_v(\vec{E}) \leq r_\alpha(v) + 1$; otherwise, v would have a higher α-rank value. Moreover, we have that either v has an indegree greater than $r_\alpha(v)$, or v can reach a node in V with indegree greater than $r_\alpha(v)$. If v has indegree greater than $r_\alpha(v)$, the property holds. Otherwise, by condition (2) in the definition of egalitarian orientation, v must have indegree greater than $r_\alpha(v) - 1$, and thus the property also holds.

Next, we prove property (2). According to the construction of $D_{\alpha,\beta}$ in the proof of Theorem 10, let $T = \{v \in V \mid \vec{d}_v(\vec{E}) > \beta\}$. Then, $D_{\alpha,\beta}$ consists of T together with all nodes that can reach T, which implies that all edges between $D_{\alpha,\beta}$ and $(U \cup V) \setminus D_{\alpha,\beta}$ are directed from $D_{\alpha,\beta}$ to $(U \cup V) \setminus D_{\alpha,\beta}$. Therefore, if $r_\alpha(x) > r_\alpha(y)$, node x belongs to a denser subgraph than y, and the edge (x, y) must be directed toward y, which proves the property. □

THEOREM 11. *The* BD-Insert-T *algorithm can correctly maintain* BD-Index.

PROOF. Since the algorithm processes each node list (i.e., each alpha value) separately, we only need to prove that within the loop for a given alpha value (lines 2–7), the algorithm correctly updates the egalitarian orientation. Once the orientation is correctly maintained, the corresponding r_α values and the BD-Index can be correctly updated based on Theorem 10.

Therefore, let the current alpha value be α. If $d_u(G) < \alpha$ (line 4), it is clear that the egalitarian orientation is updated correctly. Otherwise, if $d_u(G) > \alpha$, let (v_1, u) be the last edge on the path $v_{\min} \rightsquigarrow u$. Since $v_{\min}$ can reach v_1 or $v_{\min} = v_1$, it follows from Theorem 2 that $r_\alpha(v_{\min}) \geq r_\alpha(v_1)$. Additionally, since $\vec{d}_{v_{\min}} \leq \vec{d}_{v_1}$, we also have $r_\alpha(v_{\min}) \leq r_\alpha(v_1)$. Therefore, $r_\alpha(v_{\min}) = r_\alpha(v_1)$. Similarly, we can conclude that all nodes along the path $v_{\min} \rightsquigarrow v_1$ have the same α-rank as $v_{\min}$. Now, suppose we first reverse the path $v_{\min} \rightsquigarrow v_1$ in the original orientation. It is easy to see that the resulting orientation remains egalitarian. Then, we reverse the edge (v_1, u) (i.e., the full path $v_{\min} \rightsquigarrow u$ has been reversed). Since $v_{\min}$ is the node with the smallest indegree among those that can reach u, the resulting orientation is still egalitarian. Hence, BD-Insert-T correctly maintains the egalitarian orientation, and by Theorem 10, it also correctly updates the BD-Index. □

THEOREM 12. *The time complexity and space complexity of the* BD-Insert-T *algorithm are both* $O(p \cdot |E|)$.

PROOF. The most computationally intensive steps of the algorithm are line 5 and line 7. Line 5 can be implemented by performing a breadth-first search from node u, requiring $O(|E|)$ time. Similarly, line 7 can be executed in $O(|E|)$ time according to Theorem 10. Consequently, processing a single α value requires $O(|E|)$ time. Since the algorithm handles $(p + 1)$ distinct values of α, maintaining $\mathbb{I}_{BD}^U$ incurs $O(p \cdot |E|)$ time complexity. By symmetry, the maintenance of $\mathbb{I}_{BD}^V$ has identical complexity. Therefore, the total time complexity of BD-Insert-T is $O(p \cdot |E|)$. For space complexity, the dominant cost comes from storing the input egalitarian orientations $\vec{\mathbb{E}}$, occupying $O(p \cdot |E|)$ space. □

5.3 The deletion algorithm BD-Delete-T

The pseudo-code of our BD-Delete-T algorithm is outlined in Algorithm 7. Similar to BD-Insert-T, the BD-Delete-T algorithm processes each α value individually (line 1). For each α, it first retrieves the corresponding egalitarian orientation $\vec{E}$ (line 2). The algorithm then diverges into two cases based on whether $d_u(G) \leq \alpha$. In the first case where $d_u(G) \leq \alpha$, the algorithm simply removes the edge (v, u) from $\vec{E}$ (by definition, edge (v, u) must be directed toward u in this case) while preserving the orientation's egalitarian property and all r_α values. In the second case where $d_u(G) > \alpha$, the algorithm examines the direction of edge (u, v) in $\vec{E}$. If oriented toward u (line 6), it identifies $v_{\max}$ as the maximum-indegree node reachable from u, and reverses the path $u \rightsquigarrow v_{\max}$ (lines 7-8). Conversely, if oriented toward v (line 9), the algorithm determines $v_{\max}$ as either v itself or the node with the highest indegree among all nodes in V reachable from v (line 10), and performs path reversal only in the latter case (line 11). Next, the algorithm removes either

Algorithm 7: BD-Delete-T$(G, \mathbb{I}_{BD}, \vec{\mathbb{E}}, u, v)$

Input: A bipartite graph G and its BD-Index $\mathbb{I}_{BD}$, the egalitarian orientations, the edge (u, v) to be deleted.
Output: The updated G, $\mathbb{I}_{BD}$, and egalitarian orientations.

