Maximal Biclique Enumeration: A Prefix Tree Based Approach

Jiujian Chen[1], Kai Wang[2], Rong-Hua Li[1], Hongchao Qin[1], Xuemin Lin[2], Guoren Wang[1]

[1]*Beijing Institute of Technology;* [2]*Antai College of Economics and Management, Shanghai Jiao Tong University;*
[1]{jiujian, rhli, hcqin, wanggr}@bit.edu.cn; [2]{w.kai, xuemin.lin}@sjtu.edu.cn;

***Abstract*—Bipartite graphs are commonly used to model relationships between two distinct types of entities, such as customer-product relationships in e-commerce platforms and protein-protein interactions in bioinformatics. Enumerating all maximal bicliques from a bipartite graph is a fundamental graph mining problem that has been widely used in many real-world applications including community search and spam detection. Existing algorithms for maximal biclique enumeration can struggle to scale to large graphs with a vast number of maximal bicliques. In this paper, we propose a novel and highly-efficient algorithm for maximal biclique enumeration in bipartite graphs using prefix trees. Specifically, a prefix tree is a data structure that stores lists of elements as paths in the tree, and we observe that a maximal biclique can be represented uniquely by the vertices in one of its vertex layers and stored compactly in prefix trees. The process of our algorithm is divided into two steps. First, we find the lower layer vertices of all maximal bicliques and organize them in a prefix tree (i.e., the result tree). During this step, we transform the original time-consuming operations of checking maximality and filtering candidates for vertex sets into determining uniqueness and performing extraction from a prefix tree at each level of the recursion. Second, we use the result tree to obtain the upper layer vertices of the maximal bicliques by computing the common neighbors of vertices in the tree. In this step, we further optimize the computation for intersections of vertex sets by compressing the neighbors of each vertex and memoization. In addition, we also propose a pre-processing method based on the order of traversal on the prefix tree to reduce memory usage. We conduct extensive experiments on 10 real-world datasets, and the results demonstrate that the proposed algorithm outperforms existing solutions by up to one order of magnitude.**

***Index Terms*—graph mining, bipartite graph, maximal biclique, prefix tree**

I. Introduction

Bipartite graphs are widely used to model relationships between two disjoint sets of entities, such as people-location [1], [2], author-paper [3], [4] and customer-product [5], [6]. The vertices are divided into two layers representing distinct sets of entities, and the edges connecting the layers signify the relationships between them. A complete bipartite subgraph, also referred to as a biclique, is a well-researched pattern where each vertex in one layer has edges connecting it to every vertex in the other layer within the subgraph.

For instance, Figure 1 illustrates an example bipartite graph comprising 5 vertices in the upper layer U, 6 vertices in the lower layer V, and 21 edges. One biclique in this graph is formed by vertices u_2, u_4, u_5 and v_2, v_4, v_5, along with 9 edges. It is considered maximal as it cannot be fully contained within any other biclique. Biclique is an important component

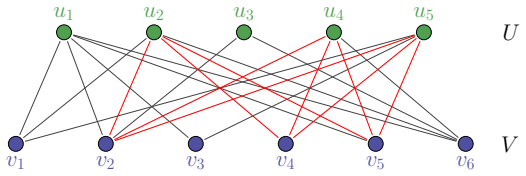


Fig. 1: A maximal biclique in a bipartite graph

in dense subgraph discovery [7]. Finding bicliques has numerous applications such as community search [8], [9], text mining [10], [11], GNN Information Aggregation [12], and social recommendation [13], [14]. In addition, more and more research in bipartite graphs needs to find maximal bicliques as a foundation [15]–[17]. However, the number of maximal bicliques can grow exponentially to the size of the graph [18], which makes the problem very challenging.

In recent years, many algorithms have been proposed for solving the maximal biclique enumeration (MBE) problem [19]–[22]. Among them, iMBEA [19] is inspired by the maximal clique enumeration problem in unipartite graphs [23], which uses a vertex set P for keeping candidate vertices and another set Q for avoiding the redundancy. Given a bipartite graph $G(U, V, E)$, the entire algorithm is executed in recursion by maintaining a biclique $B(L \in U, R \in V)$ at each level and starting with an empty biclique where it treats all vertices in U as the candidates. For each level, the algorithm tries to add vertices from P into L to check if it can become a new maximal biclique. If some vertices in Q can be added to L to enlarge the biclique, the newly generated biclique is redundant and it goes to check the next candidate vertex. Upon this procedure, PMBE [21] employs the single optimal pivot strategy [24] for pruning in biclique scenario. FMBE [20] improves the approach of checking redundancy and reduces the search space by selecting a starting vertex. ooMBEA [22] concentrates on the order of vertices, introducing the unilateral order and batch-pivots technique to accelerate the enumeration.

Nevertheless, these optimizations do not change the core workflow of MBE, and none of them focuses on improving efficiency by organizing the processed data. This motivates us to introduce new data structures for solving the MBE problem. In the literature, the prefix tree is a tree-based data structure storing lists of elements, where the common prefixes are merged to reduce memory usage. Traditionally, it is often used in the field of patterns matching and indexing [25]–[27]. In addition, some recent works [28], [29] use it for computing set containment relationships.

Notation	Definition
G	a bipartite graph
$U(G), V(G)$	the vertex set of upper (lower) layer of G
$E(G)$	the edge set of G
u, v, x, y	vertices in the bipartite graph
$u \prec u'$	precedence of upper vertices in the visiting order
d_{max}, d_{2max}	maximum degree (2-hop degree) of one vertex
X, Y	vertex sets in the bipartite graph
$B(X, Y)$	a biclique composed of vertices X and Y
$\Gamma(x), \Gamma(X)$	the neighbors (the common neighbors) of x (X)
$\mathcal{T}_{cand}, \mathcal{T}_{res}$	prefix trees composed of lower vertices

TABLE I: Notations and their definitions

In this paper, we propose novel enumeration algorithms with prefix trees. We observe that finding all maximal bicliques can be achieved by finding all the lists of vertices in one layer and then completing the other layer. Since prefix trees can store elements (i.e., vertices) compactly, we build prefix trees by these lists and transform the computations of vertex sets into operations on trees. Two kinds of prefix trees are used in our methods, a candidate tree stores the candidate maximal bicliques at each level of the recursion, and a result tree keeps the intermediate results in the lower layer for later completion. The same common prefixes are merged as one path from the root in prefix trees, hence, both the memory and traversing time are saved.

We present our methods, MBET and MBETM, with efficient extraction for the candidate trees and completion of maximal bicliques by traversing the result trees. As our approach is a trade-off with memory for execution time, we also adopt the techniques of memoization and neighbors compression to reduce redundancy, and introduce other optimizations to improve the effectiveness and the space usage. In summary, our principal contributions are listed as follows.

- We propose a novel algorithm to solve the maximal biclique enumeration problem with the prefix tree structure. The new workflow divides the enumeration into two steps, including storing intermediate results in prefix trees and completing the maximal bicliques using prefix trees.
- We adopt optimizations in the two steps. We transform the intersections of vertex sets in the first step into operations on prefix trees, and we propose optimizations for computing common neighbors by memoization and compression in the second step.
- We further optimize the memory usage by a pre-process based on the order of traversal in a prefix tree.
- We conduct comprehensive experiments on ten real-world datasets to validate the efficiency of MBET and MBETM. The results show that our approaches achieve better execution time in all datasets and outperform existing algorithms by up to one order of magnitude.

For reproducibility purposes, the source code of this paper is released at https://github.com/Haizs/MBET.

II. Preliminaries

We consider an undirected and unweighted bipartite graph $G = (U, V, E)$ where U and V are the upper layer and lower layer of G that contain two sets of disjoint vertices respectively, and $E \subseteq U \times V$ is the set of edges. The vertices in the upper layer (lower layer) are called upper vertices (lower vertices) respectively, and each vertex is assigned with a unique vertex ID. Table I summarizes all the frequently used notations. Given a vertex $x \in G$, we represent the set of neighbors of x as $\Gamma(x)$. Furthermore, when we refer to a set X of vertices, $\Gamma(X)$ denotes the common neighbors of all vertices in X, i.e., $\Gamma(X) = \cap_{x \in X} \Gamma(x)$. Conveniently, we use X for a vertex set in the upper layer and Y for the lower.

Definition 1 (Maximal biclique): Given a bipartite graph G, a biclique $B(X, Y)$ is a complete bipartite subgraph, i.e., $X \subseteq U(G)$, $Y \subseteq V(G)$, and $\forall (x \in X, y \in Y), (x, y) \in E(G)$. A biclique is a maximal biclique if it cannot be contained by any other biclique.

Problem Statement. Given a bipartite graph G, we aim to enumerate all the maximal bicliques in G.

