Efficient Mining of Maximal Bicliques in Graph by Pruning Search Space

Youngtae Kim[1], Dongyul Ra[2]
Computer and Telecommunications Engineering Division
Yonsei University, Wonju, Kangwon, South Korea

***Abstract*—In this paper, we present a new algorithm for mining or enumerating maximal biclique (MB) subgraphs in an undirected general graph. Our algorithm achieves improved theoretical efficiency in time over the best algorithms. For an undirected graph with n vertices, m edges and k maximal bicliques, our algorithm requires $O(kn^2)$ time, which is the state of the art performance. Our main idea is based on a strategy of pruning search space extensively. This strategy is made possible by the approach of storing maximal bicliques immediately after detection and allowing them to be looked up during runtime to make pruning decisions. The space complexity of our algorithm is $O(kn)$ because of the space used for storing the MBs. However, a lot of space is saved by using a compact way of storing MBs, which is an advantage of our method. Experiments show that our algorithm outperforms other state of the art methods.**

***Keywords*—Graph algorithms; maximal bicliques; maximal biclique mining; complete bipartite graphs; pruning search space; social networks; protein networks**

I. Introduction

A biclique is a graph (or a subgraph of a graph) whose vertex set can be partitioned into two component sets where every vertex in one set is adjacent to every vertex in the other set. A biclique is also referred to as a complete bipartite graph. A maximal biclique (MB) of a graph G is a biclique which cannot be not a subgraph of another biclique of G.

Nowadays social networks based on the internet or mobile communications are popular [1]. Protein interaction networks receive much attention in biomedical areas [2]. The emerging block chain technology must handle large-scale graphs [3]. In these fields, enumerating all MBs existing (as subgraphs) in a graph is very important to many practical data mining problems. As networks get large in size, efficiency in speed and space of algorithms becomes important.

In this paper, we introduce a new efficient algorithm that can enumerate all MBs in an undirected graph given as input. Henceforth, we use variables n, m and k to denote the number of vertices, edges and MBs in an input graph, respectively. The emphasis in this research is to improve performance of fully general algorithms that involves no constraints. The constraints that can be placed on the algorithms are diverse. Some algorithms accept only bipartite graphs as input. Other algorithms produce only MBs whose component sets are independent sets. There can be size constraints on the component sets. We aim to design a fully general algorithm that does not have any such constraints.

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2017R1D1A3B03031855).

Our approach is based on a new idea of exploiting search space pruning techniques to gain efficiency. In contrast to other fully general algorithms, ours looks up stored MBs to make decisions related to pruning search space, which allows to gain efficiency in time. As a result, we discovered an algorithms with $O(kn^2)$ and $O(kn)$ as time and space complexity, respectively.

Our algorithm's time complexity $O(kn^2)$ can be considered to be a significant improvement over the current state of the art $O(kmn)$ [4]. The algorithm of Li et al. [4] has been the state of the art for more than a decade and a half among the fully general algorithms. This means that improving speed of the best fully general algorithm has been quite hard. In this respect, contribution of our work is nontrivial.

The theoretical space complexity of our algorithm is $O(kn)$ due to the space required to store all MBs. This space requirement seems natural considering the fact that the MBs enumerated anyway need to be loaded into memory to allow application tasks to utilize them. In our scheme of storing MBs, a lot of space can be saved by using a compact way of storing MBs. This is due to the fact that the component vertex sets of different MBs can share their parts and thus the actual amount of space required can be quite less than that of theoretical expectation. This is another advantage of our algorithm. How much space is saved depends on the structure of the graph. It was observed in the experiments that more than 50% of the space is easily saved in case of dense input graphs.

II. Related Work

A lot of research has been done on the problem of mining all MBs in an undirected graph G. Algorithms for this purpose belong to one of three categories. Algorithms in the first category have a constraint that the input graph should be bipartite. Algorithms of the other two categories do not have the bipartite-graph constraint. The algorithms in the second category have a restriction that the components of MBs should be independent sets. In other words, they only generate maximal induced bicliques. Algorithms that do not need any constraints or restrictions belong to the third category.

Various algorithms of the first category were developed in the past [5, 6]. Makino and Uno [7] proposed an algorithm whose time complexity is $O(n^4)$ time and $O(n^2)$ space. Zhang et al. [8] recently introduced a novel efficient algorithm of time complexity $O(d^2n^2)$ where d is the maximum degree of any vertex. The space complexity is $O(\min(d,a)b)$ and a and b are

cardinalities of the two vertex partitions composing the input bipartite graph G.

Many algorithms have also been developed that belong to the second category by allowing a general undirected graph as input. One of them is the algorithm introduced by Dias et al. which requires $O(kn^4)$ time and $O(2^n)$ space [9]. However, this algorithm generates only maximal induced bicliques. If an MB consists of component sets at least one of which is not an independent vertex set, the MB is not enumerated. Kloster et al. [10] pursued improving the algorithm of Dias et al. But their algorithm is specifically designed for general graphs which are near to bipartite graphs. Their algorithm has time complexity of $O(knmh^2 3^{h/3})$ where h is the cardinality of the vertex set whose deletion from G makes G a bipartite graph. Sullivan et al. [11] attempted to even further improve the algorithm of Kloster et al. and achieved time complexity of $O(knmh)$.

There has been research on developing fully general algorithms belonging to the third category. Liu et al. [12] effectively uses the size constraints on both vertex sets to prune unpromising bicliques and to reduce the search space iteratively during the mining process. The time complexity of the proposed algorithm is $O(kdn)$, where d is the maximal degree of the vertices. But this algorithm has a size constraint in such a way that only MBs are enumerated whose components' sizes are above a threshold ms. One of those fully general algorithms with no constraints was proposed by Alexe et al. [13] which has $O(kn^3)$ and $O(kn)$ as time and space complexity, respectively. Another general algorithm in this category is that of Tomita et al. [14] whose time complexity is $O(3^{n/3})$. The state of the art algorithm in this category is the one by Li et al. [4] as mentioned in section I. This algorithm has time complexity of $O(kmn)$ and space complexity of $O(mn)$. However, this space complexity does not include the space for storing the MBs enumerated. When the MBs enumerated need to be stored to be used later by application tasks, their space complexity should be $O(kn)$.

III. Preliminary on Maximal Bicliques

We assume that an undirected graph $G = (V, E)$ is given to the algorithm where V denotes a vertex set and E an edge set. Let n and m denote the number of vertices and edges in G, respectively. We use integers between 1 and n to denote vertices. Thus $V = \{1, \dots, n\}$. An edge is represented by a set of two vertices (no order between the two). It is said that a vertex of an edge is adjacent to the other vertex of the edge. We use an adjacency list representation of G. In this representation, there is a list $L(v)$ for each vertex v in V which is an ordered list of vertices which are adjacent to v.

