# Enumeration of Billions of Maximal Bicliques in Bipartite Graphs without Using GPUs

Zhe Pan[1,2,3], Shuibing He[3], Xu Li[3], Xuechen Zhang[4], Yanlong Yin[3],
Rui Wang[1,2], Lidan Shou[1,2,3], Mingli Song[3], Xian-He Sun[5], Gang Chen[3]
[1]*The State Key Laboratory of Blockchain and Data Security, Zhejiang University*
[2]*Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security*
[3]*College of Computer Science and Technology, Zhejiang University*
[4]*Washington State University Vancouver* [5]*Illinois Institute of Technology*
{panzhe, heshuibing, fhxu, yinyanlong, rwang21, should, brooksong, cg}@zju.edu.cn,
xuechen.zhang@wsu.edu, sun@iit.edu

***Abstract*—Maximal biclique enumeration (MBE) is crucial in bipartite graph analysis. Recent studies rely on extensive set intersections on static bipartite graphs to solve the MBE problem. However, the computational subgraphs dynamically change during enumeration, leading to redundant memory accesses and degraded set intersection performance. To overcome this limitation, we propose an AdaMBE algorithm. First, we redesign its core operations using local neighborhood information derived from computational subgraphs to minimize redundant memory accesses. Second, we dynamically create computational subgraphs using bitmaps leveraging its fast bitwise operations to accelerate set intersections. Finally, we integrate them in AdaMBE. Our experimental results show that AdaMBE is 1.6×−49.7× faster than its closest CPU-based competitor and successfully enumerates all 19 billion maximal bicliques on the TVTropes dataset, a large task beyond the capabilities of existing algorithms. Notably, on certain datasets, our parallel version, ParAdaMBE, on CPUs even outperforms GMBE on GPUs by up to 5.07×.**

***Index Terms*—Bipartite graph, maximal biclique enumeration, bitmap**

## I. Introduction

Bipartite graphs are popular in diverse domains for capturing relationships between two different sets of entities. In a bipartite graph $G(U, V, E)$, there are two distinct vertex sets, $U$ and $V$, and the edges in set $E$ only connect vertices from these two sets. A biclique is a complete bipartite subgraph that includes all possible edges between the two sets of vertices. A maximal biclique is a biclique that cannot be further expanded to include more vertices. This paper aims to efficiently enumerate all maximal bicliques in large bipartite graphs, known as the MBE problem.

The MBE problem offers a versatile approach for analyzing various real-world scenarios. For instance, E-commerce giants like Alibaba, eBay, and Amazon use bipartite graphs to represent the user-purchase-item relationships with each edge denoting a purchase transaction [1]–[4]. Then, they use MBE algorithms to detect click farming [5] in which fraudulent users purchase a set of products on behalf of malicious merchants. Additionally, the MBE approach finds applications in gene expression analysis in biology [6]–[8], community search in social networks [9], [10], and information aggregation in Graph Neural Networks [11], [12]. As a result, the research on MBE has garnered significant attention from both academia and industry in recent years [6], [13]–[21].

Mainstream MBE algorithms [6], [15], [16], [18], [21]–[24] typically solve the MBE problem through backtracking using enumeration trees. The enumeration process involves extensive set intersections on static bipartite graphs. To enhance performance, researchers have proposed various techniques, such as vertex ordering [6], [16], [18], pruning [6], [16], [18], and parallelization [15], [21]. Recent efforts explored GPUs to accelerate MBE, leveraging hundreds of thousands of threads [21]. However, all of them ignore the characteristics of the computational subgraphs derived from the original bipartite graphs during enumeration, leading to limited scalability. For example, within 48 hours, they can only process bipartite graphs that contain no more than 263 million maximal bicliques, i.e., the CebWiki dataset [25] (Section IV).

Our research reveals three new observations regarding the computational subgraphs. First, the computational subgraphs dynamically change at runtime and their sizes can be much smaller than those of the original bipartite graphs. Second, the vertex in the computational subgraphs of the current node in the enumeration tree can be directly used for node generation. We do not need the neighborhood information from the original graphs. Third, the existing algorithms need to access vertices that do not belong to their corresponding computational subgraphs, leading to redundant memory accesses and degraded set intersection performance.

In this paper, we present AdaMBE, a novel MBE algorithm that exploits the characteristics of computational subgraphs to accelerate the enumeration process on CPUs. AdaMBE uses local neighborhood information derived from computational subgraphs in the key operations of MBE and a hybrid graph representation in memory to achieve both high memory efficiency and fast set intersections. Specifically, we propose two primary techniques and integrate them to enable AdaMBE.

Corresponding Author: Shuibing He

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/$31.00 ©2024 IEEE

First, we keep the current computational subgraph by maintaining local neighbors of vertices, which are essential intermediate results in recent studies [6], [15], [16], [18], [21]. Additionally, we leverage the local neighbor data to efficiently minimize accesses to unnecessary vertices, prevent repetitive set intersection operations, and direct node pruning.

Second, we employ bitmaps to represent small computational subgraphs. Recent studies [15], [16], [18], [21] avoid using the bitmap due to its extensive memory usage up to $O(|U| \times |V|)$. Because the sizes of computational subgraphs can be much smaller than those of the original static bipartite graphs, we could store the bitmaps of selective computational subgraphs in memory. Additionally, we use the high-performance bitwise operations to expedite set intersections required for key MBE operations such as node generation.

Finally, we design AdaMBE by integrating the two aforementioned approaches and then develop its parallel version, ParAdaMBE. Specifically, AdaMBE primarily employs the local neighborhood cache and its hybrid in-memory representation. It adaptively enables bitmap-based representations for small computational subgraphs.

In summary, we make the following contributions.

- We use derived computational subgraphs in the enumeration. Because the computational subgraphs corresponding to a vertex can be retrieved from its parent's neighborhood information, we don't need to access its neighborhood information in the original bipartite graph. Moreover, we store the parent's neighborhood information in a cache to support high-performance access.
- We introduce a hybrid representation of computational subgraphs in memory. AdaMBE selectively generates bitmap-based subgraphs for small computational subgraphs. This approach expedites extensive set intersections through efficient bitwise operations.
- We propose an adaptive maximal biclique enumeration algorithm, named AdaMBE, which integrates the neighborhood data cache and the bitmap representation.
- We conduct extensive experiments on real-world and synthetic datasets, including large datasets with billions of maximal bicliques. Experimental results show that AdaMBE outperforms its closest competitor by up to 49.7× and efficiently handles large datasets with billions of maximal bicliques. Our parallel version, ParAdaMBE, running on CPUs even outperforms GMBE [21] running on GPUs by up to 5.07× on three time-consuming datasets out of twelve general datasets.

## II. Background and Motivation

### A. Baseline MBE Solutions

**Problem formulation.** Our problem is defined over a bipartite graph $G(U, V, E)$, where $U$ and $V$ represent two disjoint vertex sets, and $E$ denotes the edge set with $E \subseteq U \times V$. The vertices in $U$ or $V$ are denoted by $u$ or $v$, respectively. $N(v)$ represents the set of neighbors of vertex $v$, and $\Gamma(X)$ represents the common neighbors of vertices in the vertex set $X$, i.e., $\Gamma(X) = \bigcap_{v \in X} N(v)$. $\Delta(v)$ denotes the degree of vertex $v$, i.e., the number of vertices it is connected to, and $\Delta(X)$ denotes the maximum degree among the vertices in set $X$. A biclique $B(L, R, E')$ is a complete subgraph of $G$ where $L \subseteq U$, $R \subseteq V$, and $E' = L \times R \subseteq E$. We denote the biclique $B$ as the set pair $(L, R)$ for brevity. A maximal biclique is a biclique $B(L, R)$ that is not a proper subset of any other biclique $B'(L', R')$ in $G$, namely $(L \cup R) \not\subset (L' \cup R')$. For instance, Figure 1 shows a bipartite graph $G_0$ and one of maximal bicliques in $G_0$. This paper focuses on efficiently *enumerating all maximal bicliques in large bipartite graphs*, which is known as the MBE problem.

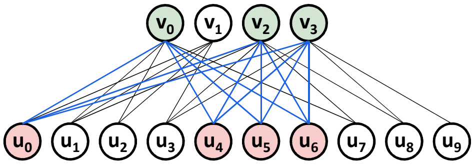


Fig. 1. An example of a bipartite graph $G_0$ and a maximal biclique $(\{u_0, u_4, u_5, u_6\}, \{v_0, v_2, v_3\})$ in $G_0$.

**Algorithm 1:** Baseline MBE Solution

```
   Input: Bipartite graph $G(U,V,E)$
   Output: All maximal bicliques
 1 biclique_search($U, \emptyset, V$);
 2 procedure biclique_search($L, R, C$):
 3     foreach $v' \in C$ do
 4         $L' \leftarrow L \cap N(v')$; $R' \leftarrow R$; $C' \leftarrow \emptyset$;
 5         foreach $v_c \in C$ do          // Node generation
 6             if $L' \cap N(v_c) = L'$ then
 7                 $R' \leftarrow R' \cup \{v_c\}$;
 8             else if $L' \cap N(v_c) \neq \emptyset$ then
 9                 $C' \leftarrow C' \cup \{v_c\}$;
10         if $R' = \Gamma(L')$ then             // Node check
11             Output($L', R'$) as a maximal biclique;
12             biclique_search($L', R', C'$)
13         $C \leftarrow C \setminus \{v'\}$;
```

**Baseline approach.** Recent works [6], [15], [16], [18], [21]–[24] commonly employ backtracking to generate an enumeration tree for MBE on a static bipartite graph $G(U, V, E)$. This approach guarantees that each maximal biclique is represented within a single node in the enumeration tree. Typically, most of these works [15], [16], [21]–[24] use a 3-tuple $(L, R, C)$ to represent each enumeration tree node, where $L$ is a subset of $U$, while $R$ and $C$ are two disjoint subsets of $V$. $(L, R)$ represents the current biclique, while $C$ contains candidate vertices for expanding $R$ during the backtracking. Each node corresponds to a unique biclique $(L, R)$ because their $R$ sets differ during backtracking. Consequently, all maximal bicliques are derived from this enumeration tree.

