Finding Maximal Cliques in Massive Networks

JAMES CHENG, Nanyang Technological University
YIPING KE, ADA WAI-CHEE FU, and JEFFREY XU YU, The Chinese University of Hong Kong
LINHONG ZHU, Institute for Infocomm Research

Maximal clique enumeration is a fundamental problem in graph theory and has important applications in many areas such as social network analysis and bioinformatics. The problem is extensively studied; however, the best existing algorithms require memory space linear in the size of the input graph. This has become a serious concern in view of the massive volume of today's fast-growing networks. We propose a general framework for designing external-memory algorithms for maximal clique enumeration in large graphs. The general framework enables maximal clique enumeration to be processed recursively in small subgraphs of the input graph, thus allowing in-memory computation of maximal cliques without the costly random disk access. We prove that the set of cliques obtained by the recursive local computation is both correct (i.e., globally maximal) and complete. The subgraph to be processed each time is defined based on a set of *base vertices* that can be flexibly chosen to achieve different purposes. We discuss the selection of the base vertices to fully utilize the available memory in order to minimize I/O cost in static graphs, and for update maintenance in dynamic graphs. We also apply our framework to design an external-memory algorithm for maximum clique computation in a large graph.

Categories and Subject Descriptors: G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Maximal clique enumeration, massive networks, scale-free networks, dynamic graphs, H*-graph, h-index

ACM Reference Format:
Cheng, J., Ke, Y., Fu, A. W.-C., Yu, J. X., and Zhu, L. 2011. Finding maximal cliques in massive networks. ACM Trans. Datab. Syst. 36, 4, Article 21 (December 2011), 34 pages.
DOI = 10.1145/2043652.2043654 http://doi.acm.org/10.1145/2043652.2043654

1. INTRODUCTION

Maximal clique enumeration [Akkoyunlu 1973; Bron and Kerbosch 1973] is a long-standing problem in graph theory. It is closely related to a number of fundamental graph problems, such as maximal independent sets (or minimal vertex covers) [Tsukiyama et al. 1977], graph coloring [Byskov 2003], maximal common induced subgraphs [Koch 2001], maximal common edge subgraphs [Koch 2001], etc. Its significance is not just limited to graph theory but also in numerous applications in various real-world networks, such as social network analysis [Faust and Wasserman 1995], hierarchy detection through email networks [Creamer et al. 2007], study of structures in

This work is supported in part by the AcRF Tier-1 Grant (M52020092) from Ministry of Education of Singapore, RGC Direct Grant for Research 2050483, and RGC CUHK No. 419008.
Authors' addresses: J. Cheng, School of Computer Engineering, Nanyang Technological University, Singapore; Y. Ke (corresponding author), A. W.-C. Fu and J. X. Yu, The Chinese University of Hong Kong, New Territories, Hong Kong; email: ypke@se.cuhk.edu.hk; L. Zhu, Data Mining Department, Institute for Infocomm Research, Singapore.

DOI 10.1145/2043652.2043654 http://doi.acm.org/10.1145/2043652.2043654

behavioral and cognitive networks [Bernard et al. 1979], statistical analysis of financial networks [Boginski et al. 2005], clustering in dynamic networks [Stix 2004], the detection of emergent patterns in terrorist networks [Berry et al. 2004], as well as various applications in computational biology [Abu-Khzam et al. 2005], including the detection of protein-protein interaction complex [Zhang et al. 2008] and clustering protein sequences [Mohseni-Zadeh et al. 2004].

Algorithms for maximal clique enumeration have been extensively studied over the decades. The optimal worst-case time complexity of *in-memory algorithms* for general graphs is shown to be $\mathcal{O}(3^{n/3})$ recently by Tomita et al. [2006], where n is the number of vertices in the input graph. The worst-case time complexity is derived based on the number of maximal cliques in a general graph in the worst case. In practice, however, the number of maximal cliques in real graphs is significantly smaller than the theoretical worst-case bound, which is especially true for those massive real-world networks that are sparse. Therefore, despite that the problem is NP-hard theoretically, many algorithms [Akkoyunlu 1973; Bron and Kerbosch 1973; Tsukiyama et al. 1977; Kose et al. 2001; Gouda and Zaki 2001; Koch 2001; Makino and Uno 2004; Stix 2004; Abu-Khzam et al. 2005; Tomita et al. 2006; Wan et al. 2006; Cazals and Karande 2008; Modani and Dey 2008; Du et al. 2009; Schmidt et al. 2009] have been proposed in the past to solve this problem and these algorithms are shown to be fast running in real-world graphs. Many of these proposed algorithms have also been applied to solve other practical problems in different application domains, such as the ones we listed earlier [Bernard et al. 1979; Faust and Wasserman 1995; Koch 2001; Berry et al. 2004; Mohseni-Zadeh et al. 2004; Stix 2004; Abu-Khzam et al. 2005; Boginski et al. 2005; Creamer et al. 2007; Zhang et al. 2008].

Although many practical algorithms have been proposed, these existing algorithms all fall into the category of in-memory algorithms. The best existing in-memory algorithms require space that is asymptotically linear in the size of the input graph. Unfortunately, many real-world networks have grown exceedingly large in recent years and are continuing to grow at a steady rate. For example, the Web graph has over 1 trillion Web pages (by Google in 2008), most social networks (e.g., Facebook, MSN) have millions to billions of users, many citation networks (e.g., DBLP, Citeseer) have millions of publications, other networks such as phone-call networks, email networks, stock-market networks, etc., are also massively large.

For processing such large graphs, *external-memory algorithms* offer a possible recourse; however, designing such an algorithm is fraught with difficulties. Maximal clique computation accesses vertices in a rather arbitrary manner. This potential random access requirement makes it difficult to divide the graph and process it in main memory in a part-by-part manner and perhaps suggests the reason for the current prevalence in in-memory algorithms for tackling this problem.

In this article, we develop an *E*xternal-*M*emory algorithm for *M*aximal *C*lique *E*numeration, called **EmMCE**. Our work focuses on the broad class of sparse graphs and in particular, *scale-free* graphs [Newman 2003; Dorogovtsev and Mendesand 2003], whose degree distribution follows a power law.