Let V_1 and V_2 be disjoint subsets of V. If every vertex in V_1 is adjacent to every vertex in V_2, then V_1 and V_2 form a biclique $[V_1, V_2]$ which is a subgraph of G. Its vertex set is $V_1 \cup V_2$. Its edge set consists of all edges connecting a vertex in V_1 and a vertex in V_2. We call V_1 and V_2 the component vertex sets. A biclique formed by components V_1 and V_2 becomes a maximal biclique (MB) if there is no vertex set $X \supset V_1$ where X and V_2 form a biclique and no vertex set $Y \supset V_2$ where V_1 and Y form a biclique. If V_1 and V_2 form an MB, V_2 and V_1 can form an MB. Thus $[V_1, V_2]$ and $[V_2, V_1]$ is actually the same MB. Later in this paper, a specific ordering will be enforced for the two components in writing an MB.

To design our algorithm, we begin with the problem of finding a maximal vertex set which can form a biclique with a given vertex set X. This set is called an occurrence set of X, which is denoted by $\mathrm{Oc}(X)$ [4]. Throughout this paper, V and E denote the vertex and edge set of the input graph G, respectively.

Definition 1: An occurrence set of $X \subseteq V$ is $\mathrm{Oc}(X) = \{v \in V \mid v \notin X$ and v is adjacent to all vertices in $X\}$.

By the definition of $\mathrm{Oc}(X)$, it is important to note that $\mathrm{Oc}(X)$ is a maximal vertex set for given X. In other words, there is no vertex set H where $H \supset \mathrm{Oc}(X)$ and H can form a biclique with X.

Theorem 1: Let $X \subseteq V$. $[X, \mathrm{Oc}(X)]$ is a biclique.

Proof: Let us select any vertex $u \in X$ and any vertex $v \in \mathrm{Oc}(X)$. By Definition 1, $(u, v) \in E$. Thus X and $\mathrm{Oc}(X)$ form a biclique. Q.E.D.

Note that $[X, \mathrm{Oc}(X)]$ is a biclique. We need to know whether this biclique is a maximal biclique or not. Closure of a vertex set X , $\mathrm{Cl}(X)$, is a maximal vertex set extended from X which forms a biclique with $\mathrm{Oc}(X)$. It is formally defined in Definition 2. Theorem 2 and 3 provide a method for deciding whether $[X, \mathrm{Oc}(X)]$ is an MB or not.

Definition 2: Closure of a vertex set $X \subseteq V$, $\mathrm{Cl}(X)$, is the occurrence set of $\mathrm{Oc}(X)$. I.e., $\mathrm{Cl}(X) = \mathrm{Oc}(\mathrm{Oc}(X))$.

Theorem 2: Let $X \subseteq V$. $[\mathrm{Cl}(X), \mathrm{Oc}(X)]$ is an MB.

Proof: $\mathrm{Cl}(X) = \mathrm{Oc}(\mathrm{Oc}(X))$. Thus $\mathrm{Oc}(X)$ and $\mathrm{Cl}(X)$ constitute a biclique by Theorem 1. There is no vertex $v \notin \mathrm{Oc}(\mathrm{Oc}(X))$ which is adjacent to all vertices in $\mathrm{Oc}(X)$ by the property of an occurrence set.

If we assume that there is a vertex $v \notin \mathrm{Oc}(X)$ which is adjacent to all vertices in $\mathrm{Oc}(\mathrm{Oc}(X))$, contradiction occurs since v is adjacent to all vertices in X and thus it should be in $\mathrm{Oc}(X)$. Thus $\mathrm{Cl}(X)$ and $\mathrm{Oc}(X)$ meet the condition of forming an MB. The theorem holds. Q.E.D.

Theorem 3: Let $X \subseteq V$. Then $X \subseteq \mathrm{Cl}(X)$.

Proof: X forms a biclique with $\mathrm{Oc}(X)$. $\mathrm{Cl}(X) = \mathrm{Oc}(\mathrm{Oc}(X))$ is a maximal set that forms a biclique with $\mathrm{Oc}(X)$. Thus $X \subseteq \mathrm{Oc}(\mathrm{Oc}(X))$. Q.E.D.

Theorem 4 states that if a vertex is added to a set, the corresponding occurrence set may lose some vertices.

Theorem 4: For any vertex set X, and a vertex $v \notin X$, if $Z = \mathrm{Oc}(X \cup \{v\})$, then $Z \subseteq \mathrm{Oc}(X)$.

Proof: Z consists of only those vertices in $\mathrm{Oc}(X)$ which are adjacent to v. If there is a vertex u in $\mathrm{Oc}(X)$ which is not adjacent to v, u does not belong to Z. Thus the theorem holds. Q.E.D.

IV. Set Enumeration Tree

A. Set Enumeration Tree as a Search Space

A graph with a large number of vertices may have a huge number of MBs. The basic strategy of enumerating all MBs is simple as follows: for each $Y \subseteq V$, compute Oc(Y), Cl(Y) and then enumerate [Y, Oc(Y)] as an MB if Y = Cl(Y). In this strategy, all subsets of V should be tried as Y. Therefore, the search space to find MBs is the power set of V. We use a set enumeration tree as the conceptual model and data structure of the search space [15].

In a set enumeration (SE) tree for a vertex set V, a unique node exists for every subset of V. Each vertex is represented by an integer label. The i^{th} vertex in V is given label i, $1 \le i \le n$. The SE tree for V = {1, ... , 4} is illustrated in Fig. 1. Every node has a vertex label. Every node represents a unique subset of V which is formed by including the vertex labels of all nodes on the path from the root to the node. For example, consider the orange node with label 4 in Fig. 1. This node represents the vertex set {1, 3, 4}. All nodes of the SE tree for V covers all subsets of V. A vertex set and its corresponding node in the SE tree is used interchangeably in this paper.

Note that a node with label b has children with labels from b + 1 to n. For a node X, the set of labels on all child nodes is called its tail set, Tail(X). In Fig. 1, Tail(X) = {2, 3, 4} for the node X = {1}.

A depth-first search (DFS) traversal scheme is used to visit all nodes in an SE tree. Fig 2 illustrates the order of node visits in DFS traversal. When the control arrives at a node for the first time, this is the visit to the node. After the control leaves a node, it may return to the node again later by backtracking from a child node. In this paper, a visit to a node stands for the first visit and not the return caused by backtracking. During the (first) visit to a node, the processing related to the node is performed. This is a kind of preorder traversal. The control at a node moves to the leftmost unvisited child of the node. If a node has no more unvisited child, the control backtracks to the parent of the node.

Definition 3: Relation Prior(X, Y) is true if and only if the visit to node X comes before the visit to node Y during DFS traversal of the SE tree. If Prior(X, Y), it is said that X is prior to Y.

Definition 4: Subtree(Y) denotes the subtree whose root is the node of vertex set Y.

For a given graph G, our algorithm does not explicitly build the SE tree of G. It uses a recursive function to implement the DFS traversal of the implicit SE tree. The basic design of our algorithm is the recursive function Basic_GenMB. Our algorithm is started by invoking the recursive function with an empty set ∅ passed to X.

