Efficient Mining of Maximum Biclique over Dynamic Bipartite Graphs

Shengli Sun[‡†], Guanming Jiang[‡†], Peng Xu[‡], Philip S.Yu[§], Weiping Li[‡*]
[‡]*School of Software and Microelectronics, Peking University, China.*
[§]*Department of Computer Science, University of Illinois at Chicago, Chicago*
{slsun,wpli}@ss.pku.edu.cn {jiangguanming,peng.xu}@stu.pku.edu.cn psyu@uic.edu

***Abstract*—Maximum biclique on bipartite graphs finds widespread applications in various fields such as fraud detection etc. Bipartite graphs keep evolving in real life while no algorithm is specifically designed for mining maximum biclique over dynamic graphs. Although this problem can be solved by simply invoking a state-of-the-art static method like *MBC** once graph changes, such an approach of simply re-calculating from scratch is inefficient. To address this issue, we propose an efficient solution for mining maximum biclique on dynamic graphs. Firstly, we improve *MBC** by adopting graph reduction strategy based on (α, β)-core and an upper-bound pruning technique to enhance refreshing efficiency. Efficient maintenance of (α, β)-core is now the key issue. Subsequently, we introduce the novel concept of α/β-order and devise an order-based algorithm to conduct (α, β)-core maintenance. As for the critical task—edge insertion processing, our algorithm demonstrates one to three orders of magnitude advantages over state-of-the-art approach. Thirdly, we provide a theoretical bound on the size of the maximum biclique in the updated graph. An effective mechanism for acquiring a sizable seed is introduced, and the refreshing rate is substantially reduced by estimating the necessity of doing so. Additionally, a local refreshing strategy is proposed, where the search space is confined to a local subgraph. Finally, extensive experiments on real-world datasets demonstrate the effectiveness and scalability of our approach.**

***Index Terms*—graph mining, bipartite graph, dynamic graph, maximum biclique**

I. Introduction

The maximum biclique, a graph model on bipartite graphs, finds significant applications in various fields, including fraud detection [1]–[3], gene expression analysis [4]–[6], and recommendation system [7], [8]. Consequently, the exploration of maximum biclique on bipartite graphs has garnered considerable research attention [9]–[11]. A bipartite graph is denoted by $G = (U, V, E)$ where $U(G)$ and $V(G)$ denote two disjoint vertex sets and $E(G) \in U \times V$ denotes the edge set. A subgraph B is a biclique if it is a complete bipartite subgraph of G, i.e., for every pair $u \in U(B)$ and $v \in V(B)$, there exists $(u, v) \in E(B)$. A biclique is a maximal biclique if it cannot be contained by any other biclique. Maximum biclique is then the maximal biclique with the largest number of edges.

Real-world bipartite graphs are continuously evolving. For instance, in user-item networks [12], [13], relationship between users and items evolves as purchasing behavior occurs; in social networks of users and communities [14], [15], interaction between users drives a change in social networks. Numerous studies have focused on maximal biclique enumeration (MBE) problem over dynamic graphs [16]–[18]. Nevertheless, there has been no specific research addressing the problem of maximum biclique search (MBS) on dynamic graphs.

In this study, we are the first to investigate the problem of mining maximum biclique on dynamic bipartite graphs. Consistent with previous research on dynamic graphs, we focus on edge insertion and deletion for the graph updating mode. Maximum biclique, as the largest maximal biclique, can be obtained through the following approach. Upon an edge insertion or deletion, the maximal biclique set is maintained using a state-of-the-art MBE algorithm, such as [16]. Subsequently, the largest biclique from the maximal biclique set is output. However, this approach is not feasible because the number of maximal bicliques is exponential to the number of vertices in the graph.

An alternative strategy is to recalculate the maximum biclique using a static MBS algorithm after graph updates. The state-of-the-art static MBS algorithm is *MBC** [9], and each invocation of *MBC** is called a *refreshing* in the rest of this article. *MBC** is based on a branch-and-bound strategy, and employs a progressive-bounding framework to estimate the lower bound of the maximum biclique, followed by graph reduction. It is also inefficient simply taking refreshing by invoke *MBC** once the graph changes. Firstly, relying solely on degree of vertex to prune search space, and effective graph reduction measures are lacking, which is critical to the algorithm's performance. Secondly, the performance of *MBC** is closely related to the size of the initial seed. However, the size of the seed returned by the heuristic method in *MBC** cannot be guaranteed. Thirdly, if a sizable seed is found, some refreshing or implementing refreshing on the entire graph become unnecessary.

Recently, (α, β)-core is employed to support personalized maximum biclique search [19]. The (α, β)-core of G consists of two vertex sets $U_S \subseteq U(G)$ and $V_S \subseteq V(G)$ where the subgraph induced by $U_S \cup V_S$ is the maximal subgraph of G, and all the vertices in U_S have degree no less than α and all the vertices in V_S have degree no less than β. As a $(\beta \times \alpha)$-biclique is a subgraph of a (α, β)-core, (α, β)-core can be used for graph reduction. For every combination

† They contribute equally to this work.
* Corresponding author.

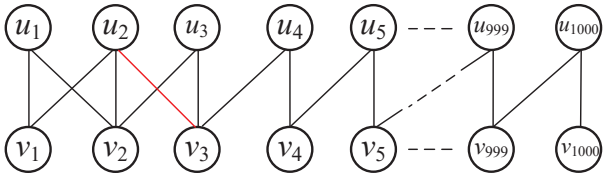


Fig. 1. A sample bipartite graph

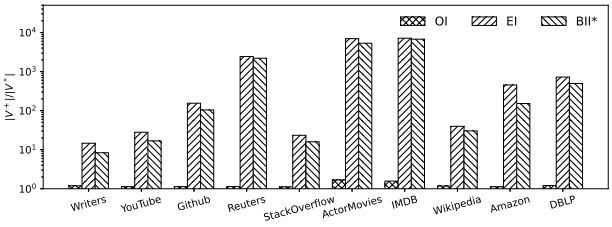


Fig. 2. Proportion of number of visited vertices

of α and β such that some vertex u exists in a (α, β)-core of the graph, the largest value of $(\alpha \times \beta)$ is then the upper bound of the size of the largest biclique that u can take part in. Thus, (α, β)-core can naturally be integrated to *MBC** to enhance refreshing efficiency. Due to the significance of (α, β)-core for graph reduction, efficient maintenance of (α, β)-core is then the key to support MBS on dynamic graphs. However, existing (α, β)-core maintaining algorithms [20], [21] are inefficient and are not applicable to our scenario. Their inefficiency lies in that they need to access too many vertices during the maintenance process, especially when handling edge insertion. For example, considering a sample graph in Fig. 1 where edge (u_2, v_3) is inserted. In order to update (2,2)-core (u_1, u_2, v_1, v_2), algorithm *BII** [20] and *EI* [21] will traverse vertices $u_3, ..., u_{1000}$ and $v_3, ..., v_{1000}$, while our proposed algorithm only visits two vertices, i.e., u_3 and v_3. As shown in the Fig. 2 on real-world large graphs, the number of vertices visited by our algorithm *OI* is 1-4 orders of magnitude less than the existing algorithms *EI* and *BII** when handling edge insertions. And for edge deletions, our algorithm also shows clear advantages. Inspired by core maintenance on unipartite graphs [22], [23], we propose a concept of α/β-order and devise an (α, β)-core maintenance method base on α/β-order. By preserving the α/β-order, computational overhead during core maintenance is substantially reduced and thus it greatly enhances the efficiency of refreshing.

Contributions. To the best of our knowledge, we are the first to address maximum biclique search problem over dynamic bipartite graphs and our principal contributions are as follows:

- *We devised an efficient algorithm, DynamicMBC, to mine maximum biclique over dynamic bipartite graphs using the following strategies.* Firstly, we conducted an in-depth study on the change pattern of the scale of maximum biclique over evolving graphs. We found that the size variation of maximum biclique was no larger than $\lfloor |B_{max}| \rfloor + 1$, where B_{max} was a maximum biclique in the graph. Secondly, we improved the *MBC** algorithm by developing a novel upper-bound-pruning technique based on vertex connectivity relationships to enhance refreshing efficiency. Thirdly, we proposed an effective mechanism to acquire a sizable seed. Refreshing rate is reduced by estimating its necessity and a local refreshing strategy is introduced that search space during refreshing is confined to a local subgraph.
- *We developed a novel order-based algorithm to efficiently maintain (α, β)-core over dynamic bipartite graphs.* The (α, β)-core is fundamental for solving the maximum biclique problem and plays a crucial role for graph reduction during refreshing. Unlike the related works [20], [21], we proposed a novel concept of α/β-order and introduced an order-based algorithm for (α, β)-core maintenance. Besides, we exploited sparsity characteristic of graphs to significantly reduce space overhead of the maintaining algorithm through thoughtful design. Compared with existing algorithms, our algorithm exhibits significant advantages in terms of time cost. Especially when handling edge insertion, our algorithm demonstrates one to three orders of magnitude improvement. The high efficiency of (α, β)-core maintenance makes the (α, β)-core based graph reduction strategy applicable to the task of maximum biclique mining on large dynamic bipartite graphs.
- *Experimental evaluation.* We conducted theoretical analysis and extensive experimental evaluation on our approach. Experiments on 10 real-world graphs demonstrated the effectiveness and scalability of our algorithm.

Organization. We introduce preliminaries and define the problem in Section II. Related work is reviewed in Section III. We present the *Baseline* algorithm for mining maximum biclique on dynamic graphs in Section IV. A novel (α, β)-core maintenance algorithm is detailed in Section V. Our efficient algorithm for mining maximum biclique on dynamic graphs, *DynamicMBC*, is proposed in Section VI. Intensive experiments are conducted in Section VII. We conclude the paper in Section VIII.

II. Preliminaries

In this section, we formally introduce symbols and definitions used in this article, and state the problem we addressing. Table I provides a summary of symbols and notations frequently used in this paper.

TABLE I
List of Symbols

SYMBOL	MEANING
G	A bipartite graph
$(u, v), e$	An edge on a bipartite graph
n, m	The number of vertices and edges on G
$N(u, G)$	The Neighbor set of a vertex u on G
$d(u, G)$	The degree of a vertex u on G
$C_{\alpha,\beta}, C'_{\alpha,\beta}$	An (α, β)-core in graph G and updated graph G'
$\beta_\alpha(u), \alpha_\beta(u)$	The α-core number and β-core number of a vertex u
B, B'	A biclique in G and G'
τ_U, τ_V	Size constraints of the maximum biclique
B_{max}, B'_{max}	A maximum biclique in G and G'
$S_{u,v}$	A subgraph of G induced by $N(u, G) \cup N(v, G)$
H	Seed, a large biclique obtained in the initial step

We consider an undirected and unweighted bipartite graph $G = (U, V, E)$, where $U(G)$ and $V(G)$ denote the two disjoint vertex sets ($U(G) \cup V(G) = \varnothing$) and $E(G) \subseteq U(G) \times V(G)$ denotes the set of edges. The number of vertices and edges

of graph G are denoted by n and m respectively, where $n = |U| + |V|$ and $m = |E|$. The size of graph G is indicated by $|G| = |E(G)|$. The set of neighbors of a vertex u of G is denoted as $N(u, G)$, and the degree of a vertex u is denoted by $d(u, G) = |N(u, G)|$. Additionally, the highest degree across all vertices is denoted as d_{max}. Given two vertex sets $U_S \subseteq U$ and $V_S \subseteq V$, the subgraph of G induced by them is denoted as $S = (U_S, V_S, E_S)$.

The dynamic bipartite graph is considered in this study, and the edge to be inserted (deleted) into (from) the graph is denoted as e or (u, v). The updated graph is denoted as G'.

Definition 1 (biclique). *Given a bipartite graph G, a biclique B is a complete bipartite subgraph of G, i.e., for each pair of $u \in U(B)$ and $v \in V(B)$, $(u, v) \in E(B)$.*

Based on Definition 1, the maximal biclique and the maximum biclique are defined as follows.

Definition 2 (maximal biclique). *Given a bipartite graph G, a biclique is a maximal biclique in G if it cannot be contained by any other biclique.*

Definition 3 (maximum biclique). *Given a bipartite graph G, a maximal biclique B_{max} is the maximum biclique in G if there is no maximal biclique B satisfies $|B| > |B_{max}|$.*

The (α, β)-core plays a crucial role in graph reduction during the process of solving the MBS problem. Here, we present some relevant definitions of the (α, β)-core.

Definition 4 ((α, β)-core). *Given a bipartite graph $G = (U, V, E)$ and two positive integers α and β, the (α, β)-core of G, denoted by $C_{\alpha,\beta}$, is a maximal subgraph induced by two vertex sets $U_S \subseteq U$ and $V_S \subseteq V$, such that $\forall u \in U_S, d(u, C_{\alpha,\beta}) \geq \alpha$ and $\forall v \in V_S, d(v, C_{\alpha,\beta}) \geq \beta$.*

Definition 5 (α/β-core number). *Given a vertex $u \in U(G) \cup V(G)$ and a positive integer α, u's α-core number, denoted as $\beta_\alpha(u)$, represents the maximum value of β so that u is contained in the corresponding $C_{\alpha,\beta}$. Symmetrically, the β-core number $\alpha_\beta(u)$ of u is the maximum value of α so that u is contained in the corresponding $C_{\alpha,\beta}$.*

A biclique B satisfies $|U(B)| \geq \beta$ and $|V(B)| \geq \alpha$ is a subgraph of the (α, β)-core. Based on Definition 5, we present bicore upper bound below, which is pivotal for graph reduction using (α, β)-core.