Since maximal clique computation requires random access to different parts of a large graph, it is a great challenge to divide the graph into smaller parts and process one part at a time, because either the result may be incorrect and incomplete, or it incurs huge cost on merging the results from different parts. To this end, we propose the notion of B^**-graph*, which is a special subgraph that enables the computation of maximal cliques locally while ensuring the global correctness of the result computed. Our algorithm EmMCE recursively constructs the B^*-graph and computes the maximal cliques from the B^*-graph. We prove both the correctness and the completeness of the result computed by EmMCE. We further introduce a special version of B^*-graph, which

is applied for efficient update maintenance of maximal cliques in dynamic graphs as well as for computing the maximum clique from a large graph.

Extensive experiments verify that our algorithm efficiently enumerates maximal cliques in large networks that are too expensive to be processed by the existing in-memory algorithms, while achieving comparable performance as an in-memory algorithm when memory is sufficient. The results also show that our algorithm is efficient for update maintenance in dynamic networks. Finally, we show that our external-memory algorithm for maximum clique computation is significantly more efficient than its in-memory counterpart for both small and large networks.

Organization. The remaining part of this article is organized as follows. Section 2 formally defines the problem and gives the basic notations. Section 3 describes the framework of an in-memory algorithm and our solution framework. Section 4 introduces the B^*-graph. Section 5 discusses in details the computation of the global maximal cliques from the B^*-graph. Section 6 gives the overall algorithm. Section 7 discusses the selection of the base vertex-set B. Section 8 discusses the update maintenance in dynamic networks. Section 9 presents the algorithm for maximum clique computation. Section 10 reports the experimental results. Section 11 discusses the related work. Section 12 concludes.

2. NOTATIONS AND PROBLEM DEFINITION

Let $G = (V, E)$ be an undirected and unlabeled graph. We define $n = |V|$ and $m = |E|$. Let b is the disk block size. In this article, we focus on sparse graphs, where $m < (b \cdot n)$. For large sparse graphs, random disk access is prohibitively expensive since data transfer from/to disk is in blocks, while the exact amount of data used for maximal clique computation is much less than the amount of data transferred for each random access.

We define the *size* of G, denoted as $|G|$, as $|G| = m$. Given a subset of vertices $S \subseteq V$, we define the *induced subgraph* of G by S as $G_S = (V_S = S, E_S = \{(u, v) : u, v \in S, (u, v) \in E\})$. We define the set of *neighbors* of a vertex v in G as $nb(v) = \{u : (u, v) \in E\}$, and the *degree* of v in G as $d(v) = |nb(v)|$. Similarly, we define $nb(v, G_S) = \{u : (u, v) \in E_S\}$ and $d(v, G_S) = |nb(v, G_S)|$.

A *clique* in G is a subset of vertices, $C \subseteq V$, such that the induced subgraph of G by C is a complete graph. C is called a *maximal clique* (*max-clique* for short) in G if there exists no clique C' in G such that $C' \supset C$.

The problem of Maximal Clique Enumeration (MCE) is

given a graph G, find the set of all maximal cliques in G.

In this article, we solve the problem of MCE for large scale-free graphs, which is defined as

given a large scale-free graph G, find the set of all maximal cliques in G with bounded memory usage.

Table I shows the notations used throughout the article.

3. IN-MEMORY ALGORITHM AND OUR SOLUTION FRAMEWORK

We first briefly discuss a general framework of existing in-memory algorithms for max-clique enumeration, explain the problem and challenge when the input graph is too large to fit in main memory, and then outline the framework of our solution.

3.1. In-Memory Algorithm Framework

Algorithm 1 sketches an in-memory algorithm for max-clique enumeration. The algorithm starts from a single vertex $v \in V$ and grows into larger cliques by checking

Table I. Notations

Symbol	Description
n	Number of vertices in a graph $G = (V, E)$
m	Number of edges in a graph $G = (V, E)$
b	Disk block size
$\|G\|$	Size of G, defined as $\|G\| = \|E\| = m$
G_S	Induced subgraph of G by a set of vertices S
$nb(v)$; $nb(v, G_S)$	The set of neighbors of a vertex v in G / G_S
$d(v)$; $d(v, G_S)$	The degree of v in G / G_S
$\mathcal{M}$	The set of maximal cliques in the whole graph G
B	Base vertex-set; a set of base vertices
B_{nb}	Neighbor vertex-set of B; $B_{nb} = \{v : v \in (nb(u) \setminus B), u \in B\}$
B^+	Extension vertex-set of B; $B^+ = B \cup B_{nb}$
G_B ; G_{B^+}	B-graph / B^+-graph; the induced subgraph of G by B / B^+
G_{B^*}	B^*-graph; $G_{B^*} = (B^+, E_{BB} \cup E_{BB_{nb}})$
$\mathcal{M}_X$	The set of X-max-cliques in G_X, X can be B^*, B^+, or B
T_{B^*}	B^*-max-clique tree; a prefix-tree to keep $\mathcal{M}_{B^*}$
C=($C_B \cup C_{B_{nb}}$)	For a clique C in G_{B^+}: C_B=($C \cap B$); $C_{B_{nb}}$=($C \cap B_{nb}$)
$comNB(X)$	The set of common B-neighbors of the vertices in X, where X is a clique in G_B
$maxCL(S)$	The set of all max-cliques in G_S
$\mathcal{M}_1$, $\mathcal{M}_2$, $\mathcal{M}_3$	Three disjoint subsets of $\mathcal{M}_{B^+}$, defined in Lemmas 5.6-5.10
$\mathcal{X}$	A set of "B" parts used to form cliques in $\mathcal{M}_3$, see Eq. (1)
$EXT(C)$	A set of B-neighbors used to extend $C \in \mathcal{X}$, see Eq. (2)
h_{max}	The h-index defined for a graph G
H	The set of H-vertices of G; $\forall v \in H, d(v) \geq h_{max}$

ALGORITHM 1: *ImMCE*

Input: a graph $G = (V, E)$
Output: the set of max-cliques, $\mathcal{M}$, in G

1. $\mathcal{M} \leftarrow \emptyset$;
2. **for each** vertex $v \in V$ **do**
3. $\quad C \leftarrow \{v\}$;
4. $\quad S \leftarrow \{u : u \in nb(v), u \succ v\}$; /* $u \succ v$: u is ordered after v */
5. $\quad$ *ImMCEstep*(C, S);
6. **return** $\mathcal{M}$;

whether v's neighbors are interconnected. In line 4 we consider only a subset S of $nb(v)$, which are the neighbors of v that are ordered after v. This is because the neighbors that are ordered before v have already been processed (we assume that in line 2 the vertices in V are processed by an order, i.e., v is processed before u if $u \succ v$). Then, the algorithm invokes Procedure 2 to grow C, which is initialized as $\{v\}$, by checking the interconnection of the vertices in S.