Algorithm Basic_GenMB (X, Oc(X), Tail(X)):
(1) if [X, Oc(X)] is an MB, mine it;
(2) for each v in Tail(X):
(3) $Y \leftarrow X \cup \{v\}$; Tail(Y) = { $u \in V \mid u \in$ Tail(X) and $u > v$};
(4) Compute Oc(Y) using Oc(X) and v;
(5) if (Oc(Y) ≠ ∅)
(6) Basic_GenMB (Y, Oc(Y), Tail(Y));

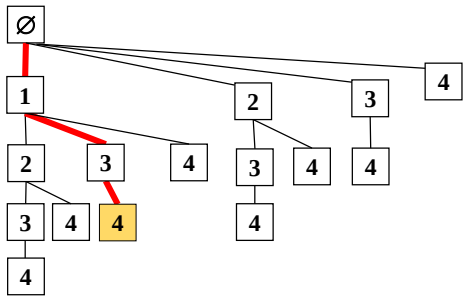

Fig. 1. Set Enumeration Tree with n = 4.

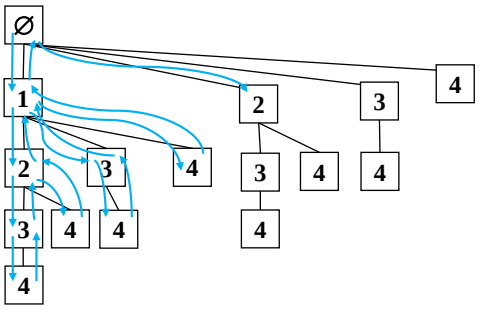

Fig. 2. Depth-First Search Traversal of the SE Tree.

Though significant pruning of search space is done at step 5 of our basic algorithm, more pruning needs to be pursued to improve efficiency. In our algorithm, it is assumed that all vertex sets are ordered sets to improve efficiency in computation. Assume the two vertex sets X and Y form an MB. We write the MB as [X, Y] if Prior(X, Y) is true. Otherwise, we write [Y, X]. In an MB, the component which is prior to the other is the first component and the other the second component. In our algorithm, an MB is mined (i.e. discovered and registered) when its first component is visited. Thus, if [X, Y] was mined before, it means Prior(X, Y) is true and the node X was visited already. When the second component of an MB is visited, the MB is not produced again to avoid duplicate mining.

B. Storing Maximal Bicliques

The techniques for achieving efficiency in our algorithm are based upon looking up the MBs already mined during the run of our algorithm. To exploit this idea, it is required to store MBs as soon as they are identified and mined. Our algorithm does not construct the SE tree explicitly. Our algorithm uses an implicit SE tree as the whole search space. When an MB is detected, it should be stored immediately. Its two component vertex sets need to be stored. To store a component, its corresponding node in the SE tree is constructed. The path corresponding to this node is also constructed. The initial part of the path is shared with other existing paths as much as possible.

A tree in which MBs are stored is called an MB tree. The MB tree is a subgraph of the SE tree. The MB tree has only the paths for the components of MBs generated so far. The two nodes representing the components of an MB point to each other by the component pointer (CP). From a node of a component of an MB, the node of the other component can be accessed instantly by using the CP pointer.

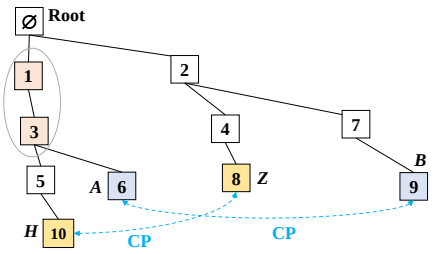

Fig. 3. MB Tree Containing Two MBs: [*H*, *Z*] and [*A*, *B*].

A snapshot of an MB tree is shown in Fig. 3. One MB, [*H*, *Z*], was stored where *H* is {1, 3, 5, 10} and *Z* {2, 4, 8}. Another MB, [*A*, *B*], also exists in the MB tree where *A* = {1, 3, 6}, *B* = {2, 7, 9}. Note that the paths of *H* and *A* share a sub-path consisting of nodes 1 and 3. The paths of an MB tree have the same structure as those of an SE tree. In making decisions about pruning search space, our algorithm needs to look up an MB produced before. Storing and looking up an MB is efficient by adopting the idea of an MB tree.

V. Exploiting Pruning Techniques

We will modify Basic_GenMB to improve efficiency by utilizing MBs stored in the MB tree and pruning search space. Note that MBs are stored as soon as they are identified during the run of the algorithm. An MB is constructed and stored as soon as its first component is visited for the first time during DFS traversal. In this section we will introduce pruning techniques exploited by our algorithm. Our algorithm is a recursive function GenMB. To run our algorithm, the function is invoked as follows: GenMB(∅, *V*, *V*, 0). The algorithm starts at the root of the SE tree. The roles of the parameters are as follows:

- *X*: a vertex set which is being visited by DFS.
- Oc(*X*): the occurrence set of *X*.
- Tail(*X*): the tail set of *X*.
- genflag: If this flag receives 1, an MB should be generated using (extended) *X* and Oc(*X*).

```
Algorithm GenMB ( X, Oc(X), Tail(X), genflag ):
(1)  if (genflag =1 and X ≠ ∅) {
(2)    Closure_extension (X, Oc(X), Tail(X)) ;
(3)    Generate and store MB [X, Oc(X)];
       } // end if
(4)  for each v in Tail(X) do {
(5)      Y ← X ∪ {v} ; Tail(Y) ← {u | u ∈ Tail(X) and v < u} ;
(6)      Compute Oc(Y) using Oc(X) and v;
(7)      if (Oc(Y) = ∅) continue; // Pruning-1
(8)      if (Oc(Y) = 2nd component of MB stored already)
             continue;   //Pruning-2
(9)      if (Oc(Y) = 1st component of MB stored already) {
(10)         Obtain Cl(Y) from node of Oc(Y) using CP link;
(11)         if (Prior(Cl(Y), Y)) continue; // Pruning-3
(12)         else {  Extend Y to Cl(Y) and update Tail(Y);
(13)                 GenMB(Y, Oc(Y), Tail(Y), 0); // Pruning-4
             } // end if
         } // end if
(14)     GenMB(Y, Oc(Y), Tail(Y), 1 ); // Pruning-5
     } // end for
```

If genflag is 1, it means that generation of an MB using *X* and Oc(*X*) is requested. *X* may not be maximal to form an MB. So *X* is extended to its closure Cl(*X*) on line 2. Tail(*X*) is updated by removing vertices added to *X*. More detailed explanation for closure extension and update will be provided later in this section. An MB consisting of Cl(*X*) and Oc(*X*) is generated and stored on line 3. If genflag is 0, it means that an MB composed of *X* and Oc(*X*) was generated previously and thus the MB should not be generated again to avoid duplication. But DFS traversal should continue for nodes in Subtree(*X*). Action "continue" on line 7 and 11 stands for "jumping to next iteration".

The loop from line 4 to 14 is to traverse SE nodes in the subtrees of children of *X*. On line 5, *Y* (a child of *X* in the SE tree) is proposed to be visited next by DFS. *Y* was not visited before. It is visited now. Thus *Y* cannot be a first component of an MB generated earlier. It is necessary to compute Oc(*Y*) (line 6).

A. Pruning scheme 1

Pruning-1: If Oc(*Y*) is an empty set where *Y* is the node to visit next on line 7 of GenMB, Subtree(*Y*) can be pruned.