In the following, we present the baseline MBE approach in Algorithm 1. The algorithm starts at a root node $(U, \emptyset, V)$ (line #1) and then recursively computes each node $(L, R, C)$ using the biclique_search procedure (line #2). Specifically, the

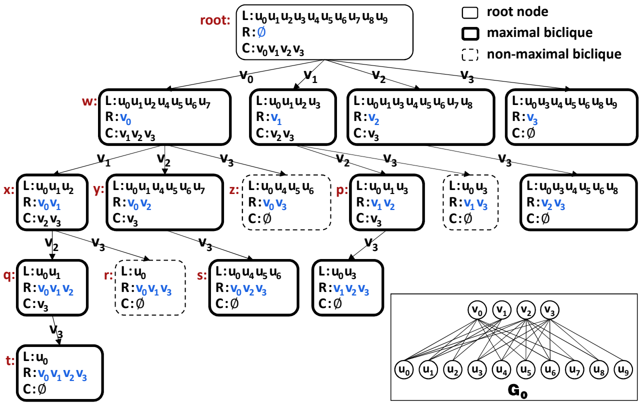
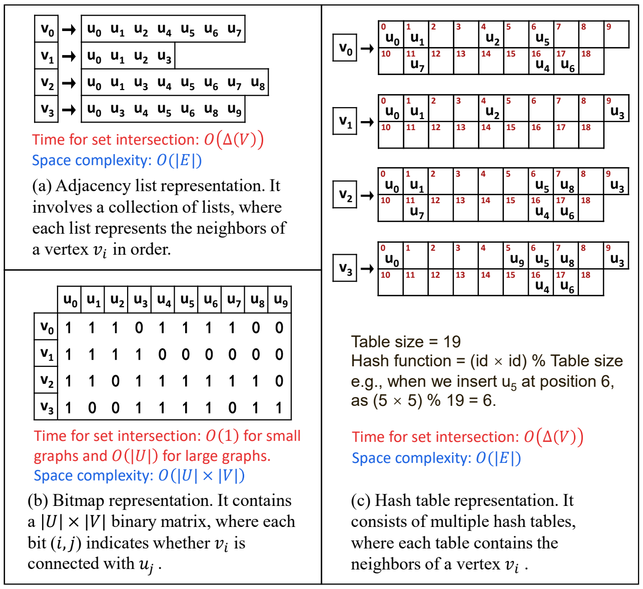

root node
maximal biclique
non-maximal biclique
root: L: $u_0u_1u_2u_3u_4u_5u_6u_7u_8u_9$ R: ∅ C: $v_0v_1v_2v_3$
$v_0$
$v_1$
$v_2$
$v_3$
w: L: $u_0u_1u_2u_4u_5u_6u_7$ R: $v_0$ C: $v_1v_2v_3$
L: $u_0u_1u_2u_3$ R: $v_1$ C: $v_2v_3$
L: $u_0u_1u_3u_4u_5u_6u_7u_8$ R: $v_2$ C: $v_3$
L: $u_0u_3u_4u_5u_6u_8u_9$ R: $v_3$ C: ∅
$v_1$
$v_2$
$v_3$
$v_2$
$v_3$
$v_3$
x: L: $u_0u_1u_2$ R: $v_0v_1$ C: $v_2v_3$
y: L: $u_0u_1u_4u_5u_6u_7$ R: $v_0v_2$ C: $v_3$
z: L: $u_0u_4u_5u_6$ R: $v_0v_3$ C: ∅
p: L: $u_0u_1u_3$ R: $v_1v_2$ C: $v_3$
L: $u_0u_3$ R: $v_1v_3$ C: ∅
L: $u_0u_3u_4u_5u_6u_8$ R: $v_2v_3$ C: ∅
$v_2$
$v_3$
$v_3$
$v_3$
q: L: $u_0u_1$ R: $v_0v_1v_2$ C: $v_3$
r: L: $u_0$ R: $v_0v_1v_3$ C: ∅
s: L: $u_0u_4u_5u_6$ R: $v_0v_2v_3$ C: ∅
L: $u_0u_3$ R: $v_1v_2v_3$ C: ∅
$v_3$
t: L: $u_0$ R: $v_0v_1v_2v_3$ C: ∅
$v_0$ $v_1$ $v_2$ $v_3$
$u_0$ $u_1$ $u_2$ $u_3$ $u_4$ $u_5$ $u_6$ $u_7$ $u_8$ $u_9$
$G_0$

Fig. 2. The enumeration tree on the bipartite graph $G_0$. Each node represents a unique biclique $(L, R)$ with distinct set $R$ (highlighted in blue) during backtracking.

procedure iterates through each vertex $v'$ in $C$ (lines #3, 13) to generate each new child node $(L', R', C')$ (lines #4-9). After node generation, the procedure checks if the corresponding biclique $(L', R')$ is maximal (line #10). If it is, the maximal biclique is reported (line #11), and exploration of new nodes continues from the node $(L', R', C')$ (line #12). The time complexity for computing each node is $O(|V|\Delta(V))$, because each node involves $O(|V|)$ set intersections and each set intersection takes $O(\Delta(V))$ time based on the adjacency lists.

**Example 1.** *Figure 2 illustrates an enumeration tree for a bipartite graph $G_0$ using Algorithm 1. We start from the root node and generate the enumeration tree through backtracking. Initially, we enter node $w$ by traversing $v_0 \in C_{root}$ (line #3)* [1]. *We know $L_w = L_{root} \cap N(v_0) = \{u_0, u_1, u_2, u_4, u_5, u_6, u_7\}$ (line #4). Subsequently, we know that $R_w = \{v_0\}$ and $C_w = \{v_1, v_2, v_3\}$ because $v_0$ connects with all vertices in $L_w$, while $v_1$, $v_2$, and $v_3$ connect with some of vertices in $L_w$ (lines #5-9). Node $w$ corresponds to a maximal biclique since $R_w = \{v_0\} = \Gamma(L_w)$ (line #10). Continuing this process, we then enter node $x$ by traversing $v_1 \in C_w$ and finally generate the complete enumeration tree. Node $z$ represents a non-maximal biclique because $L_z \cup R_z = \{u_0, u_4, u_5, u_6, v_0, v_3\} \subset L_s \cup R_s = \{u_0, u_4, u_5, u_6, v_0, v_2, v_3\}$.*

**Recent optimizations.** To improve the enumeration performance, various algorithmic optimizations have been employed, such as vertex ordering [6], [16], [18], node pruning [6], [16], [18], and parallelization [15], [21]. However, despite these efforts, the current state-of-the-art approaches still lack efficiency when applied to large-scale datasets, as they overlook the characteristics of dynamic computational subgraphs (See Section II-C). As a result, existing MBE algorithms are constrained and capable of exploring relatively small real-world datasets with millions of maximal bicliques.



$v_0$ → $u_0$ $u_1$ $u_2$ $u_4$ $u_5$ $u_6$ $u_7$
$v_1$ → $u_0$ $u_1$ $u_2$ $u_3$
$v_2$ → $u_0$ $u_1$ $u_3$ $u_4$ $u_5$ $u_6$ $u_7$ $u_8$
$v_3$ → $u_0$ $u_3$ $u_4$ $u_5$ $u_6$ $u_8$ $u_9$

Time for set intersection: $O(\Delta(V))$
Space complexity: $O(|E|)$

(a) Adjacency list representation. It involves a collection of lists, where each list represents the neighbors of a vertex $v_i$ in order.

| | $u_0$ | $u_1$ | $u_2$ | $u_3$ | $u_4$ | $u_5$ | $u_6$ | $u_7$ | $u_8$ | $u_9$ |
|---|---|---|---|---|---|---|---|---|---|---|
| $v_0$ | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| $v_1$ | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| $v_2$ | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| $v_3$ | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |

Time for set intersection: $O(1)$ for small graphs and $O(|U|)$ for large graphs.
Space complexity: $O(|U| \times |V|)$

(b) Bitmap representation. It contains a $|U| \times |V|$ binary matrix, where each bit $(i, j)$ indicates whether $v_i$ is connected with $u_j$ .

$v_0$ →

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 $u_0$ | 1 $u_1$ | 2 | 3 | 4 $u_2$ | 5 | 6 $u_5$ | 7 | 8 | 9 |
| 10 | 11 $u_7$ | 12 | 13 | 14 | 15 | 16 $u_4$ | 17 $u_6$ | 18 | |

$v_1$ →

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 $u_0$ | 1 $u_1$ | 2 | 3 | 4 $u_2$ | 5 | 6 | 7 | 8 | 9 $u_3$ |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | |

$v_2$ →

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 $u_0$ | 1 $u_1$ | 2 | 3 | 4 | 5 | 6 $u_5$ | 7 $u_8$ | 8 | 9 $u_3$ |
| 10 | 11 $u_7$ | 12 | 13 | 14 | 15 | 16 $u_4$ | 17 $u_6$ | 18 | |

$v_3$ →

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 $u_0$ | 1 | 2 | 3 | 4 | 5 $u_9$ | 6 $u_5$ | 7 $u_8$ | 8 | 9 $u_3$ |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 $u_4$ | 17 $u_6$ | 18 | |

Table size = 19
Hash function = (id × id) % Table size
e.g., when we insert $u_5$ at position 6, as (5 × 5) % 19 = 6.

Time for set intersection: $O(\Delta(V))$
Space complexity: $O(|E|)$

(c) Hash table representation. It consists of multiple hash tables, where each table contains the neighbors of a vertex $v_i$ .

Fig. 3. Three representations of the bipartite graph $G_0(U, V, E)$.

## B. Graph Representation in Memory

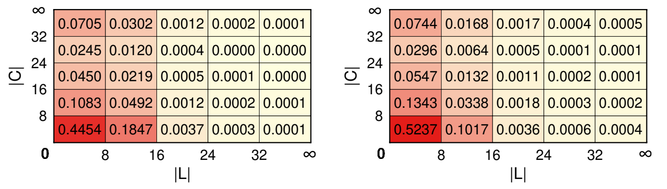
The representation of a bipartite graph plays a critical role in the performance of MBE solutions, as it directly affects the efficiency of extensive set intersections in MBE algorithms, e.g., lines #4,6,8,10 in Algorithm 1. Figure 3 depicts three common representations of bipartite graphs used in existing MBE algorithms: adjacency lists, bitmaps, and hash tables. Each data structure has its own advantages and trade-offs between memory usage and running time of set operations.

Adjacency lists. The adjacency list representation is widely employed in recent MBE algorithms designed for large graphs, such as PMBE [4], OOMBEA [18], and GMBE [21]. It excels in memory efficiency by only storing necessary connections in $O(|E|)$. However, each set intersection operation requires $O(\Delta(V))$ time complexity since it sequentially accesses the neighbors of a vertex, which are bounded by $O(\Delta(V))$.

Bitmaps. The bitmap representation is widely used in earlier research on small graphs, including LCM-MBC [23] and iMBEA [6]. It achieves the time complexity of $O(|U|)$ for each set intersection through bitwise AND operations between two $|U|$-bit bitmaps. Notably, for small graphs, it efficiently performs set intersections in $O(1)$, accomplished by several bitwise AND operations. However, it is not suitable for large sparse graphs due to its extensive memory usage (e.g., $O(|U| \times |V|)$), as it needs to store all missing edges as bit 0.

Hash tables. The hash table representation is only utilized in ParMBE [15]. It accelerates set intersections by employing efficient lookup operations, especially when the sizes of the input vertex sets vary significantly. Each set intersection operation using hash tables requires $O(\Delta(V))$ time complexity to access all neighbors of a vertex. Additionally, the memory

[1] For clarity, we use subscripts to denote the corresponding vertex sets of enumeration nodes. For instance, $C_{root}$ denotes the set $C$ of the root node.



(a) BookCrossing.

(b) Github.

Fig. 4. Distribution of CG sizes based on $|L|$ and $|C|$. Each cell denotes the frequency of occurrence for a specific combination of $|L|$ and $|C|$, normalized to the total number of occurrences.

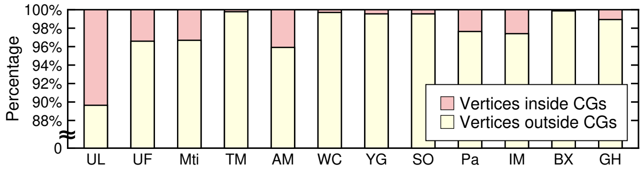


Fig. 5. Percentage of vertices inside and outside CGs on real-world datasets.

usage of hash tables can be bounded by $O(|E|)$ when employing cuckoo hashing. Although the hash table representation has similar time and space complexities to the adjacency list, it involves considerable random memory access and higher memory usage, resulting in suboptimal performance compared to the adjacency list representation.

### C. Motivation

In this section, we provide the formal definition of computational subgraphs in MBE, analyze its characteristics, and discuss limitations in existing works.

**Computational subgraphs (CG).** At the current enumeration node $(L, R, C)$, the computational subgraph in MBE refers to the subgraph formed by the vertices in $L \cup C$, along with all edges between $L$ and $C$ in the original bipartite graph.