Procedure 2 first tests if S is empty. An empty S indicates that C cannot grow bigger; we then check if C is indeed maximal before we include it in $\mathcal{M}$. If S is not empty, the algorithm (line 4) first checks if $(C \cup S)$, which is the largest clique that C can potentially grow to, is maximal. The algorithm continues only if $(C \cup S)$ is maximal, and it grows C by considering every vertex $u \in S$. The algorithm obtains S_u, a subset of vertices in S that are u's neighbors (line 6). Essentially, S_u contains the common neighbors of the vertices in $(C \cup \{u\})$. Then, the algorithm recursively invokes Procedure 2 to grow $C = (C \cup \{u\})$ by checking the interconnection of the vertices in S_u, until C can no longer grow.

Algorithm 1 in effect constructs a search tree. Most existing algorithms [Bron and Kerbosch 1973; Gouda and Zaki 2001; Koch 2001; Tomita et al. 2006; Cazals and

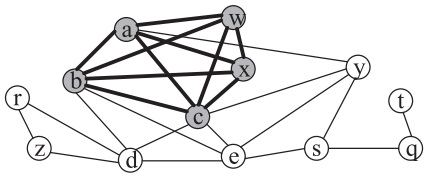


Fig. 1. An example of random access to vertices.

PROCEDURE 2: *ImMCEstep*(C, S)

1. **if**$(S = \emptyset)$
2. **if**$(C$ is maximal)
3. $\mathcal{M} \leftarrow \mathcal{M} \cup \{C\}$;
4. **else if**$(C \cup S$ is maximal)
5. **for each** $u \in S$ **do**
6. $S_u \leftarrow (S \cap nb(u))$;
7. $C \leftarrow (C \cup \{u\})$;
8. *ImMCEstep*(C, S_u);
9. $C \leftarrow (C \backslash \{u\}))$;

Karande 2008; Modani and Dey 2008; Du et al. 2009; Schmidt et al. 2009] for computing max-cliques essentially construct a search tree in a similar way as does Algorithm 1, although different algorithms have their own ways of pruning unnecessary searches and performing the maximality checking (lines 2 and 4 of Procedure 2).

As we can see in lines 5–6 of Procedure 2, the in-memory algorithm requires access to the neighbor sets of the vertices in S (i.e., $nb(u)$), which may scatter in different locations in the graph. When the input graph is too large to fit in main memory, this random access of vertices can lead to extremely high I/O cost and severely degrade the performance of the in-memory algorithm. We illustrate this problem using the following example.

Example 3.1. Figure 1 gives an example graph G which contains 13 vertices and 25 edges. Assume that the vertices are ordered in the alphabetic order. Then, according to Algorithm 1, the first max-clique to be enumerated is $C = \{a, b, c, w, x\}$. During the enumeration, the algorithm needs to access $nb(a)$, $nb(b)$, $nb(c)$, $nb(w)$ and $nb(x)$. If the graph cannot fit in main memory and these neighbor sets are stored in different blocks on disk, the algorithm requires many random accesses to retrieve the neighbor sets into main memory. For example, if the graph is stored in its adjacency list representation and the adjacency lists are stored on disk in the order of the vertices, then at least $nb(w)$ and $nb(x)$ are not stored sequentially with $nb(a)$, $nb(b)$, and $nb(c)$, and random disk access is needed assuming that they are in different disk blocks. In a large graph, vertices may have edge connection with vertices in arbitrary locations in the graph. In this situation, the amount of random access in the entire process of max-clique enumeration is huge.

3.2. Our Solution Framework

To avoid random access to arbitrary vertices in the graph, we design an algorithm that recursively computes max-cliques in a subgraph of G that fits in main memory. We outline the framework of our algorithm as follows.

—Each recursive step:
(1) Extract a subgraph G' from G, such that $|G'| < M$ (M is memory size);
(2) Compute the set of *local* max-cliques in G';

(3) Obtain and output a subset of *global* max-cliques from the local max-cliques by linking G' to G;
(4) Remove G' from G.

—Repeat the recursive step until G becomes empty.

The main idea of our algorithm is to recursively divide the graph and compute the max-cliques in each local subgraph G' separately. The concept is simple but there are significant challenges in choosing an appropriate subgraph G' and linking the computation from G' to the other part of G, while ensuring the correctness and completeness of the final result, and at the same time achieving low I/O complexity.

4. SEMI-EXTENSION SUBGRAPH

In this section, we introduce the concept of semi-extension subgraph, which serves as the local subgraph G' processed at each recursive step. This concept plays an essential role in enabling max-clique computation with bounded memory usage.

We start by defining a set of *base vertices*, namely the *base vertex-set*.

Definition 4.1 (*Base Vertex-Set*). Given a graph $G = (V, E)$, the *base vertex-set*, denoted by B, is a subset of vertices selected from V, that is, $B \subseteq V$. The vertices in B are called *base vertices*.

The base vertices are used in our algorithm as the *seeds* for max-clique enumeration at each recursive step. A set of base vertices are first selected at each step. Based on these seeds, we then obtain a subgraph, from which all max-cliques consisting of the seeds are computed. In this way, we can recursively select disjoint sets of base vertices for max-clique enumeration, until all vertices in V are selected. Thus, the use of the base vertex-sets not only enables the local computation of max-cliques in main memory, but also ensures the completeness of the final global result.