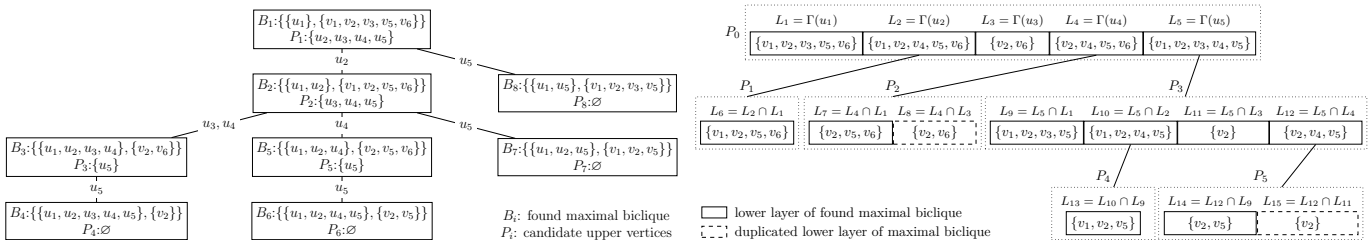
Algorithm 1: Core workflow of MBE

Input : $G = (U, V, E)$
Output: All maximal bicliques in G

```
1  foreach u ∈ U(G) do
2  |  Enumerate({u}, Γ(u), ⋃_{v∈Γ(u)} Γ(v))
3  Procedure Enumerate(X, Y, P)
4  |  foreach v ∈ P do
5  |  |  X' ← X ∪ {v}, Y' ← Y ∩ N(v), P ← P \ {v}
6  |  |  check the maximality of B(X', Y')
7  |  |  if B(X', Y') denotes a new maximal biclique then
8  |  |  |  X' ← X' ∪ {v' ∈ P | N(v') ∩ Y' = Y'}
9  |  |  |  report B(X', Y')
10 |  |  |  P' ← {v' ∈ P \ X' | N(v') ∩ Y' ≠ ∅}
11 |  |  |  if P' ≠ ∅ then
12 |  |  |  |  Enumerate(X', Y', P')
```

A. Warm Up

In bipartite graphs, recent solutions for maximal biclique enumeration [19]–[22] share a core workflow which expands the upper layer during a recursive procedure to enumerate all maximal bicliques. As shown in Algorithm 1, the main procedure is performed recursively and accepts three arguments X, Y, and P. Here X and Y are the sets of upper and lower vertices of a biclique $B(X, Y)$, respectively, and P is the set of candidate upper vertices that can be used to generate a new biclique $B(X', Y')$. During each recursion, we generate X' by adding one candidate vertex $v \in P$ into X (Lines 4 - 5). Correspondingly, we can get the lower vertex set Y' by computing the intersection of Y and $N(v)$. We need to check whether $B(X', Y')$ can become a new maximal biclique (Line 6) and then enlarge X' by adding the vertices in P that have all vertices in Y' as its neighbors (Line 8). After the biclique is found, a new set of vertices P' is created with the remaining candidates that have neighbors in Y' (Line 10), and it will be used as the set of candidate upper vertices in the next recursion level (Line 12). Figure 2a shows an example of finding maximal bicliques containing u_1 in Figure 1. Each node in the recursion tree denotes a



(a) Expansion on the upper layer

(b) Intersection and deduplication on the lower layer

Fig. 2: Partial recursion tree of maximal biclique enumeration

subproblem with new maximal biclique $B_i(X, Y)$ and its corresponding candidate set P_i. The procedure tries to move each candidate vertex $u \in P_i$ into X and check whether the expansion on the upper layer results in a new maximal biclique $B_j(X' \supseteq X \cup \{u\}, Y' = Y \cap \Gamma(u))$. The lower layer is reduced to the intersection of neighbors of the expanded vertex, and a new candidate set $P_j = \{u' \in P_i \setminus X' \mid \Gamma(u') \cap Y' \neq \varnothing\}$ is created with other candidates that have neighbors in Y'.

During the recursion, a subproblem with new candidates is derived only when it finds a new maximal biclique. For instance, P_2 has three candidate vertices u_3, u_4, u_5, but adding only u_3 to the upper layer of B_2 does not lead to a maximal biclique, prompting the expansion of B_3 by both u_3 and u_4. Previous works adopt different methods to complete the expansion and check the maximality. After adding a candidate vertex u to the upper layer and reducing the lower layer Y', FMBE [20] and PMBE [21] directly obtain the complete upper layer by computing $X' = \Gamma(Y')$ and determine the maximality by verifying $X' \subseteq X \cup P_i$. In iMBEA [19] and ooMBEA [22], an additional vertex set Q is introduced to keep track of previously used candidate vertices. A new maximal biclique is reported only if no vertex in Q has all lower vertices as neighbors (i.e., $\nexists u \in Q, Y' \subseteq \Gamma(u)$), and the complete upper layer $X' = X \cup \{u\} \cup \{u' \in P_i \mid \Gamma(u') \cap Y' = Y'\}$ is computed by further examining the remaining candidate vertices.

To mitigate the redundant computation of non-maximal bicliques, existing solutions employ pivot-based prunings [21], [22] to skip certain candidate vertices based on the containment relationships of their neighbors, and utilize optimized orders [20], [22] for candidate vertex enumeration within each subproblem. However, these approaches still encounter challenges in handling the vast number of maximal bicliques. We identify main weaknesses of existing solutions as follows.

As the enumeration process tries to expand the biclique by candidate vertices in each recursive subproblem, multiple candidate vertices can be added simultaneously for one candidate maximal biclique. Additional work is required to compute the complete set of upper vertices. In iMBEA and ooMBEA, all candidate vertices are examined every time. In PMBE and FMBE, the complete upper layer is also used to check for maximality which implies that both layers need to be obtained even if the biclique is not a new maximal one, leading to many redundant computations. On the other hand, the intersection of vertex sets during the derivation of subproblems is always achieved by comparing pairs of elements continuously. All the existing solutions lack the opportunity to obtain the intersection results by necessary elements directly.

III. Two-step Enumeration with Prefix Tree

A. Lower-layer-based Enumeration

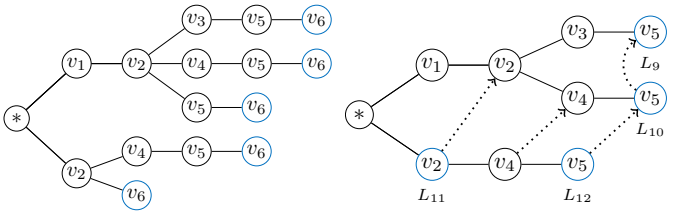
In this paper, we try to enumerate all maximal bicliques based on the lower layer. As discussed in the recursion presented in §II-A, adding a candidate vertex u into $B(X, Y)$ results in a maximal biclique whose lower layer is $Y' = Y \cap \Gamma(u)$. Then, its upper layer can be directly computed by the common neighbors of Y' (i.e., $X' = \Gamma(Y')$). According to this observation, we have the following lemma.

Lemma 1: A maximal biclique can be represented by a unique set of lower vertices, which is composed of the common neighbors of some vertices in the upper layer.

Proof: We discuss an example where Y is the non-empty set of common neighbors of an upper vertex set X. Relatively, the common neighbors of Y must have all vertices in X, i.e., $X \subseteq \Gamma(Y = \Gamma(X))$. If $B(\Gamma(Y), Y)$ is not a maximal biclique, assume there exists a vertex $v \notin Y$ that can enlarge the biclique as $B(\Gamma(Y), Y \cup \{v\})$. Then, v needs to have an edge with every vertex in $\Gamma(Y)$, and also the same for X. That means v is a common neighbor of X, which is contrary to the definition of common neighbors Y. Thus, $B(\Gamma(Y), Y)$ is a maximal biclique, and this representation omits X. □

Lemma 1 allows us to represent and enumerate the candidate maximal bicliques by their lower layers. We maintain the candidate maximal bicliques each as a set of lower vertices and the upper layers are implicitly involved as recursive derivations. Each maximal biclique in the candidate set is different in one recursive subproblem . The workload for checking maximality is then transformed into determining the uniqueness of the set of vertices in the lower layer. We derive recursive subproblems by each set only at its first occurrence in the entire recursion. Initially, each upper vertex in the bipartite graph contributes to one candidate maximal biclique and we keep its neighbors as the lower layer. Then, we obtain the candidates in each subproblem by computing intersections of lower vertex sets with the following lemma.

Lemma 2: Given two different sets of lower vertices, Y_1 and Y_2, corresponding to different candidate maximal bicliques ($B(\Gamma(Y_1), Y_1)$ and $B(\Gamma(Y_2), Y_2)$), their intersection $Y_1 \cap Y_2$ also forms the lower layer of a candidate maximal biclique if it is not empty.



(a) $\mathcal{T}_{res}$ before P_2 (b) $\mathcal{T}_{cand}$ for P_3

Fig. 3: Examples of prefix trees

Proof: We have $\Gamma(\Gamma(Y_1)) = Y_1$ and $\Gamma(\Gamma(Y_2)) = Y_2$ by the definition of maximal bicliques. Hence, $\Gamma(\Gamma(Y_1) \cup \Gamma(Y_2)) = Y_1 \cap Y_2$ as only vertices in both Y_1 and Y_2 have edges connected to all vertices in $\Gamma(Y_1)$ and $\Gamma(Y_2)$. It indicates that the intersection of Y_1 and Y_2 is composed of the common neighbors of $\Gamma(Y_1) \cup \Gamma(Y_2)$ in the upper layer, and when $Y_1 \cap Y_2 \neq \varnothing$, it can represent a candidate maximal biclique according to Lemma 1. □

Overall, only the lower layers are computed during the recursive enumeration. The upper layer of each maximal biclique can be obtained precisely once as the common neighbors of the lower vertices. Figure 2b shows the recursion tree for the lower-layer-based enumeration for Figure 1. In each recursive subproblem P_i, we list the sets of lower vertices corresponding to candidate maximal bicliques, and it initially has the neighbor sets of $u_1, \cdots, u_5$ in P_0. We enumerate each candidate and construct one recursive subproblem as a child node by performing the intersection with other candidates before it, and produce further candidates in the subproblem as the resulting proper subsets. In the figure, each box represents a maximal biclique, and dashed boxes indicate duplicated bicliques that are skipped to derive subproblems. For example, the candidate L_4 produces L_7 and L_8 by intersecting with L_1 and L_3 respectively. However, L_8 can be safely pruned because it represents the same biclique as L_3. Additionally, note that the candidate L_3 can only have intersection results identical to itself, resulting in no subproblem being derived from it.