1 **for** $\alpha = 0, 1, \ldots, p$ **do**
2   Let $\vec{E} \in \vec{\mathbb{E}}$ be the egalitarian orientation for the current α;
3   **if** $d_u(G) \leq \alpha$ **then**
4         $\vec{E} \leftarrow \vec{E} \setminus (v, u)$;
5   Continue to the next α value;
6   **if** $(v, u) \in \vec{E}$ **then**
7   Let $v_{\max} \leftarrow \arg\max_{v' \in V,\, u \text{ can reach } v'} \vec{d}_{v'}(\vec{E})$;
8   Reverse the path $u \rightsquigarrow v_{\max}$;
9   **else** // $(u, v) \in \vec{E}$
10   Let $v_{\max} \leftarrow \arg\max_{v' \in \{v' \in V | v \text{ can reach } v'\} \cup \{v\}} \vec{d}_{v'}(\vec{E})$;
11   **if** $v_{\max} \neq v$ **then** reverse the path $v \rightsquigarrow v_{\max}$;
12     $\vec{E} \leftarrow \vec{E} \setminus (u, v)$ or $\vec{E} \leftarrow \vec{E} \setminus (v, u)$;
13     $r_\alpha(\cdot) \leftarrow$ OrientationToRank$(\alpha, \vec{E})$, and update $\mathbb{I}_{BD}^U$ accordingly;
14 Update $\mathbb{I}_{BD}^V$ similarly;
15 $G \leftarrow G \setminus \{(u, v)\}$;
16 **return** $G, \mathbb{I}_{BD}, \vec{\mathbb{E}}$;

(u, v) or (v, u) from $\vec{E}$, depending on whether it points toward u or v (line 12). At this point, the resulting $\vec{E}$ is guaranteed to be an egalitarian orientation. Subsequently, the algorithm updates the r_α values of all nodes and updates the BD-Index using algorithm OrientationToRank (line 13). After processing all α values for $\mathbb{I}_{BD}^U$, the algorithm performs analogous updates for $\mathbb{I}_{BD}^V$ (line 14). Ultimately, the BD-Delete-T algorithm returns G, $\mathbb{I}_{BD}$, and $\vec{\mathbb{E}}$ (line 16).

Next, we prove the correctness of BD-Delete-T and analyze its complexity.

THEOREM 13. *The* BD-Delete-T *algorithm can correctly maintain* BD-Index.

PROOF. Similar to the correctness proof of BD-Insert-T, the correctness of BD-Delete-T also only requires showing that the algorithm correctly maintains the egalitarian orientation for each alpha value α. First, if $d_u(G) \leq \alpha$, it is straightforward to verify the correctness based on the definition of egalitarian orientation. Otherwise, if $d_u(G) > \alpha$, we consider two cases:

Case 1: $(u, v) \in \vec{E}$ (line 6). In this case, according to the proof of Theorem 8, the neighbor $v_1 \in N(u)$ that is pointed to by u and has the highest r_α satisfies $r_\alpha(v_1) = r_\alpha(u)$. Since v_1 can reach $v_{\max}$ or is equal to $v_{\max}$, we have $r_\alpha(v_1) \geq r_\alpha(v_{\max})$. On the other hand, since $v_{\max}$ has a higher indegree than v_1, it follows that $r_\alpha(v_{\max}) \geq r_\alpha(v_1)$. Therefore, $r_\alpha(v_1) = r_\alpha(v_{\max})$. Similarly, we can conclude that all nodes along the path $v_1 \rightsquigarrow v_{\max}$ have the same α-rank as $v_{\max}$. Hence, by reversing the path $u \rightsquigarrow v_{\max}$ and deleting the edge (u, v) or (v, u) from $\vec{E}$, it is easy to see that the resulting orientation remains egalitarian.

Case 2: $(v, u) \in \vec{E}$ (line 6). This case can be further divided into two subcases: (1) If $v_{\max} = v$, then the algorithm simply removes the edge (u, v) without reversing any path. This implies that v cannot reach any node in V with a higher indegree. According to the definition of egalitarian orientation, any node in V that can reach v must have indegree at least $\vec{d}_v - 1$, so decreasing v's indegree by removing (u, v) does not violate the egalitarian condition. (2) If $v_{\max} \neq v$, we can use a similar argument as in Case 1 to show that the resulting orientation remains egalitarian after the path reversal and edge deletion.

Therefore, the algorithm correctly maintains the egalitarian orientation, and by Theorem 10, it also correctly maintains the BD-Index. □

Theorem 14. *The time complexity and space complexity of the* BD-Delete-T *algorithm are both* $O(p \cdot |E|)$.

Proof. Similar to BD-Insert-T, BD-Delete-T can identify $v_{\max}$ through a single breadth-first search in $O(|E|)$ time. The rank update in line 13 also requires $O(|E|)$ time according to Theorem 10. Thus, maintaining $\mathbb{I}_{BD}^{U}$ for a single α takes $O(|E|)$ time. With $(p+1)$ distinct α values, the total time complexity for maintaining $\mathbb{I}_{BD}^{U}$ becomes $O(p \cdot |E|)$. The same complexity applies to maintaining $\mathbb{I}_{BD}^{V}$. Hence, the overall time complexity of BD-Delete-T is $O(p \cdot |E|)$. Regarding space complexity, the dominant factor is storing the egalitarian orientations, which occupies $O(p \cdot |E|)$ space. □

By leveraging the egalitarian orientation, BD-Insert-T and BD-Delete-T can update a node list in BD-Index in $O(|E|)$ time, avoiding the computationally expensive DSS++ algorithm ($O(|E|^{1.5})$ time) required by BD-Insert-S and BD-Delete-S. This complexity reduction enables BD-Insert-T and BD-Delete-T to significantly outperform their counterparts (BD-Insert-S and BD-Delete-S) when processing dynamic graphs.

6 EXPERIMENTS

Algorithms. For (α, β)-dense subgraph queries, we implement three algorithms: the baseline online algorithm Online, its optimized variant Online++, and the index-based algorithm Query-BD-Index (Algorithm 1). The Online algorithm processes each query by invoking the state-of-the-art (α, β)-dense subgraph search algorithm DSS++ [50]. Building on this, Online++ accelerates query processing through two optimizations: (1) it reuses the orientation obtained from the previous query rather than reinitializing it before the max-flow computation; and (2) it caches the results of the most recent 10 queries. When processing a new query with parameters (α, β), Online++ checks whether any cached result D_{α^+,β^+} satisfies $\alpha^+ \geq \alpha$ and $\beta^+ \geq \beta$, or D_{α^-,β^-} satisfies $\alpha^- \leq \alpha$ and $\beta^- \leq \beta$. By exploiting the hierarchical property of (α, β)-dense subgraphs, the computation can then be restricted to the subgraph outside D_{α^+,β^+} or inside D_{α^-,β^-}.