The base vertex-set B can be randomly selected from V or selected according to a certain criterion. We delay the discussion on the selection of B to Section 7. For now, let us assume that we are given a base vertex-set B. We now discuss how to obtain a local subgraph from B for max-clique enumeration. We first define the neighbor and extension of a base vertex-set.

Definition 4.2 (*Neighbor and Extension Vertex-Sets*). Given a graph $G = (V, E)$ and a base vertex-set B, we define the following two types of vertex-sets.

—The *neighbor vertex-set* of B, denoted by B_{nb}, is defined as $B_{nb} = \{v : v \in (nb(u) \backslash B), u \in B\}$.
—The *extension vertex-set* of B, denoted by B^+, is defined as $B^+ = B \cup B_{nb}$.

We also call a base vertex $u \in B$ a *B-vertex*, and a vertex $v \in B_{nb}$ a *neighbor vertex* or a *B-neighbor*.

Intuitively, in Definition 4.2 we have a base vertex-set B, from which we extend to their neighbors B_{nb}. Then, we obtain B^+ as the union of B and B_{nb}, where we use the "+" sign to indicate the extension from the B-vertices to the B-neighbors.

With these vertex-sets, we define their corresponding subgraphs as follows.

Definition 4.3 (*Base, Extension, and Semi-Extension Subgraphs*). Given a graph $G = (V, E)$ and a base vertex-set B, we define the following three types of subgraphs.

—The *base subgraph*, denoted by G_B, is defined as the induced subgraph of G by B. We also call G_B the *B-graph*.

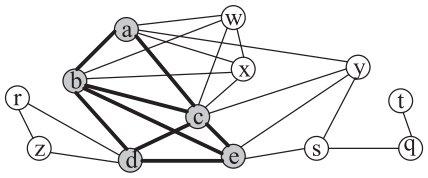


Fig. 2. A semi-extension subgraph and related concepts.

—The *extension subgraph*, denoted by G_{B^+}, is defined as the induced subgraph of G by B^+. We also call G_{B^+} the B^+*-graph*.
—The *semi-extension subgraph*, denoted by G_{B^*}, is defined as $G_{B^*} = (B^+, E_{BB} \cup E_{BB_{nb}})$, where $E_{BB} = \{(u, v) : u, v \in B, (u, v) \in E\}$ and $E_{BB_{nb}} = \{(u, v) : u \in B, v \in B_{nb}, (u, v) \in E\}$. We also call G_{B^*} the B^**-graph*.

Intuitively, the B^*-graph is a graph that "lies" between the B-graph and the B^+-graph. The B^*-graph is the same as the B^+-graph except that the B^*-graph does not contain the edges between the B-neighbors. In other words, the B^*-graph contains only those edges incident to at least one B-vertex. It is easy to see that $G_B \subseteq G_{B^*} \subseteq G_{B^+}$. The first equality holds when $B_{nb} = \emptyset$ and the second equality holds when there is no edge between the B-neighbors in G.

We use the following example to illustrate these basic concepts.

Example 4.4. Figure 2 gives an example graph G, which contains 13 vertices and 25 edges. Let $B = \{a, b, c, d, e\}$ be the set of B-vertices selected from G. The set of B-neighbors is thus $B_{nb} = \{r, s, w, x, y, z\}$, and we have $B^+ = \{a, b, c, d, e, r, s, w, x, y, z\}$. The two vertices q and t are not in B^+ since they are not incident to any vertex in B. The B-graph consists of the shaded vertices and bold edges in Figure 2, which is the induced subgraph of G by B. The B^+-graph contains all edges in G except for the two edges incident to q and t. Finally, the B^*-graph contains all edges in the B^+-graph except for the edges between the B-neighbors, that is, (w, x), (s, y), and (r, z).

We have just introduced three types of subgraphs of a graph G with respect to a base vertex-set B. In our work, we use the B^*-graph as the local subgraph processed at each recursive step for max-clique enumeration. In the following, we analyze the suitability of each of the three types of subgraphs for max-clique enumeration and explain why we choose the B^*-graph.

4.1. Why Semi-Extension Subgraph?

First of all, we find that the B-graph G_B is not suitable to be used for max-clique enumeration at each recursive step of our algorithm. The main problem with G_B is that it only gives the edge connection among the vertices in B. However, the vertices in B may form cliques with vertices in other parts of G rather than in G_B alone. Thus, max-clique enumeration based on G_B, with respect to a chosen B, at each recursive step may not only miss a large number of max-cliques but also output false max-cliques, that is, nonmaximal cliques. For example, given G_B in Figure 2, we compute two max-cliques $\{a, b, c\}$ and $\{b, c, d, e\}$. However, $\{a, b, c\}$ is not maximal and we miss the real max-cliques $\{a, b, c, w, x\}$ and $\{a, c, y\}$ since the B-vertices are not to be revisited after we finish this step.

Next, we consider the B^+-graph G_{B^+}. We first define the notion of B^+-max-cliques as follows.

Definition 4.5 (B^+*-Max-Clique*). A B^+*-max-clique* is a max-clique in G_{B^+} that consists of at least one B-vertex. The set of all B^+-max-cliques is denoted by $\mathcal{M}_{B^+}$.

The following lemma states that a B^+-max-clique locally maximal in G_{B^+} is also globally maximal in G.

LEMMA 4.6. *A B^+-max-clique is also a max-clique in G.*

PROOF. Proof by contradiction. Let C be a B^+-max-clique and u be a B-vertex in C. Suppose that C is not maximal in G, that is, there exists a max-clique C' in G such that $C' \supset C$. Then, C' must contain some vertex v, where $v \notin B^+$ (otherwise C' must be maximal in G_{B^+} and C is not). However, $v \notin B^+$ implies that v is not connected with u, which contradicts that C' is a clique. Therefore, C must be also maximal in G. □

Unlike the case with the B-graph, Lemma 4.6 ensures that the B^+-max-cliques computed from the B^+-graph are real max-cliques in G, although we still need to deal with the completeness of the result. However, as we mentioned earlier, in this work we use the B^*-graph instead of the B^+-graph. There are a few disadvantages of using the B^+-graph related to both space and time efficiency as follows.