The first strategy of pruning search space, Pruning-1, is applied on line 7 of GenMB. If Oc(*Y*) is an empty set ∅, there is no need of visiting nodes in the subtree of *Y*. This pruning is possible since the occurrence set of nodes in those subtrees will be ∅ by Theorem 5. The action of "continue" on line 7 makes the algorithm to ignore the remaining part of the loop and start the next iteration (as in C language). This has the effect of ignoring or pruning Subtree(*Y*) during DFS traversal.

Definition 5: If X can be obtained by taking zero or more consecutive elements starting from the first element of an ordered set Z, Prefix(X, Z) is true. Otherwise, Prefix(X, Z) is false.

B. Pruning scheme 2

Pruning-2: If Oc(*Y*) exists in the MB tree as a second component of an MB stored already on line 8 of GenMB, then Subtree(*Y*) can be pruned.

Let us consider a situation to which Pruning-2 can be applied. Fig. 4 has an example. By Theorem 3, Cl(*Y*) is the first component of the MB. $Y \subseteq$ Cl(*Y*) by Theorem 2. Cl(*Y*) was visited already before *Y*, which can be derived by the existence of Oc(*Y*) in an MB discovered already. Thus Prior(Cl(*Y*), *Y*). Thus ¬Prefix(*Y*, Cl(*Y*)). Symbol ¬ denotes negation. In this case, Subtree(*Y*) can be pruned safely (on line 8 of GenMB). Theorem 5 proves that this pruning is safe.

Theorem 5: Pruning-2 is safe (This action will not prevent any MB from being generated.)

Proof: Cl(Y) forms a biclique with Oc(Y) by Theorem 3. $Y \subseteq$ Cl(Y). $Y \neq$ Cl(Y) since Cl(Y) was visited already and Y is being visited now. Thus $Y \subset$ Cl(Y). Let $v \in$ (Cl(Y) – Y). Note that v is adjacent to all vertices in Oc(Y) since $v \in$ Cl(Y).

Y and Oc(Y) form a biclique by definition of an occurrence set. However, Y cannot form a maximal biclique with Oc(Y)

since a bigger set (Y ∪{v}) can form a biclique with Oc(Y). This argument can be applied to any node in Subtree(Y).

Consider any node L (other than Y) in Subtree(Y). L is obtained by adding one or more vertices to Y. (For example, consider L in Fig. 4.) L forms a biclique with Oc(L). By Theorem 4, Oc(L) ⊆ Oc(Y). Thus v ⊆ (Cl(Y) – Y) is adjacent to all vertices in Oc(L). Therefore, L∪{v} is completely connected with Oc(L). Thus L and Oc(L) can form a biclique but not a maximal biclique because a bigger set $L\cup\{v\}$ can form a biclique with Oc(L). Thus no node in Subtree(Y) can be a component of a maximal biclique. Q.E.D.

C. Pruning scheme 3

Pruning-3: If Oc(Y) exists in the MB tree as a first component of an MB and Prior(Cl(Y), Y) is true on line 11 of GenMB, then Subtree (Y) can be pruned.

Let us consider a situation to which Pruning-3 is applicable. Fig. 5 gives an example of such a situation. Cl(Y) exists in the MB tree as a second component of an MB stored already. Line 11 of our algorithm implements this pruning by executing "continue". Theorem 6 below proves that Pruning-3 is safe.

Theorem 6: Pruning-3 is safe (This action will not prevent any MB from being generated.)

Proof: An MB [Oc(Y), Cl(Y)] was stored before. Since Prior(Cl(Y), Y), Cl(Y) was visited before Y. Y ⊆ Cl(Y) by Theorem 3. Y and Cl(Y) cannot be equal since Cl(Y) was visited before and Y is now being visited. So Y ⊂ Cl(Y).

If Prefix(Y, Cl(Y)) is assumed, Prior(Y, Cl(Y)), which is a contradiction. Thus ¬Prefix(Y, Cl(Y)). There should v ∈ (Cl(Y) – Y) since Y ⊂ Cl(Y). (Y ∪{v}) can form a biclique with Oc(Y). So Y and Oc(Y) cannot form an MB.

We can apply the same argument used in the proof of Theorem 5 to conclude that Subtree(Y) can be pruned without missing any MBs. Q.E.D.

D. Pruning scheme 4

Pruning-4: If Oc(Y) exists as a first component of an MB stored already and ¬Prior(Cl(Y), Y) (line 12 of GenMB), then the DFS traversal visits a node W in Subtree(Y) and all nodes in Subtree(W) if and only if Cl(Y) ⊆ W. This leads to the fact that any Subtree(W) contained in Subtree(Y) will be pruned if ¬[Cl(Y) ⊆ (W∪Tail(W))].

Fig. 6 shows a situation to which Pruning-4 can be applied. Since Prior(Cl(Y),Y) is false, Y is being visited now but Cl(Y) has not been visited yet. But Cl(Y) exists as a second component of an MB in the MB tree. Note that Y ⊆ Cl(Y) by Theorem 3. Thus Prefix(Y, Cl(Y)). Y = Cl(Y) or Y ⊂ Cl(Y). For example, node W in Fig. 6 will be pruned since Cl(Y) = {2, 4, 6, 8} is not a subset of W = {2, 4, 7} and Tail(W) = {8, 9, 10}. Subtree(W) will be pruned since any node in it can contain Cl(Y). Note that R is not visited since R = {2, 4, 5}. But, since Tail(R) = {6, 7, 8, 9, 10}, some nodes in Subtree(R) can contain Cl(Y) and thus can be traversed. For example, Z = {2, 4, 5, 6, 8} in Subtree(R) will be visited since Z contains Cl(Y). S in Subtree(R) and all nodes in Subtree(S) will be pruned since they cannot contain Cl(Y).

Theorem 7: Pruning-4 is safe (This action will not prevent any MB from being generated.)

Proof: Let W be a node in Subtree(Y). Assume that ¬ [Cl(Y) ⊆ W]. Let v be a vertex in Cl(Y) but not in W. Thus v is adjacent to all vertices in Oc(Y). Oc(W) ⊆ Oc(Y) by Theorem 4. Thus v is adjacent to all vertices in Oc(W) because v is adjacent to all vertices in Oc(Y).

Let Q = W ∪ {v}. Oc(Q) = Oc(W) because v is adjacent to all vertices in Oc(W). Cl(W) = Oc(Oc(W)) = Oc(Oc(Q)) = Cl(Q). Cl(W) ⊇ Q. Thus Cl(W) ⊃ W. Therefore, W cannot form a maximal biclique with Oc(W).

Instead, Cl(W) forms an MB with Oc(W). W needs not to be visited. If ¬ [Cl(Y) ⊆ (W∪Tail(W))], then no node Z in Subtree(W) can satisfy Cl(Y) ⊆ Z and thus Subtree(W) can be pruned. Q.E.D.