By executing the baseline MBE approach (Algorithm 1) on the graph datasets listed in Table I of Section IV, we observe the following characteristics of computational subgraphs:

- *O1: The size of CGs dynamically changes.* It is due to variations in the sizes of $|L|$ and $|C|$ for each node. Notably, most of these CGs are relatively small. Figure 4 shows that 90% of CGs contain both $|L|$ and $|C|$ which are less than 32.
- *O2: The computational subgraph of the current enumeration node can be directly used for node generation.* Specifically, in Algorithm 1, the current node $(L, R, C)$ generates node $(L', R', C')$ by traversing $v'$. We know $L'$ is a subset of $L$ (line #4). To our observation, it is feasible to replace all neighborhood accesses on the original graph with local neighborhood accesses on the current CG. To elaborate, we derive $L \cap N(v')$ (line #4) as $L \cap N(v') = L \cap (L \cap N(v'))$ and derive $L' \cap N(v_c)$ (lines #6, 8) as $L' \cap N(v_c) = (L' \cap L) \cap N(v_c) = L' \cap (L \cap N(v_c))$, where $L \cap N(v')$ and $L \cap N(v_c)$ represent the local neighbors of vertices $v'$ and $v_c$ in the current CG.
- *O3: Existing algorithms require access to vertices outside their corresponding CGs.* Although CGs are adequate for node generation, excessive vertex access outside CGs significantly impairs performance. Figure 5 reveals that these unnecessary vertex accesses account for more than 90% of memory accesses across the majority of datasets.

**Limitations of existing works.** The existing MBE works ignore the characteristics of CGs, leading to two main drawbacks. First, these studies typically operate on the original graph, resulting in extensive access to vertices outside CGs. Additionally, exploiting memory accesses within CGs could aid in reducing repetitive set intersections and node pruning. Further elaboration will be provided in Section III-A. Second, current approaches commonly utilize the adjacency list as the default choice for representing graphs [16], [18], [21]. However, as discussed in Section II-B, set intersections on adjacency lists are less efficient than on bitmaps for smaller graphs. Fortunately, as the CG dynamically changes during enumeration, leveraging bitmaps can enhance set intersection performance on smaller CGs.

## III. AdaMBE

The design objectives of AdaMBE are (1) to use local neighborhood information derived from the computational subgraphs to accelerate the core operations of the MBE algorithm; and (2) to use bitmaps to represent computational subgraphs making a tradeoff between set intersection performance and memory efficiency.

### A. Redesign of Key Operations using Local Neighborhood Information

**Local neighbors.** Assume we have node $x$ in an enumeration tree. Vertex $v$ is a candidate vertex in $C_x$. Then, the local neighbors of vertex $v$ are the vertices that appear in both $N(v)$ and $L_x$, denoted as $N_x(v)$. $N_x(v) = N(v) \cap L_x$.

The local neighbors of vertex $v$ can be derived from the neighbors of its parent node in its corresponding computational subgraph without accessing the global neighbors of $v$ in the original bipartite graph. To improve the performance, we use local neighbors to redesign three key operations as follows:

**(1) Reducing vertex access for computing $R'$ and $C'$.** Existing MBE algorithms compute $R'$ and $C'$ by calculating $L' \cap N(v_c)$ for each candidate vertex $v_c$, as shown in lines #6 and #8 of Algorithm 1. $N(v_c)$ refers to the global neighbors of vertex $v_c$ in the original graph. In AdaMBE, instead of accessing $N(v_c)$, we only access local neighbors of $v_c$ in the computational subgraph of the parent node $x$, i.e., $N_x(v_c)$, according to characteristics O2 in Section II-C. Furthermore, because $|N_x(v_c)|$ can be far smaller than $|N(v_c)|$, the overhead of computing $L' \cap N_x(v_c)$ is much smaller than $L' \cap N(v_c)$. In order to provide fast retrieval of the intermediate data $N_x(v_c)$, we store $N_x(v_c)$ in the local neighbor cache to avoid redundant vertex accesses.

Local neighbors are necessary intermediate results in all existing MBE algorithms. Simply caching these local neighbors can enable us to optimize their usage. By utilizing the relationships between parent and child nodes, we replace global neighbors in the original graph with local neighbors in the CGs

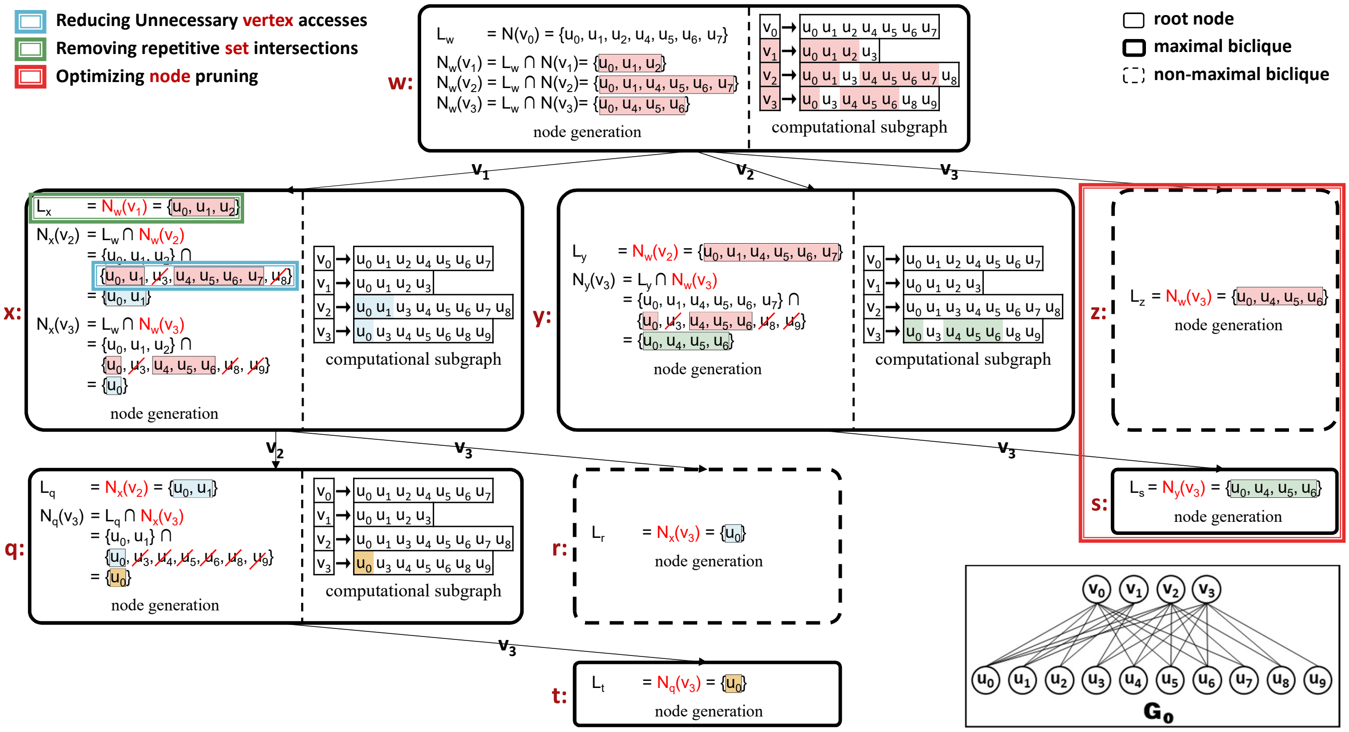

Reducing Unnecessary vertex accesses
Removing repetitive set intersections
Optimizing node pruning
root node
maximal biclique
non-maximal biclique
w:
L_w = N(v_0) = {u_0, u_1, u_2, u_4, u_5, u_6, u_7}
N_w(v_1) = L_w ∩ N(v_1)= {u_0, u_1, u_2}
N_w(v_2) = L_w ∩ N(v_2)= {u_0, u_1, u_4, u_5, u_6, u_7}
N_w(v_3) = L_w ∩ N(v_3)= {u_0, u_4, u_5, u_6}
node generation
computational subgraph
v_1
v_2
v_3
x:
L_x = N_w(v_1) = {u_0, u_1, u_2}
N_x(v_2) = L_w ∩ N_w(v_2)
= {u_0, u_1, u_2} ∩
{u_0, u_1, u_3, u_4, u_5, u_6, u_7, u_8}
= {u_0, u_1}
N_x(v_3) = L_w ∩ N_w(v_3)
= {u_0, u_1, u_2} ∩
{u_0, u_3, u_4, u_5, u_6, u_8, u_9}
= {u_0}
y:
L_y = N_w(v_2) = {u_0, u_1, u_4, u_5, u_6, u_7}
N_y(v_3) = L_y ∩ N_w(v_3)
= {u_0, u_1, u_4, u_5, u_6, u_7} ∩
{u_0, u_3, u_4, u_5, u_6, u_8, u_9}
= {u_0, u_4, u_5, u_6}
z:
L_z = N_w(v_3) = {u_0, u_4, u_5, u_6}
s:
L_s = N_y(v_3) = {u_0, u_4, u_5, u_6}
q:
L_q = N_x(v_2) = {u_0, u_1}
N_q(v_3) = L_q ∩ N_x(v_3)
= {u_0, u_1} ∩
{u_0, u_3, u_4, u_5, u_6, u_8, u_9}
= {u_0}
r:
L_r = N_x(v_3) = {u_0}
t:
L_t = N_q(v_3) = {u_0}
v_0 v_1 v_2 v_3
u_0 u_1 u_2 u_3 u_4 u_5 u_6 u_7 u_8 u_9
G_0

Fig. 6. Redesign of key operations using local neighborhood information. The local neighbors of vertices are those within the current computational subgraph (CG). We illustrate this with the subtree rooted at node $w$ in Figure 2. Local neighbors and CGs for different nodes are highlighted in different colors.

during node enumeration. Our approach significantly reduces memory accesses by 97.1% for vertices outside the CGs on average, leading to improved efficiency, as demonstrated in Section II-C. Considering that memory is usually not the bottleneck for computing-intensive MBE problems, consuming some additional memory for storing local neighbors to reduce running time is worthwhile.