First, since the B^+-graph G_{B^+} keeps all connections among the vertices in B^+, it is often too large to be put in main memory, unless B is small. And if B is small, our algorithm needs many recursions to complete the max-clique enumeration since each recursive step can only process a small number of vertices. This would result in many scans of the graph G from disk, which is very time consuming and I/O inefficient.

Compared with the B^+-graph, the B^*-graph is much smaller and therefore we can cover a much larger base vertex-set each time. The size of G_{B^+} is bounded by $(\sum_{v \in B^+} d(v))$, while the size of G_{B^*} is bounded by $(\sum_{v \in B} d(v))$. If B is randomly selected from V, then the bound on $|G_{B^+}|$ can be estimated as $(|B|d_{avg}^2)$ while the bound on $|G_{B^*}|$ is only $(|B|d_{avg})$, where d_{avg} is the average degree of the vertices in G. For many real-world networks that are scale-free [Newman 2003; Dorogovtsev and Mendesand 2003], a more detailed analysis on $|G_{B^+}|$ and $|G_{B^*}|$ is given in Sections 7.2.2 and 7.2.3. In short, the size of G_{B^+} can be significantly larger than that of G_{B^*}.

Second, the B^+-graph cannot be safely removed from the input graph without jeopardizing the completeness of the final result. For example, removing the B^+-graph from the graph given in Example 4.4 will miss the max-clique $\{q, s\}$ from the final result. On the contrary, the B^*-graph can be safely removed from the input graph at each recursive step to save the I/O cost for future max-clique computation, without compromising the completeness of the final result. We discuss how to compute the max-cliques from the B^*-graph, as well as proving the correctness and completeness of the result, in Section 5.

Third, some edges in B^+-graph may not be relevant to the computation of max-cliques at the current recursive step. Thus, loading them into main memory incurs extra I/O cost, while it may also lead to unnecessary computation and waste both CPU and memory resources.

5. COMPUTATION OF MAXIMAL CLIQUES FROM A LOCAL SUBGRAPH

In this section, we discuss how we compute the set of maximal cliques from the extracted subgraph G_{B^*}, which is the key step to our external-memory algorithm.

5.1. Local Max-Cliques: the B*-Max-Cliques

We start by first defining the notion of B^*-max-cliques, which are the local max-cliques in the B^*-graph.

Definition 5.1 (*B^*-Max-Clique*). A *B^*-max-clique* is a max-clique in G_{B^*}. We denote the set of all B^*-max-cliques by $\mathcal{M}_{B^*}$.

The following lemma states two properties of B^*-max-cliques.

LEMMA 5.2. *The following statements of B^*-max-clique are true:*

(1) *A B^*-max-clique contains at least one B-vertex.*
(2) *A B^*-max-clique contains at most one B-neighbor.*

PROOF. Since each B-neighbor in G_{B^*} is connected to at least one B-vertex and there is no edge between any two B-neighbors in G_{B^*}, a B^*-max-clique containing a B-neighbor must also contain at least an incident B-vertex, which proves the first statement. Note that a single B-neighbor v cannot form a max-clique in G_{B^*} since there exists a larger clique that consists of v and the B-vertex(es) v is connected to. The second statement holds since there is no edge among the B-neighbors. □

We next present a data structure that we use to keep the set of B^*-max-cliques. Since two cliques may share common vertices, we define a prefix-tree structure to represent common vertices in the cliques as common paths.

Definition 5.3 (*B^*-max-clique tree*). Given the B^*-graph G_{B^*} of a graph G, let $\prec$ be a total order on B^+, where $\forall u \in B$ and $\forall v \in B_{nb}$, $u \prec v$. The *B^*-max-clique tree*, T_{B^*}, of G_{B^*} is a prefix tree defined as follows.

—The root of T_{B^*} is a virtual vertex λ, where $\forall v \in B^+$, $\lambda \prec v$.
—The children of a vertex in T_{B^*} are ordered by $\prec$.
—All vertices in a root-to-leaf path in T_{B^*} are ordered by $\prec$.
—The set of root-to-leaf paths in T_{B^*} has a one-to-one correspondence to the set of B^*-max-cliques. A root-to-leaf path $\langle \lambda, u, \ldots, v \rangle$ corresponds to a B^*-max-clique $\{u, \ldots, v\}$.

We define $\prec$ by simply assigning each vertex a unique ID and ordering them by their IDs, where the ID of a B-vertex is always smaller than that of a B-neighbor.

By Definition 5.3, we have the following lemma.

LEMMA 5.4. *The following statements of T_{B^*} are true:*

(1) *A B-neighbor can only be a leaf in T_{B^*}.*
(2) *All children of λ are B-vertices.*

PROOF. Lemma 5.2 states that a B^*-max-clique contains at most one B-neighbor. By the definition of the order $\prec$ and the tree T_{B^*}, a B-neighbor can only be a leaf in T_{B^*}.

Similarly, all children of λ are B-vertices since a B^*-max-clique contains at least one B-vertex as stated in Lemma 5.2 and all B-vertices are ordered before B-neighbors in a root-to-leaf path in T_{B^*}. □

Most existing in-memory algorithms for computing max-cliques can be modified with small overhead to compute B^*-max-cliques from G_{B^*} by constructing a T_{B^*}. For example, the search tree constructed by Algorithm 1 is almost a B^*-max-clique tree. We do not go into the details of these existing algorithms, but highlight two improvements that we can make by employing the unique properties of T_{B^*}.

Given a path $p = \langle \lambda, u, \ldots, v \rangle$ in T_{B^*}, let S be the set of vertices that can be used to potentially grow p from v. If $S \subseteq B_{nb}$, by Statement 1 of Lemma 5.4, we can directly create S as the set of children of v. Second, unlike a normal prefix tree or a backtracking search tree, by Statement 2 of Lemma 5.4, we only need to consider B-vertices when creating the children of λ. These two improvements can be a huge save of unnecessary checkings and comparisons since $|B| \ll |B_{nb}|$ in most cases.