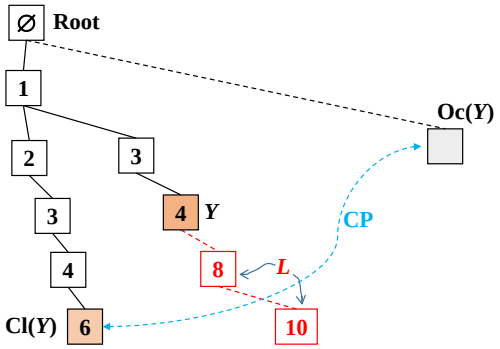

Fig. 4. Situation where Pruning-2 is Applicable.

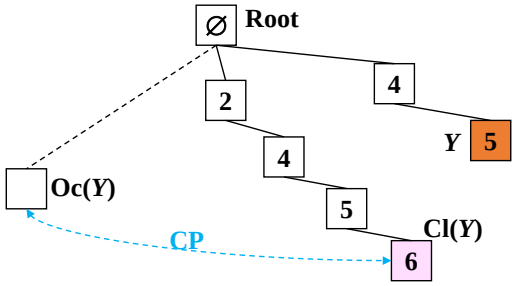

Fig. 5. Situation where Pruning-3 is Applicable.

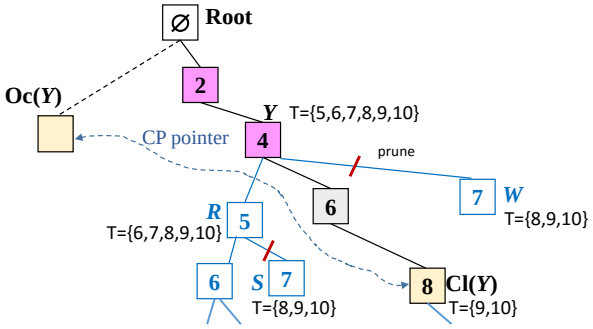

Fig. 6. Pruning-4 is Possible at Y; n =10, T:Tail.

To implement Pruning-4, our algorithm executes the operations on lines 12, 13 implemented by the next procedure.

Procedure for Pruning-4:
(i) // Extend *Y* to Cl(*Y*) and update Tail(*Y*) by using next for loop.
for each *v* in Tail(*Y*):
if (*v* in Cl(*Y*)) { add *v* to *Y*; remove *v* from Tail(*Y*); }
(ii) GenMB(*Y*, Oc(*Y*), Tail(*Y*), 0) ; // recursive invocation of itself

This procedure accomplishes pruning search space as stated in Pruning-4. Let us use a simple example in Fig. 7 to understand how our algorithm works related with Pruning-4. Let n = 10. After extension and update operations applied to Fig. 6, $Y = \{2, 4\}$ becomes $Y' = \{2, 4, 6, 8\}$ as in Fig. 7. Tail(Y') = {5, 7, 9, 10}. Note that Tail(Y') has extra vertices 5 and 7 in addition to {9, 10}, the normal tail set of Y'. Thus the tail set becomes unorthodox. GenMB(Y', Oc(Y), Tail(Y'), 0) is called. Zero is passed to genflag to suppress MB generation using Y' and Oc(Y). This call results in visiting only nodes Z in Subtree(Y) where $Z \supset$ Cl(Y) and all nodes in Subtree(Z). The nodes W (in Subtree(Y)) and Subtree(W) will be pruned if $\neg$ [Cl(Y) $\subseteq$ ($W \cup$Tail(W))].

Fig. 8 illustrates nodes in Subtree(Y={2,4}) in the MB tree of Fig. 7 that will be visited by the algorithm. So the parts of Subtree(Y) not covered by traversals in this figure are pruned. First, node Y of Fig. 8(a) and its subtree will be traversed. Secondly, Y of Fig. 8(b) and its subtree will be traversed. Then Y of Fig. 8(c) and its subtree will be traversed. Finally, Y of Fig. 8(d) and 8(e) and their subtree will be traversed.

Theorem 8: Procedure for Pruning-4 accomplishes pruning suggested by Pruning-4.

Proof: Let Y be a node at which Pruning-4 condition is met. We need to verify that Pruning-4 operations achieve the effect that Subtree(W) for any node W in subtree(Y) is pruned if $\neg$ [Cl(Y) $\subseteq$ (W$\cup$Tail(W))].

Extension operation updates Y and Tail(Y) accordingly (see step i). Let Y' and Tail(Y') denote the updated results. Thus Y' = Cl(Y). Tail(Y') contains all elements of Tail(Y) except those added to Y'. Then GenMB(Y', Oc(Y), Tail(Y'), 0) is invoked.

All nodes W that will be visited as a result of this invocation satisfy that Cl(Y) $\subseteq$ W since Y' = Cl(Y) and W is obtained by adding some nodes in Tail(Y') to Y'. Q.E.D.

E. Pruning scheme 5

Pruning-5: If neither Y nor Oc(Y) exist as a component of an MB stored already in the MB tree, the same type of pruning as Pruning-4 should be done. The algorithm will visit a node W in Subtree(Y) and all nodes in Subtree(W) if and only if Cl(Y) $\subseteq$ W. As a result, any Subtree(W) contained in Subtree(Y) is pruned if $\neg$[Cl(Y) $\subseteq$ ($W \cup$Tail(W))]

If the conditions required by the pruning strategies introduced so far are not satisfied by Y, our algorithm will arrive at line 14. This happens when neither Y nor Oc(Y) does exist as a component of an MB stored already in the MB tree. (Note that Y cannot be a first component of a stored MB since Y is being visited now and thus was not visited before.) In this case, it is guaranteed that Y is a prefix of Cl(Y), which is proved by Theorem 9. Thus the situation of this Y is quite similar to that of pruning case 4. In both cases, Y is a prefix of Cl(Y). In the current case, Cl(Y) will be a first component while, in case 4, Cl(Y) is a second component of an MB. In the current case, [Cl(Y), Oc(Y)] was not generated and thus will be generated. In case 4, an MB [Oc(Y), Cl(Y)] was generated already. For the current Y, Pruning-5 will be carried out which is similar to Pruning-4.

Note that Oc(Y) was not visited yet. Otherwise, it should exist as a component of an MB in the MB tree. Thus Prior(Y, Oc(Y)) is true. Fig. 9 shows an example of Y on line 14. An MB [Y, Oc(Y)] cannot be proposed as an MB because it is not known yet if Y = Cl(Y) or not. Oc(Y) $\neq \varnothing$ (determined on line 7). Thus Cl(Y) and Oc(Y) form an MB. But the algorithm is visiting node Y. The algorithm invokes a recursive call: GenMB(Y, Oc(Y), Tail(Y), 1). At the start of new instance of GenMB invoked by this call, the two operations are carried out (lines 2, 3 of GenMB): extending Y to Cl(Y) by executing procedure Closure_extension shown below, and generating and storing MB [Cl(Y), Oc(Y)].

Procedure Closure_extension performs the same task of step i of "Proceudre for Pruning-4" shown before. The only difference is that Cl(Y) is not known in this case and thus needs to be computed in using steps (1) and (2). Subtree(W) for any node W in subtree(Y) can be pruned if $\neg$[Cl(Y) $\subseteq$ ($W \cup$Tail(W))] where Y here is before being extended by Closure_extension.