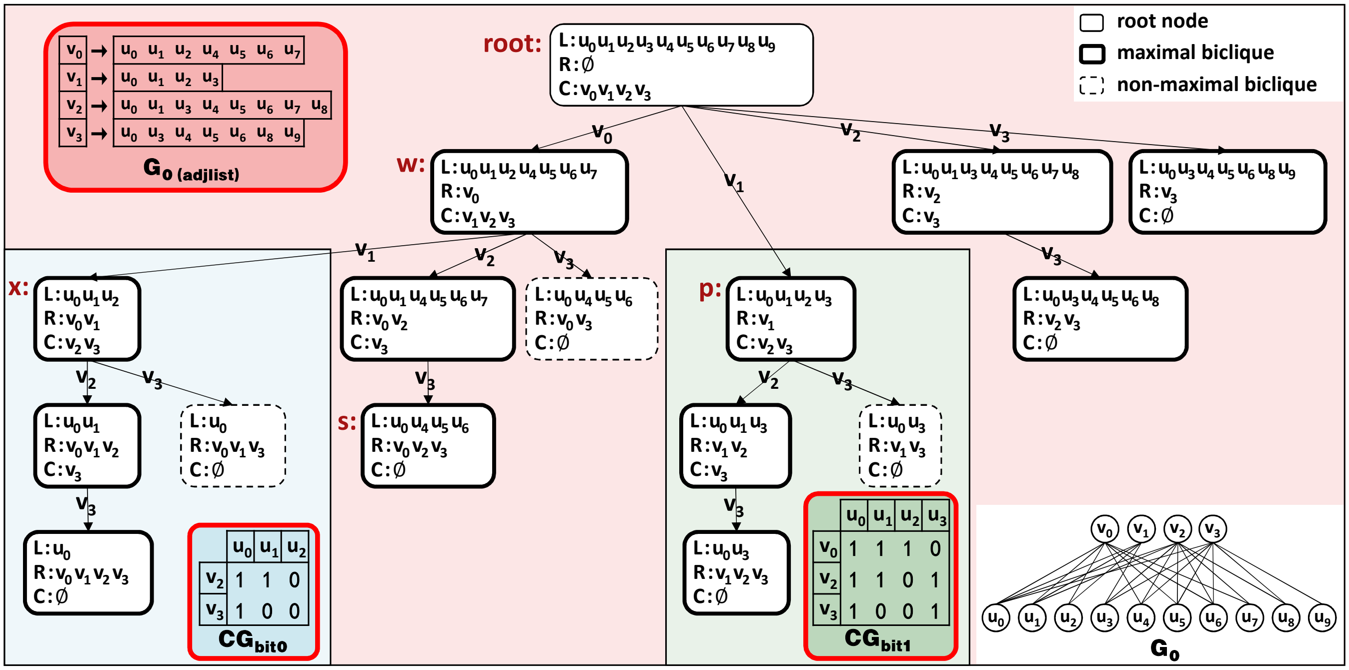
**(2) Removing repetitive set intersection for computing $L'$.** We have identified that there are repetitive set intersections for computing $L'$. Notably, the operation $L \cap N(v')$ in line #4 of Algorithm 1 precisely yields the local neighbors of $v'$ in the computational subgraph of the node $(L, R, C)$. Consequently, we can remove these repetitive set intersections by directly accessing local neighbors of $v'$ in the local neighbor cache. Despite being a common occurrence in existing MBE algorithms, this issue has yet to be addressed.

**(3) Pruning enumeration nodes before node generation.** According to our observation, we note that the local neighbors of each vertex consistently correspond to the set $L$ of an enumeration node, as illustrated in line #4 of Algorithm 1. Thus, we can prove that identical local neighbors will result in nodes with the same set $L$, leading to redundant nodes. More formally, when node $q$ is a child node of node $p$ and local neighborhood sizes of vertex $v$ are equal in both nodes, i.e., $|N_p(v)| = |N_q(v)|$, node $p$ can safely prune the node enumerated by traversing vertex $v$. This is because the equality in neighborhood sizes indicates that $N_p(v)$ and $N_q(v)$ are identical, given that $N_p(v)$ always contains all vertices in $N_q(v)$. Thus, we can achieve node pruning by ensuring that each unique local neighborhood generates a node only once.

**Example 2.** *Figure 6 demonstrates how AdaMBE works using local neighborhood information from node $w$ in Figure 2. First, AdaMBE reduces unnecessary vertex accesses outside the current CG by utilizing only local neighbors cached in CGs. For instance, node $x$ computes $N_x(v_2)$ using local neighbors $N_w(v_2)$ cached in the CG of its parent node $w$, effectively preventing access to unnecessary vertices like $u_3$ and $u_8$ outside the current CG. Next, AdaMBE removes repetitive set intersections by directly obtaining set $L$ for the current node in the CG of its parent node. For instance, node $x$ obtains $L_x$ by directly accessing local neighbors of $v_1$, i.e., $N_w(v_1)$, in the CG of its parent node $w$. Finally, AdaMBE optimizes node pruning by comparing local neighborhood sizes between parent and child nodes, leading to efficient pruning decisions. For instance, node $w$ prunes its child node $z$ enumerated by vertex $v_3$ because local neighborhood sizes of $v_3$ are equal in both nodes $w$ and $y$, i.e., $|N_w(v_3)| = |N_y(v_3)| = 4$.*

### *B. Hybrid In-Memory Representation of CGs*

The selection of the graph representations is a trade-off between computation and memory efficiency. We could utilize the adjacency list representation for large CGs leveraging its memory efficiency while employing the bitmap representation for small CGs leveraging its computational efficiency. Previous MBE works only use one data structure in enumeration. They



| | $u_0$ | $u_1$ | $u_2$ |
|---|---|---|---|
| $v_2$ | 1 | 1 | 0 |
| $v_3$ | 1 | 0 | 0 |

| | $u_0$ | $u_1$ | $u_2$ | $u_3$ |
|---|---|---|---|---|
| $v_0$ | 1 | 1 | 1 | 0 |
| $v_2$ | 1 | 1 | 0 | 1 |
| $v_3$ | 1 | 0 | 0 | 1 |

Fig. 7. Hybrid in-memory representation of CGs in MBE on the bipartite graph $G_0$. We represent the entire graph using adjacency lists for memory efficiency. When $|L|$ of the current node is less than or equal to 4 and $C$ is not empty, we create bitmaps dynamically. These bitmaps accelerate set intersections, enabling faster generation of child nodes.

cannot achieve the best performance because the size of CGs changes during enumeration as discussed in Section II.

In AdaMBE, our idea is to represent the original bipartite graph using adjacency lists and create CGs on the fly using bitmaps when the current CGs are small. Through efficient bitwise operations on these bitmaps, we can accelerate set intersections for MBE computations. Specifically, we create bitmap-based CGs dynamically as the current CG tends to shrink during enumeration. In Algorithm 1, each node $(L, R, C)$ only needs to access neighbors of vertices in $L$ (line #10) and $C$ (lines #4,6,8). As $L' \subset L$ and $C' \subset C$, we can infer that the CG induced by vertices in the child node (i.e., in $L'$ and $C'$) is always a subgraph of the CG induced by vertices in the parent node (i.e., in $L$ and $C$). Thus, we dynamically create bitmap-based subgraphs when the CG is smaller than our defined threshold and reuse them in the child nodes. In the design, we need to answer the following two key questions.

**(1) How do we create and utilize bitmap-based subgraphs?** Given a node $(L^*, R^*, C^*)$, we create a subgraph $CG_{bit}(U', V', E')$ using the bitmap from the original graph $G(U, V, E)$. To create $U'$, we include all vertices in $L^*$ because vertices in $U \setminus L^*$ cannot exist in $L'$ and do not contribute to intermediate results in Algorithm 1 (lines #4,6,8). To create $V'$, we include all vertices that connect with any vertex in $L^*$ excluding vertices in $R^*$, denoted as $\bigcup_{u \in L^*} N(u) - R^*$. This is because vertices that do not connect with any vertex in $L^*$ cannot exist in $C'$ or $\Gamma(L')$ and do not contribute to intermediate results in Algorithm 1. We exclude vertices in $R^*$ from $V'$ since they always appear in $R'$ and $\Gamma(L')$ (line #10) of all child nodes. This reduces redundant computations. $E'$ contains all edges between $U'$ and $V'$. When the size of $U'$ (i.e., $|L^*|$) is bounded by a small constant $\tau$, each set intersection operation in lines #4,6-9,13 can be computed in $O(\tau) = O(1)$ using bitwise AND operations. We compute $\Gamma(L')$ by checking if $L' \cap N(v)$ equals $L'$ for all vertices $v \in V'$. This node check takes $O(\tau\Delta(U)) = O(\Delta(U))$ because $V'$ contains at most $\tau\Delta(U)$ vertices, and each vertex takes a set intersection in $O(1)$. In summary, creating and utilizing bitmap-based subgraphs enables efficient computation of intermediate results in Algorithm 1, where each node takes $O(\Delta(U))$ time in total.

**(2) When do we create bitmap-based subgraphs?** To optimize set intersection operations and improve computational efficiency, we create bitmap-based subgraphs under two conditions for the node $(L^*, R^*, C^*)$: when $|L^*|$ is less than or equal to a threshold $\tau$, and when $C^*$ is not empty. These conditions are crucial because $|L^*|$ directly affects the time required for set intersections, while $C^*$ influences the reuse of the bitmap in child nodes. However, selecting an appropriate threshold $\tau$ is challenging. A larger $\tau$ increases set intersection time (i.e., $O(\tau)$) and memory usage for each subgraph. Conversely, a smaller $\tau$ leads to the creation of more small subgraphs, which limits opportunities for bitmap reuse in child nodes. Hence, we must strike a balance between the efficiency of set intersection computations and the utilization of bitmaps when determining $\tau$. We empirically set the threshold $\tau$ to 64 (See Section IV-D). This allows for efficient set intersection operations, as each operation only requires a single bitwise AND between two 64-bit long long integers.

**Example 3.** *Figure 7 illustrates the application of the hybrid in-memory representation of CGs in MBE on the bipartite graph $G_0$. In this example, we set $\tau$ to 4. Initially, the graph $G_0$ is represented using the adjacency list as the default method (referred to as $G_{0(adjlist)}$). When we enter node $x$, we observe that $|L_x| = 3 < \tau$, and thus create a bitmap-based subgraph $CG_{bit0}$. In $CG_{bit0}(U_0, V_0, E_0)$, $U_0$ contains all vertices in $L_x$, namely $u_0, u_1$, and $u_2$. $V_0$ contains $v_2$ and $v_3$ since they have connections with some vertices in $L_x$. However, $v_0$ and $v_1$ are not part of $V_0$ because they belong to $R_x$. Consequently, all child nodes of $x$ can expedite set intersections by utilizing the bitmap in $CG_{bit0}$. Similarly, at node $p$, we create another bitmap-based subgraph $CG_{bit1}$, benefiting all child nodes of node $p$. Although $|L_s| = 4 = \tau$, we avoid creating the subgraph at node $s$ because $C_s$ is empty, which means no child node can reuse the bitmap.*

### C. Adaptive MBE Algorithm

**Proposed AdaMBE.** To explore an efficient MBE algorithm, we propose AdaMBE as demonstrated in Algorithm 2. For clarity, we highlight the utilization of local neighbor information in blue and the bitwise AND (&) operations in red. Given a bipartite graph $G(U, V, E)$, AdaMBE initially sorts vertices in $V$ based on their degrees in ascending order (line #1) due to its high efficiency as shown in Section IV-D. Next, AdaMBE recursively calls the `AdaMBE_search` procedure. For large CGs where $|L_p| > \tau$, AdaMBE accelerates enumeration using local neighborhood information (lines #8-23). This involves a redesign of key operations in lines #9,12-16,18, leading to reduced unnecessary vertex access, repetitive set operations, and redundant nodes. Conversely, for small CGs with $|L_p| \le \tau$ (lines #5-6, 24-40), AdaMBE expedites set operations using bitwise AND (&) operations (lines #26,29,34,36). We set $\tau$ to 64 by default based on experimental results in Section IV-D.

**Parallel version of AdaMBE.** To enhance performance, we devise a parallel version of AdaMBE called ParAdaMBE. It harnesses the power of the Intel TBB library [26] to compute multiple nodes concurrently in the enumeration tree. By employing loop unrolling techniques, ParAdaMBE achieves efficient parallelization of the for loops, maximizing computational performance.

Unlike existing MBE techniques that overlook the characteristics of CGs, AdaMBE utilizes them to accelerate vertex accesses and set operations. Additionally, by adapting to the sizes of CGs, AdaMBE substantially improves performance with high memory efficiency. Despite operating on CPUs with limited parallel computational resources compared to GPUs, AdaMBE remains competitive on datasets containing billions of maximal bicliques, due to its efficient memory usage and adaptive strategies.