5.2. From B*-Max-Cliques to Global Max-Cliques

A B^*-max-clique C may not be a real max-clique in G; that is, C is maximal locally in G_{B^*} but may not be maximal globally in G. In this subsection, we discuss how we compute the global max-cliques from the B^*-max-cliques.

The key is to compute the set of B^+-max-cliques, $\mathcal{M}_{B^+}$, from T_{B^*}. Our result is based on the following theorem.

THEOREM 5.5. *Let $\mathcal{M}$ be the set of max-cliques in G. Let $\mathcal{M}_0$ be the set of max-cliques in G that consist of at least a B-vertex, that is, $\mathcal{M}_0 = \{C : C \in \mathcal{M}, C \cap B \neq \emptyset\}$. Then, $\mathcal{M}_{B^+} = \mathcal{M}_0$.*

PROOF. First, Lemma 4.6 shows that $\mathcal{M}_{B^+} \subseteq \mathcal{M}_0$. Next, $\forall C \in \mathcal{M}_0$, there exists a vertex $u \in (C \cap B)$. Since C is a clique, for any other vertex v in C such that $v \neq u$, we have v and u are connected. Since $u \in B$, we have either $v \in B$ or $v \in B_{nb}$, meaning that all vertices in C are in B^+. Therefore, we have $C \in \mathcal{M}_{B^+}$ and hence $\mathcal{M}_0 \subseteq \mathcal{M}_{B^+}$. Thus, $\mathcal{M}_{B^+} = \mathcal{M}_0$. □

Theorem 5.5 is important because it enables us to compute a subset of $\mathcal{M}$ separately from a subgraph of G, output it, and move on to computing another subset of $\mathcal{M}$ for another subgraph in the remaining part of G, and so on recursively until we finish the whole graph.

5.2.1. Categorizing B^+-Max-Cliques. We now move on to devise a way to compute the B^+-max-cliques from T_{B^*}. We first define some notation used in the subsequent discussions.

Given a clique C in G_{B^+}, we define $C_B = (C \cap B)$ and $C_{B_{nb}} = (C \cap B_{nb})$. Since C is a clique in G_{B^+} and $B^+ = (B \cup B_{nb})$, we have $C = (C_B \cup C_{B_{nb}})$. Let $\mathcal{M}_B$ be the set of all max-cliques in G_B. Given a clique X in G_B, we define the set of common B-neighbors of the vertices in X as $comNB(X) = \{v : v \in B_{nb}, \forall u \in X, (u, v) \in E\}$. In particular, if C_B is a path in T_{B^*}, $comNB(C_B)$ defines the set of B-neighbor leaves sharing the same path C_B. Finally, given a set of vertices S, we define $maxCL(S)$ to be the set of all max-cliques in G_S, where $S \subseteq V$ and G_S is the induced subgraph of G by S.

With the preceding notations, we identify three disjoint categories of B^+-max-cliques as follows. Let $C = (C_B \cup C_{B_{nb}})$ be a B^+-max-clique.

(1) "$C_{B_{nb}} = \emptyset$": the set of B^+-max-cliques in this category is defined as $\mathcal{M}^1_{B^+} = \{C : C \in \mathcal{M}_{B^+}, C_{B_{nb}} = \emptyset\}$.
(2) "$C_{B_{nb}} \neq \emptyset$ and $C_B \in \mathcal{M}_B$": the set of B^+-max-cliques in this category is defined as $\mathcal{M}^2_{B^+} = \{C : C \in \mathcal{M}_{B^+}, C_{B_{nb}} \neq \emptyset, C_B \in \mathcal{M}_B\}$.
(3) "$C_{B_{nb}} \neq \emptyset$ and $C_B \notin \mathcal{M}_B$": the set of B^+-max-cliques in this category is defined as $\mathcal{M}^3_{B^+} = \{C : C \in \mathcal{M}_{B^+}, C_{B_{nb}} \neq \emptyset, C_B \notin \mathcal{M}_B\}$.

Recall that our objective in this subsection is *to obtain $\mathcal{M}_{B^+}$ from T_{B^*}, or equivalently from $\mathcal{M}_{B^*}$*. Therefore, in the remaining part of this subsection, we first define three sets of cliques $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$ that can be obtained from $\mathcal{M}_{B^*}$. We then prove that $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$ are sound and complete with respect to the earlier-defined three categories of B^+-max-cliques, respectively. We further prove that $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$ give the complete set of $\mathcal{M}_{B^+}$ in Theorem 5.12. Finally, we show how $\mathcal{M}_{B^+}$ can be computed from T_{B^*} in Theorem 5.14.

We first define $\mathcal{M}_1$. Intuitively, $\mathcal{M}_1$ contains all max-cliques in $\mathcal{M}_{B^*}$ that are also in $\mathcal{M}_{B^+}$.

LEMMA 5.6. *Let $\mathcal{M}_1 = \mathcal{M}_B \cap \mathcal{M}_{B^*}$. Then, $\mathcal{M}_1 = \mathcal{M}^1_{B^+}$.*

PROOF. The proof is given in the appendix. □

Essentially, each $C \in \mathcal{M}_1$ corresponds to a root-to-leaf path in T_{B^*}, where the leaf is a B-vertex. Thus, $\mathcal{M}_1$ can be readily obtained from T_{B^*}. We further illustrate the concept by the following example.

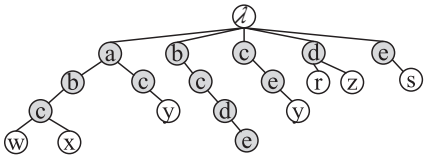


Fig. 3. T_{B^*} of G in Figure 2.

Example 5.7. Figure 3 gives the B^*-max-clique tree T_{B^*} (with the B-vertices shaded) computed from the G_{B^*} of the example graph G in Figure 2. Each root-to-leaf path in T_{B^*} represents a B^*-max-clique and thus there are totally eight B^*-max-cliques. The $\mathcal{M}_B$ consists of only two cliques $\{a, b, c\}$ and $\{b, c, d, e\}$ ($\mathcal{M}_B = \{abc, bcde\}$ for short), which can be obtained from T_{B^*} too.