Definition 6 (bicore upper bound). *Given a vertex $u \in U(G)$, $cub_u(l, r) = \max_{\alpha \in [l,r]} \alpha \cdot \beta_\alpha(u)$. Given a vertex $v \in V(G)$, $cub_v(l, r) = \max_{\beta \in [l,r]} \beta \cdot \alpha_\beta(v)$.*

Lemma 1. *Given a vertex $u \in U(G)$, for any biclique B containing u and satisfying $\alpha = |V(B)| \in [l, r]$, $|B| \leq cub_u(l, r)$. Similarly, given a vertex $v \in V(G)$, for any biclique B containing v and satisfying $\beta = |U(B)| \in [l, r]$, $|B| \leq cub_v(l, r)$.*

Due to space limitations, most of proofs of lemmas and theorems in this article are omitted thereafter, and they can be found in the full version[1].

[1]https://github.com/wjjw2006/dynamic-maximum-biclique

According to Lemma 1, if we have identified a large biclique B, all vertices with bicore upper bound no larger than $|B|$ can be removed. Utilizing the (α, β)-core, we can efficiently reduce the search space during the refreshing process. Thus, the maintenance of the (α, β)-core is fundamental for mining maximum biclique on dynamic graphs. Similar to [9], we aim to identify the maximum biclique B_{max} that satisfies the size constraints τ_U and τ_V, i.e., $|U(B_{max})| > \tau_U$ and $|V(B_{max})| > \tau_V$. We formulate our problems below.

Problem Statement Given a bipartite graph G, size constraints τ_U and τ_V, and an inserted/deleted edge (u, v), we aim to develop an efficient solution to address two closely related problems: (1) once an edge is inserted (deleted) into (from) G, the graph evolves from G to G'. Then, we report the size of the maximum biclique on G' and return a maximum biclique of G'; (2) after an edge insertion/deletion, we maintain (α, β)-core on G' to support graph reduction during refreshing.

III. RELATED WORK

Cohesive subgraph mining on bipartite graphs. The (α, β)-core concept was initially introduced in [24]. Subsequent studies, such as [3] and [25] extended the linear time complexity algorithm for mining k-cores in uni-partite graphs to bipartite graphs, introducing an online computation algorithm for (α, β)-cores. [26] proposed an index-based approach to compute (α, β)-cores, reducing query time through offline index construction. Considering the dynamic nature of real-world bipartite graphs, [20] examined the variations of (α, β)-cores on dynamic bipartite graphs and introduced a locality-based (α, β)-core maintenance algorithm. [21] suggested maintaining (α, β)-core via bicore number maintenance.

The maximum biclique search problem is concerned with identifying the biclique with the maximum number of edges over bipartite graphs, it is a NP-complete problem as demonstrated in [10]. [27] and [11] were early explorations of MBS problem, but they do not apply to large-scale graphs. [9] proposed a progressive boundary framework and graph reduction techniques to reduce the search space for addressing the MBS problem on large-scale graphs. Furthermore, [19] investigated a related problem to identify a maximum biclique containing a given query vertex, and proposed an algorithm *PMBC-OL** to address this problem. *PMBC-OL** performs a search in a small subgraph around the specific query vertex, while *MBC** performs a search in the entire graph. Our goal in this article is consistent with algorithm *MBC**, that is to find the largest biclique on the global graph, but the scenario we considered is a dynamic graph.

Cohesive subgraph mining on dynamic uni-partite graphs. A k-core represents a prominent graph model over uni-partite graphs. [28] introduced a quadratic algorithm for this purpose, while [29] proposed the algorithm *Traversal* that are linear in the size of the subcore. [22] and [23] further reduces the search space during maintenance through k-order. Nonetheless, these algorithms are not directly applicable to (α, β)-core maintenance due to structure difference in the graph

Algorithm 1: *Baseline*

Input: G, τ_U, τ_V, graph updating operator
Output: B'_{max}
compute (α, β)-core and bicore upper bound offline;
acquire B_{max} on G;
repeat
  receive the graph updating operator;
  update G to reach G';
  update (α, β)-core and bicore upper bound;
  if the operator is insertion of edge (u, v) **then**
    $H \leftarrow B_{max}$;
    $B'_{max} \leftarrow$ *IMBC**(G', H, τ_U, τ_V);
  else // edge deletion processing
    if $\{u, v\} \subseteq B_{max}$ **then**
      $H_1 \leftarrow \varnothing$, $H_2 \leftarrow \varnothing$;
      if $U(B_{max}) > \tau_U$ **then** $H_1 \leftarrow B_{max} \setminus \{u\}$;
      if $V(B_{max}) > \tau_V$ **then** $H_2 \leftarrow B_{max} \setminus \{v\}$;
      $H \leftarrow$ the larger one between H_1 and H_2;
      $B'_{max} \leftarrow$ *IMBC**(G', H, τ_U, τ_V);
    else
      $B'_{max} \leftarrow B_{max}$;
  report $|B'_{max}|$ and B'_{max};
  $B_{max} \leftarrow B'_{max}$;
until termination of graph updating;

models. Specifically, during the maintenance of (α, β)-core, the change of vertices' core number may exceed 1 and the updating conditions of core number of the vertices on two layers are different.

Clique also stands as a classical cohesive graph model in uni-partite graphs. The maximum clique search (MCS) problem is a long-standing NP-hard problem. [30] stands as the sole study devoted to addressing MCS on dynamic graphs up to now. This work established a theoretical bound on size variation of the maximum clique when edge insertion/deletion happens, and proposed innovative measures to reduce refreshing rate and refreshing overhead. This work gives inspiration to our algorithm. Nevertheless, the structure of bipartite graph and unipartite graph is obviously different, maximum biclique mining on dynamic bipartite graph has its own unique challenges. The magnitude of change of maximum clique/biclique varies substantially between unipartite and bipartite graph. Effective measures should be carefully designed to efficiently support maximum biclique search.

IV. The Baseline Algorithm for Mining Maximum Biclique on Dynamic Bipartite Graphs

In this section, we propose a baseline algorithm for mining maximum biclique on dynamic bipartite graphs. In order to enhance refreshing efficiency, we improve *MBC** by adopting graph reduction strategy based on (α, β)-core and an upper-bound pruning technique. In addition, we get a sizable seed at a lower cost, which makes refreshing work at a higher starting point.

A. The Baseline Algorithm

To provide a reasonable baseline algorithm, we no longer acquire a seed through the heuristic method used in *MBC** but instead by updating the previous maximum biclique to get a sizable seed.

Lemma 2. *Given a bipartite graph G and an inserted edge (u, v), B_{max} is at least a biclique on G'.*

Algorithm 2: *BranchUB*$(U', V', C'_V, \tau_U, \tau_V)$

$S[i] \leftarrow 0$ for $i = 1$ to $|U'|$;
for each $v \in C'_V$ **do**
  $k \leftarrow |N(v, G) \cap U'|$;
  $S[i] \leftarrow S[i] + 1$ for $i = 1$ to k;
$ans \leftarrow 0$;
for $i = \tau_U$ **to** $|U'|$ **do**
  if $|V'| + S[i] < \tau_V$ **then break**;
  $ans \leftarrow max\{ans, i \times (|V'| + S[i])\}$;
return ans;

Algorithm 3: *IMBC**

Input: G, a large seed H, size constraints τ_U and τ_V
Output: maximum biclique B_{max}
$B_0^* \leftarrow H$;
if $B_0^* = \varnothing$ **then** $B_0^* \leftarrow$ *InitMBC*(G, τ_U, τ_V) in *MBC**;
$\tau_V^0 \leftarrow dmaxU(G)$; $k \leftarrow 0$;
while $\tau_V^k > \tau_V$ **do**
  compute τ_U^{k+1} and τ_V^{k+1} according to *MBC**;
  $G_{k+1} \leftarrow$ Reduce$(G, \tau_U^{k+1}, \tau_V^{k+1})$, remove vertices with bicore upper bound not greater than $|B_k^*|$ in Reduce1Hop;
  $B_{k+1}^* \leftarrow$BranchBound$(U(G_{k+1}), \varnothing, V(G_{k+1}), \varnothing, B_k^*, \tau_U^{k+1}, \tau_V^{k+1})$;
  $k \leftarrow k + 1$;
return B_k^*;
procedure BranchBound$(U, V, C_V, X_V, B^*, \tau_U, \tau_V)$
  if $|V| \geq \tau_V$ **and** $|U| \times |V| > |B^*|$ **then**
    $B^* \leftarrow (U, V, U \times V)$;
  while $C_V \neq \varnothing$ **do**
    $v^* \leftarrow C_V$.pop();
    if $cub_{v^*}(\tau_U, |U|) \leq |B^*|$ **then**
      $X_V \leftarrow X_V \cup \{v^*\}$; **continue**;
    $U' \leftarrow \{u \in U \cap N(v^*, G) | cub_u(|V|, d(u, G)) > |B^*|\}$;
    compute V', C'_V, and X'_V according to *MBC**;
    if U', V', C'_V, and X'_V can lead to a larger biclique according to *MBC** and *BranchUB*$(U', V', C'_V, \tau_U, \tau_V) > |B^*|$ **then** BranchBound$(U', V', C'_V, X'_V, B^*, \tau_U, \tau_V)$;
    $X_V \leftarrow X_V \cup \{v^*\}$;
  return B^*;

Lemma 3. *Given a bipartite graph G and a deleted edge (u, v), if $\{u, v\} \nsubseteq B_{max}$, B_{max} is still a maximum biclique on G'. Otherwise, each of $B_{max} \setminus \{u\}$ and $B_{max} \setminus \{v\}$ is at least a biclique on G'.*

According to Lemma 2 and Lemma 3, a relatively large seed can be rapidly obtained from the previous maximum biclique. In the case of edge insertion, B_{max} can serve as an initial seed H. In the case of edge deletion, if B_{max} is not hit by the deleted edge, it remains a maximum biclique on G'; otherwise, considering the size constraints, if $|U(B_{max})| > \tau_U \wedge |V(B_{max})| > \tau_V$, both $B_{max} \setminus \{u\}$ and $B_{max} \setminus \{v\}$ are legal, and then we select the larger one of them as the seed. If only one of the conditions $|U(B_{max})| = \tau_U$ and $|V(B_{max})| = \tau_V$ holds, then $B_{max} \setminus \{u\}$ or $B_{max} \setminus \{v\}$ is legal, we choose the legal one as the seed. Only when $|U(B_{max})| = \tau_U \wedge V(|B_{max}|) = \tau_V$, both $B_{max} \setminus \{u\}$ and $B_{max} \setminus \{v\}$ are illegal. Therefore, we output an empty set and use the heuristic method in *IMBC** to get a valid seed.

Based on this strategy, the pseudocode of our algorithm *Baseline* is presented in Algorithm 1.

B. Improvements to the MBC* Algorithm

As *MBC** does not utilize (α, β)-core for graph reduction, we introduce an (α, β)-core based reduction strategy into it. According to Definition 6 and Lemma 1 in Section II, and taking into account size constraints τ_U and τ_V, we introduce graph reduction based on (α, β)-core into both the One-Hop Graph Reduction and the branch-and-bound process of

the *MBC** algorithm in the following manner. When we have obtained a sub-optimal solution B^*, in the One-Hop Graph Reduction phase, we consider not only the vertex's degree but also remove all vertices $u \in U(G_i)$ satisfying $cub_u(\tau_V^i, d(u, G_i)) \leq |B^*|$ and vertices $v \in V(G_i)$ satisfying $cub_v(\tau_U^i, d(v, G_i)) \leq |B^*|$ from G_i. In the BranchBound procedure, $(U, V, U \times V)$ defines a partial biclique. C_V is the set of candidate vertices that can be possibly added to V, and X_V is the set of vertices that has been used and should be excluded from V. If a candidate vertex v^* satisfies $cub_{v^*}(\tau_U, |U|) \leq |B^*|$, we can skip v^*. In addition, while computing U', we can exclude all vertices u from U that do not satisfy $cub_u(|V|, d(u, G)) > |B^*|$. The query for bicore upper bound is a range maximum query problem. In this article, we use the Sparse Table algorithm [31] to handle queries for bicore upper bound. Sparse Table algorithm requires preprocessing time and space overhead of $O\left(\sum_{x \in U(G) \cup V(G)} (d(x, G) \cdot \log(d(x, G)))\right)$ and supports querying the bicore upper bound of any vertex in $O(1)$ time.

When we consider a specific search branch, the α/β-core number of a vertex may be influenced by vertices outside the branch, leading to the potential failure of the bicore upper bound. However, it is relatively costly to perform core decomposition on the subgraph corresponding to each branch. To minimize the number of branches while balancing the costs, we propose an upper-bound pruning technique, which is based on the definition of branch upper bound.