For index construction, we implement the Build-BD-Index algorithm (Algorithm 2). To maintain the BD-Index dynamically, we implement the space-efficient algorithms BD-Insert-S (Algorithm 3) and BD-Delete-S (Algorithm 4), as well as the time-efficient algorithms BD-Insert-T (Algorithm 6) and BD-Delete-T (Algorithm 7). As a baseline comparison, we consider recomputing the BD-Index from scratch using Build-BD-Index after each update, denoted as Recomputing. All algorithms are implemented in C++ with O3 optimization. Our experiments are conducted on a Linux system with a 2.2GHz AMD 3990X 64-Core CPU and 256GB of memory.

Datasets. As shown in Table 1, we evaluate the proposed algorithms on 10 real-world datasets: Actor (AC), IMDB (IM), Hepph (HE), Amazon (AM), Flickr (FL), Epinions (EP), Patent (PA), Pokec (PO), Wiki (WI), and Livejournal (LI). All datasets are publicly available from the Koblenz Network Collection (http://www.konect.cc/).

Table 1: Statistics of datasets.

1K=1,000, 1M=1,000,000

Dataset	Category	$\lvert U\rvert$	$\lvert V\rvert$	$\lvert E\rvert$	p
AC	affiliation	127.8K	383.6K	1.5M	12
IM	affiliation	303.6K	896.3K	3.8M	20
HE	citation	24.5K	28.1K	4.6M	371
AM	rating	2.1M	1.2M	5.7M	23
FL	affiliation	396.0K	103.6K	8.5M	134
EP	rating	120.5K	755.8K	13.7M	120
PA	citation	2.1M	3.3M	16.5M	35
PO	social	1.6M	1.2M	22.3M	25
WI	authorship	953.5K	5.9M	30.6M	156
LI	affiliation	3.2M	7.5M	112.3M	104

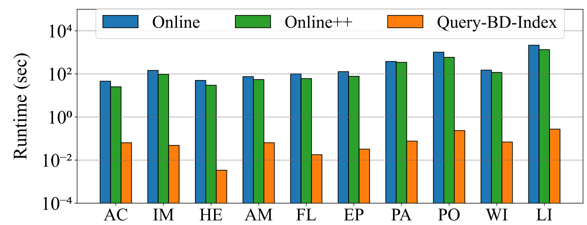
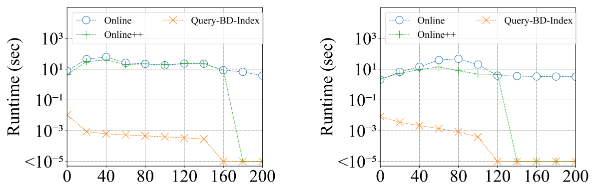


Figure 4: Runtime of different query algorithms (total time of 100 random queries).

6.1 Performance studies on static graphs

Exp-1: Query processing time of different algorithms. In this experiment, we evaluate the performance of (α, β)-dense subgraph query algorithms, including Online, Online++, and Query-BD-Index. For each dataset, we execute 100 queries with α and β parameters uniformly sampled from $[0, p]$, measuring the total processing time. The results are shown in Figure 4. Query-BD-Index achieves a speedup of 3 to 4 orders of magnitude over Online and Online++. For example, on the HE dataset, the running times of Online, Online++, and Query-BD-Index are 49.84 seconds, 29.82 seconds, and 0.0034 seconds, respectively. On the large-scale dataset LI with over 100 million edges, the maximum query time of Online, Online++, and Query-BD-Index across all 100 queries are 60.27 seconds, 39.86 seconds, and only 0.018 seconds, respectively. This striking gap highlights that both online algorithms fail to satisfy the stringent real-time response requirements of practical applications (typically under 0.5 seconds [25]), whereas our index-based method fully meets the requirement. Although Online++ reduces redundant computations and achieves moderate improvements over Online, its performance gain is far from sufficient to meet real-time demands, further underscoring the necessity of our index-based approach.

Exp-2: Query processing time of different algorithms with varying α and β. In this experiment, we evaluate the performance of Online, Online++, and Query-BD-Index across different (α, β) values. The results on the LI dataset are shown in Figure 5 (other datasets exhibit similar trends). As observed, the index-based algorithm Query-BD-Index consistently outperforms the online algorithms across all parameter settings, achieving a speedup of 2 to 5 orders of magnitude. For Online++, it is query-efficient only when (α, β) are large enough such that $D_{\alpha,\beta} = \emptyset$ (where its cache-based



(a) α = 100, varying β.

(b) β = 100, varying α.

Figure 5: Running time with varying α and β on dataset LI.

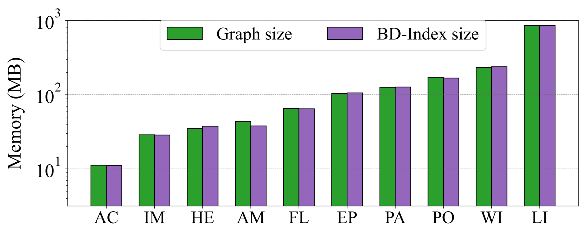


Figure 6: Memory usage of graph and BD-Index**.**

Table 2: The construction time of BD-Index**.**

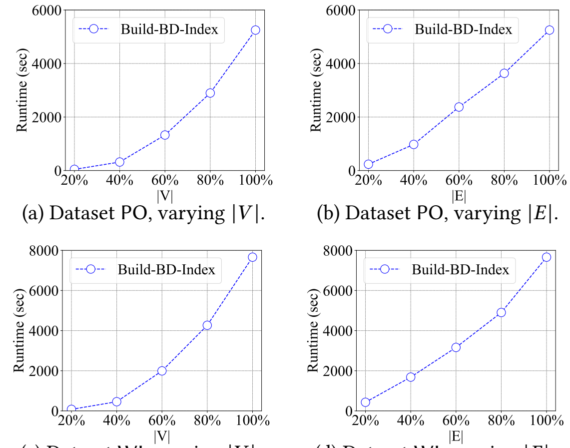
Dataset	Runtime (sec)	Dataset	Runtime (sec)
AC	76.2	EP	3,952.0
IM	469.2	PA	2,198.0
HE	2,646.6	PO	5,250.7
AM	405.7	WI	7,659.8
FL	3,010.5	LI	65,194.9

optimization is effective); when (α, β) are smaller, its performance is as slow as Online. These results demonstrate that, compared to the online algorithms, Query-BD-Index maintains high efficiency across all (α, β) combinations.