The set of B^+-max-cliques obtained from the G_{B^+} in Figure 2 is $\mathcal{M}_{B^+} = \{abcwx, acy, bcde, cey, drz, esy\}$.

By Lemma 5.6, $\mathcal{M}_1 = (\mathcal{M}_B \cap \mathcal{M}_{B^*}) = \{bcde\}$, which is the only root-to-leaf path in T_{B^*} with a non-B-neighbor leaf.

We now define $\mathcal{M}_2$. Intuitively, for each clique C in $\mathcal{M}_2$, its B-vertices (i.e., C_B) are in $\mathcal{M}_B$; or equivalently, its B-vertices form a max-clique in G_B.

LEMMA 5.8. *Let* $\mathcal{M}_2 = \{C_1 \cup C_2 : C_1 \in (\mathcal{M}_B \backslash \mathcal{M}_1), C_2 \in maxCL(comNB(C_1))\}$. *Then,* $\mathcal{M}_2 = \mathcal{M}^2_{B^+}$.

PROOF. The proof is given in the appendix. □

Essentially, the C_B of each $C \in \mathcal{M}_2$ is locally maximal in G_B. According to Lemma 5.8, we can compute $\mathcal{M}_2$ as follows. For each path in T_{B^*} that corresponds to each $C_1 \in \mathcal{M}_B$ and has at least a B-neighbor leaf (since $C_1 \in (\mathcal{M}_B \backslash \mathcal{M}_1)$), compute $maxCL(comNB(C_1))$, and output $C = (C_1 \cup C_2)$ for each $C_2 \in maxCL(comNB(C_1))$. We will explain how to check whether a path in T_{B^*} corresponds to a clique in $\mathcal{M}_B$ later in this section.

We further illustrate the concept by the following example.

Example 5.9. (Continued from Example 5.7) By Lemma 5.8, we have $(\mathcal{M}_B \backslash \mathcal{M}_1) = \{abc\}$. Therefore, the C_1 in $\mathcal{M}_2$ can only be abc. Then $comNB(abc) = \{w, x\}$, which are the common B-neighbor leaves of paths in T_{B^*} containing abc. Since w and x are connected in G, we have $maxCL(comNB(abc)) = \{wx\}$. And thus $\mathcal{M}_2 = \{abcwx\}$.

Finally, we define $\mathcal{M}_3$. Intuitively, for each clique C in $\mathcal{M}_3$, its B-vertices form a clique in G_B but the clique is not maximal in G_B. This set of B-vertices in C then forms a max-clique in G_{B^+} together with their common B-neighbors.

We first define two notations, $\mathcal{X}$ and $EXT(\cdot)$, which are used in the definition of $\mathcal{M}_3$. The set $\mathcal{X}$ contains the candidate vertex-sets that are potentially the C_B of some max-cliques. We enumerate the proper subsets of a max-clique in $\mathcal{M}_B$ that have at least one common B-neighbor as

$$\begin{aligned}\mathcal{X} = \{C_1 : {} & C_1 \subset C, C \in \mathcal{M}_B, C_1 \neq \emptyset, comNB(C_1) \neq \emptyset, \text{ and} \\ & \nexists C_1' \subseteq C', C' \in \mathcal{M}_B, \text{s.t. } C_1 \subset C_1', comNB(C_1) = comNB(C_1')\}. \end{aligned} \quad (1)$$

The last condition ensures that each $C_1 \in \mathcal{X}$ is not subsumed by its proper superset when forming a clique with the common B-neighbors. This avoids enumerating some cliques that are not maximal.

Then, for each $C_1 \in \mathcal{X}$, we use $EXT(C_1)$ to denote the set of B-neighbors that can be used to extend C_1, that is, to be the $C_{B_{nb}}$ of some max-cliques. It is defined as

$$\begin{aligned} EXT(C_1) = \{&C_2 : C_2 \in maxCL(comNB(C_1)), \text{ and} \\ &\nexists C' \in \mathcal{M}_2, \text{ s.t. } C' \supset (C_1 \cup C_2), \text{ and} \\ &\nexists C_1'' \in \mathcal{X}, \text{ s.t. } C_1'' \supset C_1,\ C_2 \in EXT(C_1'')\}. \end{aligned} \tag{2}$$

The last two conditions are for the maximality checking of $(C_1 \cup C_2)$ in $\mathcal{M}_2$ and $\mathcal{X}$, respectively.

LEMMA 5.10. *Let* $\mathcal{M}_3 = \{C_1 \cup C_2 : C_1 \in \mathcal{X}, C_2 \in EXT(C_1)\}$. *Then,* $\mathcal{M}_3 = \mathcal{M}^3_{B^+}$.

PROOF. The proof is given in the appendix. □

Essentially, each $C \in \mathcal{M}_3$ differs from a clique in $\mathcal{M}_1$ or $\mathcal{M}_2$ in that C_B is not maximal in G_B. This category of B^+-max-cliques is not as straightforward to compute as the first two categories. We discuss the details in Section 5.2.2.

We further illustrate the concept by the following example.

Example 5.11. (Continued from Examples 5.7 and 5.9) By Lemma 5.10, we have $\mathcal{X} = \{ac, ce, d, e\}$. A naive way is to enumerate all proper subsets of each clique in $\mathcal{M}_B$. However, many of them are subsumed by their proper supersets in $\mathcal{X}$ or $\mathcal{M}_B$. For example, a is subsumed by ac since $comNB(a) = comNB(ac) = \{w, x, y\}$. Then, for each $C_1 \in \mathcal{X}$, we compute $EXT(C_1)$. For example, considering ac, $maxCL(comNB(ac)) = \{wx, y\}$ but $EXT(ac) = \{y\}$. Note that $wx \in maxCL(comNB(ac))$ is excluded from $EXT(ac)$ because $acwx$ is checked to be nonmaximal with respect to $abcwx \in \mathcal{M}_2$. Similarly, we have $EXT(ce) = \{y\}$, $EXT(d) = \{rz\}$, and $EXT(e) = \{sy\}$. Thus, by Lemma 5.10, $\mathcal{M}_3 = \{acy, cey, drz, esy\}$.