Definition 7 (branch upper bound). Given a search branch (U', V', C'_V, X'_V), the upper bound on the size of a potential biclique this branch can generate is $\max_{i \in [1, |U'|]} i \times (|V'| + S[i])$, where $S[i] = |\{v | v \in C'_V \wedge |N(v, G) \cap U'| \geq i\}|$.

To enhance *MBC**, we can directly replace $|U'| \times (|V'| + |C'_V|)$ with branch upper bound in size pruning rule. This strategy is called an upper-bound pruning technique. The upper bound $|U'| \times (|V'| + |C'_V|)$ simply assumes that U' and $V' \cup C'_V$ can induce a biclique, while our branch upper bound consider specific connection relationships of vertices in C'_V within a branch. We use Example 1 to further illustrate the effectiveness of the branch upper bound strategy.

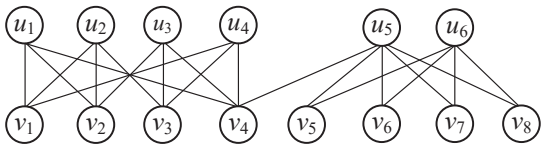


Fig. 3. Running example for branch upper bound

Example 1. *We consider a search branch that $V' = \{v_2\}$, $U' = \{u_1, u_2, u_3\}$, and $C'_V = \{v_1, v_3, v_4\}$ in Fig. 3. Assuming that we have found a biclique B satisfying $|B| = 8$, we cannot remove any vertex in this branch due to the bicore upper bound of each vertex is 9. Using $|U'| \times (|V'| + |C'_V|) = 12$ as the upper bound calculation result, we cannot skip this branch. However, as $S[1] = 3$, $S[2] = 3$, and $S[3] = 0$, the branch upper bound of this branch is $8 \leq |B|$. Thus, we can skip this branch.*

Considering the size constraints, Algorithm 2 provides the computation process for branch upper bound. For each vertex $v \in C'_V$, we calculate its number of neighbors within the branch and denote it as k. Then, for i from 1 to k, we increment $S[i]$ by 1 according to Definition 7 (lines 2-4). In lines 6-8, we calculate the branch upper bound and exclude the illegal cases of $i < \tau_U$ or $|V'| + S[i] < \tau_V$. As the maximum number of vertices in U' is d_{max} and the maximum degree of vertices in the graph is also d_{max}, Algorithm 2 can run in $O(d_{max}^2)$ time.

Based on the enhancements mentioned above, Algorithm 3 shows the details of improved algorithm *IMBC**. Compared with *MBC**, *IMBC** introduces graph reduction based on (α, β)-core in lines 6 and 15-17, and introduces an upper-bound pruning technique in line 19.

Thereafter, we use *IMBC** to implement refreshing.

V. A Novel Algorithm for (α, β)-core Maintenance

Maintaining (α, β)-core is a crucial step for mining maximum biclique on dynamic graphs. Due to the inadequate performance of existing algorithms, in this section, we propose an order-based (α, β)-core maintenance algorithm to efficiently support refreshing. First, we provide theoretical foundation of our algorithm. Then, we propose *Order-Insert* algorithm for handling edge insertion and *Order-Delete* algorithm for handling edge deletion. Finally, we rely on graph sparsity to reduce the space overhead of the algorithm.

A. Theoretical Foundation

We initially analyze the dynamic updating property of α-core number and β-core number when an edge insertion or deletion occurs, and obtained Lemmas 4 to 6. Due to the symmetric nature of α-core number and β-core number, we focus on illustrating α-core number in the sequel.

Lemma 4. *Given a bipartite graph G and an inserted (deleted) edge (u, v), for any positive integer α, let $b_\alpha = \max x \; s.t. \; |\{w \mid w \in N(u, G') \wedge \beta_\alpha(w) \geq x\}| \geq \alpha$. If such x does not exist, then $b_\alpha = 0$. Then $\beta_\alpha(u)$ needs to be updated to b_α.*

Lemma 5. *Given a bipartite graph G and an inserted (deleted) edge (u, v), for any integer α and $w \in U(G) \cup V(G)$ except for u, $\beta_\alpha(w)$ can change at most 1.*

Lemma 4 and Lemma 5 bound the updating range of α-core number. Next, Lemma 6 tells us which vertices' α-core number may be updated.

Lemma 6. *Given a bipartite graph G and an inserted (deleted) edge (u, v), for any integer α, let $\beta = \min\{b_\alpha, \beta_\alpha(v)\}$ ($\beta = \min\{\beta_\alpha(u), \beta_\alpha(v)\}$). It follows that the α-core number of a vertex w can only change if $\beta_\alpha(w) = \beta$ and there exists a reachable path from w to either u or v where all vertices along the path have an α-core number equal to β.*

Lemma 6 restricts the range of vertices with their α-core number updated to a connected subgraph near the edge (u, v) when inserting or deleting the edge.

Thereafter, we focus on the order of removing vertices during core decomposition, and define α-order and β-order.

Based on this, we design the order-based algorithm for (α, β)-core maintenance.

Definition 8 (α/β-order). *Given a bipartite graph G, for a fixed α and any pair of vertices x and y, the definition of α-order $\preceq_\alpha$ is as follows: If $\beta_\alpha(x) < \beta_\alpha(y)$, then $x \preceq_\alpha y$; if $\beta_\alpha(x) = \beta_\alpha(y)$, and $\beta_\alpha(x)$ is computed before $\beta_\alpha(y)$ during core decomposition, then $x \preceq_\alpha y$. For a fixed β and any pair of vertices x and y, the definition of β-order $\preceq_\beta$ is as follows: If $\alpha_\beta(x) < \alpha_\beta(y)$, then $x \preceq_\beta y$; if $\alpha_\beta(x) = \alpha_\beta(y)$, and $\alpha_\beta(x)$ is computed before $\alpha_\beta(y)$ during core decomposition, then $x \preceq_\beta y$.*

α/β-order satisfies certain properties. We take α-order as an example. For a fixed α, a vertex sequence $w_1, w_2, \ldots, w_n$ that satisfies α-order is an instance of the many sequences that the core decomposition algorithm can produce. α-order satisfies transitivity, meaning if $x \preceq_\alpha y$ and $y \preceq_\alpha z$, then $x \preceq_\alpha z$. For a vertex x and its neighbor w, if $w \preceq_\alpha x$, w is denoted a predecessor neighbor of x; if $x \preceq_\alpha w$, w is denoted a successor neighbor of x.

Using $Oa_{\alpha,\beta}$ to denote the sequence of vertices in α-order with the α-core number equal to β, we can obtain a vertex sequence $Oa_\alpha : Oa_{\alpha,0}Oa_{\alpha,1}Oa_{\alpha,2}\ldots$, which satisfies $Oa_{\alpha,i} \preceq_\alpha Oa_{\alpha,j}$ for $i < j$. In other words, the α-order can be determined from the sequence Oa_α. Similar properties hold for β-order. For a fixed β, we use $Ob_{\beta,\alpha}$ to represent the sequence of vertices in β-order with the β-core number equal to α. For any α and β, Oa_α and Ob_β can be obtained using algorithm *ComShrDecom* proposed in [26].

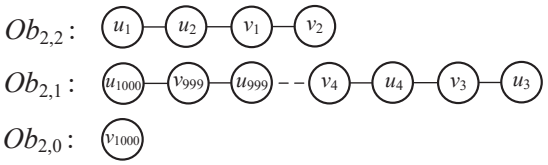


Fig. 4. The β-order for $\beta = 2$ in G shown in Fig. 1

Example 2. *In Fig. 4, we show a β-order for $\beta = 2$ in the graph G without edge insertion (u_2, v_3) (shown in Fig. 1) by showing $Ob_{2,0}$, $Ob_{2,1}$, and $Ob_{2,2}$. Only a vertex v_{1000} is in $Ob_{2,0}$. In $Ob_{2,1}$, $u_{1000} \preceq_\beta v_{999} \preceq_\beta u_{999} \preceq_\beta \ldots \preceq_\beta v_4 \preceq_\beta u_4 \preceq_\beta v_3 \preceq_\beta u_3$. In $Ob_{2,2}$, $u_1 \preceq_\beta u_2 \preceq_\beta v_1 \preceq_\beta v_2$. The β-order for $\beta = 2$ is one of the orders computed by ComShrDecom for the graph G.*

Next, we introduce our algorithms: *Order-Insert* for handling edge insertion and *Order-Delete* for handling edge deletion.

B. Order-Insert Algorithm for Handling Edge Insertion

Given a bipartite graph G, when an edge is inserted, for a given $\alpha(\beta)$, the key to (α, β)-core maintenance is to compute the set of vertices where the α-core number (β-core number) of the vertex must be updated. This set of vertices is denoted as V^*. Additionally, we represent V^+ as the minimal set of vertices that need to be visited repeatedly in order to identify V^*.

The key to improving the efficiency of the insertion algorithm is to minimize V^+ as much as possible while accurately calculating V^*. In order to reduce the size of V^+, we introduce the concept of remaining degree and candidate degree.

Definition 9 (remaining degree). *Given a bipartite graph G, for a fixed α (β) and any vertex x on G, the remaining degree of x $rd_\alpha(x) = |\{y | y \in N(x, G) \wedge x \preceq_\alpha y\}|$ ($rd_\beta(x) = |\{y | y \in N(x, G) \wedge x \preceq_\beta y\}|$). The remaining degree indicates the number of successor neighbors of vertex x in Oa_α (Ob_β) that potentially support the current $\alpha(\beta)$-core number to increase.*

Definition 10 (candidate degree). *Given a bipartite graph G, for a fixed α (β) and any vertex x on G, the candidate degree of x $cd_\alpha(x) = |\{y | y \in N(x, G) \wedge y \preceq_\alpha x \wedge y \in V^*\}|$ ($cd_\beta(x) = |\{y | y \in N(x, G) \wedge y \preceq_\beta x \wedge y \in V^*\}|$). The candidate degree indicates the number of predecessor neighbors of vertex x in Oa_α (Ob_β) that could potentially increase $\alpha(\beta)$-core number.*

The sum of a vertex's remaining degree and candidate degree indicates the number of neighbors supporting its core number to increase. The initial value of a vertex's remaining degree can be computed through the algorithm *ComShrDecom*, which equals the vertex's degree upon removal during the core decomposition. Furthermore, based on the definition of the $\alpha(\beta)$-order and remaining degree, we can derive Lemma 7, offering a guide to $\alpha(\beta)$-order maintenance.

Lemma 7. *Given a bipartite graph G, the α-order is determined by the sequence Oa_α if and only if, for any β, each vertex x in $Oa_{\alpha,\beta}$ satisfies that if $x \in U(G)$, then $rd_\alpha(x) < \alpha$, or if $x \in V(G)$, then $rd_\alpha(x) \leq \beta$. Symmetrically, for a fixed β, the β-order is determined by the sequence Ob_β if and only if, for any α, each vertex x in $Ob_{\beta,\alpha}$ satisfies that if $x \in U(G)$, then $rd_\beta(x) \leq \alpha$, or if $x \in V(G)$, then $rd_\beta(x) < \beta$.*

Based on the aforementioned analysis, the fundamental idea of the *Order-Insert* algorithm can be outlined as follows: taking α-core number update as an example, for each α, once β is established according to Lemma 6, the vertices in $Oa_{\alpha,\beta}$ are sequentially traversed in α-order starting from u (assuming $u \preceq_\alpha v$). When vertex w is visited, the sum $rd_\alpha(w) + cd_\alpha(w)$ is at its maximal. If it satisfies $rd_\alpha(w) + cd_\alpha(w) \geq \alpha$ for $w \in U(G)$, or $rd_\alpha(w) + cd_\alpha(w) > \beta$ for $w \in V(G)$, then w is included in V^* and its successor neighbors wait for traversal; otherwise, w is deemed not to belong to V^*, potentially leading to the removal of other vertices from V^*. Upon traversing all vertices with an α-core number of β, the computation of V^* comes to an end. Following this, the α-core number of all vertices in V^* is increased by 1. Additionally, the maintenance of Oa_α and remaining degree is necessary for all vertices. A similar approach is applied to updating β-core number.