B. From Vertex Sets to Prefix Tree

To efficiently store the lower layers and identify duplicates, we utilize the structure of prefix trees to maintain the results. Formally, a prefix tree is a rooted tree, where every node in it except the root is assigned a vertex and all outgoing links from one node point to its children nodes with different vertices. We build a prefix tree by all found sets of lower vertices corresponding to maximal bicliques, where each set is inserted as a tree path in the ascending order of vertex IDs. We name it the **result tree** and use $\mathcal{T}_{res}$ for notation.

A new maximal biclique is determined by whether its lower layer contributes to a new path in the result tree. For instance, Figure 3a illustrates the result tree before processing the recursive subproblem P_2. Each path from the root to a blue node represents the lower layer of one maximal biclique. It includes the first four candidates in P_0 as well as the candidate in P_1. Then, the path corresponding to $L_8 : *-v_2-v_6$ already exists, indicating that it is a duplicated result.

Algorithm 2: Our proposed algorithm MBET

Input : $G = (U, V, E)$
Output: All maximal bicliques in G

1 **Algorithm** `MBET(`G`)`
2 **sort** vertices in V by non-ascending degrees
3 $\mathcal{T}_{res} \leftarrow \varnothing$ `// an empty result tree`
4 **compute** the visiting order of U `// lexicographical order of` $\Gamma(u \in U)$
5 **foreach** $u \in U$ *in the visiting order* **do**
6 $\mathcal{T}_{cand} \leftarrow \{\Gamma(u) \cap \Gamma(u') \mid u' \in U \wedge u' \prec u\}$
7 `FindLower(`$\mathcal{T}_{cand}$, $\mathcal{T}_{res}$`)` `// enumerate lower layers of maximal bicliques`
8 $\mathcal{T}'_{res}$, $\mathcal{T}_{res} \leftarrow$ **partition** $\mathcal{T}_{res}$ as inserting $\Gamma(u)$
9 `CompleteUpper(Root(`$\mathcal{T}'_{res}$`)` , U, $\varnothing$`)` `// compute upper layers and reclaim obsolete part of` $\mathcal{T}_{res}$

In each recursive subproblem, we also build a prefix tree to maintain the lower layers of candidate maximal bicliques. We name it the **candidate tree** and use $\mathcal{T}_{cand}$ for notation. The derivation of a candidate maximal biclique is achieved by using its corresponding path to build a new candidate tree for the derived subproblem. The intersection of sets of lower vertices is transformed into an **extraction** operation using a path: we build a new prefix tree by connecting tree nodes that have the same vertices in the path and preserving the ancestor-descendant relationships of the nodes. Rather than compare each node on other paths, this efficient extraction only visits the necessary nodes . For instance, Figure 3b illustrates the candidate tree for the subproblem P_3, where dotted links connect tree nodes with the same vertices. While performing the extraction of L_{12} and following these links, the tree nodes with v_1 and v_3 would not be visited.

Algorithm 2 shows the outline of our algorithm MBET. Initially, the lower vertices are sorted by degrees and the result tree is empty (Line 2-3). To optimize the memory consumption of the result tree, we process maximal bicliques containing each upper vertex sequentially in a **visiting order**, which is the lexicographical order of their neighbors (Line 4-5). The process of each vertex is divided into two steps. In the first step, we build an initial candidate tree with upper vertices before u in the visiting order, find lower layers of all maximal bicliques derived from it and store them in the result tree (Line 6-7). We then partition the result tree by inserting the corresponding path of neighbors of u and split the tree into two parts (Line 8). The first part contains paths that are no longer used in the subsequent process. To optimize memory usage, we traverse this portion of the result tree to compute the upper layers of maximal bicliques and then reclaim the memory occupied by these obsolete paths (Line 9).

IV. Computing the Result Tree

In this section, we present the first step of our approach which finds the lower layers of maximal bicliques and stores

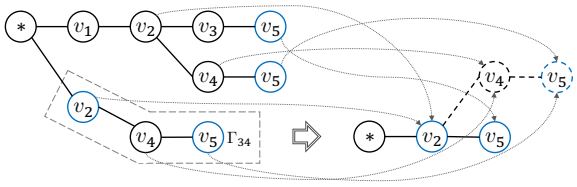


Fig. 4: Derivation of a recursive subproblem by extraction

them in the result tree. We detail the extraction operation for the corresponding path of a candidate maximal biclique.

A. Efficient Extraction for Candidate Tree

Given a candidate tree representing the lower vertex sets in a subproblem, we expect to construct the intersection results for each set directly by vertices in the set. By traversing the candidate tree and having a path corresponding to one candidate to derive the subproblem, we can jump to other nodes which have the same vertices in the path and use them to build the extracted candidate tree. For instance, Figure 4 shows a derivation by the path of L_{12}. The dotted lines with arrows indicate the relationships of extracted nodes in the derived candidate tree. The dashed links and nodes form the same path of L_{12} and are pruned from the new tree. Specifically, we divide the process of extraction into three steps, aiming at only visiting the necessary nodes in the candidate tree.

The first step is to build external information on the candidate tree before we choose any set of lower vertices to extract. We build additional directed links, named **jump links**, connecting all nodes with the same vertex so that we can visit the required nodes by following the links. To maintain the hierarchical structure of the tree, we assign each node a unique index by performing a DFS traversal from the root, and a **range index** which denotes the beginning and end of the indices inside the subtree of that node. Moreover, the end of the path for the lower layer of one candidate maximal biclique is marked as an **end-node**, and we count how many end-nodes existed in the subtree of each node. Then, for each path indicating a new maximal biclique, we perform the extraction efficiently by forking the required nodes directly. Keeping the ancestor-descendant relationships, nodes with the same vertex are merged to construct an extracted tree. At last, we reduce the maintained information and prune the useless path.

B. Recursive Enumeration of Lower Layers

Algorithm 3 shows the algorithm for finding the lower vertex sets of all maximal bicliques. The main procedure `FindLower` accepts the candidate tree for traversal and the result tree for storing the lower layer of all maximal bicliques and checking uniqueness. Initially, the result tree is empty and the candidate tree is constructed by computing intersections of neighbor sets for each upper vertex in the graph. For every $node$ in a prefix tree, `Father`$(node)$ and `Child`$(node)$ return the father node and the children list respectively. It invokes `Build` and `Traverse` for each recursive subproblem to find new maximal bicliques and extract derived candidates. Throughout the process, we keep the following data in each $node$ in the candidate tree:

Algorithm 3: Find lower layers of all maximal bicliques

Input : $\mathcal{T}_{cand}$, $\mathcal{T}_{res}$
Output: update $\mathcal{T}_{res}$ by lower layers of maximal bicliques

Procedure `FindLower`$(\mathcal{T}_{cand}, \mathcal{T}_{res})$
  $Cnt_{dfs} \leftarrow 0$, $Prev \leftarrow \{\}$
  `Build(Root(`$\mathcal{T}_{cand}$`))`
  `Traverse(Root(`$\mathcal{T}_{cand}$`),` $\mathcal{T}_{res}$`)`
Procedure `Build`$(node)$
  $idx \leftarrow Cnt_{dfs}$, $Cnt_{dfs} \leftarrow Cnt_{dfs} + 1$
  $node_{prev} \leftarrow Prev[node_{vid}]$, $Prev[node_{vid}] \leftarrow node$
  $node_{cntE} \leftarrow 1$ **if** $node_{isE}$ **else** 0
  foreach $node' \in$ `Child`$(node)$ **do**
    $node_{cntE} \leftarrow node_{cntE}$+`Build`$(node')$
  $node_{idx} \leftarrow [idx, Cnt_{dfs}]$
  return $node_{cntE}$
Procedure `Traverse`$(node, \mathcal{T}_{res})$
  if $node_{isE}$ **then**
    $path :=$ nodes **from** $root$ **to** $node$
    if *check uniqueness* of $path$ by $\mathcal{T}_{res}$ **then**
      $\mathcal{T}'_{cand} \leftarrow$ **build** an empty prefix tree
      foreach $pnode \in path \setminus root$ **do**
        while $pnode$ **do**
          $pnode' \leftarrow$ **find** or **create** deepest node in $\mathcal{T}'_{cand}$ s.t. $pnode_{vid} = pnode'_{vid} \wedge pnode_{idx} \subset$ `Father`$(pnode')_{idx}$
          $pnode'_{idx} \leftarrow pnode'_{idx} \cup pnode_{idx}$
          $pnode'_{cntE} \leftarrow pnode'_{cntE} + pnode_{cntE}$
          $pnode \leftarrow pnode_{prev}$
      `Reduce(Root(`$\mathcal{T}'_{cand}$`))`
      prune $path$ in $\mathcal{T}'_{cand}$
      if $|\mathcal{T}'_{cand}| > 1$ **then**
        `FindLower(`$\mathcal{T}'_{cand}$`,` $\mathcal{T}_{res}$`)`
  foreach $node' \in$ `Child`$(node)$ **do**
    `Traverse(`$node'$`,` $\mathcal{T}_{res}$`)`
Procedure `Reduce`$(node)$
  if $node_{cntE} > \sum$ `Child`$(node)_{cntE}$ **then**
    $node_{isE} \leftarrow true$
  foreach $node' \in$ `Child`$(node)$ **do**
    `Reduce(`$node'$`)`

- $node_{vid}$: the vertex ID which it is assigned.
- $node_{prev}$: the previous node pointed by jump links.
- $node_{idx}$: the range index of the subtree.
- $node_{isE}$: whether it is an end-node or not.
- $node_{cntE}$: count of end-nodes within the subtree.