Procedure Closure_extension (*Y*, Oc(*Y*), **Tail(*Y*)):**
(1) Scan all adjacency lists of vertices in Oc(*Y*);
(2) By scanning, occurrence count is obtained for each vertex;
(3) for each *v* in Tail(*Y*):
(4) if (Count of *v* = |Oc(*Y*)|)
{ Add *v* to *Y*; Remove *v* from Tail(*Y*); }
end for;

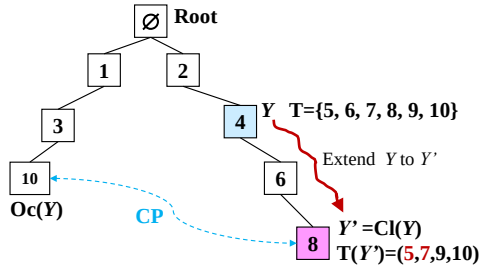

Fig. 7. Extension and update of Y and Tail (Y) to Y' and Tail (Y').

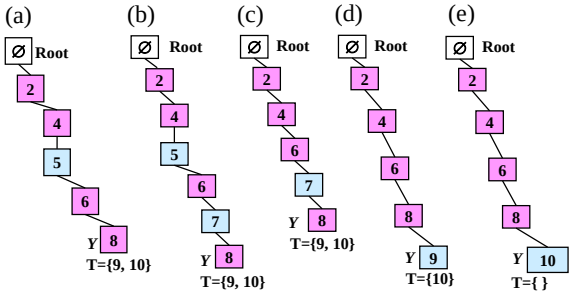

Fig. 8. More Examples of nodes visited while Pruning-4 is Done.

An example of pruning caused by Pruning-5 is shown in Fig. 9 by red line segments. Fig. 10 illustrates additional examples of pruning caused by Pruning-5. Green nodes and their subtrees are traversed; red lines indicate pruning of subtrees. Justifying Pruning-5 can be done using the same arguments used to justify Pruning-4.

Theorem 9: If neither Y nor Oc(Y) exists as a component of an MB stored already, Y is a prefix of Cl(Y).

Proof: A prefix of a node in the SE tree is visited before the node in DFS traversal because of the structure of SE tree. Oc(Y) ≠ ∅ since the test on line 7 was false. Thus Cl(Y) and Oc(Y) can constitute an MB.

If Cl(Y) had been visited already, Oc(Y) should exist as a component of an MB since Cl(*Y*) and Oc(*Y*) can form an MB (by Theorem 2). Because Oc(*Y*) does not exist as a component of an MB in the MB tree, Cl(*Y*) ⊇ *Y* was not visited yet. Therefore, *Y* is a prefix of Cl(*Y*). Q.E.D.

F. Correctness of our algorithm

Theorem 10 proves the correctness of our algorithm as an MB enumerator. In other words, there is no MB that is not generated by our algorithm.

Theorem 10: Our algorithm GenMB enumerates all MBs in a graph given as input.

Proof: DFS traversal implemented by GenMB visits all subsets of *V* in an SE tree if there is no pruning. *Y* proposed on line 5 of GenMB is the vertex set being visited.

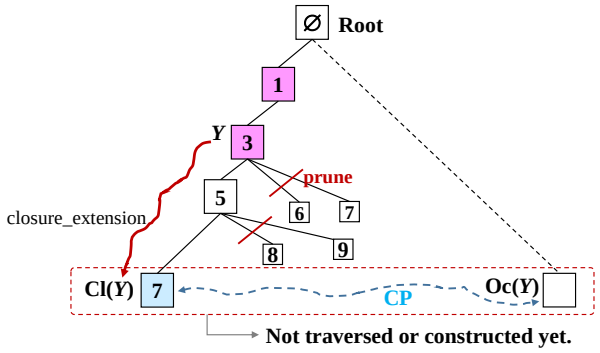

Fig. 9. Situation to which Pruning-5 is Applicable.

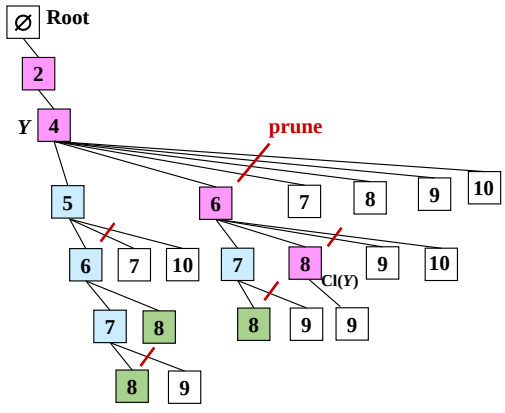

Fig. 10. More Examples for Pruning-5.

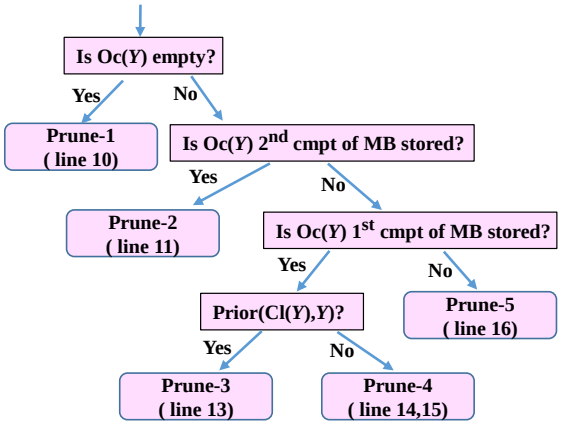

Fig. 11. Decision Tree to Decide Processing for *Y*.

In GenMB, a decision tree shown in Fig. 11 is used to select a case for *Y*. This *Y* will be assigned to one of 5 cases at the leaf in the decision tree. It is necessary to verify that there is no loss of MB after performing actions in any case.

We verified above that the pruning actions for each case are safe and thus no MB is lost from being generated. So the theorem holds. Q.E.D.

VI. Performance Evaluation

In this section, we determine computational complexity of our algorithm GenMB. The result of experimental comparison with current state of the art algorithm is also provided. Let $n = |V|$, and $m = |E|$. Let k be the total number of MBs in the given graph.

A. Theoretical Performance Evaluation

Our algorithm uses a recursive function named GenMB. Let $T_{GenMB}(X)$ denote the amount of time required for executing function GenMB when it is invoked with *X* passed to the first input parameter. The time complexity of our algorithm, $T(n, m)$, will be equal to that of $T_{GenMB}(\emptyset)$ which is the time taken by the initial invocation GenMB(∅, *V*, *V*, 0) to finish. An analytical solution for $T_{GenMB}(\emptyset)$ is hard to obtain since systematic progression cannot be formulated for the case of GenMB.

GenMB is invoked in two places in its procedure as follows: (i) line 13 (*X* receives a second component of an MB generated already), and (ii) line 14 (*X* will be a first component of an MB immediately by closure extension.) For each MB, GenMB is called twice (when DFS is visiting its two components). Thus the number of invocations of GenMB is equal to 2*k*. We first compute the time taken by one instance of GenMB, $T_{instance}$, which does not include the time to wait for the return from the recursive call issued during the run of the instance. Then $T(n, m)$ can be obtained from the result of multiplying $T_{instance}$ and 2*k*.