## IV. Evaluation

### A. Experimental Setup

**Platform.** We run all experiments on a machine equipped with four Intel Xeon(R) Gold 5318Y 2.10GHz CPUs (24 cores per CPU), 128GB of main memory, and an NVIDIA A100 GPU [27] with 40GB of global memory. The CPUs have a combined system bandwidth of 204.8 GB/s, achieved through 8 memory channels operating at 25.6 GB/s each [28]. In contrast, the A100 GPU delivers a significantly higher system bandwidth of 1,555 GB/s [29]. The operating system is Linux kernel-5.4.0.

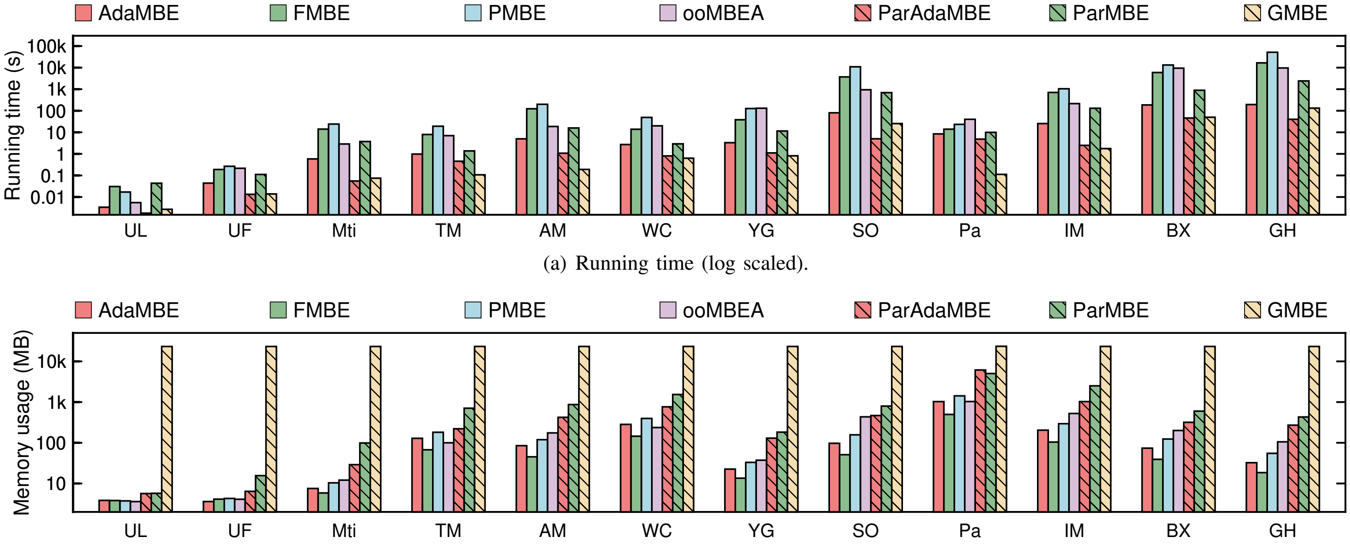
**Algorithm 2:** AdaMBE Algorithm

```
   Input: Bipartite graph $G(U, V, E)$
   Output: All maximal bicliques
1  Sort vertices in $V$ based on their degrees in ascending order;
2  AdaMBE_search($U, \emptyset, V, G$);
3  procedure AdaMBE_search($L_p, R_p, C_p, CG_p$):
4      if $|L_p| \le \tau$ and $C_p$ is not empty then
5          Create bitmap $CG_{bit}(U_{bit}, V_{bit}, E_{bit})$;
6          AdaMBE_search_bit($L_p, R_p, C_p, CG_{bit}$);
7          Return;
8      foreach $v' \in C_p$ do
9          $L_q \leftarrow N_p(v')$;
10         $R_q \leftarrow R_p$; $C_q \leftarrow \emptyset$; $CG_q \leftarrow \emptyset$;
11         foreach $v_c \in C_p$ do
12             $N_q(v_c) \leftarrow L_q \cap N_p(v_c)$;
13             $CG_q$.insert($N_q(v_c)$);
14             if $N_p(v_c)$ = $N_q(v_c)$ then    // Node prune
15                 $CG_p$.delete($N_p(v_c)$);
16             if $N_q(v_c)$ = $L_q$ then
17                 $R_q \leftarrow R_q \cup \{v_c\}$;
18             else if $N_q(v_c) \neq \emptyset$ then
19                 $C_q \leftarrow C_q \cup \{v_c\}$;
20         if $R_q = \Gamma(L_q)$ then
21             Output($L_q, R_q$) as a maximal biclique;
22             AdaMBE_search($L_q, R_q, C_q, CG_q$)
23         free($CG_q$); $C_p \leftarrow C_p \setminus \{v'\}$;
24 procedure AdaMBE_search_bit($L_p, R_p, C_p, CG_{bit}$):
25     foreach $v' \in C_p$ do
26         $L_q \leftarrow L_p$ & $N_{bit}(v')$;
27         $is\_maximal \leftarrow$ True;
28         foreach $v'' \in V_{bit} \setminus (R_p \cup C_p)$ do
29             if $L_q = L_p$ & $N_{bit}(v'')$ then // Node check
30                 $is\_maximal \leftarrow$ False;
31         if $is\_maximal$ then    // Node generation
32             $R_q \leftarrow R_p$; $C_q \leftarrow \emptyset$;
33             foreach $v_c \in C_p$ do
34                 if $L_q$ & $N_{bit}(v_c) = L_q$ then
35                     $R_q \leftarrow R_q \cup \{v_c\}$;
36                 else if $L_q$ & $N_{bit}(v_c) \neq 0$ then
37                     $C_q \leftarrow C_q \cup \{v_c\}$;
38             Output($L_q, R_q$) as a maximal biclique;
39             AdaMBE_search_bit($L_q, R_q, C_q, CG_{bit}$)
40         $C_p \leftarrow C_p \setminus \{v'\}$;
```

**Baselines.** We evaluate both serial and parallel MBE algorithms. For serial algorithms, we compare AdaMBE against three state-of-the-art algorithms: FMBE [15], PMBE [16], and ooMBEA [18]. We run these algorithms on a single

TABLE I
REAL-WORLD DATASET STATISTICS.

| General Datasets | Category | Type | $\|U(G)\|$ | $\|V(G)\|$ | $\|E(G)\|$ | Maximal Bicliques |
|---|---|---|---|---|---|---|
| Unicode (UL) | Feature | Country-Hosts-Language | 614 | 254 | 1,255 | 460 |
| UCforum (UF) | Intersection | User-Post-Forum | 899 | 522 | 7,089 | 16,261 |
| MovieLens (Mti) | Feature | Tag-Assignment-Movie | 16,528 | 7,601 | 71,154 | 140,266 |
| Teams (TM) | Affiliation | Athlete-Membership-Team | 901,130 | 34,461 | 1,366,466 | 517,943 |
| ActorMovies (AM) | Affiliation | Movie-Appearance-Actor | 383,640 | 127,823 | 1,470,404 | 1,075,444 |
| Wikipedia (WC) | Feature | Article-Inclusion-Category | 1,853,493 | 182,947 | 3,795,796 | 1,677,522 |
| YouTube (YG) | Affiliation | User-Membership-Group | 94,238 | 30,087 | 293,360 | 1,826,587 |
| StackOverflow (SO) | Rating | User-Favorite-Post | 545,195 | 96,680 | 1,301,942 | 3,320,824 |
| DBLP (Pa) | Authorship | Author-Authorship-Publication | 5,624,219 | 1,953,085 | 12,282,059 | 4,899,032 |
| IMDB (IM) | Affiliation | Movie-Appearance-Actor | 896,302 | 303,617 | 3,782,463 | 5,160,061 |
| BookCrossing (BX) | Interaction | User-Rating-Book | 340,523 | 105,278 | 1,149,739 | 54,458,953 |
| Github (GH) | Authorship | User-Membership-Project | 120,867 | 56,519 | 440,237 | 55,346,398 |
| **Large Datasets** | **Category** | **Type** | $\|U(G)\|$ | $\|V(G)\|$ | $\|E(G)\|$ | **Maximal Bicliques** |
| CebWiki (ceb) | Authorship | User-Edit-Article | 8,483,068 | 3,132 | 11,792,890 | 263,138,916 |
| TVTropes (DBT) | Feature | Work-HasFeature-Trope | 87,678 | 64,415 | 3,232,134 | 19,636,996,096 |



(a) Running time (log scaled).

(b) Memory usage (log scaled).

Fig. 8. Overall evaluation on twelve general datasets. Parallel MBE algorithms are represented with diagonal lines.

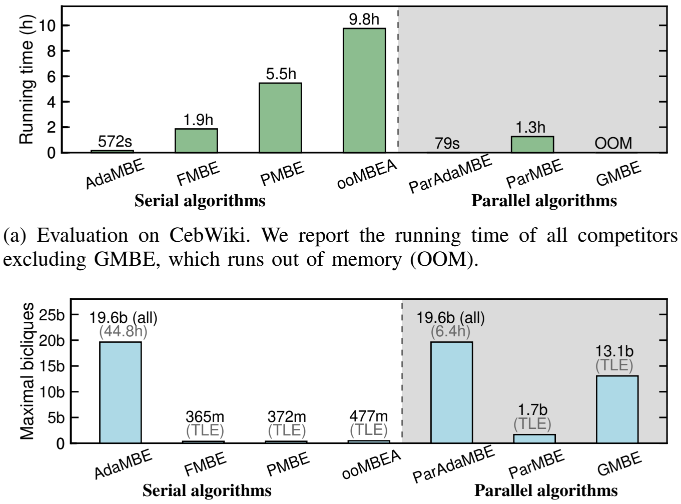
core. For parallel algorithms, we compare ParAdaMBE with two advanced algorithms: ParMBE [15] and GMBE [21]. We run ParAdaMBE and ParMBE on a 96-core CPU machine using 96 threads, and GMBE on an A100 GPU. We obtain all competitors in the repository [30] and run them with the default configurations in their papers. We default $\tau$ to 64 in our AdaMBE and ParAdaMBE.

**Datasets.** We utilize both real-world and synthetic datasets. Real-world datasets are sourced from the KONECT repository [25], encompassing diverse domains. Table I summarizes the statistics of these datasets, comprising twelve commonly used *general* datasets in recent research [15], [16], [18], [21], along with two *large* datasets containing over 200 million maximal bicliques. We arrange all datasets in ascending order based on their maximal biclique count and designate the vertex set with fewer vertices as $V$, as done in [21]. Synthetic datasets are generated by sampling edges from the extensive LiveJournal dataset ($|U|$=7,489,073, $|V|$=3,201,203, $|E|$=112,307,385), which is also obtained from the KONECT repository.

**Metrics.** We measure the running time and memory usage for all competitors on the datasets. The running time excludes the graph loading from the disk and the memory usage represents the maximum memory occupation of the corresponding process. Given that certain algorithms may require considerable time for execution, we have set a 48-hour time limit to prevent Time Limit Exceeded (TLE) errors, as done in [18].

## B. Overall Evaluation

**General datasets.** Figure 8(a) shows that our serial algorithm, AdaMBE, consistently outperforms the next-best serial competitor by 1.6×−49.7× on average, and our parallel algorithm, ParAdaMBE, is 1.3×−33.7× faster than other parallel competitors. Particularly, on the most time-consuming dataset, Github, AdaMBE completes the task in 194 seconds,



(a) Evaluation on CebWiki. We report the running time of all competitors excluding GMBE, which runs out of memory (OOM).