Exp-3: Index space usage. This experiment evaluates the memory consumption of BD-Index compared to the graph size. Given that each edge requires storing two endpoints (4 bytes each), we compute the graph size as $8|E|$ bytes. Figure 6 shows the comparative results. As seen, BD-Index exhibits near-identical memory requirements to the graph size across all datasets. For example, on dataset LI, the graph occupies 856.3MB while BD-Index requires 856.8MB. These results align well with the theoretical $O(|E|)$ space complexity of BD-Index, highlighting its highly space-efficient advantage.

Exp-4: Index construction time. Table 2 presents the construction times of BD-Index using the Build-BD-Index algorithm acorss all datasets. As shown, BD-Index can be efficiently constructed at various scales. For smaller datasets such as AC and IM, the index is built in under 500 seconds. For mid-sized graphs like HE, AM, and PA, construction completes in a few thousand seconds. Notably, even on the largest dataset LI, which contains over 112 million edges, the index is constructed in approximately 18 hours (65,194.9 seconds). These results demonstrate that BD-Index can be constructed within reasonable time even for large-scale graphs, and the practical performance of Build-BD-Index significantly outperforms its worst-case time complexity of $O(p \cdot |E|^{1.5} \cdot \log |U \cup V|)$.

Exp-5: Scalability test of index construction. We evaluate the scalability of the Build-BD-Index algorithm by constructing BD-Index for subgraphs containing {20%, 40%, 60%, 80%, 100%} of the original vertices or edges. Figure 7 shows the runtime results



(a) Dataset PO, varying $|V|$.

(b) Dataset PO, varying $|E|$.

(c) Dataset WI, varying $|V|$.

(d) Dataset WI, varying $|E|$.

Figure 7: Scalability test of Build-BD-Index **algorithm.**

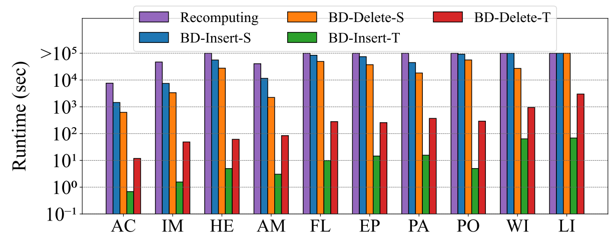


Figure 8: Runtime of maintenance algorithms of BD-Index **(total time for processing 100 random edge deletions and insertions).**

for PO and WI, with other datasets exhibiting similar trends. The runtime increases smoothly and predictably with the growth of $|V|$ and $|E|$, indicating that Build-BD-Index scales well with both graph size dimensions. In all cases, the growth trend remains stable, with no sudden spikes or inefficiencies observed. These results confirm the strong scalability of our Build-BD-Index algorithm.

6.2 Index maintenance on dynamic graphs

Exp-6: Runtime of index maintenance algorithms. Here we evaluate the runtime of maintenance algorithms for BD-Index, including the baseline Recomputing, space-efficient algorithms (BD-Insert-S and BD-Delete-S), and time-efficient algorithms (BD-Insert-T and BD-Delete-T). For each dataset, we perform 100 random edge updates (deletions followed by re-insertions) and measure the total processing time. The results are presented in Figure 8.

As seen, the baseline algorithm Recomputing is extremely slow, completing within the 10^5-second runtime limit only for datasets AC, IM, and AM. It is approximately one order of magnitude slower than the space-efficient algorithms and 2–4 orders of magnitude slower than the time-efficient algorithms. For space-efficient algorithms, BD-Insert-S and BD-Delete-S exhibit comparable runtimes, but both exceed the 10^5-second limit on the large dataset LI. In contrast, the time-efficient algorithms BD-Insert-T and BD-Delete-T are substantially faster, achieving 3–4 orders of magnitude and

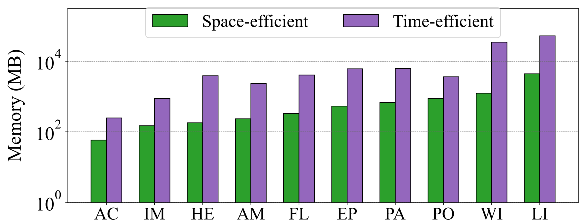
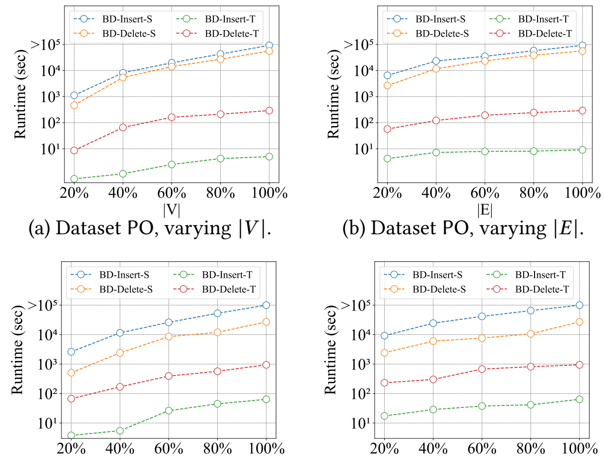


Figure 9: Memory usage of two maintenance approaches.