The procedure `Build` performs DFS starting from the root node of the candidate tree. During the traversal, we keep the last visited node for each vertex, ensuring that the jump links have the order reversed to which we visit the lower layer of candidate maximal bicliques. It uses a $Prev$ array to track the last visited node of each vertex. When entering a node, the count of visited nodes (Cnt_{dfs}) is recorded in idx, and $node_{prev}$ is pointed to the last visited node by $Prev[node_{vid}]$. Both the Cnt_{dfs} and $Prev$ are then updated and the recursion goes into the children nodes. While backtracking, we have the count of end-nodes inside this subtree and save it in $node_{cntE}$. The range index $node_{idx}$ is updated by previously recorded idx and current Cnt_{dfs} which is the maximal index inside this subtree.

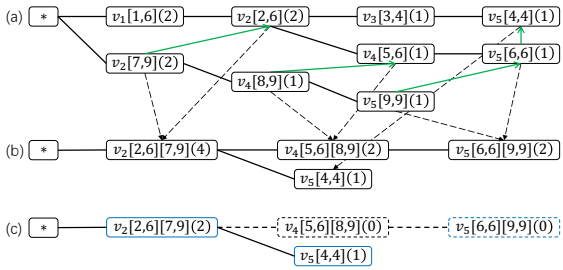


Fig. 5: Steps of extraction for Figure 4

We traverse the candidate tree again as shown in procedure `Traverse`. When we encounter an end-node, it means the path from the root to the current node forms the lower layer of a candidate maximal biclique. Then, we check its uniqueness by trying to insert the path to the result tree. If it indicates a new maximal biclique, we use each node on the path to extract a new candidate tree. We perform fork and merge operations as described in Line 19-23, $pnode$ starts with every node on the path except the root. For each $pnode$, we look for a node with maximum depth in the new tree where it is assigned with the same vertex and satisfies that $pnode$'s range index is contained by the node's father. Or we create a new node if it does not exist. We say such a node in the new tree is a forked node of $pnode$. As the jump links have reversed order to our traversal, the range indices decrease with $pnode$ moving forward. The forking of nodes in the jump links starting from a node on the path can be computed by trying to add a child to the forked nodes of nodes in those jump links starting from a previous node on the path. Then we merge the range indices and the counts of end-nodes into the forked node of $pnode$.

However, the marks for end-nodes are lacking because only some of the end-nodes are forked. We then use the merged counts to recover them. As shown in Line 30-34, we remark end-nodes by performing DFS on the newly created tree. When a node has a merged count more than the sum of its children, it means either itself is an end-node or it is the last node remaining for the paths corresponding to some candidates. Therefore, it needs to be treated as an end-node in the extracted candidate tree. In the last step, we need to prune the path in the extracted tree which is as same as $path$ because we only need to keep proper subsets. We prune the path from the last node corresponding to $path$. We minus the count of end-nodes in all nodes on the path by the count in the last node, and then remove those nodes with zero counts.

Example of an extraction. Figure 5 depicts the steps of the extraction by the candidate path $\{v_2, v_4, v_5\}$ in Figure 4. We illustrate the jump links in green and the end-nodes in blue. For each node in step (a), we show the range index in square brackets and the count of end-nodes in parentheses. For instance, there are 6 nodes in the subtree of the node assigned v_1 and two of them are end-nodes, and it is the first node visited by the DFS. So the range index of it is $[1, 6]$ and the count is 2.

Then, the range indices are merged in step (b) and the counts are updated in steps (b) and (c). The dashed arrows depict the merging of nodes assigned with the same vertices in the new tree. Two nodes assigned v_2 are merged into a single node as it is the first vertex in $path$, and v_5 exists in two nodes in the new tree because there are two nodes of v_5 in the subtree of a node assigned v_4 and one directly in the subtree of a node assigned v_2.

In step (c), we remark the end-nodes which correspond to sets $\{v_2\}$, $\{v_2, v_5\}$ and $\{v_2, v_4, v_5\}$. Note that the set of lower vertices $\{v_1, v_2, v_4, v_5\}$ results in the same path $\{v_2, v_4, v_5\}$ in the extracted tree. The dashed nodes in Figure 5 denote the removal of useless nodes due to the same path. Hence, only sets of $\{v_2\}$ and $\{v_2, v_5\}$ remain in the final extracted candidate tree.

Algorithm 4: Complete maximal bicliques by result tree

Input : $G = (U, V, E)$, $\mathcal{T}_{res}$
Output: Report all maximal bicliques

```
1 Procedure CompleteUpper(node, X, Y)
2     Y' ← Y ∪ node_vid
3     X' ← X ∩ Γ(node_vid)
4     if node_isE then
5         report B(X', Y')
6     foreach node' ∈ Child(node) do
7         CompleteUpper(node', X', Y')
```

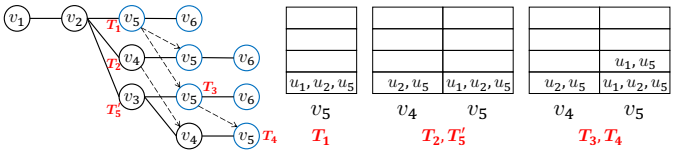
Correctness analysis. Overall, once a unique path corresponding to a candidate maximal biclique is found in the candidate tree and the extracted tree has nodes other than the root, we move to the next level of the recursion, invoking `Build` and `Traverse` continuously. Each time we enter the procedure `FindLower`, we have a candidate tree consisting of the lower layers of all candidate maximal bicliques in the recursive subproblem. Initially, we build a candidate tree with the neighbor set of each upper vertex in the bipartite graph. By traversing in the candidate tree, we compute the intersection of each pair of sets of lower vertices for the candidates and derive subproblems for non-empty results. This recursive process will compute the intersections of neighbor sets of $2^{|U(G)|}$ sets of upper vertices, which correspond to the common neighbors of any subset of $U(G)$. According to Lemma 1, any maximal biclique is formed by the common neighbors of a subset of $U(G)$ and the common neighbors of the common neighbors. Therefore, our recursion can find all the maximal bicliques in the given bipartite graph.

V. Completing the Upper Layer

After the first step of our approach, we obtain the lower layers of maximal bicliques in the result tree. The upper layers are then computed through a traversal of the tree. Additionally, we present optimizations for the intersection of vertex sets in this section.

A. Traverse the Result Tree

Remind that each path from the root to an end-node forms the lower layer of a maximal biclique, and the upper layer can be directly computed as the common neighbors of vertices in



(a) Traverse the result tree (b) Store memoized sets in stacks

Fig. 6: Intersection memoization during traversal

the path. Hence, we can perform a DFS to traverse the result tree and maintain both layers of the biclique while visiting each node. Algorithm 4 outlines the process for reporting all maximal bicliques through the traversal. We employ two vertex sets, X and Y, to store both layers of the biclique during traversal. The algorithm begins from the root node, with $Y = \varnothing$ and $X = U$ representing the set of common neighbors. Each time we visit a node, the assigned vertex of it is added to Y and X is intersected by the neighbors of the vertex (Line 2-3). When a node is an end-node, the current biclique is reported as a maximal result (Line 5). The process is then repeated for each child of the node, using the updated sets X' and Y'.

In figure 6a, we show the traversal of a partial result tree from Figure 1. Each blue node represents an end-node for the lower layer of a maximal biclique. For the top two nodes, we have $X_1' = \{u_1, u_2, u_5\}$, $Y_1' = \{v_1\}$ and $X_2' = \{u_1, u_2, u_5\}$, $Y_2' = \{v_1, v_2\}$ respectively. When traversing the node assigned with v_3, we obtain $X_3' = \{u_1, u_5\}$ as the intersection of X_2' and $\{u_1, u_5\}$ (i.e., v_3's neighbors). Furthermore, we visit its direct child node with v_5, computing the intersection of its neighbors and X_3', which reports a maximal biclique $B(\{u_1, u_5\}, \{v_1, v_2, v_3, v_5\})$.

The elimination of duplicated computation for the upper layer is achieved as each node in the result tree is visited exactly once, and the total workload is further reduced through the merging of prefixes. Inspired by the computation of upper layers during the traversal, the following lemma is intuitively derived, leading to our optimizations in the next part.

Lemma 3: Given two upper vertex sets X' and X'' that are computed from the result tree when visiting the tree nodes p' and p'' respectively, and p' is an ancestor node of p'', then $X'' \subseteq X'$.

B. Optimizations for Neighbor Intersections

Based on Lemma 3, we propose optimizations based on the memoization of intersection results and the compression of neighbor sets, exploiting the ancestor-descendant relationships in the result tree.

Intersection memoization. In practice, memoization is an optimization technique that involves trading space for time by storing intermediate results in memory for future use. We leverage this technique by caching some sets of common neighbors within specific subtrees of the result tree. More precisely, for any given subtree, we can store the computed sets of common neighbors on certain direct children of the root node, and then utilize them for the intersection computations on some grandchildren nodes.

v_1	u_1	u_2	u_5	comp.
v_2	1	1	1	{0,3}{2,1}
v_3	1	0	1	{0,1}{2,1}
v_4	0	1	1	{0,2}{2,1}
v_5	1	1	1	{0,3}{2,1}
v_6	1	1	0	{0,3}

path	comp.
v_1, v_2, v_5	{0,3}{2,1}
v_1, v_2, v_5, v_6	{0,3}
v_1, v_2, v_3, v_5	{0,1}{2,1}
v_1, v_2, v_3, v_4	{2,1}
v_1, v_2, v_3, v_4, v_5	{2,1}

TABLE II: Neighbors in bits and compression

Figure 6a also illustrates the memoization during the traversal. The vertex in each node always has a greater ID than its father, and we traverse into its children in the descending order of their vertex IDs. We mark the time points of visiting some nodes as $T_1 \cdots T_4$ in the figure, and T_5' denotes the time point of backtracking from the subtree of the node. We use dashed lines with arrows to indicate nodes that share the same vertex in their siblings' grandchildren. In the example, we can reuse the memoized sets of common neighbors of v_4 once and those of v_5 three times, and the last node of v_5 utilizes the common neighbors in the previous node under v_3, which is deeper in the tree.