Obtaining $T_{instance}$ requires to compute the time taken by each step of the function. It takes $O(m)$ to perform procedure Closure_extension on line 2, which is shown by Theorem 11. Storing an MB involves storing two components in the MB tree and connect them by a CP pointer. By Theorem 12, it takes $O(n)$ time to store an MB (on line 3).

To represent a vertex set, say X, a bit-sequence array can be used for fast processing. Let A be the bit-sequence array used to represent X. It has n elements. If $v \in X$, $A[v] = 1$. Otherwise, $A[v] = 0$. It takes a constant time to check if a vertex v is in X, since it is needed just to check the value of $A[v]$.

Theorem 11: It takes O(m) time for executing Closure_extension (on line 2).

Proof: Let us consider function Closure_extension. At step 1, adjacency lists of vertices in Oc(Y) are scanned. We prepare an array $Ar[1..n]$ of n integers (initialized to 0) for storing counts of vertices. $Ar[i]$ has the count of vertex i encountered during scanning. During scanning, if an element with value of v is encountered, $Ar[v]$ is incremented. It takes O(m) time for this scanning and counting since the number of elements in adjacency lists is 2m.

Y and Tail(Y) are represented by bit-sequence arrays. Thus step 4 takes O(1) time. Since $|\text{Tail}(Y)| \leq n$, step 3 and 4 of Closure_extension take $O(n)$ time. Thus time complexity of Closure_extension is O($m + n$). Since m is usually much bigger than n, it can be said that Closure_extension takes O(m) time. Q.E.D.

Theorem 12: It takes O(n) to store an MB in the MB tree.

Proof: Let X be a component of an MB to be stored in the MB tree. Note that the MB tree is a subgraph of the SE tree. Only the nodes in the paths for components of MBs generated are actually constructed. Let $X = \{a_1, a_2, \ldots, a_r\}$.

Assume that the prefix $(a_1, \ldots, a_{j-1})$ of the path for X already exists in the MB tree and a_{j-1} does not have a child with label a_j. Let a pointer P point to the root node ∅ of the MB tree at first. Locate a child of P whose label is a_1. This takes a_1 operations at most since node ∅ may have children with all labels from 1 to a_1. Let P point to a_1 node.

Among children of a_1, a node with label a_2 should be identified. This may take $(a_2 - a_1)$ operations at most since the node with label a_1 may have children with labels from $a_1 + 1$, ... , a_2. Let P point to a_2 node. By proceeding in this way, P will point to a node of label a_{j-1} eventually. Then searching for node with a_j among children of node with a_{j-1} will be tried but fail, which takes $(a_j - a_{j-1})$ operations at most.

A node with label a_j should be created and attached to node a_{j-1}. For each element in the sequence $(a_{j+1}, \ldots, a_r)$, a node is created and attached to its predecessor's node. So the number of operations required is $a_1 + (a_2 - a_1) + (a_3 - a_2) + \cdots + (a_j - a_{j-1}) + (r - j) = a_j + r - j$. All a_j, r, j is upper-bounded by n. It takes O(n) to store a component of an MB. Creating a CP link takes O(1). Storing an MB involves storing two components and creating a CP pointer. Thus it takes O(n) to store an MB. Q.E.D.

It takes O(n) time to execute line 5 since scanning Tail(X) can be done in O(n) time. Theorem 13 shows that it takes O(n) to compute Oc(Y) from Oc(X) on line 6.

Theorem 13: Let $Y = X \cup \{v\}$ and $v \in V$. Oc(X) is given. It takes O(n) to compute Oc(Y).

Proof: $\text{Oc}(Y) = \text{Oc}(X) \cap L(v)$ where the adjacency list $L(v) = \{u \in V \mid u \text{ is adjacent to } v\}$. Since Oc(X) and L(v) are ordered lists, intersection of them can be done in this way. We use two pointers p1 and p2 to point to elements of Oc(X) and L(v), respectively. Initially they are made to point to the leftmost element of their set. The next loop is repeated until either p1 or p2 falls off the end of their set:

- Use p2 to scan L(v) left to right to find a next element which is not less than the element of p1,
- If the elements of p1 and p2 are the same, the element of p1 is added to the intersection result,
- p1 is made to point to the next element of Oc(X).

Thus computing intersection can be done in $|\text{Oc}(X)| + |L(v)|$ steps. Since n is the upper bound of $|\text{Oc}(X)|$ and $|L(v)|$, Oc(Y) can be computed in O(n) time. Q.E.D.

It takes O(1) to carry out the empty-set test on line 7. Theorem 14 shows that it takes O(n) for the look-up operations on lines 8 and 9.

Theorem 14: It takes O(n) time to look up a component of an MB stored already.

Proof: It is needed to find a path corresponding to Y in the MB tree. Let $Y = \{a1, \ldots, ar\}$. It is certain that $ar \leq n$. As assumed before, Y is ordered. To locate the nodes for all vertices in Y, the number of nodes to be probed depends only on the final element ar because of the characteristics of the SE tree.

For example, let $Y = \{2, 5, 8\}$. The path for Y can be found in this way: we try to find a node with 2 among children of the root, which may require scanning 2 nodes at most (labels 1 and 2); then we try to find a node with label 5 among children of 2 which may require scanning of $(5 - 2)$ nodes at most (with labels 3, 4, 5); we try to find a node with 8 among children of 5, which may require scanning of $(8 - 5)$ nodes at most. In total, the maximum number of operations required is $2 + (5 - 2) + (8 - 5) = 8$.

The maximum number of nodes to scan is equal to a_r. Thus the number of operations required is of the order of a_r. Note that $a_r \leq n$. Thus O(n) time is taken at most to locate a node of a vertex set. Finally the node with a_r is checked if it has a CP pointer, which takes O(1) time. Q.E.D.

It takes O(1) time to obtain Cl(Y) on line 10 via CP pointer in the node of Oc(Y). It takes O(n) to test Prior(Cl(Y), Y) since the two lists are scanned using two pointers to find the first position at which the elements do not match. Thus it takes O(n) to execute line 11. Theorem 15 shows that it takes O(n) to do extension and update on line 12.

Theorem 15: It takes O(n) to carry out extension and update operations on line 12.

Proof: The operations to perform is step i of Procedure for Pruning-4. The bit-sequence array representations introduced above are used to implement vertex sets Y, Tail(Y) and Cl(Y). Thus it takes O(1) to test if $v \in$ Cl(Y).

TABLE. I. EXPERIMENTAL RESULTS

Input graphs		R_1	R_2	R_3	B_1	B_2
graphspec.	n	100	100	300	992	3,890
	m	496	2,476	8,971	1,138	7,729
	k	721	696,000	2.38×10^6	52	5,165
Exec. time (sec)	Li [4]	0.043	269	3,130	1.48	186
	Ours	0.036	99.8	1,474	1.47	65

Adding v to Y takes O(1) since it can be done by setting the corresponding element of the array of Y to 1. Similarly, removing v from Tail(Y) takes O(1). The number of iterations of the loop is not more than |Tail(Y)|. Thus it takes O(n) to execute the loop. Q.E.D.

The amount of time taken to invoke a function on line 13 and 14 requires O(1) time (since the time to wait for the return from the called function is not included).