(a) Dataset PO, varying $|V|$. (b) Dataset PO, varying $|E|$.

(c) Dataset WI, varying $|V|$. (d) Dataset WI, varying $|E|$.

Figure 10: Scalability test of maintenance algorithms (measuring by the total time for processing 100 random edge deletions and insertions).

1–2 orders of magnitude speedups, respectively, over their space-efficient counterparts. For example, on dataset PO, the total runtimes of BD-Insert-S, BD-Delete-S, BD-Insert-T, and BD-Delete-T for processing 100 edge deletions and insertions are 91,860 seconds, 55,972 seconds, 4.9 seconds, and 291 seconds, respectively, corresponding to speedups of 18,747× and 192× for insertion and deletion. These results demonstrate the superior efficiency of our proposed BD-Insert-T and BD-Delete-T.

Additionally, we observe that BD-Insert-T is about an order of magnitude faster than BD-Delete-T. This performance difference stems from the inherent complexity of edge deletion operations in BD-Delete-T, compared to the relatively straightforward implementation of BD-Insert-T. These results show that maintaining BD-Index for edge insertions is more efficient than for deletions, highlighting our algorithm's practical advantages in real-world applications where graph updates primarily consist of edge insertions.

Exp-7: Memory usage of two maintenance approach. We evaluate the memory overhead of two maintenance strategies for BD-Index: the space-efficient approach and the time-efficient approach. The total memory consumption comprises the BD-Index size, the maintenance process overhead, and the storage for egalitarian orientations (used only in the time-efficient approach). The results are shown in Figure 9. As seen, the time-efficient approach consumes 4 to 28 times more memory than the space-efficient one. For example, on the largest dataset LI, the space-efficient method uses 4,413 MB, while the time-efficient method requires 52,597 MB. Notably, even for massive graphs like LI (containing over 100 million edges), the time-efficient method's memory overhead remains practical at approximately 51 GB, well within modern server-grade hardware capacities. On the other hand, in terms of runtime, the time-efficient approach achieves up to four orders of magnitude speedup over the space-efficient method. This highlights that our time-efficient approach offers a favorable time-space trade-off.

Exp-8: Scalability test of index maintenance algorithms. This experiment evaluates the scalability of our maintenance algorithms using the subgraphs generated in Exp-5. For each subgraph, we perform 100 random edge deletions followed by re-insertions, measuring the total runtime. Figure 10 shows the results on PO and WI, with other datasets exhibiting similar trends. As shown, the space-efficient algorithms BD-Insert-S and BD-Delete-S exhibit slow runtimes, and even encounter timeout issues (>10^5 seconds) when the graph becomes large. In contrast, the time-efficient algorithms BD-Insert-T and BD-Delete-T maintain fast performance with gradual runtime increases as graphs grow. These findings demonstrate the superior scalability of the time-efficient algorithms in handling edge updates on large-scale graphs.

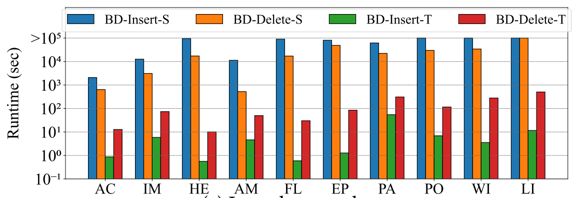
Exp-9: Effect of different edge update strategies. This experiment evaluates the maintenance algorithms under different edge update strategies. We define the degree of an edge as the sum of its endpoints' degrees. The edges are then partitioned into three categories: low-degree (edges with lowest-1/3 degree), medium-degree (edges with middle-1/3 degree), and high-degree (edges with highest-1/3 degree), representing regions with different density in the graph. For each category, we randomly select 100 edges for deletion and re-insertion, forming distinct update strategies. Figure 11 shows the total runtime across all datasets under different update strategies. The time-efficient algorithms BD-Insert-T and BD-Delete-T consistently outperform the space-efficient algorithms BD-Insert-S and BD-Delete-S by 1-5 orders of magnitude. Notably, our proposed algorithms exhibit minimal runtime variation across different edge selection strategies. In particular, the time-efficient algorithms maintain high performance regardless of update edge type, confirming their robustness.

6.3 Case studies

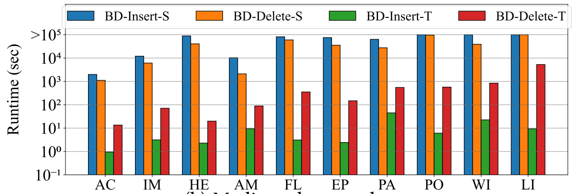
To better simulate the update and query patterns of real streaming systems, we conduct a case study on the real-world temporal AM-app dataset[1], which is an Amazon user–product review network for the appliances category. The dataset contains $|E| = 2.1$M edges, $|U| = 1.7$M users, and $|V| = 94.3$K products, where each edge is associated with a timestamp representing a review.

First, we simulate edge updates in an insertion-only mode: specifically, the latest 2,000 edges (covering the period from 2023-08-13 to 2023-09-13) are incrementally inserted into the graph in chronological order. During the insertion process, we generate 20 query sessions at random time points. Each session lasts 10 seconds with queries distributed randomly over its duration. By varying the number of queries per session, we simulate different query frequencies. In our setup, the system processes queries and updates sequentially in chronological order, and subsequent operations enter a queue and await prior operations' completion. All reported times are therefore measured as turnaround times, defined as the duration from issue to completion of each query or update. The results are presented

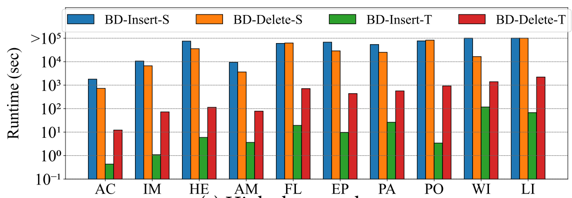
[1]Data source: https://amazon-reviews-2023.github.io/data_processing/0core.html.



(a) Low-degree edges.