We maintain a stack for each vertex to store the memoized sets. The set of common neighbors in a node is added to the stack of its assigned vertex if it needs to be kept, while it is removed after the traversal leaves its parent node. Figure 6b shows the state of stacks at the marked time points. At T_1 and T_2, we push $\Gamma(\{v_1, v_2, v_5\}) = \{u_1, u_2, u_5\}$ and $\Gamma(\{v_1, v_2, v_4\}) = \{u_2, u_5\}$ respectively. Then at T_3, $\Gamma(\{v_1, v_2, v_3, v_5\})$ is computed by $\Gamma(\{v_1, v_2, v_3\}) \cap \{u_1, u_2, u_5\}$, and the result $\{u_1, u_5\}$ is also pushed to the stack. Afterward, we can use this result to compute $\Gamma(\{v_1, v_2, v_3, v_4, v_5\})$ at T_4, and it does not update the stack due to no siblings' grandchild being assigned with v_5. Finally, before leaving the subtree at T_5', the top element in the stack of v_5 is removed, and both v_4 and v_5 have only one memoized set in their stacks.

To identify the sets of common neighbors that require memoization, we traverse the result tree and keep track of the last visited node assigned with each vertex. A node is marked for memoization if the lowest common ancestor of itself and the last visited node of the same vertex is its parent. During the procedure `CompleteUpper`, we store the sets of common neighbors in stacks as necessary, and the size of memoized sets decreases with the increasing depth of the tree.

Neighbors compression. For each direct child node of the root of the result tree, the set of common neighbors X' is composed of the neighbors of the vertex on that node. Since the sets of common neighbors in all nodes beneath it are subsets of X', we use indices in the sorted list of X' to represent the vertices in each subset. Specifically, we employ a bitset to store the common neighbors, where each vertex in X' is represented by a single bit to indicate the presence or absence.

Table II demonstrates the use of bitsets to represent sets of common neighbors for the partial result tree in Figure 6a. For the neighbors of v_1 (i.e., u_1, u_2, and u_5), we use three bits to indicate the state of each vertex. For example, the set of common neighbors for v_1 and v_4 is $\{u_2, u_5\}$, which is represented by the bitset 011. We then divide each bitset based

on the number of bits that can be held by an integer and convert consecutive bits to integers where the lower bits denote smaller indices. We store each integer in a pair, along with the index of its first bit. For instance of 2 bits here, the neighbors of v_4 are represented as $\{0, (10)_2 = 2\}$ and $\{2, (01)_2 = 1\}$.

The compressed representation of neighbor lists for each vertex is different in every subtree under the root of the result tree, owing to the different assigned vertex on the root node of the subtree. Hence, we only build compressed neighbor lists for vertices that exist in the subtree. The vertex set X in Algorithm 4 is also replaced by a compressed list, and computing the intersection is to perform a bitwise AND operation for each pair of the same beginning index and append the non-zero results into a new compressed list.

We combine the compression of neighbors with the intersection memoization. We push those compressed neighbor lists into stacks if they can be used for further computations. Table II shows some examples of intersection results. For instance, the common neighbors of path $\{v_1, v_2, v_5\}$ are $\{u_1, u_2, u_5\}$, and the compressed format is $\{0, 3\}\{2, 1\}$, where the second pair is removed when we traverse into v_6 and the third bit becomes zero. To obtain the common neighbors of $\{v_1, v_2, v_3, v_4, v_5\}$, we compute intersection of $\{2, 1\}$ from $\{v_1, v_2, v_3, v_4\}$ and $\{0, 1\}\{2, 1\}$ from the stack of v_5, and only the second pair remains.

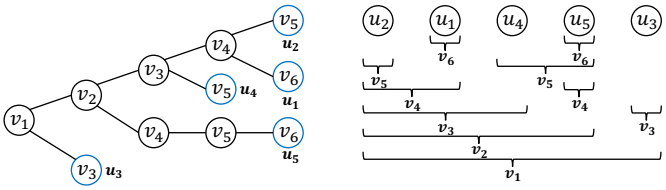
VI. Pre-process and Space Optimization

To enhance the performance of our approaches, we propose pre-processing techniques that are based on the order of vertices and introduce methods for reducing space usage.

A. Reclaim Obsolete Memory

During the recursion for finding the lower layers of maximal bicliques, the order of lower vertices in a path in candidate trees is determined by their ascending vertex IDs. Both the insertion into the result tree and the extraction operation for derivation preserve the order of vertices in paths. To enhance the merging of prefixes on prefix trees, we sort all the lower vertices on the graph by non-ascending degrees and assign new vertex IDs to each, ranging from 0 to $|V|$.

Remind that each candidate tree is traversed using DFS to obtain the lower layers of candidate maximal bicliques, we define the order of visiting end-nodes in a candidate tree as the **visiting order** for the paths of candidate maximal bicliques. Considering an initial candidate tree built using the neighbors of each upper vertex on the graph, we assign each upper vertex to the end-node of a path that corresponds to its neighbors in the tree. For instance, we mark the upper vertices based on the paths of their neighbors in Figure 7a which is constructed from Figure 1 after sorting and assigning new vertex IDs to the lower vertices. According to their assigned nodes, we obtain the visiting order for all upper vertices, which is $\{u_2, u_1, u_4, u_5, u_3\}$ in this example. We use notations like $u_2 \prec u_1$ to denote the precedence in this order. Based on properties of prefix trees, the visiting order for upper vertices coincides with the lexicographical order of their neighbors.



(a) Paths for upper vertices (b) Intervals over visiting order

Fig. 7: Visiting order (lexicographical order of neighbors)

As described in Algorithm 2, we process the upper vertices sequentially in the visiting order. By building an initial candidate tree using the neighbors of vertex u, we ensure that all the found maximal bicliques derived from it have u in the upper layer. Furthermore, after the recursion process, we partition the result tree based on the neighbors of u and remove the part which is no longer used to reclaim memory. We apply the same definition of the visiting order to the result tree and present the following lemma.

Lemma 4: Let u be an upper vertex, p be a path corresponding to $\Gamma(u)$, and P be the set of all paths before p in the visiting order of the result tree. All candidate paths generated in the recursive subproblems derived from p do not include any $p' \in P$ for uniqueness checks.

Proof: Considering the extraction operation for derivation, p forks nodes from the prior paths in the visiting order where the nodes have the same vertices in p, and the traversal by DFS visits the children of a node in the ascending order of their vertex IDs. Hence, before the last pruning step in §IV-A, p is the first path in the visiting order of the extracted tree. Inserting the extracted candidates into the result tree only contributes to paths after p in the visiting order, and the same manner establishes recursively for derived subproblems. Therefore, the uniqueness checks for candidates in subproblems will never use any $p' \in P$. □

Consequently, when selecting each $u \in U$ in the visiting order, the neighbors of u can partition the result tree into two parts, where the first part contains only paths that are no longer required and can be safely destroyed. However, there is still a portion of the result tree that remains in memory until all upper vertices have been processed. In the subsequent part, we will explore how to enable the complete removal of the result tree each time we finish processing one upper vertex.

B. Narrow the Search Space

As a side-effect of MBET, there is no limitation on the upper vertices during the first step as we only maintain the lower layers. To address this issue, we propose a variant of our algorithm with an additional restriction on the upper vertices. Specifically, when starting from each upper vertex u, we enumerate maximal bicliques with the upper vertices before u in the visiting order. Note that during this process, if a biclique can be enlarged by a vertex u' that meets $u \prec u'$, then u' must have edges connected to all vertices in the lower layer of the biclique and we should omit such candidate bicliques. To implement this restriction, we slightly modify

Algorithm 5: Space optimized algorithm MBETM

Input : $G = (U, V, E)$
Output: All maximal bicliques in G

1 **Algorithm** `MBETM(`G`)`
2 **sort** vertices in V by non-ascending degrees
3 **compute** the visiting order of U with intervals
4 **foreach** $u \in U$ *in the visiting order* **do**
5 $\mathcal{T}_{res} \leftarrow \{\Gamma(u)\}$
6 $\mathcal{T}_{cand} \leftarrow \{\Gamma(u) \cap \Gamma(u') \mid u' \in U \wedge u' \prec u\}$
7 `FindLower(`$\mathcal{T}_{cand}$, $\mathcal{T}_{res}$`)` with extended check using the intervals for deduplication
8 `CompleteUpper(Root(`$\mathcal{T}_{res}$`)` , U, $\varnothing$`)`

the uniqueness check while traversing the candidate tree. In Line 16 of Algorithm 3, we also check that the set of assigned vertices on $path$ is not a subset of the neighbors of any vertex after the initial selected upper vertex in the visiting order.

Lemma 4 is also established as we select $u \in U$ in the visiting order, and the result tree generated by u will not include bicliques with upper vertices after u so that it can be safely destroyed. Moreover, we take advantage of the visiting order to speed up the additional workload for the restriction on upper vertices. Specifically, as the order of upper vertices corresponds to end-nodes on a prefix tree, some consecutive vertices may have the same prefix for their neighbors in the tree. For each lower vertex v, we save some intervals of indices over the visiting order to indicate that v is a neighbor of those upper vertices. Each interval is recorded as large as possible. Figure 7b depicts the intervals that meet the candidate tree in Figure 7a. The vertex v_4 has neighbors $\{u_1, u_2, u_5\}$, hence it covers the first, second, and fourth indices of the visiting order. We merge them as two intervals $[0, 1]$ and $[3, 3]$ for v_4, and similarly, v_1 is represented by only one interval $[0, 4]$ which covers all the upper vertices. Therefore, a $path$ is a subset only if the intervals of every vertex in $path$ cover the same index which is greater than the index of the initially selected upper vertex. For example, assuming we have a set of lower vertices $\{v_1, v_4\}$ generated from the candidate tree initialized by u_4, whose index is 2, it is not a result currently as the index 3 is covered by intervals of both v_1 and v_4.