By adding the amounts of time for lines from 5 to 14, it is found out that one iteration of the loop of line 4 requires O(n) time. This loop iterates |Tail(X)| number of times. |Tail(X)| ≤ n. It takes O(1) time for line 1, O(m+n) for line 2 and O(n) for line 3. Therefore, $T_{instance} = c_1n + c_2m + c_3n^2$ for some constants c_1, c_2 and c_3. By multiplying 2k and $T_{instance}$, we obtain time complexity of our algorithm which is $T(n) = O(kn^2)$.

The number of nodes used to store a component of an MB is less than or equal to n. Thus O(n) space is required for storing an MB. So O(kn) space is required to store all MBs. Each active instance of GenMB uses storage for the variables such as X, Oc(X), Y, Oc(Y), Tail(X), Tail(Y), Cl(Y). However, each of them requires O(n) space. The maximum number of instances of GenMB existing in memory at the same time is equal to the height of the SE tree. Thus it is n. So it needs $O(n^2)$ space to store variables used by all active instances of GenMB. Therefore, it requires $O(kn+n^2)$ space by our algorithm. It can be approximated to O(kn) because usually n is much smaller than k.

As a conclusion, we obtain $T(n) = O(kn^2)$ and $S(n) = O(kn)$ as time and space complexity of our algorithm, respectively.

B. Empirical Performance Evaluation

We performed experimentations to confirm the time complexity analysis results of our algorithm. We implemented our algorithm and measured its speed. The current state of the art algorithm is that of Li et al. [4]. This algorithm was implemented to compare its efficiency with ours. We compared the amounts of time required by the two algorithms as shown in Table I. We used five graphs to test the algorithms.

Graphs were generated randomly (marked with R's in Table I) to be used as input. In addition to these artificial graphs, real life protein interaction networks were obtained from the biological repository, BioGrid (marked with B's in Table I) and used as input to test the algorithms [16]. The experimental result shows that our algorithm takes less amount of time than that of Li et al. [4]. This conforms to the theoretical analysis of our algorithm given in the previous subsection. However, if the total number k of MBs is small, efficiency of our algorithm is not manifested fully.

C. Discussions

As far as theoretical time complexity is concerned, our algorithm is superior to any fully general algorithms. The algorithm of Li et al. [4] has been state of the art for more than a decade and a half. Eventually we propose a new state of the art algorithm described in this paper. Another advantage of our algorithm is that a lot of space can be saved by storing all MBs by using paths in a set enumeration tree.

VII. CONCLUSION

In this paper, a new efficient algorithm is proposed for mining all maximal bicliques in an arbitrary undirected graph with n vertices, m edges and k maximal bicliques. The time complexity of ours is $O(kn^2)$ which is a significant improvement over O(kmn) the current state of the art performance [4]. This improvement is made possible by pruning search space extensively in our method. To be able to apply pruning techniques, maximal bicliques are stored as soon as they are discovered. They are looked up to make pruning decisions. Our algorithm requires O(kn) space which is used for storing all MBs. If the MBs need to be loaded into memory after generation to be available for application tasks, any algorithm cannot but require O(kn) space. Because the paths for components of maximal bicliques share a lot of nodes, the actual amount of storage used by our algorithm is less than that expected by theoretical analysis.

Nowadays the networks appearing in the fields of social networks and protein networks have a huge size. Parallelizing the MB-mining algorithms is vital to achieve practical systems [17]. This topic is included in our near future research.

REFERENCES

[1] A. Z. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. L. Wiener, "Graph structure in the web," Computer Networks, 33(1-6) pp. 309–320, June 2000.

[2] D. Bu, Y. Zhao, L. Cai, H. Xue, X. Zhu, H. Lu, J. Zhang, S. Sun, L. Ling, N. Zhang, G. Li, R. Chen, "Topological structure analysis of the protein interaction network in budding yeast," Nucleic Acids Research, vol. 31, no. 9, pp. 2443–2450, May 2003.

[3] C. G. Akcora, M. F. Dixon, Y. R. Gel, M. Kantarcioglu, "Bitcoin risk modeling with blockchain graphs," Economics Letters, vol. 173 pp. 138-142, Dec. 2018.

[4] J. Li, G. Liu, H. Li, L. Wong, "Maximal biclique subgraphs and closed pattern pairs of the adjacency matrix: a one-to-one correspondence and mining algorithms," IEEE Trans. on Knowledge and Data Engineering, vol. 19, no. 12, pp. 1625–1637, Dec. 2007.

[5] M. J. Zaki, C. Hsiao, "Charm: An efficient algorithm for closed itemset mining," In Proceedings of 2nd SIAM International Conference on Data Mining, Arlington, Virginia, pp. 398–416, April 2002.

[6] M. J. Sanderson, A. C. Driskell, R. H. Ree, O. Eulenstein, S. Langley, "Obtaining maximal concatenated phylogenetic data sets from large sequence databases," Molecular Biology Evol., vol. 20, no. 7, pp. 1036–1042, May 2003.

[7] K. Makino, T. Uno, "New algorithms for enumerating all maximal cliques," In Proceedings of 9th Scandinavian Workshop on Algorithm Theory (SWAT 2004), Springer-Verlag, pp. 260-272, July 2004.

[8] Y. Zhang, C. A. Phillips, G. L. Rogers, E. J. Baker, E. J. Chesler, 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. 110, April 2014.

[9] V.M. Dias, C.M. de Figueiredo, J.L. Szwarcfiter, "Generating bicliques of a graph in lexicographic order," Theoretical Computer Science, vol. 337, pp. 240-248, June 2005.

[10] K. Kloster, A. van der Poel, B. D. Sullivan, " Mining Maximal Induced Bicliques using Odd Cycle Transversals," In Proceedings of the 2019 SIAM International Conference on Data Mining, pp. 324-333, 2019.

[11] B. D. Sullivan, A. van der Poel, T. Woodlief, "Faster biclique mining in near-bipartite graphs," Analysis of Experimental Algorithms, Springer International Publishing, pp 424-453 , Nov. 2019.

[12] G. Liu, K. Sim, J. Li, "Efficient mining of large maximal bicliques," In Proceedings of the 8th international conference on Data Warehousing and Knowledge Discovery, pp. 437-448, Sep. 2006.

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

[14] E. Tomita, A. Tanaka, H. Takahashi, "The worst-case time complexity for generating all maximal cliques and computational experiments," Theoretical Computer Science, vol. 363 pp. 28–42, Oct. 2006.

[15] R. Rymon, "Search through systematic set enumeration," In Proceedings of 3rd International Conference on Principles of Knowledge Representation and Reasoning, Cambridge, MA, pp. 539-590, Oct. 1992.

[16] C. Stark, B. J. Breitkreutz, T. Reguly, L. Boucher, A. Breitkreutz, M. Tyers, "Biological general repository for interaction datasets (BioGRID)," http://thebiogrid.org/download.php.

[17] A. P. Mukherjee, S. Tirthapura, "Enumerating maximal bicliques from a large graph using mapreduce," IEEE Transactions on Services Computing, vol. 10 , no. 5, pp. 771-784 , May 2017.