According to the computation-sharing theory [20], when updating the $\alpha(\beta)$-core number for $\alpha(\beta) \leq \delta$, it can simultaneously update the $\beta(\alpha)$-core number for $\beta(\alpha) > \delta$, where δ is the maximum value that keeps $C_{\delta,\delta}$ non-empty, with δ not exceeding $\sqrt{m}$. Therefore, we only need to maintain Oa (Ob) and the remaining degree, and use the above strategy to update core number for α (β) from 1 to δ. In the *Order-Insert* algorithm, we maintain δ by monitoring Oa and Ob;

Algorithm 4: *Order-Insert*

Input: G, an inserted edge (u, v)
Output: Updated α-core number and β-core number on G'

1 $G' \leftarrow G \cup (u, v)$; update δ;
2 **for** $\alpha = 1$ **to** δ **do**
3 $b_\alpha \leftarrow max\ x\ s.t.\ |\{w | w \in N(u, G') \wedge \beta_\alpha(w) \geq x\}| \geq \alpha$;
4 **if** $b_\alpha > \beta_\alpha(u)$ **then**
5 move u from $Oa_{\alpha,\beta_\alpha(u)}$ to the front of Oa_{α,b_α};
6 $\beta_\alpha(u) \leftarrow b_\alpha$;
7 update the rd_α of u and its neighbours;
8 $r \leftarrow u$ (assuming $u \preceq_\alpha v$);
9 $\beta \leftarrow \beta_\alpha(r)$; $rd_\alpha(r) \leftarrow rd_\alpha(r) + 1$;
10 $\mathcal{H} \leftarrow$an empty min-heap by α-order; $V^* \leftarrow \varnothing$; $V^+ \leftarrow \varnothing$;
11 **if** r can be included in V^* **then** $\mathcal{H}$.push(r);
12 **while** $\mathcal{H} \neq \varnothing$ **do**
13 $w \leftarrow \mathcal{H}$.top(); add w to V^+;
14 **if** w can be included in V^* **then**
15 remove w from $Oa_{\alpha,\beta}$ and add it to V^*;
16 **for** each $x \in N(w, G')$ **do**
17 **if** $\beta_\alpha(x) = \beta \wedge w \preceq_\alpha x$ **then**
18 $cd_\alpha(x) \leftarrow cd_\alpha(x) + 1$;
19 **if** $x \notin \mathcal{H}$ **then** $\mathcal{H}$.push(x);
20 **else if** $cd_\alpha(w) \neq 0$ **then**
21 $rd_\alpha(w) \leftarrow rd_\alpha(w) + cd_\alpha(x)$; $cd_\alpha(x) \leftarrow 0$;
22 $p \leftarrow w$;
23 *RemoveCandidates_*$\alpha(G', V^*, \mathcal{H}, w, p, \alpha, \beta)$;
24 **if** $rd_\alpha(u) = \alpha$ **then**
25 $x \leftarrow$ the second vertex in V^*;
26 swap the positions of x and u in V^*;
27 $rd_\alpha(u) \leftarrow rd_\alpha(u) - 1$; $rd_\alpha(x) \leftarrow rd_\alpha(x) + 1$;
28 **for** each $w \in V^*$ **do**
29 $cd_\alpha(w) \leftarrow 0$; $\beta_\alpha(w) \leftarrow \beta_\alpha(w) + 1$;
30 **if** $w \in V(G')$ **then**
31 **for** $i = \beta + 1$ **to** $\delta + 1$ **do**
32 **if** $\alpha_i(w) < \alpha$ **then** $\alpha_i(w) \leftarrow \alpha$;
33 **else break**;
34 add all the vertices in V^* to the front of $Oa_{\alpha,\beta+1}$ in α-order(the order vertices added to V^*)
35 **for** $\beta = 1$ **to** δ **do**
36 lines 3-34 by swapping u, U, α and Oa with v, V, β and Ob;

once δ increases, we also need to obtain $Oa_{\delta+1}$ and $Ob_{\delta+1}$.

Algorithm 4 outlines the steps of the *Order-Insert* algorithm. Initially, graph G and δ get update, followed by maintaining α-core number and β-core number.

The steps for updating α-core number are as follows: for each α, initially, b_α is computed according to Lemma 4, then updating $\beta_\alpha(u)$, and maintaining Oa_α alongside the remaining degree of vertex u and its neighbors (lines 3-7). Subsequently, r is set as the vertex preceding in α-order from u and v. Let $\beta = \beta_\alpha(r)$, and $rd_\alpha(r)$ is increased by 1 to reflect the insertion of the edge (u, v) (lines 8-9). Next, r is added to the min-heap $\mathcal{H}$ (line 10) if r meets the criteria for inclusion in V^* as previously analyzed (lines 10-11). For each vertex w extracted from the top of $\mathcal{H}$, the value of $rd_\alpha(w) + cd_\alpha(w)$ is computed. If it meets the criteria for inclusion in V^*, w is added to both V^* and V^+. Subsequently, for each successor neighbor x of w with $\beta_\alpha(x) = \beta$, $cd_\alpha(x)$ increases by 1, and x is added to $\mathcal{H}$ for pending processing (lines 14-19). If the condition is not met, w is solely added to V^+, $rd_\alpha(w)$ is updated, and vertices in V^* no longer satisfying the criteria are handled by the *RemoveCandidates_*α procedure (lines 20-23). When $\mathcal{H}$ is empty, the computation of V^* concludes. In the final stage, special cases where $rd_\alpha(u) = \alpha$ might lead to non-satisfaction with Lemma 7 are addressed initially (lines 24-27). Then we update the α-core number and β-core number of vertices in V^*, and ultimately maintain Oa_α (lines 28-34). The steps for updating β-core number are similar to those for updating α-core number (lines 35-36).

Algorithm 5: *RemoveCandidates_*$\alpha(G', V^*, \mathcal{H}, w, p, \alpha, \beta)$

1 $\mathcal{Q} \leftarrow$ an empty queue;
2 **for** each $x \in N(w, G') \cap V^*$ **do**
3 $rd_\alpha(x) \leftarrow rd_\alpha(x) - 1$;
4 **if** x cannot be included in V^* **then** $\mathcal{Q}$.push(x);
5 **while** $\mathcal{Q} \neq \varnothing$ **do**
6 $x \leftarrow \mathcal{Q}$.pop();
7 $rd_\alpha(x) \leftarrow rd_\alpha(x) + cd_\alpha(x)$; $cd_\alpha(x) \leftarrow 0$;
8 remove w from V^* and insert it after p in $Oa_{\alpha,\beta}$;
9 $p \leftarrow x$;
10 **for** each $y \in N(x, G')$ such that $\beta_\alpha(y) = \beta$ **do**
11 **if** $w \preceq_\alpha y$ **then**
12 $cd_\alpha(y) \leftarrow cd_\alpha(y) - 1$;
13 **if** $cd_\alpha(y) = 0$ **then** remove y from $\mathcal{H}$;
14 **else if** $x \preceq_\alpha y \wedge y \in V^*$ **then**
15 $cd_\alpha(y) \leftarrow cd_\alpha(y) - 1$;
16 **if** y cannot be included in V^* and $y \notin \mathcal{Q}$ **then**
17 $\mathcal{Q}$.push(y);
18 **else if** $y \preceq_\alpha x \wedge y \in V^*$ **then**
19 $rd_\alpha(y) \leftarrow rd_\alpha(y) - 1$;
20 lines 16-17;

Algorithm 5 describes the procedure *RemoveCandidates_*α for updating α-core number. First, an empty queue $\mathcal{Q}$ is initialized. Subsequently, we decrease the $rd_\alpha(x)$ by 1 for each predecessor neighbor x of w that has been added to the set V^*, and enqueue x that no longer satisfy the inclusion criteria of V^* (lines 1-4). Following this, the procedure iterates over all vertices x and their neighbors y that are to be removed from V^* (lines 5-20). In each iteration, we update $rd_\alpha(x)$ and $cd_\alpha(x)$ first and then remove x from V^*. We further update the remaining degree and candidate degree of x's predecessor and successor neighbors in V^* respectively, and remove or add vertices from the heap $\mathcal{H}$ based on the fulfillment of the criteria. To maintain the structure Oa_α, the procedure deletes x from $Oa_{\alpha,\beta}$ in each iteration and inserts x after p in $Oa_{\alpha,\beta}$, where p represents either w or the vertex that was previously moved before x in $Oa_{\alpha,\beta}$.

The procedure *RemoveCandidates_*β handling for updating β-core number follows a similar framework, requiring only swapping u, U, α and Oa with v, V, β and Ob.

Theorem 1. *Given a bipartite graph G and an inserted edge (u, v), the time complexity of algorithm Order-Insert is $O(\delta \cdot |E^+| \cdot \log |E^+|)$, where $|E^+| = \sum_{w \in V^+} d(w, G')$. The space complexity of Order-Insert is $O(\delta \cdot n)$.*

C. Order-Delete Algorithm for Handling Edge Deletion

The key to enhancing edge deletion processing efficiency is also to reduce V^+.

Definition 11 (max-core degree). *Given a bipartite graph G, for a fixed $\alpha(\beta)$ and any vertex x on G, the max-core degree of x $mcd_\alpha(x) = |\{y | y \in N(x, G) \wedge \beta_\alpha(x) \leq \beta_\alpha(y)\}|$ ($mcd_\beta(x) = |\{y | y \in N(x, G) \wedge \alpha_\beta(x) \leq \alpha_\beta(y)\}|$). The max-core degree indicates the number of neighbors supporting x to maintain its current $\alpha(\beta)$-core number.*

Maintaining the max-core degree facilitates the swift identification of vertices need to be added into V^*. The initial max-core degree value can be calculated through core decomposition by traversing each vertex and its neighbors. Additionally, when hybrid edge insertion and deletion management is required, the max-core degree of the vertices and their neighbors

in V^* must be updated during in the final stage of the *Order-Insert* algorithm.

The fundamental idea of the *Order-Delete* algorithm is as follows: taking α-core number update as an example, for each α, once β is computed according to Lemma 6, the initial step involves updating $mcd_\alpha(u)$ and $mcd_\alpha(v)$ to account for the edge insertions. Then, propagation commences from vertices u and v. If vertex w satisfies $mcd_\alpha(w) < \alpha$ for $w \in U(G')$ or $mcd_\alpha(w) < \beta$ for $w \in V(G')$, indicating insufficient neighbor support to maintain its α-core number, then the algorithm incorporates w into V^* and updates the max-core degree of all its neighbors. This iteration continues until all vertices lacking sufficient neighbors are included in V^*. A similar approach is applied to updating β-core number. We only need to use the above strategy to update the core number for α (β) from 1 to δ due to the computation-sharing theory.

Algorithm 6 shows the details of the *Order-Delete* algorithm. The algorithm initially updates the bipartite graph and δ, and then maintains α-core number and β-core number.

Algorithm 6: *Order-Delete*

Input: G, an deleted edge (u, v)
Output: Updated α-core number and β-core number of G'

1 $G' \leftarrow G \cup (u, v)$; update δ;
2 **for** $\alpha = 1$ **to** δ **do**
3 **if** $\beta_\alpha(u) \leq \beta_\alpha(v)$ **then** $mcd_\alpha(u) \leftarrow mcd_\alpha(u) - 1$;
4 line 3 by swapping u with v;
5 $rd_\alpha(u) \leftarrow rd_\alpha(u) - 1$ (assuming $u \preceq_\alpha v$);
6 $\beta \leftarrow min\{\beta_\alpha(u), \beta_\alpha(v)\}$;
7 **if** $\beta = 0$ **then continue**;
8 $\mathcal{Q} \leftarrow$ an empty queue; $V^* \leftarrow \varnothing$;
9 **if** $\beta_\alpha(u) = \beta \wedge mcd_\alpha(u) < \alpha$ **then** $\mathcal{Q}$.push(u);
10 **if** $\beta_\alpha(v) = \beta \wedge mcd_\alpha(v) < \beta$ **then** $\mathcal{Q}$.push(v);
11 **while** $\mathcal{Q} \neq \varnothing$ **do**
12 $w \leftarrow \mathcal{Q}$.pop(); $V^* \leftarrow V^* \cup \{w\}$;
13 $\beta_\alpha(w) \leftarrow \beta_\alpha(w) - 1$;
14 **for** each $x \in N(w, G')$ **do**
15 **if** $x \notin V^* \wedge x \notin Q \wedge \beta_\alpha(x) = \beta$ **then**
16 $mcd_\alpha(x) \leftarrow mcd_\alpha(x) - 1$;
17 **if** x can be included in V^* **then** $\mathcal{Q}$.push(x);
18 **for** each $w \in (V^* \cap V(G'))$ **do**
19 **for** $i = \delta$ **to** (w, G') **do**
20 **if** $\alpha_i(w) > \alpha - 1$ **then** $\alpha_i(w) \leftarrow \alpha - 1$;
21 **else break**;
22 recompute $mcd_\alpha(w)$ for each w in V^*;
23 **for** each $w \in V^*$ in order vertices added to V^* **do**
24 $rd_\alpha(w) \leftarrow 0$;
25 **for** each $x \in N(w, G')$ **do**
26 **if** $\beta_\alpha(x) = \beta \wedge x \preceq_\alpha w$ **then** $rd_\alpha(x) \leftarrow rd_\alpha(x) - 1$;
27 **if** $\beta_\alpha(x) \geq \beta \vee x \in V^*$ **then** $rd_\alpha(w) \leftarrow rd_\alpha(w) + 1$;
28 $V^* \leftarrow V^* \setminus \{w\}$;
29 remove w from $Oa_{\alpha,\beta}$ and add it to the back of $Oa_{\alpha,\beta-1}$;
30 $b_\alpha \leftarrow max\ x\ s.t.\ \left|\{w | w \in N(u, G') \wedge \beta_\alpha(w) \geq x\}\right| \geq \alpha$;
31 **if** $b_\alpha < \beta_\alpha(u)$ **then**
32 move u from $Oa_{\alpha,\beta_\alpha(u)}$ to the back of Oa_{α,b_α};
33 $\beta_\alpha(u) \leftarrow b_\alpha$;
34 update the rd_α and mcd_α of u and its neighbours;
35 **for** $\beta = 1$ **to** δ **do**
36 lines 3-34 by swapping u, U, α and Oa with v, V, β and Ob;

Lines 3-34 present the updating process of α-core number: for each α, the algorithm updates the remaining degree and max-core degree of vertices u and v to reflect the insertion of edge (u, v) initially (lines 3-5). Subsequently, we set $\beta = \min\{\beta_\alpha(u), \beta_\alpha(v)\}$ and initialize $\mathcal{Q}$ and V^* (lines 6-8). Then we individually evaluate whether u and v meet the criteria for inclusion in V^* and add them to $\mathcal{Q}$ (lines 9-10). Following this, the vertices in $\mathcal{Q}$ are iterated (lines 11-17). In each iteration, a vertex w is extracted from $\mathcal{Q}$ and added to V^*. Then, we update the $mcd_\alpha(x)$ of its neighbors x and include x in $\mathcal{Q}$ if it meets the conditions for being added to V^*. When $\mathcal{Q}$ becomes empty, the iteration terminates, and the computation of V^* is completed. The α-core number and max-core degree of the vertices in V^* are updated, and maintenance of Oa_α and remaining degree is performed as required (lines 18-29). Finally, according to Lemma 4, an update is made to $\beta_\alpha(u)$, and the maintenance of Oa_α along with the max-core degree and remaining degree of u and its neighbors is carried out (lines 30-34). The process for updating β-core number is similar to those for updating α-core number (lines 35-36).