We present the optimized algorithm, MBETM, in Algorithm 5. Both the result tree and the candidate tree are initialized each time for a selected upper vertex (Line 5-6). During the recursion, we check for subsets using the intervals and perform the completion of the upper layers on the entire tree (Line 7-8). This algorithm requires additional computation time for generating the intervals and checking uniqueness, and also requires more initial memory for storing the intervals. However, it greatly reduces the memory usage of the result tree as it is limited to each upper vertex. It is a viable option for large graphs where memory usage is a concern.

C. Complexity Analysis

Since candidate trees serve a vital purpose in our recursive enumeration, we first discuss the size bound for each candidate tree. We use notations $d_{max}(U)$, $d_{max}(V)$, and d_{max} to represent the maximum degree of each layer and the entire graph, respectively, and use notation d_{2max} for the maximum 2-hop degree. The initial candidate tree for each upper vertex is built by its neighbors and the intersections with other vertices. Hence, the length of a candidate path is bounded by $O(d_{max}(U))$, and the length of jump links from a tree node is bounded by $O(d_{max}(V))$. Consequently, the overall size of each candidate tree is $O(d^2_{max})$. Note that the merging of prefixes can omit many nodes in practice, and the extraction process reduces the tree size as the recursion proceeds.

Given a bipartite graph with the number of maximal bicliques $\mathcal{B}$, the total time complexity of MBET and MBETM is $O(\mathcal{B}d^3_{max}log(d_{2max}))$. Here $log(d_{2max})$ is related to the structure of prefix trees. An additional workload is required to insert and search for outgoing links of one tree node, and the outgoing degree is bounded by the maximum 2-hop degree of lower vertices. We then discuss the complexity of computing the lower and upper layers, respectively.

Each time we find the lower layer of a new maximal biclique, we need to perform an extraction operation to build a new candidate tree, traverse the tree and check the uniqueness of each candidate path on it. The extraction process starts from each node on the path corresponding to the maximal biclique and visits all nodes following the jump links. For the jump links starting from one node, we visit every node on all previous jump links at most once. Hence, the total time of the extraction for a path is $O(d^2_{max}(U)d_{max}(V))$. To check the uniqueness of each candidate path, MBET requires $O(|path|)$ for the insertion into the result tree and MBETM requires an extra $O(|path|d_{max}(V))$ for the extended check using intervals where each interval is used at most once. The complexity of checking the uniqueness of all paths on a candidate tree is $O(d_{max}(U)d^2_{max}(V))$, resulting in an overall cost of $O(\mathcal{B} \cdot d^3_{max})$ for the enumeration of lower layers.

Completing the upper layer for each maximal biclique requires $O(d_{max}(U)d_{max}(V))$ to find the common neighbors of the lower vertices. We visit each node on the result tree once and both the compression and intersection operations have a linear time complexity corresponding to the length of the neighbor lists. The memoization on prefix trees reduces duplicated intersection computation, and the average length of the compressed lists is further decreased as we eliminate pairs of all-zero bits. As a result, the cost of completing all the upper layers is much smaller in practice than the overall bound $O(\mathcal{B} \cdot d^2_{max})$.

The space complexity depends on the result tree and candidate trees. During the recursion for each upper vertex, there are at most $O(d_{max}(V))$ candidate trees in memory at the same time, as each derivation inherently adds one upper vertex to all the candidate maximal bicliques in the extracted tree. The total space usage for candidate trees is $O(d^3_{max})$ but the average tree size is much smaller in practice. As for the result tree, MBET has no limitation for the search space, resulting in a space usage of $O(\sum(|V(B)|))$. MBETM utilizes a new result tree for each upper vertex, leading to a space usage of $O(\max_u \sum(|V(\bar{B}(u))|))$, where $\bar{B}(u)$ denotes maximal

Dataset	$\|U\|$	$\|V\|$	$\|E\|$	$\|MB\|$	iMBEA	PMBE	FMBE	ooMBEA	MBET	MBETM
YouTube	30,087	94,238	293,360	1,826,587	641	905	196.4	57.93	4.96	5.25
BibSonomy	204,673	767,447	2,499,057	2,507,269	7169	5694	691.6	82.62	9.21	9.43
StackOverflow	96,678	545,195	1,301,942	3,320,824	15298	110083	17028.0	426.22	11.40	14.67
DBLP	1,953,085	5,624,219	12,282,059	4,899,032	—	—	35.9	26.43	14.63	17.31
IMDB	303,617	896,302	3,782,463	5,160,061	35312	11714	1557.5	103.92	19.72	21.20
Amazon	1,230,915	2,146,057	5,743,258	7,551,358	—	119745	93436.3	340.81	38.34	49.04
BookCrossing	105,278	340,523	1,149,739	54,458,953	54778	44647	23240.2	3976.42	213.29	228.42
Github	56,519	120,867	440,237	55,346,398	123020	—	48292.8	4384.24	203.56	258.71
TV Tropes	64,415	87,678	3,232,134	19,636,996,096	∼2m (24h)	∼2m (24h)	∼84m (24h)	∼594m (24h)	OOM (4h)	∼5690m (24h)
LiveJournal	3,201,203	7,489,073	112,307,385	>15b	<1m (24h)	<1m (24h)	∼2m (24h)	∼941m (24h)	OOM (6h)	∼2078m (24h)

TABLE III: Data statistics and overall execution time (seconds)

bicliques whose lower layers are formed by u and other vertices before u in the visiting order.

VII. Experiments

A. Setup

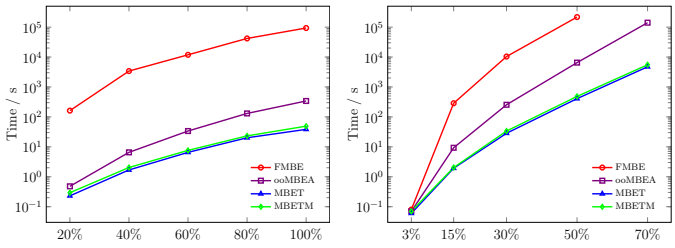
Datasets. We choose 10 real-world datasets from KONECT [30] for our experiments. The detailed statistics are shown in Table III. We show the number of upper vertices ($|U|$), the number of lower vertices ($|V|$), the number of edges ($|E|$), and the number of maximal bicliques ($|MB|$) in the table.

Algorithms. We compare our algorithms MBET and MBETM with four prior works in recent years: iMBEA [19], PMBE [21], FMBE [20], and ooMBEA [22]. All of them run against one single core. For iMBEA and PMBE, we re-implement them in C++, and our versions achieve better performance than their open-source version and Java version, respectively. We also implement FMBE by ourselves as they do not provide code. For ooMBEA, we adopt their open-source code for evaluation. All the algorithms are compiled using GCC with the optimization level set to O3. Each algorithm may swap the two sides of graphs or reorder the vertices for its convenience of faster enumeration. All the experiments are conducted on a computer equipped with AMD Ryzen Threadripper 3990X CPU and 128GB RAM. For each algorithm, we report the average result over 10 runs and terminate the algorithm if it runs more than 48 hours.

B. Overall Comparison

Table III shows the time of all algorithms in seconds, and "—" means it cannot be finished within 48 hours. Besides the normal graphs, we also examine the overall running on TV Tropes and LiveJournal, which both have several billions of maximal bicliques and all algorithms cannot finish in a few days. We report the average number of maximal bicliques each algorithm enumerates within 24 hours.

In general, our algorithms outperform others on all the datasets. For iMBEA and PMBE, both of them need to take more than 24 hours to complete the enumeration for Amazon, Github, and DBLP, and they are always slower than FMBE and ooMBEA. FMBE takes the approach of reducing the initial candidate set by two-hop neighbors, and performs exceptionally well on the sparsest graph, DBLP. On StackOverflow, BookCrossing, and Github, our algorithms significantly outperform FMBE and ooMBEA by several orders of magnitude. In addition, on LiveJournal, MBETM is the only algorithm that



(a) Percentage of vertices (Amazon) (b) Percentage of edges (TV Tropes)

Fig. 8: Execution time on subgraphs

can enumerate maximal bicliques in billions scale. Note that MBETM has an affordable overhead compared with MBET as it needs extra workloads due to the extended check for deduplication. But it solves the problem of MBET running out of memory on graphs with a vast number of results. In the following experiments, we omit the algorithms iMBEA and PMBE as they are hard to handle large graphs.

Scalability and varying density. We examine the scalability of algorithms and depict the trends of execution time in Figure 8a. We randomly select 20%, 40%, 60% and 80% of vertices for both layers of the graph and built the induced subgraph by them on the dataset Amazon. Our proposed algorithms are always faster than state-of-the-art algorithms. As the size of graphs grows, the gap between ooMBEA and our algorithms is expanding, meaning that MBET and MBETM can be easier to handle larger datasets compared to other algorithms.

To evaluate the effects of density, we utilize the dataset TV Tropes to generate subgraphs by randomly selecting 3%, 15%, 30%, 50%, and 70% of the edges while keeping all vertices in the original graph. As depicted in Figure 8b, ooMBEA exhibits similar execution times to our algorithms on sparse graphs but gets closer to FMBE as they become denser, verifying the results of overall execution time on the datasets BookCrossing and Github. Regarding our algorithms, MBETM is always a little slower than MBET on all induced graphs, indicating that the overhead is irrelevant to either scalability or density.