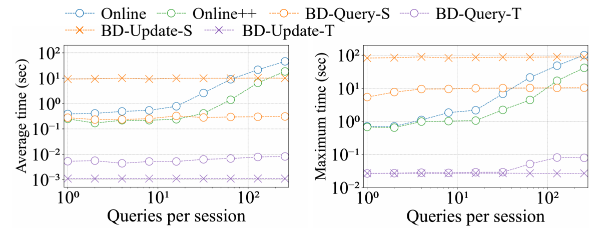


(b) Medium-degree edges.

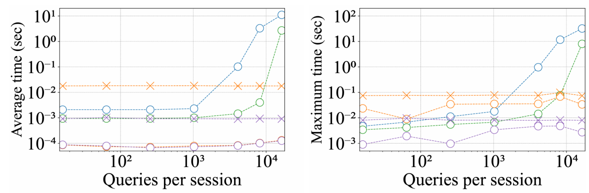


(c) High-degree edges.

Figure 11: Runtime of maintenance algorithms with different edge selection strategies (total time of 100 random edge deletions and insertions).



(a) Average and maximum turnaround time in insertion-only case.



(b) Average and maximum turnaround time in first-in-first-out case.

Figure 12: Query turnaround time and update turnaround time of different algorithms on dataset AM-app**.**

in Figure 12a (note that the online algorithms do not require update time). BD-Query-S and BD-Update-S denote the query and update times of BD-Index under the space-efficient maintenance strategy, while BD-Query-T and BD-Update-T represent the corresponding times under the time-efficient strategy. We observe that as queries become frequent (≈100 queries per session, equivalent to ≈10 queries per second), the average query turnaround times of the online algorithms Online and Online++ quickly rises beyond 10 seconds, and their maximum turnaround times exceeds 40 seconds, indicating severe queuing delays that render them impractical for frequent-query workloads. In contrast, the space-efficient strategy maintains low average query times (≈0.3 seconds), but suffers from high maximum query times (≈10 seconds) due to its slow update processing (queries issued immediately after an update must wait for the update to complete). The time-efficient strategy, on the other hand, achieves consistently low query and update times, with the average and maximum query times remaining below 0.01 seconds and 0.1 seconds, respectively.

Second, we consider another update mode, first-in-first-out. Starting from an empty graph, we insert 100,000 edges in chronological order (spanning from 2002-11-18 to 2014-07-01) while maintaining a sliding time window of one year (i.e., edges older than one year are removed during the insertion process). The query generation follows the same procedure as in the insertion-only case, and the results are presented in Figure 12b. As the query frequency increases (≈10^4 queries per session, equivalent to ≈10^3 queries per second), the maximum query turnaround time of Online and Online++ rise dramatically to 32 seconds and 8 seconds, respectively. In contrast, the query turnaround times of the space-efficient and time-efficient strategies never exceed 0.1 seconds and 0.01 seconds, respectively. This efficiency stems from the fact that BD-Index achieves optimal $O(|R|)$ query time, enabling direct retrieval of the target subgraph without redundant computation.

7 RELATED WORKS

Cohesive subgraph models in bipartite graphs. There exists a wide range of cohesive models for bipartite graphs, such as biclique [10, 29, 42, 44] and its relaxed variant, the k-biplex [13, 45, 46], as well as quasi-biclique models [19, 27, 37]. In addition, butterfly-based model k-bitruss [40, 41, 51] and degree-based model (α, β)-core [16, 26, 28] have also been studied. For identifying the densest regions of a graph, the densest subgraph model is widely used [8, 22, 31, 32]. In terms of connectivity, the k-neighbor connectivity model has been proposed [23]. Recently, the (α, β)-dense subgraph model was introduced [50] to effectively capture bipartite graph density structures with relatively efficient computation. For dynamic graphs, there exist algorithms for maintaining bicliques [15] and (α, β)-cores [28]. To the best of our knowledge, we are the first to investigate efficient solutions for (α, β)-dense subgraph search in both static and dynamic graphs.

Density-based subgraph models in unipartite graphs. In unipartite graphs, the most fundamental problems related to graph density is the densest subgraph search problem [6, 8, 9, 18], which aims to find a subgraph that maximizes the ratio between the number of edges and the number of nodes. To address diverse application requirements, this problem has been extended to several variants, such as locally-dense decomposition [38], top-k densest subgraphs [30, 34], anchored densest subgraphs [14, 43], densest k-subgraph [2, 7], and density decomposition [49]. In dynamic settings, many algorithms have been proposed for maintaining density-based structures, such as for the densest subgraph [3, 4, 17, 35, 36], top-k densest subgraphs [33], and density decomposition [49]. However, existing techniques designed for unipartite settings cannot be directly applied to the (α, β)-dense subgraph search problem.

8 CONCLUSION

This paper studies the problem of efficient (α, β)-dense subgraph search and maintenance in bipartite graphs. First, leveraging the hierarchical property of (α, β)-dense subgraphs, we introduce the concepts of α-rank and β-rank to capture their inclusion relationships.

Using these ranks, we organize nodes into p compact node lists, forming a novel index structure called BD-Index, which achieves optimal query time $O(|D_{\alpha,\beta}|)$ with linear space complexity $O(|E|)$. We also propose an index construction algorithm with time complexity $O(p \cdot |E|^{1.5} \cdot \log |U \cup V|)$. To handle dynamic updates, we present two maintenance strategies. The space-efficient method maintains the index in $O(p \cdot |E|^{1.5})$ time and $O(|E|)$ space. The time-efficient method employs egalitarian orientations to reduce update time to $O(p \cdot |E|)$ while using $O(p \cdot |E|)$ space. Experiments on 10 real-world datasets demonstrate the efficiency and scalability of our solutions in both static and dynamic graphs.