Theorem 2. *Given a bipartite graph G and a deleted edge (u, v), the time complexity of algorithm Order-Delete is $O(\delta \cdot |E^*|)$, where $|E^*| = \sum_{w \in V^*} d(w, G')$. The space complexity of Order-Delete is $O(\delta \cdot n)$.*

D. Graph sparsity based optimization

In real-world graphs, $\delta \cdot n$ may be much larger than the graph size. In this subsection, we exploit the sparsity of graphs to reduce the algorithm's space cost.

As described in [32], real-world networks are usually very sparse and follow a power-law degree distribution. This implies that for larger values of α and β, the majority of vertices on graph are not within $C_{\alpha,\beta}$. It means that the α-core number and β-core number of most vertices are both 0 for such α and β and the number of vertices with an α-core number (β-core number) of 0 increases as α (β) increases.

Algorithm 7: *HandleZero_*$\alpha(G', \alpha, u, v)$

1 **if** $\alpha < d(u, G')$ **then**
2 $\beta_\alpha(v) \leftarrow 1$; $rd_\alpha(v) \leftarrow 1$; $mcd_\alpha(v) \leftarrow 1$;
3 add v to the front of $Oa_{\alpha,1}$; update $mcd_\alpha(u)$;
4 **else**
5 $cnt \leftarrow 0$;
6 add u to the front of $Oa_{\alpha,1}$;
7 **for** each $w \in N(u, G')$ **do**
8 **if** $\beta_\alpha(w) = 0$ **then**
9 $\beta_\alpha(w) \leftarrow 1$; $rd_\alpha(w) \leftarrow 1$; $mcd_\alpha(w) \leftarrow 1$;
10 add w to the front of $Oa_{\alpha,1}$;
11 **else**
12 update $mcd_\alpha(w)$; $cnt \leftarrow cnt + 1$;
13 $\beta_\alpha(u) \leftarrow 1$; $mcd_\alpha(u) \leftarrow |N(u, G')|$; $rd_\alpha(u) \leftarrow cnt$;

We treat such vertices differently by no longer maintaining their remaining degree and max-core degree. Consequently, we no longer maintain $Oa_{\alpha,0}$ for $\alpha \leq \delta$ and $Ob_{\beta,0}$ for $\beta \leq \delta$. When an insertion of edge (u, v) causes certain vertices' α-core number to no longer be 0, it must meet one of the following two cases: (1) $\alpha < d(u, G')$ and $\beta_\alpha(v) = 0$; (2) $\alpha = d(u, G')$. We can handle these cases using Algorithm 7. A similar strategy is also applied to updating β-core number.

Through the strategies outlined above, the order-based algorithm's space complexity can be reduced to

$$O\left(\sum\nolimits_{x \in U(G') \cup V(G')} (mnd(x, G') + d(x, G'))\right) \quad (1)$$

where $mnd(x, G') = max_{y \in N(x,G')} d(y, G')$. And the time complexity of *Order-Insert* and *Order-Delete* algorithm can be decreased to $O\left(k \cdot |E^+| \cdot log\,|E^+|\right)$ and $O\left(k \cdot |E^*|\right)$ respectively, where $k = min\left(\delta, max(d\left(u, G\right), d\left(v, G\right))\right)$.

Based on algorithm *Order-Delete* and *Order-Insert*, (α, β)-core can be maintained efficiently. Thus, graph reduction

based on (α, β)-core is now applicable to the task of mining maximum biclique on large dynamic bipartite graphs.

VI. An Efficient Algorithm for Mining Maximum Biclique on Dynamic Bipartite Graphs

A maximum biclique should be a maximal biclique first, but the seed obtained in *Baseline* cannot be guaranteed to be a maximal biclique on G'. Furthermore, graph updating is likely to trigger refreshing in *Baseline*. Specifically, a refresh is always invoked for each edge insertion or when a deleted edge affects a previous result. Lastly, although pruning some vertices based on the seed is feasible, refreshing process *IMBC** still faces a vast search space.

Hence, in this section, we aim to enhance the *Baseline* from three key aspects. Initially, we expand the seed obtained in *Baseline* at a reasonable cost until it forms a maximal biclique. Secondly, we assess the necessity of refreshing beforehand to minimize unnecessary operations. Thirdly, we devise an efficient refreshing strategy that significantly reduces search space during refreshing.

A. Strategy to obtain a maximal biclique seed

We aim to expand the seed obtained in *Baseline* to form a maximal biclique at a modest cost. Our strategy is to attempt to incorporate common neighbors of H into it.

First, considering the scenario of edge insertion, Algorithm 8 outlines the process of expanding H. If $u \notin B_{max}$ and $v \notin B_{max}$, then B_{max} is already a maximal biclique. Otherwise, assuming $u \in B_{max}$, v can be added to H by verifying if it is a common neighbor of all vertices in $U(B_{max})$. Theorem 3 guarantees that the output of Algorithm 8 is a maximal biclique.

Algorithm 8: *InstInit*

Input: (u, v), B_{max}
Output: a large maximal biclique on G'
$H \leftarrow B_{max}$;
if $u \in H$ **then**
  if v is adjacent to every vertex in $U(H)$ on G' **then**
    $H \leftarrow H \cup \{v\}$;
else if $v \in H$ **then**
  if u is adjacent to every vertex in $V(H)$ on G' **then**
    $H \leftarrow H \cup \{u\}$;
return H;

Theorem 3. *When an edge insertion occurs, let H be the output of Algorithm 8 when given input B_{max}, then H forms a maximal biclique on G'.*

Then, considering edge deletion, Algorithm 9 shows the process to expand H. When the deleted edge hits B_{max}, if $B_{max} \setminus \{u\}$ ($B_{max} \setminus \{v\}$) is legal, all common neighbors of vertices in $U(B_{max} \setminus \{u\})$ ($V(B_{max} \setminus \{v\})$) are included into $B_{max} \setminus \{u\}$ ($B_{max} \setminus \{v\}$) to form a larger seed. Theorem 4 guarantees that the output of Algorithm 9 is a maximal biclique.

Theorem 4. *When an edge deletion occurs, let H be the output of Algorithm 9 when given input B_{max}, then H forms a maximal biclique on G'.*

The worst-case time complexity of Algorithms 8 and 9 are $O(d_{max})$ and $O(d_{max}^2)$ respectively. Thus, the obtained seed

Algorithm 9: *DelInit*

Input: (u, v), B_{max}
Output: a large maximal biclique on G'
if $\{u, v\} \subseteq B_{max}$ **then**
  $H_1 \leftarrow \varnothing$, $H_2 \leftarrow \varnothing$;
  if $U(B_{max}) > \tau_U$ **then**
    $H_1 \leftarrow B_{max} \setminus \{u\}$;
    for each $y \in V(G') \setminus V(H_1)$ adjacent to $U(H_1)$ on G' **do**
      $H_1 \leftarrow H_1 \cup \{y\}$;
  if $V(B_{max}) > \tau_V$ **then**
    $H_2 \leftarrow B_{max} \setminus \{v\}$;
    for each $x \in U(G') \setminus U(H_2)$ adjacent to $V(H_2)$ on G' **do**
      $H_2 \leftarrow H_2 \cup \{x\}$;
  $H \leftarrow$ the larger one between H_1 and H_2;
else $H \leftarrow B_{max}$;
return H;

is guaranteed to be a maximal biclique and is not smaller than those generated in the *Baseline*. Building upon this, we analyze the size of B'_{max} in the next subsection to assess the necessity of refreshing.

B. Analysis on the Bound of $|B'_{max}|$

Updating the graph (one edge insertion/deletion) may cause a change in the size of the maximum biclique. We attempt to quantify the extent of this change between these two consecutive graphs. By analyzing B_{max} and B'_{max}, Theorems 5 and 6 respectively provide insights into the cases of edge insertion and deletion.

Theorem 5. *If a graph G turns into G' by inserting edge (u, v), then $|B_{max}| \leq |B'_{max}| \leq |B_{max}| + \left\lfloor \sqrt{|B_{max}|} \right\rfloor + 1$.*

Proof: According to Lemma 2, B_{max} is a biclique on G', thus it is evident that $|B_{max}| \leq |B'_{max}|$. Next, we establish the upper bound. By Lemma 3, both $B'_{max} - \{u\}$ and $B'_{max} - \{v\}$ are bicliques on G. Let's assume that the size of $B'_{max} - \{v\}$ is not less than the size of $B'_{max} - \{u\}$. Denoting $B = B'_{max} - \{v\}$, $a = |U(B)|$, $b = |V(B)|$, it follows that $a \leq b + 1$; otherwise, the size of $B'_{max} - \{u\}$ would be $(a-1)(b+1)$, and $(a-1)(b+1) = ab + a - b - 1 > ab$, leading to a contradiction. Since B is a biclique on G, with $ab \leq |B_{max}|$, we can derive $a \leq \left\lfloor \sqrt{|B_{max}|} \right\rfloor + 1$. Hence, $|B'_{max}| = a(b+1) = |B| + a \leq |B_{max}| + \left\lfloor \sqrt{|B_{max}|} \right\rfloor + 1$.

Theorem 6. *If a graph G turns into G' by deleting edge (u, v), then $|B_{max}| - \left\lfloor \sqrt{|B_{max}|} \right\rfloor \leq |B'_{max}| \leq |B_{max}|$.*

Proof: According to Lemma 3, if the edge (u, v) is not part of B_{max}, then $|B'_{max}| \leq |B_{max}|$. If (u, v) is present in B_{max}, both $B_{max} - \{u\}$ and $B_{max} - \{v\}$ are bicliques on G', leading to $|B_{max}| - \min\{|U(B_{max})|, |V(B_{max})|\} \leq |B'_{max}|$. Given that $\min\{|U(B_{max})|, |V(B_{max})|\} \leq \left\lfloor \sqrt{|B_{max}|} \right\rfloor$, it follows that $|B_{max}| - \left\lfloor \sqrt{|B_{max}|} \right\rfloor \leq |B'_{max}|$.

C. Reduction of refreshing rate

When the size of seed H does not reach the upper bound of B'_{max}, bicliques larger than H may still exist on G'. To minimize unnecessary refreshing, we strive to accurately estimate the size of B'_{max} to determine whether refreshing is needed.

When an edge insertion occurs, Theorem 7 shows that a larger biclique must include the inserted edge.

Theorem 7. *Given a graph G and an inserted edge (u, v), if $|B'_{max}| > |B_{max}|$, then $\{u, v\} \subseteq B'_{max}$.*

Based on Theorem 7, bicliques larger than H can only exist in the subgraph $S'_{u,v}$ induced by $N(v, G')$ and $N(u, G')$. Our approach involves estimating the upper bound on the size of the biclique within this subgraph using bicore upper bound and branch upper bound. Initially, if the bicore upper bound of u or v is not greater than $|H|$, refreshing is unnecessary. Otherwise, all vertices $p \in U(S'_{u,v})$ with $cub_p(\tau_V, d(p, S'_{u,v})) \leq |H|$ and $q \in V(S'_{u,v})$ with $cub_q(\tau_U, d(q, S'_{u,v})) \leq |H|$ are removed from $S'_{u,v}$. Then, $S'_{u,v}$ is considered as a search branch extended by either u or v. We execute *BranchUB*$(N(v, S'_{u,v}), \varnothing, N(u, S'_{u,v}), \tau_U, \tau_V)$ and *BranchUB*$(N(u, S'_{u,v}), \varnothing, N(v, S'_{u,v}), \tau_V, \tau_U)$, and select the minimum value as the estimated result. We refer to this upper bound estimation process as *EstimateUB*$(u, v, |H|)$. Only when *EstimateUB*$(u, v, |H|) > |H|$, refreshing is required to explore the potential larger bicliques.

When deleting an edge, it is difficult to determine the specific location where a larger biclique may appear, then we evaluate the necessity of refreshing based on the maximum value of the bicore upper bound of the vertices on the updated graph. We illustrate our effort to reduce refreshing rate by following Example 3.