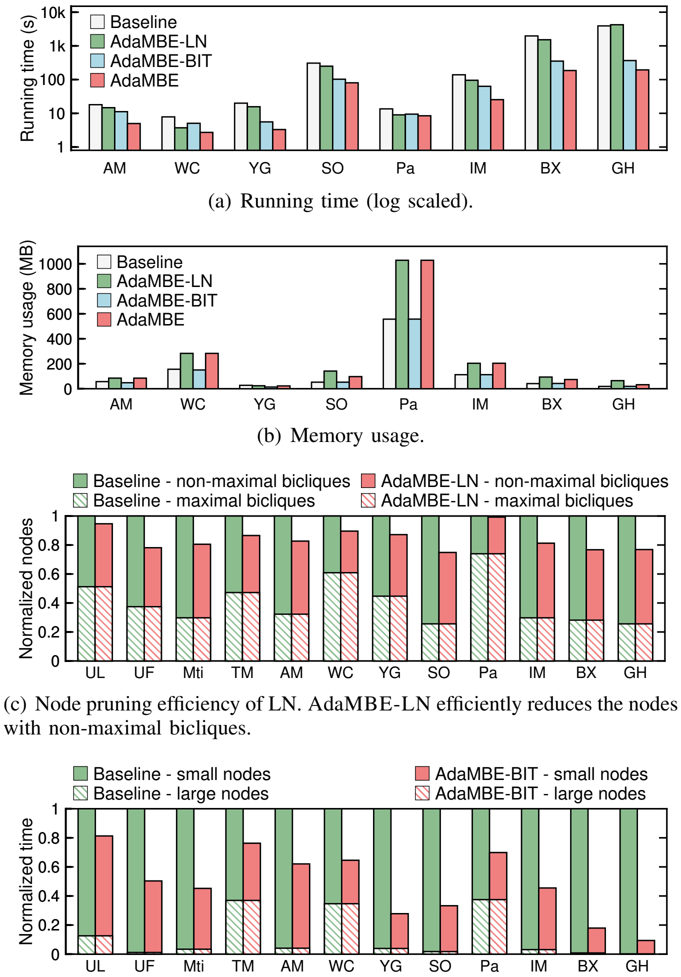
(b) Evaluation on TVTropes. We report the maximal biclique count after 48 hours. The acronym "TLE" stands for "Time Limit Exceeded."

Fig. 9. Overall evaluation on two large datasets.

while the other serial competitors require over 9,638 seconds. Similarly, ParAdaMBE finishes in 40 seconds, while the other parallel competitors take over 132 seconds. By effectively utilizing the characteristics of CGs, our 96-thread CPU algorithm ParAdaMBE is even up to 5.07× faster than the cutting-edge GPU algorithm GMBE on an A100 GPU on StackOverflow, BookCrossing, and Github. ParAdaMBE underperforms GMBE on TM, AM, and Pa because they contain a large number of small workloads, which are well-suited for extensive parallel processing on GPUs.

Figure 8(b) shows that AdaMBE reduces memory usage by 28.0% and 29.8% on average, respectively, compared to serial algorithms such as PMBE and OOMBEA. In comparison to parallel algorithms like ParMBE and GMBE, ParAdaMBE diminishes memory consumption by 29.7% and 96.4% on average, respectively. GMBE requires the largest memory due to its pre-allocation of memory for thousands of threads on the GPU [21]. Although FMBE performs slightly better than AdaMBE in terms of memory usage, its running time significantly lags behind AdaMBE. Consequently, we can conclude our approaches achieve notable performance improvements while maintaining high memory efficiency.

**Large datasets.** We further evaluate our algorithms on large datasets with over 200 million maximal bicliques. Figure 9(a) shows that all other competitors take several hours to finish MBE on the CebWiki dataset, while our AdaMBE and ParAdaMBE require 572 seconds and 79 seconds, respectively. As no existing competitor completes enumeration within TLE on the TVTropes dataset with 19.6 billion maximal bicliques, we report the maximal biclique counts for all algorithms after 48 hours. Figure 9(b) shows that the GPU algorithm GMBE enumerates 66.5% of the total maximal bicliques, while other competitors enumerate less than 8.5% within 48 hours. Notably, AdaMBE and ParAdaMBE enumerate all maximal bicliques in 44.8 hours and 6.4 hours, respectively, significantly outperforming existing competitors.



(a) Running time (log scaled).

(b) Memory usage.

(c) Node pruning efficiency of LN. AdaMBE-LN efficiently reduces the nodes with non-maximal bicliques.

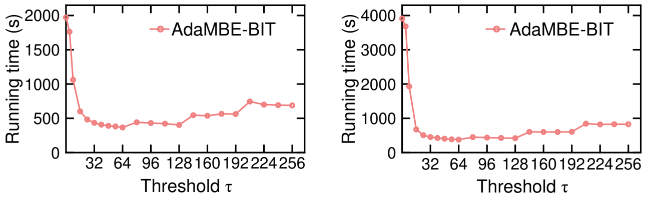
(d) Time breakdown of BIT. Small nodes represent the nodes with $|L|$ smaller than $\tau$. AdaMBE-BIT reduces the running time of these nodes.

Fig. 10. Breakdown Analysis.

## C. Breakdown Analysis

For simplicity, we denote the approach discussed in Section III-A as LN (indicating the utilization of Local Neighborhood information), and the approach in Section III-B as BIT (representing the use of BITmap on small subgraphs). We evaluate their effectiveness using three AdaMBE variants: Baseline (disabling both LN and BIT in AdaMBE), AdaMBE-LN (enabling LN only), and AdaMBE-BIT (enabling BIT only).

**Running time and memory usage.** Figure 10(a) shows both LN and BIT consistently accelerate Baseline on eight larger datasets. AdaMBE outperforms other variants by combining both approaches effectively. Notably, on Github, AdaMBE reduces the running time of Baseline from 3,911 seconds to 194 seconds. Figure 10(b) illustrates that LN increases memory usage due to storing local neighbors at runtime. However, since MBE is a computationally intensive task, utilizing limited memory to expedite computation is a reasonable trade-off. For



(a) BookCrossing.

(b) Github.

Fig. 11. Impact of threshold ($\tau$) in BIT.

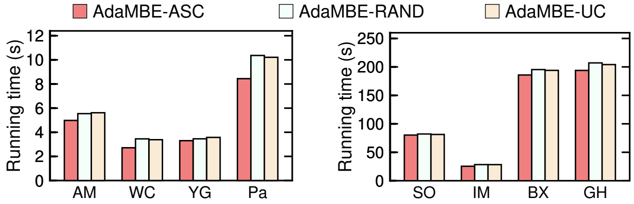


Fig. 12. Impact of vertex ordering.

all datasets, AdaMBE requires less than 1.3 GB of memory (on CebWiki), which is acceptable for a contemporary server.

**Effect of the LN approach.** LN enhances three key operations using local neighborhood information. We assess the effectiveness of these three aspects respectively. First, LN eliminates all vertex accesses outside computational subgraphs, reducing over 90% of vertex accesses across most datasets, as demonstrated in Figure 5 of Section II-C. Second, LN eliminates all repetitive set intersections for computing set $L$, as detailed in Section III-A. Third, LN exhibits node pruning efficiency in Figure 10(c): it reduces 25% of nodes with non-maximal bicliques on average across twelve general datasets.

**Effect of the BIT approach.** BIT targets accelerating computation for small nodes $(L, R, C)$ whose $|L|$ is equal to or less than the threshold $\tau$. Figure 10(d) shows the running time breakdown of Baseline and AdaMBE-BIT on twelve general datasets. While the computation time for large nodes remains the same in Baseline and AdaMBE-BIT, BIT reduces computation time for small nodes by over 50% across most datasets. Note that BIT achieves a remarkable 91% reduction on Github, resulting in a 10.7× acceleration of AdaMBE-BIT compared to Baseline.

### D. Sensitivity Analysis

**Impact of threshold ($\tau$) in the BIT approach.** Figure 11 shows the performance of AdaMBE-BIT with various values of $\tau$ on two datasets; other datasets exhibit similar trends. As $\tau$ increases from 4 to 64, the running time consistently decreases due to fewer created subgraphs and increased opportunities for reusing bitmaps. Notably, when $\tau$ is not greater than 64, the set intersection time remains unchanged, as each set intersection only requires a single bitwise AND operation between two 64-bit long long integers. However, when $\tau$ exceeds 64, the running time increases due to the additional time required for each set intersection. Furthermore, the running time is shorter when $\tau$ is a multiple of 64, as these cases enhance bitmap reuse without increasing the time for each set intersection. Consequently, AdaMBE defaults $\tau$ to 64.

TABLE II
SYNTHETIC DATASET STATISTICS. "LJX" REPRESENTS X% OF LIVEJOURNAL'S EDGES ARE USED.

| Datasets | $\lvert U(G)\rvert$ | $\lvert V(G)\rvert$ | $\lvert E(G)\rvert$ | Max. Bicliques |
|---|---|---|---|---|
| LJ10 | 2,301,031 | 1,421,088 | 11,227,130 | 7,430,705 |
| LJ20 | 2,704,651 | 2,357,485 | 22,456,757 | 61,836,924 |
| LJ30 | 3,163,966 | 2,889,804 | 33,686,334 | 343,257,225 |
| LJ40 | 3,894,262 | 2,992,774 | 44,917,368 | 1,524,229,722 |
| LJ50 | 4,572,628 | 3,057,410 | 56,150,150 | 6,387,845,280 |

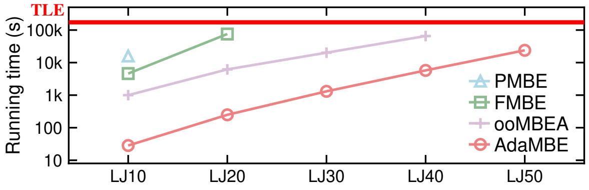
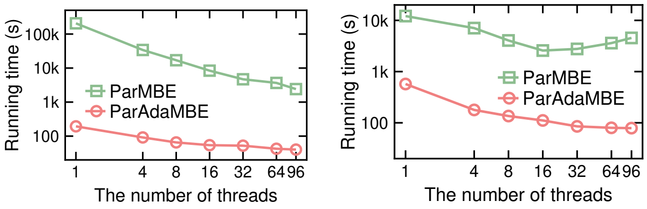


Fig. 13. Impact of dataset size (log scaled).



(a) Github

(b) CebWiki

Fig. 14. Impact of number of threads (log scaled).

**Impact of vertex ordering.** Figure 12 shows the performance of AdaMBE with three vertex ordering schemes. AdaMBE-ASC sorts vertices in ascending order based on their degrees, AdaMBE-RAND sorts vertices randomly, and AdaMBE-UC employs the unilateral order proposed by recent work OOMBEA [18], which introduces additional overhead due to the computation of their proposed "k unilateral core." Across eight large general datasets, we consistently observe that AdaMBE-ASC outperforms both AdaMBE-RAND and AdaMBE-UC. Consequently, we choose AdaMBE-ASC as the default vertex ordering.

**Impact of dataset size.** We assess the efficiency of AdaMBE using five synthetic datasets listed in Table II. Figure 13 shows AdaMBE outperforms all serial competitors by over 11.4×. Note that the absence of plotted points in the figure indicates the running time exceeds the time limit (TLE). Only AdaMBE completes MBE on LJ50, with over 6 billion maximal bicliques, in just 6.6 hours, while other competitors fail to complete it within 48 hours. These results indicate that AdaMBE is the optimal choice for large datasets.