REFERENCES

[1] Mohammad Allahbakhsh, Aleksandar Ignjatovic, Boualem Benatallah, Amin Beheshti, Elisa Bertino, and Norman Foo. 2013. Collusion Detection in Online Rating Systems. In *Web Technologies and Applications - 15th Asia-Pacific Web Conference, APWeb 2013, Sydney, Australia, April 4-6, 2013. Proceedings (Lecture Notes in Computer Science)*, Yoshiharu Ishikawa, Jianzhong Li, Wei Wang, Rui Zhang, and Wenjie Zhang (Eds.), Vol. 7808. Springer, 196–207. https://doi.org/10.1007/978-3-642-37401-2_21

[2] Yuichi Asahiro, Kazuo Iwama, Hisao Tamaki, and Takeshi Tokuyama. 2000. Greedily Finding a Dense Subgraph. *J. Algorithms* 34, 2 (2000), 203–221.

[3] Bahman Bahmani, Ashish Goel, and Kamesh Munagala. 2014. Efficient Primal-Dual Graph Algorithms for MapReduce. In *WAW 2014*, Vol. 8882. Springer, 59–78.

[4] Bahman Bahmani, Ravi Kumar, and Sergei Vassilvitskii. 2012. Densest Subgraph in Streaming and MapReduce. *Proc. VLDB Endow.* 5, 5 (2012), 454–465.

[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 *WWW 2013*. 119–130.

[6] Digvijay Boob, Yu Gao, Richard Peng, Saurabh Sawlani, Charalampos E. Tsourakakis, Di Wang, and Junxing Wang. 2020. Flowless: Extracting Densest Subgraphs Without Flow Computations. In *WWW*. 573–583.

[7] Nicolas Bourgeois, Aristotelis Giannakos, Giorgio Lucarelli, Ioannis Milis, and Vangelis Th Paschos. 2013. Exact and approximation algorithms for densest k-subgraph. In *WALCOM 2013*. Springer, 114–125.

[8] Moses Charikar. 2000. Greedy approximation algorithms for finding dense components in a graph. In *APPROX 2000s*, Vol. 1913. Springer, 84–95.

[9] Chandra Chekuri, Kent Quanrud, and Manuel R. Torres. 2022. Densest Subgraph: Supermodularity, Iterative Peeling, and Flow. In *SODA*. SIAM, 1531–1555.

[10] Jiujian Chen, Kai Wang, Ronghua Li, Hongchao Qin, Xuemin Lin, and Guoren Wang. 2024. Maximal Biclique Enumeration: A Prefix Tree Based Approach. In *ICDE 2024*. IEEE, 2544–2556.

[11] Calvin Chi, Yuting Ye, Bin Chen, and Haiyan Huang. 2021. Bipartite graph-based approach for clustering of cell lines by gene expression-drug response associations. *Bioinform.* 37, 17 (2021), 2617–2626.

[12] Francesco Colace, Massimo De Santo, Luca Greco, Vincenzo Moscato, and Antonio Picariello. 2015. A collaborative user-centered framework for recommending items in Online Social Networks. *Comput. Hum. Behav.* 51 (2015), 694–704.

[13] Qiangqiang Dai, Rong-Hua Li, Donghang Cui, Meihao Liao, Yu-Xuan Qiu, and Guoren Wang. 2024. Efficient Maximal Biplex Enumerations with Improved Worst-Case Time Guarantee. *Proc. ACM Manag. Data* 2, 3 (2024), 135.

[14] Yizhou Dai, Miao Qiao, and Lijun Chang. 2022. Anchored Densest Subgraph. In *SIGMOD 2022*. ACM, 1200–1213.

[15] Apurba Das and Srikanta Tirthapura. 2018. Incremental Maintenance of Maximal Bicliques in a Dynamic Bipartite Graph. *IEEE Trans. Multi Scale Comput. Syst.* 4, 3 (2018), 231–242.

[16] Danhao Ding, Hui Li, Zhipeng Huang, and Nikos Mamoulis. 2017. Efficient Fault-Tolerant Group Recommendation Using alpha-beta-core. In *CIKM 2017*. ACM, 2047–2050.

[17] Alessandro Epasto, Silvio Lattanzi, and Mauro Sozio. 2015. Efficient Densest Subgraph Computation in Evolving Graphs. In *WWW 2015*. ACM, 300–310.

[18] Andrew V Goldberg. 1984. *Finding a maximum density subgraph.* Technical Report. University of California Berkeley, Berkeley, CA, USA.

[19] Dmitry I. Ignatov. 2019. Preliminary Results on Mixed Integer Programming for Searching Maximum Quasi-Bicliques and Large Dense Biclusters. In *ICFCA*, Vol. 2378. 28–32.

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

[21] Barrie Kersbergen, Olivier Sprangers, and Sebastian Schelter. 2022. Serenade - Low-Latency Session-Based Recommendation in e-Commerce at Scale. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 150–159. https://doi.org/10.1145/3514221.3517901

[22] Samir Khuller and Barna Saha. 2009. On Finding Dense Subgraphs. In *ICALP 2009*, Vol. 5555. Springer, 597–608.

[23] Ravi Kumar, Andrew Tomkins, and Erik Vee. 2008. Connectivity structure of bipartite graphs via the KNC-plot. In *WSDM 2008*. ACM, 129–138.

[24] Michael Ley. 2002. The DBLP Computer Science Bibliography: Evolution, Research Issues, Perspectives. In *SPIRE 2002*, Vol. 2476. Springer, 1–10.

[25] Greg Linden, Brent Smith, and Jeremy York. 2003. Amazon.com Recommendations: Item-to-Item Collaborative Filtering. *IEEE Internet Comput.* 7, 1 (2003), 76–80. https://doi.org/10.1109/MIC.2003.1167344

[26] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. 2019. Efficient (a,β)-core Computation: an Index-based Approach. In *WWW 2019*. ACM, 1130–1141.

[27] Xiaowen Liu, Jinyan Li, and Lusheng Wang. 2008. Quasi-bicliques: Complexity and Binding Pairs. In *COCOON*, Vol. 5092. 255–264.

[28] Wensheng Luo, Qiaoyuan Yang, Yixiang Fang, and Xu Zhou. 2023. Efficient Core Maintenance in Large Bipartite Graphs. *Proc. ACM Manag. Data* 1, 3 (2023), 208:1–208:26.