Example 3. *The evolvement of a dynamic bipartite graph is illustrated in Fig. 5. The red lines represent edges to be inserted and dashed lines indicate edges to be deleted. Graph updates follow the sequence of deletion (u_1, v_1), insertion (u_3, v_5), (u_8, v_8), and (u_1, v_1). (τ_U, τ_V) is set to $(1, 1)$.*

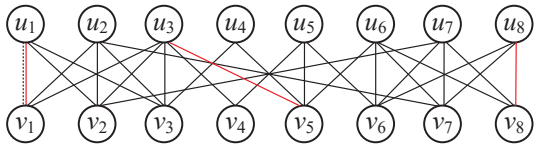


Fig. 5. Evolvement of a dynamic bipartite graph

Initially, B_{max} is $\{u_1, u_2, u_3, v_1, v_2, v_3\}$. When (u_1, v_1) is deleted, a large seed H, i.e. $\{u_1, u_2, u_3, v_2, v_3\}$, is derived from the previous result. Since the maximum value of the bicore upper bound among all vertices is 6, refreshing is not required. When (u_3, v_5) is inserted, S'_{u_3,v_5} is equal to $\{u_3, u_4, u_5, u_6, v_2, v_3, v_4, v_5\}$. As $|H|$ is 6 and the computation result for EstimateUB$(u_3, v_5, 6)$ is 4, refreshing is spared. When (u_8, v_8) is inserted, S'_{u_8,v_8} is equal to $\{u_6, u_7, u_8, v_6, v_7, v_8\}$. Due to EstimateUB$(u_8, v_8, 6)$ is 9 and $|H|$ is 6, refreshing is triggered and $\{u_6, u_7, u_8, v_6, v_7, v_8\}$ is returned as the answer. Finally, (u_1, v_1) is inserted. Now, $\min\{cub_{u_1}(1, 3), cub_{v_1}(1, 3)\} = 9$ and $|H|$ is 9. Thus, no refreshing is required. Compared with Baseline, the number of refreshing in DynamicMBC is reduced from 4 to 1.

D. Reduction of refreshing overhead

When the refreshing condition is triggered, a refreshing has to be executed. Though there is some graph reduction in the refreshing process, the search space remains vast. We aim to restrict the search space to a local subgraph whenever feasible, which is called a local refreshing strategy. In light of the relatively small size of local subgraph, the algorithm avoids executing the costly Two-Hop Graph Reduction step in *MBC** when performing local refreshing.

When considering edge insertion scenarios, according to Theorem 7, we only need to conduct a search within subgraph $S'_{u,v}$. The challenge lies in how we handle edge deletions. When the previous answer is hit by an edge deletion and $|H| < |B_{max}|$, there may still exist bicliques on G' larger than $|H|$. We observe that such bicliques must be associated with the location of the previous inserted edges.

In our algorithm, we have established a data structure called the Edge Buffer Set(EBS) to store the inserted edges. The EBS consists of several Edge Buffer(EB), where each $EB[i]$ is a list, holding the inserted edges that could potentially generate a biclique with an upper bound size of i. We save edges that can generate bicliques with size ranging from $EL + 1$ to EU, where EU maintains the upper bound size of bicliques that can be discovered by traversing the subgraph defined by EBS and is equal to $|B_{max}|$. EL equals the size of the larger seed that can be generated according to Lemma 3, assuming that one deletion edge hits B_{max}.

We adopt a hybrid approach of local and global refreshing to handle edge deletions. When an edge deletion occurs and the refreshing is necessary, if $|H| \geq EL$ is satisfied, a local refreshing is performed. We only need to search the subgraph defined by all edges in $EB[i]$, where i ranges from $|H| + 1$ to EU, to obtain all potential bicliques larger than $|H|$. Otherwise, a global refreshing is conducted.

During the warm-up phase, EU is initialized to $|B_{max}|$ and EL is initialized accordingly. Next, we identify all maximal bicliques with size greater than EL and select a pair of vertices (w.r.t. p and q) from the two vertex sets of each maximal biclique, adding the edge (p, q) to EBS. For each edge (u, v) insertion, if a biclique of size k is found through local refreshing, or the *EstimateUB*(u, v, EL) is calculated as k, it indicates that the insertion of edge (u, v) may produce a biclique with an upper bound size of k. Then the edge is inserted into $EB[k]$. Furthermore, if an edge insertion results in $|B'_{max}| > |B_{max}|$, EU is updated to $|B'_{max}|$ and EL is updated accordingly. This process ensures the correctness of the aforementioned local refreshing strategy.

During local refreshing, we iterate through the edges in EBS starting from $EB[i], i = EU$. For each edge e in $EB[i]$, if only a biclique of size k (where $k < i$) is found through local refreshing, e is moved from $EB[i]$ to $EB[k]$. If k is not greater than EL, e is pruned. Example 4 exemplifies the details of local and global refreshing.

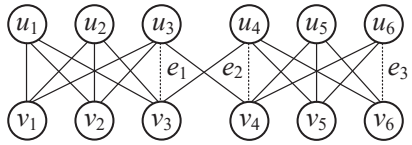


Fig. 6. Refreshing mode when an edge is deleted

Example 4. *The evolvement of a dynamic bipartite graph*

Algorithm 10: *InstProc*

Input: (u, v), B_{max}
Output: B'_{max}
1 call *InstInit*$((u, v), B_{max})$ to get H;
2 $Cub \leftarrow \min\{cub_u(\tau_V, d(u, G')), cub_v(\tau_U, d(v, G'))\}$;
3 **if** $|H| < Cub \wedge |H| <$ *EstimateUB*$(u, v, |H|)$ **then**
4 $S'_{u,v} \leftarrow$ the subgraph defined by (u, v);
5 $H \leftarrow$ *IMBC**$(S'_{u,v}, H, \tau_U, \tau_V)$;
6 **if** $|H| > |B_{max}|$ **then**
7 $EU \leftarrow |H|$; accordingly update EL;
8 $e.cnt \leftarrow (u, v)$, $e.flg \leftarrow$ false; $EB[|H|]$.push(e);
9 **else**
10 $k \leftarrow$ *EstimateUB*(u, v, EL);
11 **if** $k > EL$ **then**
12 $e.cnt \leftarrow (u, v)$, $e.flg \leftarrow$ false, $EB[k]$.push(e);
13 **return** H;

Algorithm 11: *DelProc*

Input: (u, v), B_{max}
Output: B'_{max}
1 $Cub \leftarrow \min\{cub_u(\tau_V, d(u, G')), cub_v(\tau_U, d(v, G'))\}$;
2 **for** $i = EL + 1$ **to** Cub **do**
3 **for** each $e \in EB[i]$ s.t. $e.flg =$ false **do**
4 $(p, q) \leftarrow e.cnt$;
5 **if** $p \in N(v, G) \wedge q \in N(u, G)$ **then** $e.flg \leftarrow$ true ;
6 call *DelInit*$((u, v), B_{max})$ to get H;
7 $maxCub \leftarrow$ maximum bicore upper bound of vertices on G';
8 **if** $|H| < |B_{max}| \wedge |H| < maxCub$ **then**
9 **if** $|H| \geq EL$ **then** $H \leftarrow$ LocalRfr(H);
10 **else** $H \leftarrow$ *IMBC**(G', H, τ_U, τ_V);
11 **return** H;
12 **procedure** LocalRfr(H)
13 **for** $i = EU$ **to** $|H| + 1$ **do**
14 **for** each $e \in EB[i]$ **do**
15 $(p, q) \leftarrow e.cnt$; $S'_{p,q} \leftarrow$ the subgraph defined by (p, q);
16 **if** $e.flg =$ true **then**
17 **if** *EstimateUB*$(p, q, |H|) > |H|$ **then**
18 $H \leftarrow$ *IMBC**$(S'_{p,q}, H, \tau_U, \tau_V)$;
19 **else** $H \leftarrow$ *IMBC**$(S'_{p,q}, H, \tau_U, \tau_V)$;
20 **if** $|H| = EL$ **then** remove e from $EB[i]$;
21 **else if** $|H| < i$ **then** move e from $EB[i]$ to $EB[|H|]$;
22 **return** H;

is shown in Fig. 6. Graph updates by sequentially deleting (u_3, v_3), (u_4, v_4), *and* (u_6, v_6). (τ_U, τ_V) *is set to* $(1, 1)$.

In the warm-up stage of DynamicMBC, B_{max} *is* $\{u_1, u_2, u_3, v_1, v_2, v_3\}$. *Thus,* EU *and* EL *are initialized to* 9 *and* 6, *respectively. There are two maximal bicliques with a size larger than* EL, $\{u_1, u_2, u_3, v_1, v_2, v_3\}$ *and* $\{u_4, u_5, u_6, v_4, v_5, v_6\}$. *Two pair of vertices, w.r.t.* $\{u_1, v_1\}$ *and* $\{u_4, v_4\}$, *are randomly selected from two bicliques. Then* (u_1, v_1) *and* (u_4, v_4) *are inserted to* $EB[9]$. *When* (u_3, v_3) *is deleted, the previous answer is hit. As* $|H|$ *is* 6 *and equal to EL, local refreshing is triggered. From the subgraph defined by* (u_4, v_4), $\{u_4, u_5, u_6, v_4, v_5, v_6\}$ *is found as the result. When* (u_4, v_4) *is deleted, the previous answer is hit. As* $|H|$ *is* 6 *and equal to* EL, *local refreshing is triggered. Since no subgraph defined by any edge stored in* EBS *contains a biclique larger than* H, $H = \{u_4, u_5, u_6, v_5, v_6\}$ *is returned as the result, and both* (u_1, v_1) *and* (u_4, v_4) *are deleted from EBS. When* (u_6, v_6) *is deleted, the previous answer is hit. As* $|H|$ *is* 4, *which is lower than* EL, *global refreshing has to be conducted.* $\{u_1, u_2, u_3, v_1, v_2\}$ *is finally found and output as the answer.*

In the real world, the graph updating model always alternates between edge insertion and deletion. Before a local refreshing is triggered by an edge deletion, large bicliques generated by previous edge insertions may be affected by subsequent edge deletions. Consequently, it is possible to further refine the search space defined by the EBS. For each edge stored in $EB[i]$, w.r.t. (p, q), before performing a local refreshing on the subgraph defined by (p, q), a screening to check if a biclique of size i may still exist on the subgraph.

In our approach, edges stored in EBS are packed as objects. Each edge, denoted as e, consists of two attributes: $e.cnt$ and $e.flg$. The former attribute $e.cnt$ represents the content of the edge, such as (p, q). The latter attribute $e.flg$, a boolean variable, indicates whether the edge requires screening. If screening is necessary, $e.flg$ is set to true; otherwise, it is set to false. Upon each edge deletion event, for each edge e stored in $EB[i]$, if the removal of edge (u, v) affects the subgraph defined by e and both u and v have bicore upper bound not less than i, $e.flg$ is set to true. During local refreshing, for each edge e in $EB[i]$, if $e.flg$ is true, we first compute the *EstimateUB*$(p, q, |H|)$ to determine if bicliques larger than $|H|$ the subgraph defined by e. If not, e is moved to $EB[|H|]$, and if $|H|$ is not greater than EL, e is pruned. This process narrows down the search space defined by EBS.

Algorithm 12: *DynamicMBC*

Input: G, τ_U, τ_V, graph updating operator
Output: B'_{max}
1 compute (α, β)-core and bicore upper bound offline;
2 acquire B_{max} on G;
3 $EU \leftarrow |B_{max}|$; accordingly initialize EL;
4 acquire M on G;
5 **for** each $B \in M$ **do**
6 choose a pair of vertices from B, w.r.t. p and q;
7 $e.cnt \leftarrow (p, q)$, $e.flg \leftarrow$ false, $EB[|B|]$.push(e);
8 **repeat**
9 receive the graph updating operator;
10 update G to reach G';
11 update (α, β)-core and bicore upper bound;
12 **if** the operator is insertion of edge (u, v) **then**
13 $B'_{max} \leftarrow$ *InstProc*$((u, v), B_{max})$;
14 **else**
15 $B'_{max} \leftarrow$ *DelProc*$((u, v), B_{max})$;
16 report $|B'_{max}|$ and B'_{max};
17 $B_{max} \leftarrow B'_{max}$;
18 **until** termination of graph updating;

E. Sketch of DynamicMBC

The sketch of our proposed approach *DynamicMBC* is illustrated in Algorithms 10-12. These algorithms present the procedure to process edge insertion, to process edge deletion, and the main program of *DynamicMBC*, respectively.

Algorithm 10 presents the procedure to handle edge insertion. The procedure begins by executing *InstInit* to acquire the seed H (line 1). Subsequently, the necessity of refreshing is evaluated (lines 2-4). If deemed necessary, a local refreshing on the subgraph defined by (u, v) is conducted to strive to obtain a biclique larger than H (lines 5-6). Next, if a biclique larger than $|B_{max}|$ is obtained, the EU and EL are updated accordingly (lines 8-9). Otherwise, the potential of edge (u, v) insertion to generate a biclique larger than EL is assessed, and if so, it is added to EBS (lines 10-12). Finally, H is returned as the maximum biclique on G' (line 13).