C. Breakdown Analysis

To take a deeper dive into our algorithms, we break down one execution into three parts: pre-processing, finding the lower layers, and then the upper layers, as shown in Table IV. Both MBET and MBETM use the same technique of prefix trees to enumerate, but the search space is different.

Dataset	MBET			MBETM		
	Pre	Lower	Upper	Pre	Lower	Upper
YouTube	0.05	4.04	0.87	0.07	4.30	0.88
DBLP	4.64	6.90	3.09	6.99	7.21	3.11
IMDB	0.81	10.74	8.17	1.22	11.03	8.94
Amazon	1.77	20.69	15.88	2.59	24.07	22.37
Github	0.08	179.84	23.65	0.11	240.48	18.13

TABLE IV: Breakdown analysis

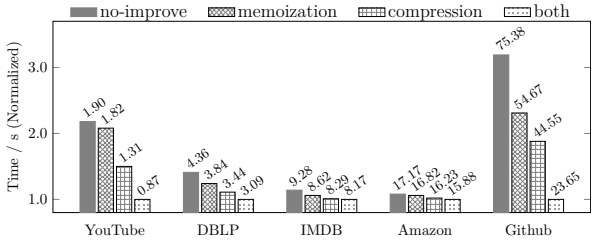


Fig. 9: Effects of improvements for intersection

Time of each part. For pre-processing, both MBET and MBETM sort vertices and compute the visiting order, while MBETM needs more time to compute and save the intervals over the order. We do not need to visit each neighbor of the upper vertex in MBET when it is the only vertex in a subtree that can determine its position in the visiting order. It can lead to a 1.5x difference in the time of pre-processing. Finding the lower layer of all maximal bicliques is the most fundamental part of our algorithms. It takes up to 93% of the overall time for the graph Github. In this part, MBETM needs more time than MBET since they enumerate the same number of bicliques while MBETM has an extended check for uniqueness. The difference in time of completing the upper layers varies in different graphs. MBETM needs more time on Amazon but less time on Github. This is because the result tree in MBETM is computed entirely and destroyed immediately each time we find the maximal bicliques for one upper vertex. The search space and traversal on the result tree are narrowed. However, on the larger graph Amazon, the advantages of merging prefixes and improvements for intersections in MBETM are not as effective as in MBET.

Optimizations for the neighbor set intersection. We also evaluated the impact of our optimizations on the intersection of neighbors. As described in §V-B, we propose two methods: memoization of previous intersection results and bitwise compression for neighbor lists. We first run the version without both methods and then with only one of them each. We use MBET for the comparison, as MBETM does not alter the logic of this process. The results are presented in Figure 9, where the height of each bar is normalized to the fastest one, and the actual time is labeled on the top.

Across all datasets, using memoization leads to faster execution time compared to the version without any optimizations. However, it is worth noting that compressing the neighbors often introduced a greater speedup. This is because while memoization saves time in computing intersections, it requires additional time and memory to store and access the memoized results. On the other hand, bitwise compression significantly reduces the number of comparison for intersection computations and save more time as the size of the neighbor list is reduced. Consequently, employing memoization with compressed neighbors further improved performance. In practice, combining both optimizations achieved a maximum speedup of more than 3× on the dataset Github.

Dataset	FMBE	ooMBEA	MBET	MBETM
Youtube	73	36	178	112
BibSonomy	583	151	671	615
StackOverflow	350	432	424	379
DBLP	3564	980	3560	3771
IMDB	784	508	836	800
Amazon	1601	1308	1901	1658
BookCrossing	265	189	4295	867
Github	106	102	2038	309
TV Tropes (48h)	418	1333	>128G	22432
LiveJournal (48h)	15949	12351	>128G	44299

TABLE V: Memory consumption (MB)

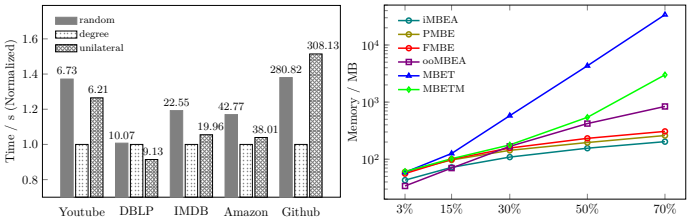
Effects of vertex orders. We also discuss the effects of different orders of vertices on the execution time. We run MBET with random order of vertices and also adopt the unilateral order from ooMBEA for comparison. Figure 10a shows the execution time after pre-processing, normalized to the time we report in Table IV. The random order always takes more time than sorting as non-ascending degrees, but for DBLP, the unilateral order has a little better performance as it is designed for sparse graphs.

However, there is a big gap in dense graphs such as Github, where using unilateral order even performed worse than the random order. An important reason is that our algorithms depend on the merging of prefixes and the order of non-ascending degrees can produce more common paths in prefix trees. Note that the unilateral order aims at minimizing the number of two-hops neighbors that do not align with the idea of merging. Also, it takes more cost to sort as the unilateral order than by degrees. As a result, we adopt the non-ascending degrees for sorting vertices in our algorithms.

D. Memory Consumption

Table V shows the memory usage on all the datasets, reported as the high-water mark of a Linux process for each algorithm. We can see that our algorithms have more space usage than existing approaches since we utilize the prefix tree structure to accelerate the enumeration process. Note that MBETM reduces the space usage of MBET in many cases such as the dense graph Github and the huge graph LiveJournal.

Varing density. We also examined the memory usage for algorithms by varying graph densities. Figure 10b shows the relationship between memory and density for all algorithms on the subgraphs of TV Tropes, as used in Figure 8b. From sparse to dense, the memory requirements for our algorithms exhibit exponential growth. While initially MBETM requires slightly more memory than MBET due to pre-processing, the benefit of narrowing the search space becomes increasingly evident as the density increases.



(a) Execution time varying reorder (b) Memory usage varying density

Fig. 10: Effectiveness comparisons

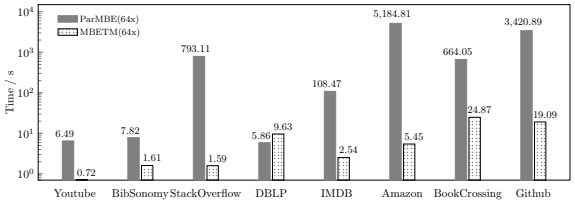


Fig. 11: Execution time for parallel enumeration

E. Parallel Enumeration

We simply extend our algorithm MBETM into a multi-thread environment and compare it with the existing parallel enumeration algorithm, ParMBE [20]. After the pre-processing in MBETM, each thread fetches one upper vertex in the visiting order and performs the enumeration starting from it. Figure 11 shows the execution time of MBETM and ParMBE in 64 threads. Our algorithm outperforms the existing parallel algorithm in most datasets, except the DBLP where the pre-processing takes much time and occupies the major portion of the whole execution.

VIII. Related Works

Maximal biclique enumeration. Earlier approaches to finding maximal bicliques in bipartite graphs often involve transforming the problem into other domains, such as maximal clique enumeration in unipartite graphs [31], [32] and frequent closed itemset mining in transactional databases [33]–[35]. Inspired by the Bron-Kerbosch algorithm [23] for the problem of maximal clique enumeration, [19] proposes iMBEA with a depth-first search manner for enumerating maximal bicliques. Based on that, PMBE [21] adopts the pivot pruning technique to reduce the search space, and ooMBEA [22] explores the order-based and batch-pivot-based techniques to achieve better performance, especially on sparse graphs. For concurrent enumeration of maximal bicliques, [36] clusters the graph into small subgraphs and processes different subgraphs in parallel using the MapReduce framework. Additionally, [20] proposes ParMBE with improvements in reordering vertices, and the sequential version, FMBE, is also introduced showing that it achieves practical speedup.

Enumerating maximal bicliques in unipartite graphs has also been the subject of research for many years. The Consensus algorithm [37] shares a similar idea of focusing on the lower layer of maximal bicliques. It maintains a set of simple bicliques and gradually expanding it through transformations of absorption and consensus adjunction. However, the algorithm neither employs a depth-first search manner nor gives attention to the organization of intermediate results. Moreover, [38] introduces the MineLMBC algorithm, based on the divide-and-conquer method, which prunes the search space by size constraints during recursion. [39] discusses maximal induced bicliques in general graphs, and solving the enumeration problem with maximal independent sets.

Various biclique problems in bipartite graphs. Some other research works focus on mining constrained bicliques in bipartite graphs. [40] investigates bicliques with a size constraint t for one layer while maximizing coverage on the other layer, and proposes a heuristic algorithm for finding the top-k results. One more classic problem is finding bicliques with fixed-size vertices, known as the (p, q)-biclique problem. [41] discusses the density and other properties of (p, q)-bicliques in bipartite graphs. [12] presents an enumeration algorithm suitable for large-scale sparse graphs, with optimizations such as pre-allocated arrays and vertex renumbering. A special case where both layers have only two vertices is studied as counting butterflies [42], [43], which is also a fundamental structure.

Another biclique problem is finding the largest biclique where both layers have the same size (maximum balanced biclique), which is proven to be NP-hard in bipartite graphs [44]. To tackle this problem, both heuristic algorithms [45], [46] and exact algorithms [16], [47] have been proposed. Another NP-hard problem is finding the biclique with the maximum number of edges [48] (maximum edge biclique). [49] proposes an integer programming-based method, and [8] presents a progressive bounding framework to narrow the search space. Recently, an extended problem has been studied in [50], which aims to find the maximum biclique containing a specific vertex by online algorithms and index-based techniques.