[29] Bingqing Lyu, Lu Qin, Xuemin Lin, Ying Zhang, Zhengping Qian, and Jingren Zhou. 2020. Maximum Biclique Search at Billion Scale. *Proc. VLDB Endow.* 13, 9 (2020), 1359–1372.

[30] Chenhao Ma, Reynold Cheng, Laks V. S. Lakshmanan, and Xiaolin Han. 2022. Finding Locally Densest Subgraphs: A Convex Programming Approach. *Proc. VLDB Endow.* 15, 11 (2022), 2719–2732.

[31] Chenhao Ma, Yixiang Fang, Reynold Cheng, Laks V. S. Lakshmanan, and Xiaolin Han. 2022. A Convex-Programming Approach for Efficient Directed Densest Subgraph Discovery. In *SIGMOD 2022*. ACM, 845–859.

[32] Chenhao Ma, Yixiang Fang, Reynold Cheng, Laks V. S. Lakshmanan, Wenjie Zhang, and Xuemin Lin. 2020. Efficient Algorithms for Densest Subgraph Discovery on Large Directed Graphs. In *SIGMOD 2020*. ACM, 1051–1066.

[33] Muhammad Anis Uddin Nasir, Aristides Gionis, Gianmarco De Francisci Morales, and Sarunas Girdzijauskas. 2017. Fully Dynamic Algorithm for Top-k Densest Subgraphs. In *CIKM 2017*. ACM, 1817–1826.

[34] Lu Qin, Rong-Hua Li, Lijun Chang, and Chengqi Zhang. 2015. Locally Densest Subgraph Discovery. In *KDD*. 965–974.

[35] Atish Das Sarma, Ashwin Lall, Danupon Nanongkai, and Amitabh Trehan. 2012. Dense Subgraphs on Dynamic Networks. In *DISC 2012*, Vol. 7611. Springer, 151–165.

[36] Saurabh Sawlani and Junxing Wang. 2020. Near-optimal fully dynamic densest subgraph. In *STOC 2020*. ACM, 181–193.

[37] Kelvin Sim, Jinyan Li, Vivekanand Gopalkrishnan, and Guimei Liu. 2009. Mining maximal quasi-bicliques: Novel algorithm and applications in the stock market and protein networks. *Stat. Anal. Data Min.* 2, 4 (2009), 255–273.

[38] Nikolaj Tatti. 2019. Density-Friendly Graph Decomposition. *ACM Trans. Knowl. Discov. Data* 13, 5 (2019), 54:1–54:29.

[39] Jun Wang, Arjen P. de Vries, and Marcel J. T. Reinders. 2006. Unifying user-based and item-based collaborative filtering approaches by similarity fusion. In *SIGIR 2006*. ACM, 501–508.

[40] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2020. Efficient Bitruss Decomposition for Large-scale Bipartite Graphs. In *ICDE 2020*. IEEE, 661–672.

[41] Yue Wang, Ruiqi Xu, Xun Jian, Alexander Zhou, and Lei Chen. 2022. Towards Distributed Bitruss Decomposition on Bipartite Graphs. *Proc. VLDB Endow.* 15, 9 (2022), 1889–1901.

[42] Xiaowei Ye, Rong-Hua Li, Qiangqiang Dai, Hongchao Qin, and Guoren Wang. 2023. Efficient Biclique Counting in Large Bipartite Graphs. *Proc. ACM Manag. Data* 1, 1 (2023), 78:1–78:26.

[43] Xiaowei Ye, Rong-Hua Li, Lei Liang, Zhizhen Liu, Longlong Lin, and Guoren Wang. 2024. Efficient and Effective Anchored Densest Subgraph Search: A Convex-programming based Approach. In *KDD 2024*. ACM, 3907–3918.

[44] Ziqi Yin, Qi Zhang, Wentao Zhang, Rong-Hua Li, and Guoren Wang. 2023. Fairness-aware Maximal Biclique Enumeration on Bipartite Graphs. In *ICDE 2023*. IEEE, 1665–1677.

[45] Kaiqiang Yu, Cheng Long, Shengxin Liu, and Da Yan. 2022. Efficient Algorithms for Maximal k-Biplex Enumeration. In *SIGMOD 2022*. ACM, 860–873.

[46] Kaiqiang Yu, Cheng Long, Deepak P, and Tanmoy Chakraborty. 2023. On Efficient Large Maximal Biplex Discovery. *IEEE Trans. Knowl. Data Eng.* 35, 1 (2023), 824–829.

[47] Qiang Zhang, Zhipeng Teng, Disheng Wu, and Jiayin Wang. 2024. An Enhanced Batch Query Architecture in Real-time Recommendation. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management, CIKM 2024, Boise, ID, USA, October 21-25, 2024*, Edoardo Serra and Francesca Spezzano (Eds.). ACM, 5078–5085. https://doi.org/10.1145/3627673.3680034

[48] Yalong Zhang, Ronghua Li, Qi Zhang, Hongchao Qin, Lu Qin, and Guoren Wang. 2024. Efficient Algorithms for Pseudoarboricity Computation in Large Static and Dynamic Graphs. *Proc. VLDB Endow.* 17, 11 (2024), 2722–2734.

[49] Yalong Zhang, Ronghua Li, Qi Zhang, Hongchao Qin, and Guoren Wang. 2024. Efficient Algorithms for Density Decomposition on Large Static and Dynamic Graphs. *Proc. VLDB Endow.* 17, 11 (2024), 2933–2945.

[50] Yalong Zhang, Rong-Hua Li, Qi Zhang, Hongchao Qin, Lu Qin, and Guoren Wang. 2025. Density Decomposition of Bipartite Graphs. *Proc. ACM Manag. Data* 3, 1, Article 30 (Feb. 2025), 25 pages.

[51] Zhaonian Zou. 2016. Bitruss Decomposition of Bipartite Graphs. In *DASFAA 2016*, Vol. 9643. Springer, 218–233.