Algorithm 11 presents the procedure to handle edge deletion. First, for each edge e stored in EBS that satisfies $e.flg =$ false, if the subgraph defined by e is hit by the deleted edge (u, v), $e.flg$ is set to true (lines 1-5). Then, *DelInit* is executed

to obtain the seed H (line 6). If a refreshing is deemed necessary, a local refreshing is conducted under the condition $|H| \geq EL$; otherwise, a global refreshing has to be performed (lines 7-10). Finally, H is returned as the maximum biclique on G' (line 11). Lines 12 to 22 outline the specific strategy for local refreshing.

The main program of *DynamicMBC* is presented in Algorithm 12, which is made up of a warm-up stage and a working stage. In the warm-up stage, first, we compute (α, β)-core and bicore upper bound of the vertices on the original graph. Second, the maximum biclique B_{max} is identified, and EU and EL are initialized according to B_{max}. Then, the maximal biclique set M is determined, where M is defined as $\{B||B| > EL \wedge B$ is a maximal biclique$\}$. Finally, each biclique B in M is recorded in EBS. In the working stage, our algorithm keeps monitoring updates on the graph. The (α, β)-core and bicore upper bound of vertices are updated first, and then the procedure to handle edge deletion or insertion is called.

F. Cost Analysis of DynamicMBC

The *EstimateUB* can be executed in $O(d_{max}^2)$ time. The cost of maintaining the (α, β)-core and bicore upper bound is denoted by C_{gr}. The expense of refreshing depends on the method of execution. For ease of discussion, we denote the cost incurred by refreshing on a graph with m edges as $\Theta(G(m), \tau)$, where τ represents the input threshold, i.e., the size of the seed.

The refreshing rate when a refreshing is triggered by edge insertion is denoted as p_{il}, hence, $0 \leq p_{il} \leq 1$. Furthermore, since refreshing operates on the subgraphs defined by edges, the refreshing cost can be represented as $\Theta(G(\eta \cdot m), \tau)$. The time complexity of *InstInit* is $O(d_{max})$. Subsequently, the cost of handling an edge insertion in *DynamicMBC* is given by (2). *DynamicMBC* operates in global and local refreshing modes to handle edge deletions. The frequencies of executing global and local refreshing are respectively denoted as p_{dg} and p_{dl}, thus, $0 \leq p_{dg} + p_{dl} \leq 1$. The cost of local refreshing is represented as $\Theta(G(\gamma \cdot m), \tau)$. The time complexity of delInit is $O(d_{max}^2)$. Therefore, the cost of handling an edge deletion in *DynamicMBC* is given by (3).

$$O\left(C_{gr} + d_{max}^2 + p_{il} \cdot \Theta(G(\eta \cdot m), \tau)\right) \quad (2)$$

$$O\left(C_{gr} + d_{max}^2 + p_{dl} \cdot \Theta(G(\gamma \cdot m), \tau) + p_{dg} \cdot \Theta(G(m), \tau)\right) \quad (3)$$

The cost for *Baseline* to handle an edge insertion is $O(C_{gr} + \Theta(G(m), \tau))$, while the cost to handle an edge deletion is $O(C_{gr} + p_{del} \cdot \Theta(G(m), \tau))$. Here, the refreshing rate when a refreshing is triggered by edge deletion is denoted as p_{del}, hence, $0 \leq p_{dg} + p_{dl} \leq p_{del} \leq 1$.

VII. EXPERIMENTS

A. Setup

Algorithms. (α, β)-core maintaining algorithms: 1) The edge insertion algorithms *BiCore-Index-Ins** (*BII**) introduced in [20], *Edge-Insert* (*EI*) introduced in [21] and *Order-Insert* (*OI*) introduced in Section V; 2) The edge deletion algorithms *BiCore-Index-Rem** (*BIR**) introduced in [20], *Edge-Delete* (*ED*) introduced in [21] and *Order-Delete* (*OD*) introduced in Section V.

Maximum biclique search algorithms: *MBC** proposed in [9] and *IMBC** proposed in Section IV.

Maximum biclique mining algorithms on dynamic graphs: *Baseline* proposed in Section IV and *DynamicMBC* proposed in Section VI.

The algorithms are implemented in C++ and compiled using g++ 9.4.0 with the optimization level set to O3. The experiments are conducted on a Linux server with Intel Xeon 2.50GHz CPU and 128GB RAM.

TABLE II
SUMMARY OF DATASETS

Dataset	n_U	n_V	m	M_O	$\delta \cdot n$
Writers	89,356	46,213	144,340	436,322	948,983
YouTube	94,238	30,087	293,360	1,315,653	2,610,825
Github	56,519	120,867	440,237	2,320,710	7,095,440
Reuters	21,557	38,677	978,446	3,044,910	4,818,720
StackOverflow	545,196	96,680	1,301,942	7,485,427	14,763,148
ActorMovies	127,823	383,640	1,470,404	4,935,545	7,671,945
IMDB	303,617	896,302	3,782,463	15,992,210	28,798,056
Wikipedia	1,853,493	182,947	3,795,796	20,746,091	38,692,360
Amazon	1,230,915	2,146,057	5,743,258	33,895,517	91,178,244
DBLP	5,624,219	1,953,085	12,282,059	45,735,346	113,659,560

Datasets. We choose 10 real-world datasets from KONECT [33] for our experiments and all duplicate edges have been removed. The summary of datasets is displayed in Table II. n_U and n_V denote the number of vertices in each vertex set, and m represents the number of edges. M_O is half of the calculation result of (1).

B. (α, β)-core maintenance

In this subsection, we compare the performance of our proposed (α, β)-core maintenance algorithms, *OI* and *OD*, with existing algorithms *BII**, *BIR**, *EI* and *ED*. For each experiment, we randomly sample 5,000 edges from the graph for insertion or deletion and reported the average time taken to process a single edge.

Performance of handling one edge insertion. Fig. 7 shows the runtime of algorithms *BII**, *EI*, and *OI* to handle one edge insertion. The performance of algorithms *EI* and *BII** is comparable. Our algorithm *OI* exhibits a one-order-of-magnitude advantage across all datasets, and achieves three-orders-of-magnitude improvement on ActorMovies and IMDB. Fig. 2 illustrates why *OI* is so efficient. The ratio of the number of visited vertices to the number of candidate vertices in *OI* is 1 to 4 orders of magnitude fewer than which in *EI* and *BII**.

Performance of handling one edge deletion. Fig. 8 shows the performance of algorithms *BIR**, *ED*, and *OD* in handling one edge deletion. On the Writer and the six largest datasets, algorithms *OD* and *ED* demonstrate significantly superior performance compared to *BII**. Across other datasets, the performance of these three algorithms is comparable. Our algorithm *OD* shows clear advantages over *ED* on Writers, ActorMovies, IMDB, and DBLP, while showing similar performance on other datasets. However, *ED* needs extra time for bicore number maintenance.

Scalability evaluation. In this experiment, we evaluate the scalability of all the (α, β)-core maintaining algorithms.

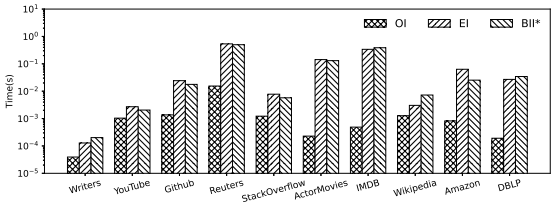


Fig. 7. Performance of handling edge insertion

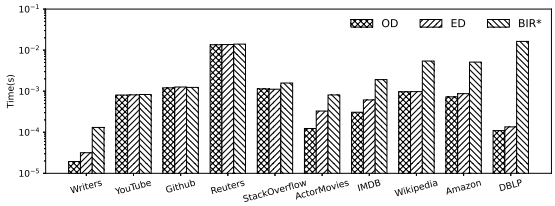
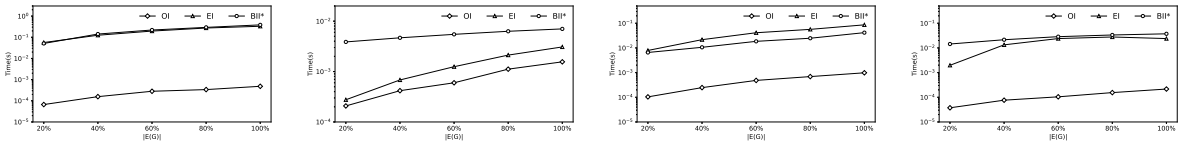
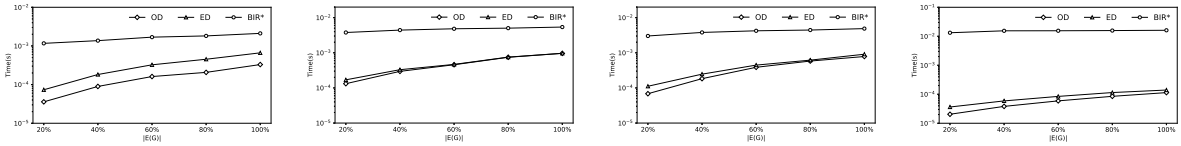


Fig. 8. Performance of handling edge insertion



(a) IMDB (b) Wikipedia (c) Amazon (d) DBLP

Fig. 9. Scalability of edge insertion algorithms



(a) IMDB (b) Wikipedia (c) Amazon (d) DBLP

Fig. 10. Scalability of edge deletion algorithms

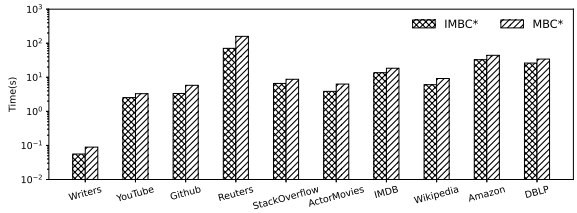


Fig. 11. Search time Evaluation

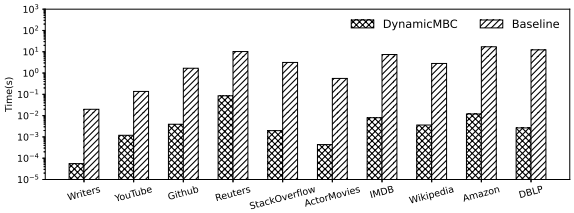
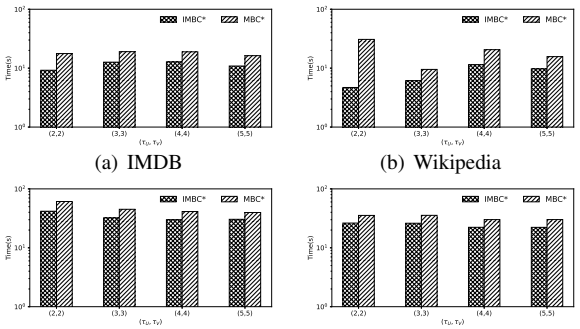


Fig. 13. Perfomance of handling one update



(a) IMDB (b) Wikipedia (c) Amazon (d) DBLP

Fig. 12. Search B_{max} by Varying τ_U and τ_V

The four largest graphs, i.e., IMDB, Wikipedia, Amazon, and DBLP are used in the experiment. The number of edges $|E(G)|$ randomly sampled from the original graph is varying from 20% to 100%.

The scalability evaluation results of the edge insertion algorithm are presented in Fig. 9. As the graph scale increases, the running time of all three algorithms, *OI*, *EI*, and *BII**, grows smoothly. *OI* has better performance in all cases compared to the other two algorithms.

The scalability evaluation results of the edge deletion algorithm are presented in Fig. 10. As the graph scale increases, the running time of all three algorithms, *OD*, *ED*, and *BIR**, grows smoothly. In all cases, the running time of algorithms *OD* and *ED* is faster than that of *BIR**. On IMDB and DBLP, algorithm *OD* consistently shows better performance than *ED* for different graph sizes, while on the other two datasets, their performances are similar.

Space cost evaluation. In this experiment, we evaluate the space cost of our order-based algorithm. Equation (1) in Section IV-C gives out the algorithm's space complexity, and we compare it with m and $\delta \cdot n$. Due to the consideration of vertices on two sides in this equation while the coefficient of 2 within $O(m)$ and $O(\delta \cdot n)$ has been hidden, we half the computation outcome and denote it as M_O. The results are detailed in Table II for all graphs.

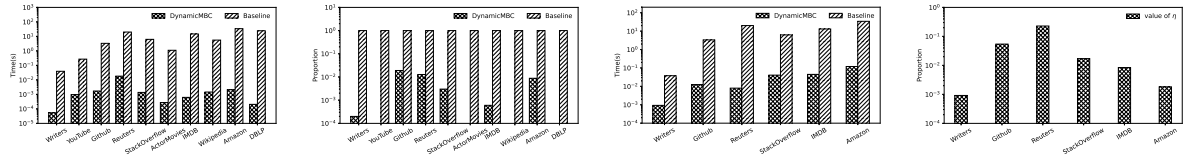
The computed results of M_O across all graphs are notably smaller than $\delta \cdot n$, ranging from 3.0 to 5.9 times the value of m. Such a level of space cost is considered acceptable for large-scale graphs.

C. Maximum biclique search

In this subsection, we compare our improved Maximum biclique search algorithm *IMBC** with the original algorithm *MBC**. We initially assess the performance of both algorithms under constraints (τ_U, τ_V) set at (3,3) across all datasets. Subsequently, we conducted experiments on the four largest datasets by varying the constraints from (2,2) to (5,5).