**Impact of number of threads.** Figure 14 shows the performance of ParAdaMBE and ParMBE with 1 to 96 threads. Each thread is assigned to a separate CPU core. Our machine is equipped with 8 independent memory channels, providing a total system bandwidth of 204.8GB/s. ParAdaMBE always outperforms ParMBE on both Github and CebWiki. Additionally, the running time decreases sub-linearly as the number of threads increases, indicating its good performance across various thread configurations.

## V. Related work

**Maximal biclique enumeration.** Many MBE algorithms accelerate the enumeration process by optimizing search order [6], [16], [18], pruning unproductive nodes [6], [16], [18], and utilizing parallelization [15], [21]. Recent researchers focus on variants of the MBE problem, such as $(p, q)$ biclique enumeration [11], [12], [31], maximal similar biclique enumeration [32], fair biclique enumeration [33], maximal $\tau$-biclique enumeration on uncertain graphs [34], and maximum biclique finding [2], [3], [16], [35]–[37]. However, all these algorithms neglect the local neighborhood information of computational subgraphs. In contrast, we utilize this information to accelerate key operations in enumeration and expedite set intersections on small subgraphs using bitmaps. Moreover, our AdaMBE can be applied to various biclique finding problems, including finding maximum edge biclique [3], maximum balanced biclique [36], and personalized maximum biclique [4].

**Graph representation and redundancy optimization.** Previous studies employ various methods for graph representation, including bitmaps [6], [23], adjacency lists [38]–[42], and hash tables [15]. However, these approaches often struggle to achieve both computation and memory efficiency simultaneously. In contrast, our approach employs hybrid representations, utilizing adjacency lists for large graphs to save memory and bitmaps for smaller graphs to speed up computation. Our hybrid representation can be easily used for various subgraph enumeration problems like maximal clique enumeration [43], maximal quasi-clique enumeration [44], and (p,q)-clique enumeration [12]. This is because, like the MBE problem, their computational subgraphs shrink during enumeration. While recent graph mining algorithms focus on reducing redundancies either at the set [38], [45] or enumeration node [18] levels, we leverage the local neighborhood information to address redundancies at the vertex, set operation, and enumeration node levels at the same time.

## VI. Conclusion

Maximal biclique enumeration (MBE) has posed a significant challenge in the analysis of large-scale bipartite graphs. In this paper, we propose AdaMBE, a novel adaptive MBE approach for enumerating billions of maximal bicliques on CPUs. AdaMBE utilizes local neighborhood information in computational subgraphs to reduce unnecessary vertex accesses, repetitive set intersections, and useless node enumeration. Furthermore, it applies an adaptive hybrid in-memory representation (i.e., adjacency lists and bitmaps) for computational subgraphs to achieve both computation and memory efficiency. Our results show that AdaMBE is 1.6×−49.7× faster than its closest competitor and is capable of handling large datasets with billions of maximal bicliques. Our 96-thread CPU algorithm ParAdaMBE is up to 5.07× faster than the cutting-edge GPU algorithm GMBE on an A100 GPU for three time-consuming datasets out of twelve general datasets.

Note that although AdaMBE is designed to run on CPUs, it can also be extended to a GPU implementation. We leave the exploration of AdaMBE on GPUs for future work.

## Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments and feedbacks. This work was supported in part by the National Key Research and Development Program of China under Grant 2023YFB4502100, the Major Projects of Zhejiang Province under Grant LD24F020012, the National Science Foundation of China under Grant 62172361, the National Key Research and Development Program of China under Grant 2021ZD0110700, and the US National Science Foundation under CNS 2216108.

## 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, pp. 196–207.

[2] B. Lyu, L. Qin, X. Lin, Y. Zhang, Z. Qian, and J. Zhou, “Maximum biclique search at billion scale,” *Proceedings of the VLDB Endowment (PVLDB)*, vol. 13, no. 9, pp. 1359–1372, 2020.

[3] ——, “Maximum and top-k diversified biclique search at scale,” *The VLDB Journal*, vol. 31, no. 6, pp. 1365–1389, 2022.

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

[5] Z. Li, P. Hui, P. Zhang, J. Huang, B. Wang, L. Tian, J. Zhang, J. Gao, and X. Tang, “What happens behind the scene? towards fraud community detection in e-commerce from online to offline,” in *Companion Proceedings of the Web Conference*, 2021, pp. 105–113.

[6] Y. Zhang, C. A. Phillips, G. L. Rogers, E. J. Baker, E. J. Chesler, and M. A. Langston, “On finding bicliques in bipartite graphs: A novel algorithm and its application to the integration of diverse biological data types,” *BMC bioinformatics*, vol. 15, no. 1, p. 110, 2014.

[7] T. Siswantining, A. Bustamam, O. Swasti, and H. S. Al-Ash, “Analysis and prediction of protein interactions between hiv-1 protein and human protein using lcm-mbc algorithm combined with association rule mining,” *Communications in Mathematical Biology and Neuroscience*, vol. 2021, p. 64, 2021.

[8] Y. Liu, “Computational methods for identifying microrna-gene regulatory modules,” in *Handbook of Statistical Bioinformatics*, 2022, pp. 187–208.

[9] T. Alzahrani and K. Horadam, “Finding maximal bicliques in bipartite networks using node similarity,” *Applied Network Science (ANS)*, vol. 4, pp. 1–25, 2019.

[10] Z. Chen, Y. Zhao, L. Yuan, X. Lin, and K. Wang, “Index-based biclique percolation communities search on bipartite graphs,” in *IEEE 39th International Conference on Data Engineering (ICDE)*, 2023, pp. 2699–2712.

[11] J. Yang, Y. Peng, and W. Zhang, “(p, q)-biclique counting and enumeration for large sparse bipartite graphs,” *Proceedings of the VLDB Endowment (PVLDB)*, vol. 15, no. 2, pp. 141–153, 2021.

[12] J. Yang, Y. Peng, D. Ouyang, W. Zhang, X. Lin, and X. Zhao, “(p, q)-biclique counting and enumeration for large sparse bipartite graphs,” *The VLDB Journal*, pp. 1–25, 2023.

[13] G. Alexe, S. Alexe, Y. Crama, S. Foldes, P. L. Hammer, and B. Simeone, “Consensus algorithms for the generation of all maximal bicliques,” *Discrete Applied Mathematics*, vol. 145, no. 1, pp. 11–21, 2004.

[14] Y. He, R. Li, and R. Mao, “An optimized mbe algorithm on sparse bipartite graphs,” in *International Conference on Smart Computing and Communication (SmartCom)*, 2018, pp. 206–216.

[15] A. Das and S. Tirthapura, "Shared-memory parallel maximal biclique enumeration," in *IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2019, pp. 34–43.

[16] A. Abidi, R. Zhou, L. Chen, and C. Liu, "Pivot-based maximal biclique enumeration," in *Proceedings of the 29th International Conference on International Joint Conferences on Artificial Intelligence (IJCAI)*, 2020, pp. 3558–3564.

[17] C. Qin, M. Liao, Y. Liang, and C. Zheng, "Efficient algorithm for maximal biclique enumeration on bipartite graphs," in *Advances in Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*, vol. 1075, 2020, pp. 3–13.

[18] L. Chen, C. Liu, R. Zhou, J. Xu, and J. Li, "Efficient maximal biclique enumeration for large sparse bipartite graphs," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 15, no. 8, pp. 1559–1571, 2022.

[19] 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.

[20] R. Sun, Y. Wu, C. Chen, X. Wang, W. Zhang, and X. Lin, "Maximal balanced signed biclique enumeration in signed bipartite graphs," in *IEEE 38th International Conference on Data Engineering (ICDE)*, 2022, pp. 1887–1899.

[21] Z. Pan, S. He, X. Li, X. Zhang, R. Wang, and G. Chen, "Efficient maximal biclique enumeration on gpus," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2023, pp. 1–13.

[22] G. Liu, K. Sim, and J. Li, "Efficient mining of large maximal bicliques," in *International Conference on Data Warehousing and Knowledge Discovery (DaWaK)*, 2006, pp. 437–448.

[23] 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 Transactions on Knowledge and Data Engineering (TKDE)*, vol. 19, no. 12, pp. 1625–1637, 2007.

[24] A. P. Mukherjee and S. Tirthapura, "Enumerating maximal bicliques from a large graph using mapreduce," *IEEE Transactions on Services Computing*, vol. 10, no. 5, pp. 771–784, 2016.

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

[26] Intel, "oneapi threading building blocks," https://github.com/oneapi-src/oneTBB, 2024.

[27] NVIDIA, "Nvidia a100 tensor core gpu," https://www.nvidia.com/en-gb/data-center/a100/, 2024.

[28] Samsung, "M393a2k43db3-cwe," https://semiconductor.samsung.com/dram/module/rdimm/m393a2k43db3-cwe/, 2024.

[29] NVIDIA, "Nvidia a100 tensor core gpu specification," https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf, 2024.

[30] Z. Pan, https://github.com/ISCS-ZJU/GMBE, 2023.

[31] X. Ye, R.-H. Li, Q. Dai, H. Qin, and G. Wang, "Efficient biclique counting in large bipartite graphs," *Proceedings of the ACM on Management of Data*, vol. 1, no. 1, pp. 1–26, 2023.

[32] K. Yao, L. Chang, and J. X. Yu, "Identifying similar-bicliques in bipartite graphs," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 15, no. 11, pp. 3085–3097, 2022.

[33] Z. Yin, Q. Zhang, W. Zhang, R.-H. Li, and G. Wang, "Fairness-aware maximal biclique enumeration on bipartite graphs," in *IEEE 39th International Conference on Data Engineering (ICDE)*, 2023, pp. 1665–1677.

[34] J. Wang, J. Yang, Z. Ma, C. Zhang, S. Yang, and W. Zhang, "Efficient maximal biclique enumeration on large uncertain bipartite graphs," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2023.

[35] J. Wang, J. Yang, C. Zhang, and X. Lin, "Efficient maximum edge-weighted biclique search on large bipartite graphs," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2022.

[36] L. Chen, C. Liu, R. Zhou, J. Xu, and J. Li, "Efficient exact algorithms for maximum balanced biclique search in bipartite graphs," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2021, pp. 248–260.

[37] Y. Zhao, Z. Chen, C. Chen, X. Wang, X. Lin, and W. Zhang, "Finding the maximum $k$-balanced biclique on weighted bipartite graphs," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2022.

[38] T. Shi, M. Zhai, Y. Xu, and J. Zhai, "Graphpi: High performance graph pattern matching through effective redundancy elimination," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020, pp. 1–14.

[39] X. Chen, R. Dathathri, G. Gill, L. Hoang, and K. Pingali, "Sandslash: A two-level framework for efficient graph pattern mining," in *Proceedings of the ACM International Conference on Supercomputing (ICS)*, 2021, p. 378–391.

[40] X. Chen *et al.*, "Efficient and scalable graph pattern mining on gpus," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022, pp. 857–877.

[41] T. Shi, J. Zhai, H. Wang, Q. Chen, M. Zhai, Z. Hao, H. Yang, and W. Chen, "Graphset: High performance graph mining through equivalent set transformations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2023.

[42] M. Almasri, I. E. Hajj, R. Nagi, J. Xiong, and W.-m. Hwu, "Parallel k-clique counting on gpus," in *Proceedings of the 36th ACM International Conference on Supercomputing (ICS)*, 2022, pp. 21:1–21:14.

[43] M. Almasri, Y.-H. Chang, I. E. Hajj, R. Nagi, J. Xiong, and W.-m. Hwu, "Parallelizing maximal clique enumeration on gpus," in *32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2023, pp. 162–175.