IX. Conclusion

In this paper, we study the problem of enumerating maximal bicliques on bipartite graphs. We present a method separating the search process into two steps and utilizing prefix trees for efficient enumeration. Our proposed algorithm, called MBET, introduces the candidate tree and the result tree to find and save the lower layers of all maximal bicliques in the first step. Then it uses the saved results to compute the upper layer of each maximal biclique. We also propose a variant algorithm, MBETM, adding some extra workloads to significantly save memory usage. Extensive experiments are conducted on real-world datasets to demonstrate the superior performance of our methods compared to existing algorithms.

X. Acknowledgment

This work was partially supported by (i) the National Key Research and Development Program of China 2020AAA0108503, and (ii) NSFC Grants U2241211, 62072034, 62302294. Rong-Hua Li is the corresponding author of this paper.

REFERENCES

[1] 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.

[2] X. Chen, K. Wang, X. Lin, W. Zhang, L. Qin, and Y. Zhang, "Efficiently answering reachability and path queries on temporal bipartite graphs," *Proc. VLDB Endow.*, vol. 14, no. 10, pp. 1845–1858, 2021.

[3] M. Ley, "The DBLP computer science bibliography: Evolution, research issues, perspectives," in *SPIRE*, vol. 2476, 2002, pp. 1–10.

[4] D. Zhou, S. A. Orshanskiy, H. Zha, and C. L. Giles, "Co-ranking authors and documents in a heterogeneous network," in *ICDM*, 2007, pp. 739–744.

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

[6] X. Li and H. Chen, "Recommendation as link prediction in bipartite graphs: A graph kernel-based machine learning approach," *Decis. Support Syst.*, vol. 54, no. 2, pp. 880–890, 2013.

[7] V. E. Lee, N. Ruan, R. Jin, and C. C. Aggarwal, "A survey of algorithms for dense subgraph discovery," in *Managing and Mining Graph Data*, ser. Advances in Database Systems, C. C. Aggarwal and H. Wang, Eds. Springer, 2010, vol. 40, pp. 303–336.

[8] B. Lyu, L. Qin, X. Lin, Y. Zhang, Z. Qian, and J. Zhou, "Maximum biclique search at billion scale," *Proc. VLDB Endow.*, vol. 13, no. 9, pp. 1359–1372, 2020.

[9] K. Wang, W. Zhang, X. Lin, Y. Zhang, L. Qin, and Y. Zhang, "Efficient and effective community search on large-scale bipartite graphs," in *ICDE*, 2021, pp. 85–96.

[10] R. Yoshinaka, "Towards dual approaches for learning context-free grammars based on syntactic concept lattices," in *Developments in Language Theory - 15th International Conference*, vol. 6795, 2011, pp. 429–440.

[11] A. S. Muhammad, P. Damaschke, and O. Mogren, "Summarizing online user reviews using bicliques," in *42nd International Conference on Current Trends in Theory and Practice of Computer Science*, vol. 9587, 2016, pp. 569–579.

[12] J. Yang, Y. Peng, and W. Zhang, "(p, q)-biclique counting and enumeration for large sparse bipartite graphs," *Proc. VLDB Endow.*, vol. 15, no. 2, pp. 141–153, 2021.

[13] T. Santos, S. Walk, R. Kern, M. Strohmaier, and D. Helic, "Activity archetypes in question-and-answer (Q8A) websites - A study of 50 stack exchange instances," *ACM Trans. Soc. Comput.*, vol. 2, no. 1, pp. 4:1–4:23, 2019.

[14] W. Zhang, Z. Chen, C. Dong, W. Wang, H. Zha, and J. Wang, "Graph-based tri-attention network for answer ranking in CQA," in *AAAI*, 2021, pp. 14 463–14 471.

[15] A. Das and S. Tirthapura, "Incremental maintenance of maximal bicliques in a dynamic bipartite graph," *IEEE Trans. Multi Scale Comput. Syst.*, vol. 4, no. 3, pp. 231–242, 2018.

[16] L. Chen, C. Liu, R. Zhou, J. Xu, and J. Li, "Efficient exact algorithms for maximum balanced biclique search in bipartite graphs," in *SIGMOD*, 2021, pp. 248–260.

[17] 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.

[18] E. Prisner, "Bicliques in graphs I: bounds on their number," *Comb.*, vol. 20, no. 1, pp. 109–117, 2000.

[19] Y. Zhang, C. A. Phillips, G. L. Rogers, E. J. Baker, E. J. Chesler, and M. A. Langston, "On finding bicliques in bipartite graphs: a novel algorithm and its application to the integration of diverse biological data types," *BMC Bioinform.*, vol. 15, p. 110, 2014.

[20] A. Das and S. Tirthapura, "Shared-memory parallel maximal biclique enumeration," in *HiPC*, 2019, pp. 34–43.

[21] A. Abidi, R. Zhou, L. Chen, and C. Liu, "Pivot-based maximal biclique enumeration," in *IJCAI*, 2020, pp. 3558–3564.

[22] L. Chen, C. Liu, R. Zhou, J. Xu, and J. Li, "Efficient maximal biclique enumeration for large sparse bipartite graphs," *Proc. VLDB Endow.*, vol. 15, no. 8, pp. 1559–1571, 2022.

[23] C. Bron and J. Kerbosch, "Finding all cliques of an undirected graph (algorithm 457)," *Commun. ACM*, vol. 16, no. 9, pp. 575–576, 1973.

[24] E. Tomita, A. Tanaka, and H. Takahashi, "The worst-case time complexity for generating all maximal cliques and computational experiments," *Theor. Comput. Sci.*, vol. 363, no. 1, pp. 28–42, 2006.

[25] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *ACM SIGMOD International Conference on Management of Data*, 2000, pp. 1–12.

[26] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[27] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: Artful indexing for main-memory databases," in *ICDE*, 2013, pp. 38–49.

[28] R. Jampani and V. Pudi, "Using prefix-trees for efficiently computing set joins," in *DASFAA*, vol. 3453, 2005, pp. 761–772.

[29] Y. Luo, G. H. L. Fletcher, J. Hidders, and P. D. Bra, "Efficient and scalable trie-based algorithms for computing set containment relations," in *ICDE*, 2015, pp. 303–314.

[30] J. Kunegis, "KONECT: the koblenz network collection," in *WWW*, 2013, pp. 1343–1350.

[31] K. Makino and T. Uno, "New algorithms for enumerating all maximal cliques," in *9th Scandinavian Workshop on Algorithm Theory*, vol. 3111, 2004, pp. 260–272.

[32] A. Gély, L. Nourine, and B. Sadi, "Enumeration aspects of maximal cliques and bicliques," *Discret. Appl. Math.*, vol. 157, no. 7, pp. 1447–1459, 2009.

[33] M. J. Zaki and C. Hsiao, "CHARM: an efficient algorithm for closed itemset mining," in *SIAM International Conference on Data Mining*, 2002, pp. 457–473.

[34] T. Uno, M. Kiyomi, and H. Arimura, "LCM ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets," in *FIMI*, 2004.

[35] J. Li, G. Liu, H. Li, and L. Wong, "Maximal biclique subgraphs and closed pattern pairs of the adjacency matrix: A one-to-one correspondence and mining algorithms," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 12, pp. 1625–1637, 2007.

[36] A. P. Mukherjee and S. Tirthapura, "Enumerating maximal bicliques from a large graph using mapreduce," *IEEE Trans. Serv. Comput.*, vol. 10, no. 5, pp. 771–784, 2017.

[37] G. Alexe, S. Alexe, Y. Crama, S. Foldes, P. L. Hammer, and B. Simeone, "Consensus algorithms for the generation of all maximal bicliques," *Discret. Appl. Math.*, vol. 145, no. 1, pp. 11–21, 2004.

[38] G. Liu, K. Sim, and J. Li, "Efficient mining of large maximal bicliques," in *Data Warehousing and Knowledge Discovery, 8th International Conference*, vol. 4081, 2006, pp. 437–448.

[39] D. Hermelin and G. Manoussakis, "Efficient enumeration of maximal induced bicliques," *Discret. Appl. Math.*, vol. 303, no. 1, pp. 253–261, 2021.

[40] A. Abidi, L. Chen, C. Liu, and R. Zhou, "On maximising the vertex coverage for top $-k$ t-bicliques in bipartite graphs," in *ICDE*, 2022, pp. 2346–2358.

[41] M. Mitzenmacher, J. Pachocki, R. Peng, C. E. Tsourakakis, and S. C. Xu, "Scalable large near-clique detection in large-scale networks via sampling," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015, pp. 815–824.

[42] S. Sanei-Mehri, A. E. Sariyüce, and S. Tirthapura, "Butterfly counting in bipartite networks," in *KDD*, 2018, pp. 2150–2159.

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

[44] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[45] Y. Wang, S. Cai, and M. Yin, "New heuristic approaches for maximum balanced biclique problem," *Inf. Sci.*, vol. 432, pp. 362–375, 2018.

[46] M. Li, J. Hao, and Q. Wu, "General swap-based multiple neighborhood adaptive search for the maximum balanced biclique problem," *Comput. Oper. Res.*, vol. 119, p. 104922, 2020.

[47] Y. Zhou, A. Rossi, and J. Hao, "Towards effective exact methods for the maximum balanced biclique problem in bipartite graphs," *Eur. J. Oper. Res.*, vol. 269, no. 3, pp. 834–843, 2018.

[48] M. Dawande, P. Keskinocak, J. M. Swaminathan, and S. R. Tayur, "On bipartite and multipartite clique problems," *J. Algorithms*, vol. 41, no. 2, pp. 388–403, 2001.

[49] M. Sözdinler and C. C. Özturan, "Finding maximum edge biclique in bipartite networks by integer programming," in *CSE*, 2018, pp. 132–137.

[50] K. Wang, W. Zhang, X. Lin, L. Qin, and A. Zhou, "Efficient personalized maximum biclique search," in *ICDE*, 2022, pp. 498–511.