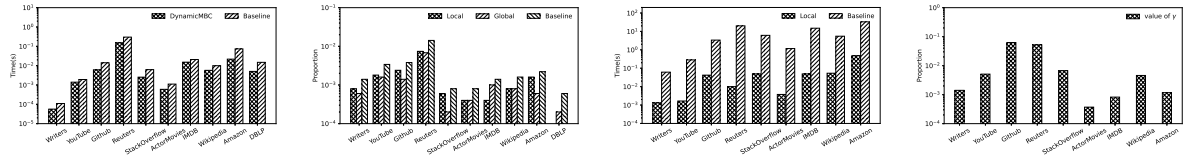
Search time Evaluation. Fig. 11 shows the evaluation results of algorithms *MBC** and *IMBC**. *IMBC** is 30% to 125% faster than *MBC** across all datasets.

Varying τ_U and τ_V constraints. Fig. 12 shows the performance of *IMBC** and *MBC** under different constraints. In all cases, *IMBC** outperforms *MBC** by at least 30%.



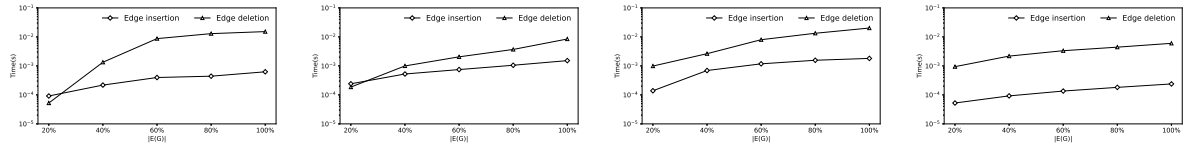
(a) Perfomance of edge insertion (b) Refreshing rate (c) Refreshing overhead (d) Local search space

Fig. 14. Analysis of edge insertion



(a) Perfomance of edge deletion (b) Refreshing rate (c) Refreshing overhead (d) Local search space

Fig. 15. Analysis of edge deletion



(a) IMDB (b) Wikipedia (c) Amazon (d) DBLP

Fig. 16. Scalability of *DynamicMBC*

These experiments prove that our improvements to *MBC** are effective and applicable under various conditions.

D. Maximum biclique mining on dynamic graphs

In this subsection, we evaluate the performance of our maximum biclique maintenance algorithms, *Baseline* and *DynamicMBC*. All experiments are conducted under the constraint of (τ_U, τ_V) set to (3,3).

Comprehensive performance. First of all, we evaluate the comprehensive performance of *Baseline* and *DynamicMBC*. Graph updating tests are conducted 10,000 times on each of the graphs and the updating sequence is randomly generated while edge insertion to deletion ratio is 1. The algorithm performance is measured by the average time spent to process one update. The performance of the two algorithms is illustrated in Fig. 13. *DynamicMBC* outperforms *Baseline* by two to three orders of magnitude across all datasets.

In the following paragraphs, we separately analyze the performance concerning edge insertion and edge deletion to investigate why *DynamicMBC* outperforms *Baseline*.

Analysis of edge insertion. Fig. 14 presents the analysis results for edge insertions. As shown in Fig. 14(a), the performance of *DynamicMBC* in handling edge insertions is more than two orders of magnitude better than *Baseline*. This improvement is attributed to the acquisition of larger seeds and the lower refreshing rates resulting from upper bound estimations. As shown in Fig. 14(b), across all datasets, the refreshing rate of *DynamicMBC* does not exceed 1/50, and even drops to 0 on YouTube, ActorMovies, Wikipedia, and DBLP. However, the refreshing rate of *Baseline* is always 1. Furthermore, even when *DynamicMBC* triggers a refreshing, its search space is typically very small due to the local refreshing strategy. As shown in Fig. 14(d), the search space of local refreshing is usually less than 1/10 of the size of entire graph. The small search space leads to significantly lower refreshing overhead compared to *Baseline* as shown in Fig. 14(c). Local refreshing is two to three orders of magnitude faster than refreshing in *Baseline*.

Analysis of edge deletion. Fig. 15 presents the analysis results for edge deletions. As shown in Fig. 15(a), *DynamicMBC* also exhibits a certain advantage over *Baseline*. The improvement ranges from 37% to about 3.4 times. This phenomenon is primarily attributed to the low refreshing rates. As depicted in Fig. 15(b), the refreshing rate of *Baseline* is mostly less than 1/100, while the sum of the local refreshing rate and global refreshing rate of *DynamicMBC* usually equals that of *Baseline*.

The primary factor enabling *DynamicMBC* to gain an advantage is the local refreshing supported by EBS. As shown in Fig. 15(d), the proportion of the search space of local refreshing to the size of graph is less than 1/10. Such a search space reduction results in a significantly lower overhead of local refreshing compared to refreshing in *Baseline*. As shown in Fig. 15(c), the performance of local refreshing is two to three orders of magnitude better than the refreshing in *Baseline*. Throughout the experimental process, the maximum number of edges stored in EBS across all datasets is 15. This indicates that we have incurred a slight and appropriate cost for implementing local refreshing in edge deletions within *DynamicMBC*.

Scalability Evaluation. In this experiment, we evaluate the scalability of algorithm *DynamicMBC*. The four largest graphs are used in the experiment and the number of edges $|E(G)|$ are varied from 20% to 100% of the original graph. On each sampled subgraph, we also sampled 10,000 edges and generated random sequences for graph updates where insertion to deletion ratio is set to 1. We separately analyze the scalability of *DynamicMBC* in handling one edge insertion and deletion and the results are as demonstrated in Fig. 16. The running time of *DynamicMBC* generally increases smoothly

with the scale of the graph.

VIII. Conclusion

In this paper, we study the problem of mining maximum biclique over dynamic bipartite graphs. Initially, we introduce a rapid seed acquisition method to adapt static MBS algorithms to dynamic environments and improve state-of-the-art static algorithm *MBC** to accelerate refreshing. Subsequently, We introduce an order-based algorithm for (α, β)-core maintenance to support (α, β)-core based graph reduction on large-scale dynamic bipartite graphs. Then, we design a mechanism for acquiring a larger seed and provide a bound on the size of the maximum biclique in the updated graph. By estimating the necessity of refreshing to reduce refreshing rate and implementing a local refreshing strategy to decrease the refreshing overhead, we propose our efficient solution *DynamicMBC*. Finally, extensive experiments on real-world datasets demonstrate the effectiveness and scalability of our approach.

Acknowledgment

This work was supported by the Natural Science Foundation of China under Grant No.62372013, and the Natural Science Foundation of Jiangsu Province under Grant No. BK20151132.

References

[1] M. Allahbakhsh, A. Ignjatovic, B. Benatallah, S.-M.-R. Beheshti, E. Bertino, and N. Foo, "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 15*. Springer, 2013, pp. 196–207.

[2] A. Beutel, W. Xu, V. Guruswami, C. Palow, and C. Faloutsos, "Copycatch: stopping group attacks by spotting lockstep behavior in social networks," in *Proceedings of the 22nd international conference on World Wide Web*, 2013, pp. 119–130.

[3] D. Ding, H. Li, Z. Huang, and N. Mamoulis, "Efficient fault-tolerant group recommendation using alpha-beta-core," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, 2017, pp. 2047–2050.

[4] J. Liu and W. Wang, "Op-cluster: Clustering by tendency in high dimensional space," in *Third IEEE international conference on data mining*. IEEE, 2003, pp. 187–194.

[5] M. Langston, E. J. Chesler, and Y. Zhang, "On finding bicliques in bipartite graphs: a novel algorithm with application to the integration of diverse biological data types," in *Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS 2008)(HICSS)*, vol. 1, 2008, p. 473.

[6] A. Kershenbaum, A. Cutillo, C. Darabos, K. Murray, R. Schiaffino, and J. H. Moore, "Bicliques in graphs with correlated edges: From artificial to biological networks," in *Applications of Evolutionary Computation: 19th European Conference, EvoApplications 2016, Porto, Portugal, March 30–April 1, 2016, Proceedings, Part I 19*. Springer, 2016, pp. 138–155.

[7] G. Liu, K. Sim, and J. Li, "Efficient mining of large maximal bicliques," in *Data Warehousing and Knowledge Discovery: 8th International Conference, DaWaK 2006, Krakow, Poland, September 4-8, 2006. Proceedings 8*. Springer, 2006, pp. 437–448.

[8] K. Wang, W. Zhang, X. Lin, Y. Zhang, L. Qin, and Y. Zhang, "Efficient and effective community search on large-scale bipartite graphs," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 85–96.

[9] B. Lyu, L. Qin, X. Lin, Y. Zhang, Z. Qian, and J. Zhou, "Maximum biclique search at billion scale," *Proceedings of the VLDB Endowment*, 2020.

[10] R. Peeters, "The maximum edge biclique problem is np-complete," *Discrete Applied Mathematics*, vol. 131, no. 3, pp. 651–654, 2003.

[11] E. Shaham, H. Yu, and X.-L. Li, "On finding the maximum edge biclique in a bipartite graph: a subspace clustering approach," in *Proceedings of the 2016 SIAM International Conference on Data Mining*. SIAM, 2016, pp. 315–323.

[12] J. Wang, A. P. De Vries, and M. J. Reinders, "Unifying user-based and item-based collaborative filtering approaches by similarity fusion," in *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, 2006, pp. 501–508.

[13] H. Wang, C. Zhou, J. Wu, W. Dang, X. Zhu, and J. Wang, "Deep structure learning for fraud detection," in *2018 IEEE international conference on data mining (ICDM)*. IEEE, 2018, pp. 567–576.

[14] R. EL BACHA and T. T. Zin, "Ranking of influential users based on user-tweet bipartite graph," in *2018 IEEE International Conference on Service Operations and Logistics, and Informatics (SOLI)*. IEEE, 2018, pp. 97–101.

[15] S. Eubank, H. Guclu, V. Anil Kumar, M. V. Marathe, A. Srinivasan, Z. Toroczkai, and N. Wang, "Modelling disease outbreaks in realistic urban social networks," *Nature*, vol. 429, no. 6988, pp. 180–184, 2004.

[16] Z. Ma, Y. Liu, Y. Hu, J. Yang, C. Liu, and H. Dai, "Efficient maintenance for maximal bicliques in bipartite graph streams," *World Wide Web*, vol. 25, no. 2, pp. 857–877, 2022.

[17] A. Das and S. Tirthapura, "Incremental maintenance of maximal bicliques in a dynamic bipartite graph," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 4, no. 3, pp. 231–242, 2018.

[18] R. Wang, M. Liao, and C. Qin, "An efficient algorithm for enumerating maximal bicliques from a dynamically growing graph," in *Advances in Natural Computation, Fuzzy Systems and Knowledge Discovery: Volume 2*. Springer, 2020, pp. 329–337.

[19] K. Wang, W. Zhang, X. Lin, L. Qin, and A. Zhou, "Efficient personalized maximum biclique search," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2022, pp. 498–511.

[20] B. Liu, L. Yuan, X. Lin, L. Qin, W. Zhang, and J. Zhou, "Efficient (α, β)-core computation in bipartite graphs," *The VLDB Journal*, vol. 29, no. 5, pp. 1075–1099, 2020.

[21] W. Luo, Q. Yang, Y. Fang, and X. Zhou, "Efficient core maintenance in large bipartite graphs," *Proceedings of the ACM on Management of Data*, vol. 1, no. 3, pp. 1–26, 2023.

[22] Y. Zhang, J. X. Yu, Y. Zhang, and L. Qin, "A fast order-based approach for core maintenance," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 2017, pp. 337–348.

[23] B. Guo and E. Sekerinski, "Simplified algorithms for order-based core maintenance," *The Journal of Supercomputing*, pp. 1–32, 2024.

[24] A. Ahmed, V. Batagelj, X. Fu, S.-H. Hong, D. Merrick, and A. Mrvar, "Visualisation and analysis of the internet movie database," in *2007 6th International Asia-Pacific Symposium on Visualization*. IEEE, 2007, pp. 17–24.

[25] M. Cerinšek and V. Batagelj, "Generalized two-mode cores," *Social Networks*, vol. 42, pp. 80–87, 2015.

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

[27] S. Shahinpour, S. Shirvani, Z. Ertem, and S. Butenko, "Scale reduction techniques for computing maximum induced bicliques," *Algorithms*, vol. 10, no. 4, p. 113, 2017.

[28] R.-H. Li, J. X. Yu, and R. Mao, "Efficient core maintenance in large dynamic graphs," *IEEE transactions on knowledge and data engineering*, vol. 26, no. 10, pp. 2453–2465, 2013.

[29] A. E. Saríyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek, "Streaming algorithms for k-core decomposition," *Proceedings of the VLDB Endowment*, vol. 6, no. 6, pp. 433–444, 2013.

[30] S. Sun, W. Li, Y. Wang, W. Liao, and S. Y. Philip, "Continuous monitoring of maximum clique over dynamic graphs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 4, pp. 1667–1683, 2020.

[31] A. S. Scribe and P. Christiano, "2 rmq and lca."

[32] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," *ACM SIGCOMM computer communication review*, vol. 29, no. 4, pp. 251–262, 1999.

[33] J. Kunegis, "Konect: the koblenz network collection," in *Proceedings of the 22nd international conference on world wide web*, 2013, pp. 1343–1350.