[44] L. Qiao, R. Li, Z. Zhang, Y. Yuan, G. Wang, and H. Qin, "Maximal quasi-cliques mining in uncertain graphs," *IEEE Transactions on Big Data*, vol. 9, no. 1, pp. 37–50, 2023.

[45] C. Gui, X. Liao, L. Zheng, and H. Jin, "Cyclosa: Redundancy-free graph pattern mining via set dataflow," in *USENIX Annual Technical Conference (ATC)*, 2023, pp. 71–85.

# Appendix: Artifact Description/Artifact Evaluation

## Artifact Description (AD)

### I. Overview of Contributions and Artifacts

*A. Paper's Main Contributions*

In this paper, we present a new maximal biclique enumeration algorithm on CPUs, named AdaMBE, which exploits computational subgraphs in the enumeration process. We make the following contributions.

$C_1$ We use derived computational subgraphs in the enumeration. Because the computational subgraphs corresponding to a vertex can be retrieved from its parent's neighborhood information, we don't need to access its neighborhood information in the original bipartite graph. Moreover, we store the parent's neighborhood information in a cache to support high-performance access.

$C_2$ We introduce a hybrid representation of computational subgraphs in memory, which selectively generates bitmap-based subgraphs for small computational subgraphs. This approach expedites extensive set intersections through efficient bitwise operations.

$C_3$ We propose an adaptive maximal biclique enumeration algorithm, named AdaMBE, which integrates the neighborhood data cache and the bitmap representation. We also propose the parallel version of AdaMBE, which is called ParAdaMBE.

For simplicity, we denote the approach discussed in $C_1$ as LN (indicating the utilization of Local Neighborhood information), and the approach in $C_2$ as BIT (representing the use of BITmap on small subgraphs). We evaluate their effectiveness using three AdaMBE variants: Baseline (disabling both LN and BIT in AdaMBE), AdaMBE-LN (enabling LN only), and AdaMBE-BIT (enabling BIT only).

*B. Computational Artifacts*

$A_1$ https://doi.org/10.5281/zenodo.12042681(Gihub repository: https://github.com/ISCS-ZJU/AdaMBE.git)

| Artifact ID | Contributions Supported | Related Paper Elements |
|---|---|---|
| $A_1$ | $C_1$ , $C_2$ and $C_3$ | Figures 8-14 |

The following is the directory structure of $A_1$ :

- **README.md:** This file contains a detailed step-by-step "Try out AdaMBE" guide.
- **src/:** This directory contains the source code of AdaMBE and its variants Baseline, AdaMBE-LN and AdaMBE-BIT.
- **scripts/:** This directory has scripts to run experiments.
- **baselines/:** This directory contains the source codes of the comparison baselines, i.e., PMBE, FMBE, ParMBE, ooMBEA, and GMBE.
- **preprocess/:** This directory has scripts to preprocess the graph datasets.
- **fig/:** This directory contains the scripts to generate figures.

### II. Artifact Identification

*A. Computational Artifact $A_1$*

*Relation To Contributions*

This artifact provides the source code and scripts to reproduce all the experimental results to verify both $C_1$ , $C_2$ and $C_3$.

*Expected Results*

AdaMBE should be faster than all comparison baselines on all datasets due to the improvement of $C_1$ and $C_2$. And the memory usage of AdaMBE should be lower than PMBE and ooMBEA but higher than FMBE.

ParAdaMBE should be faster than ParMBE on all datasets and faster than GMBE on most datasets. The memory usage of ParAdaMBE should be lower than ParMBE and GMBE.

The numbers of nodes with non-maximal bicliques can be efficiently reduced by AdaMBE-LN due to the improvement of $C_1$. AdaMBE-BDS significantly reduces the running time of small nodes due to the improvement of $C_2$. AdaMBE-LN and AdaMBE-BDS should be faster than Baseline, and AdaMBE should be faster than them.

AdaMBE outperforms all serial competitors (i.e. FMBE, PMBE, and ooMBEA) under different dataset size. Compared to ParMBE, ParAdaMBE has better scalability as the number of threads increases.

This computational artifact also provides scripts for reproducing experimental results that guide the selection of an optimal vertex order (which should be ascending order based on degrees of vertices) and the threshold value $\tau$ (which should be 64). Notably, the threshold $\tau$ determines the ideal moment to create a bimap-based subgraph.

*Expected Reproduction Time (in Minutes)*

It will take roughly 30 minutes, 2 hours, and 1 minute to complete Artifact Setup, Artifact Execution, and Artifact Analysis respectively. In the step Artifact Execution, the running progress and the expected running time of each experiment would be printed to the file *scripts/progress.txt* automatically.

*Artifact Setup (incl. Inputs)*

*Hardware:* We run all experiments on a machine equipped with four Intel Xeon(R) Gold 5318Y 2.10GHz CPUs (24 cores per CPU), 128GB of main memory, and an NVIDIA A100 GPU with 40GB of global memory. The operating system is Ubuntu 20.04.6 LTS.

*Software:* The following lists the packages applied in this computational artifact.

- zplot 1.41: https://github.com/z-plot/z-plot

- oneTBB 1.1: https://github.com/oneapi-src/oneTBB - Nvidia driver 510.85.02: https://www.nvidia.com/download/driverResults.aspx/192513/en-us/

*Datasets:* We use 14 real-world datasets for our evaluation. The following provides the urls for all the datasets.

- Unicode: http://konect.cc/files/download.tsv.unicodelang.tar.bz2

- UCforum: http://konect.cc/files/download.tsv.opsahl-ucforum.tar.bz2

- MovieLens: http://konect.cc/files/download.tsv.movielens-10m_ti.tar.bz2

- Teams: http://konect.cc/files/download.tsv.dbpedia-team.tar.bz2

- ActorMovies: http://konect.cc/files/download.tsv.actor-movie.tar.bz2

- Wikipedia: http://konect.cc/files/download.tsv.wiki-en-cat.tar.bz2

- YouTube: http://konect.cc/files/download.tsv.youtube-groupmemberships.tar.bz2

- StackOverflow: http://konect.cc/files/download.tsv.stackexchange-stackoverflow.tar.bz2

- DBLP: http://konect.cc/files/download.tsv.dblp-author.tar.bz2

- IMDB: http://konect.cc/files/download.tsv.actor2.tar.bz2

- BookCrossing: http://konect.cc/files/download.tsv.bookcrossing_full-rating.tar.bz2

- Gihub: http://konect.cc/files/download.tsv.github.tar.bz2

- CebWiki: http://konect.cc/files/download.tsv.edit-cebwiki.tar.bz2

- TVTropes: http://konect.cc/files/download.tsv.dbtropes-feature.tar.bz2

*Installation and Deployment:* The following lists the compilers to build and run $A_1$.

- GCC/G++ 10.3.0
- GNU Make 4.2.1
- CMake 3.22.0
- CUDA toolkit 11.7
- Python 3.8.10

*Artifact Execution*

**Step 1: Prepare datasets.**

We use 14 real-world datasets for our evaluation. Users can download and preprocess them with the script in the directory *preprocess/*. After executing the script, users can find the preprocessed datasets in the new directory *datasets/*.

**Step 2: Execute scripts to generate results.**

We provide the scripts to automatically generate the experimental results of Figures 8-14 in *scripts/*. All the experimental results would be printed to the file *scripts/results.txt* and the results required to generate the figures would be printed to the data file in the corresponding subdirectories in *fig/*.

*Artifact Analysis (incl. Outputs)*

We plot the figures from raw data with python. We provide the scripts to generate figures in the directory *fig/*. The generated figures would be found under the directory *fig/*.

# Artifact Evaluation (AE)

## A. Computational Artifact $A_1$

### Artifact Setup (incl. Inputs)

We provide a docker image for the convenience to deploy this project. You should install some packages and download the docker image with the following commands before you run the docker image.

```
sudo apt install nvidia-docker2
docker pull fhxu00/adambe
```

To run the docker image, you should execute the following command. To measure the performance of GMBE, assure that the host machine has installed the nvidia driver with the version mentioned above.

```
docker run -it --gpus all --name adambe-test fhxu00/
    adambe bash
```

We have prepared the source code in the directory $\sim /AdaMBE - public$ and downloaded all testing datasets in the docker image. Now you just need to compile the source code and run the testing scripts in the docker image.

Using the following commands, one can easily compile the AdaMBE and baselines. The generated executable files are located at $bin/$.

```
# Compiling AdaMBE
bash ./scripts/compile-adambe.sh
# Compiling baselines. [GPU_TYPE] denotes the GPU
    used by GMBE (refer to https://github.com/fhxu00
    /MBE-GPU.git).
bash ./scripts/compile-baselines.sh [GPU_TYPE]
```

You can run AdaMBE and its variants with the following command-line options.

```
./bin/MBE_ALL
 -i: The path of input dataset file.
 -s: Select one AdaMBE version to run. 5: AdaMBE-BIT
     , 6: AdaMBE-LN, 7: AdaMBE, 8: ParAdaMBE.
 -t: Number of threads used to run AdaMBE, only
     useful for ParAdaMBE.
 -o: The ordering technique used. 1: random, 2:
     increasing, 3: unilateral order mentioned in
     ooMBEA.
```

### Artifact Execution

To limit the time for reproduction, we provide scripts to produce only key results of the paper. You just need to run the following command.

```
bash ./scripts/gen-simplified-results.sh
```

The script will produce the experimental results of fig-8 and fig-10 on small datasets.

We also provide the scripts to generate all the experimental results of Figure 8-14 in the directory scripts/. You can execute the scripts as following, which will take a long time to complete.

```
# Running on a machine with a CPU and a GPU
bash ./scripts/gen-fig-8.sh

# Running on any machine
bash ./scripts/gen-fig-9.sh

# Running on any machine
bash ./scripts/gen-fig-10.sh

# Running on any machine
bash ./scripts/gen-fig-11.sh

# Running on any machine
bash ./scripts/gen-fig-12.sh

# Running on any machine
bash ./scripts/gen-fig-13.sh

# Running on a machine with a CPU
bash ./scripts/gen-fig-14.sh
```

The scripts will conduct experiments to produce results of all figures int the paper, and the results will be printed to the data file in the corresponding subdirectories in $fig/$.

### Artifact Analysis (incl. Outputs)

We provide the script to generate figures in the directory $fig/$ with the results generated in above. You can execute the script as following.

```
cd fig/
bash genfig.sh
```

Then you will find the figures under the directory $fig/$. If you execute the simplified version script in the stage of artifact execution, only the fig-8 and fig-10 will be generated.

AdaMBE should be faster than all comparison baselines on all datasets. And the memory usage of AdaMBE should be lower than PMBE and ooMBEA but higher than FMBE. ParAdaMBE should be faster than ParMBE on all datasets and faster than GMBE on most datasets. The memory usage of ParAdaMBE should be lower than ParMBE and GMBE. All these results should be verfified in the fig-8 and fig-9.

The numbers of nodes with non-maximal bicliques can be efficiently reduced by AdaMBE-LN. AdaMBE-BDS significantly reduces the running time of small nodes. AdaMBE-LN and AdaMBE-BDS should be faster than Baseline, and AdaMBE should be faster than them. All these results should be verfified in the fig-10.

The fig-11 and fig-12 should verify that the performance is best when threshold is set to 64 and the vertex ordering is ascending order respectively.

AdaMBE outperforms all serial competitors (i.e. FMBE, PMBE, and ooMBEA) under different dataset size, which should be verfied in the fig-13.

Compared to ParMBE, ParAdaMBE has better scalability as the number of threads increases, which should be verfied in the fig-14.