We state the completeness and soundness of $(\mathcal{M}_1 \cup \mathcal{M}_2 \cup \mathcal{M}_3)$ with respect to the whole set $\mathcal{M}_{B^+}$ in the following theorem.

THEOREM 5.12. $\mathcal{M}_{B^+} = (\mathcal{M}_1 \cup \mathcal{M}_2 \cup \mathcal{M}_3)$, *where* $\mathcal{M}_1$, $\mathcal{M}_2$ *and* $\mathcal{M}_3$ *are defined in Lemmas* 5.6–5.10.

PROOF. By the categorization, $\mathcal{M}^1_{B^+}$, $\mathcal{M}^2_{B^+}$ and $\mathcal{M}^3_{B^+}$ are disjoint and $(\mathcal{M}^1_{B^+} \cup \mathcal{M}^2_{B^+} \cup \mathcal{M}^3_{B^+})$ gives exactly $\mathcal{M}_{B^+}$. By Lemmas 5.6–5.10, we have $(\mathcal{M}_1 \cup \mathcal{M}_2 \cup \mathcal{M}_3) = \mathcal{M}_{B^+}$. □

As an example, it is easy to see from Examples 5.7–5.11 that $(\mathcal{M}_1 \cup \mathcal{M}_2 \cup \mathcal{M}_3)$ gives exactly $\mathcal{M}_{B^+}$.

Now we show that T_{B^*} (plus the knowledge of the edges between the B-neighbors) is sufficient to compute $\mathcal{M}_{B^+}$. We only need partial $G_{B_{nb}}$ but do not keep $G_{B_{nb}}$ in main memory. Before we discuss the computation of $\mathcal{M}_{B^+}$ from T_{B^*}, we first show that $\mathcal{M}_B$ can be obtained from T_{B^*} by the following lemma.

LEMMA 5.13. $\forall C \in \mathcal{M}_B$, *there exists a root-to-leaf path* $p \in T_{B^*}$ *such that* C *is the set of* B*-vertices on* p.

PROOF. For each $C \in \mathcal{M}_B$, let $C = \{v_1, \ldots, v_k\}$ where $v_1 \prec \cdots \prec v_k$. First, if $comNB(C) = \emptyset$, then there must exist a root-to-leaf path $p = \langle \lambda, v_1, \ldots v_k \rangle$ in T_{B^*} since C is maximal in G_B and the vertices in C have no common B-neighbors. Next, if $comNB(C) \neq \emptyset$, that is, $\exists u \in comNB(C)$, then $p = \langle \lambda, v_1, \ldots, v_k, u \rangle$ must be a root-to-leaf path in T_{B^*} since C is maximal in G_B and a B^*-max-clique contains at most one B-neighbor. □

THEOREM 5.14. $\mathcal{M}_{B^+}$ *can be computed from* T_{B^*} *and* $G_{B_{nb}}$.

PROOF. By Lemma 5.13, every $C \in \mathcal{M}_B$ exists in T_{B^*}. Therefore, $\mathcal{M}_B$ can be computed from T_{B^*} by removing all B-neighbor leaves and checking the maximality of all remaining paths (this can be incorporated into the maximality checking when constructing T_{B^*} without any extra cost). Thus, $\forall C \in \mathcal{M}_{B^+}$, C_B can be obtained from T_{B^*}. On the other hand, the set of common B-neighbors, $comNB(C_B)$, can be obtained from the B-neighbor leaves in T_{B^*}. Since $comNB(C_B) \subseteq B_{nb}$, the corresponding $C_{B_{nb}}$ can be computed from the part of $G_{B_{nb}}$ that gives $G_{comNB(C_B)}$. □

5.2.2. Computing B^+-Max-Cliques from B^*-Max-Cliques. We now discuss the algorithm to compute $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$, as shown in Algorithm 3.

ALGORITHM 3: *Compute-B^+-Max-Cliques*

Input: T_{B^*} (and partial $G_{B_{nb}}$)
Output: $\mathcal{M}_{B^+}$

1. Initialize $\mathcal{M}_1 = \mathcal{M}_2 = \mathcal{M}_3 = \emptyset$;
2. **for each** root-to-leaf path $p = \langle \lambda, v_1, \ldots, v_k \rangle$ in T_{B^*} **do**
3. **if** $(v_k \in B)$
4. $\mathcal{M}_1 \leftarrow \mathcal{M}_1 \cup \{(v_1, \ldots, v_k)\}$; /* by Lemma 5.6 */
5. **else**
6. $C_1 \leftarrow (v_1, \ldots, v_{k-1})$;
7. **if** $(C_1 \in \mathcal{M}_B)$ /* by Lemma 5.8 */
8. Compute $maxCL(children(v_{k-1}))$;
9. $\mathcal{M}_2 \leftarrow \mathcal{M}_2 \cup \{C_1 \cup C_2 : C_2 \in maxCL(\ children(v_{k-1}))\}$;
10. Compute $\mathcal{X}$; /* see Eq. (1) */
11. **for each** $C_1 \in \mathcal{X}$ **do** /* by Lemma 5.10 */
12. Compute $EXT(C_1)$; /* see Eq. (2) */
13. $\mathcal{M}_3 \leftarrow \mathcal{M}_3 \cup \{C_1 \cup C_2 : C_2 \in EXT(C_1)\}$;
14. **return** $\mathcal{M}_{B^+} \leftarrow (\mathcal{M}_1 \cup \mathcal{M}_2 \cup \mathcal{M}_3)$;

It is straightforward to obtain both $\mathcal{M}_1$ and $\mathcal{M}_2$, by performing a Depth-First Search (DFS) on T_{B^*} (lines 2–9). We do not store explicitly the set $\mathcal{M}_B$ and search it to check whether $C_1 \in \mathcal{M}_B$ (line 7). Instead, we mark each vertex u whose root-to-u path forms a clique in $\mathcal{M}_B$ when we construct T_{B^*}. Thus, we only need to check whether v_{k-1} is marked in line 7. We explain how to compute $maxCL(\cdot)$ later.

To obtain $\mathcal{M}_3$, we first compute $\mathcal{X}$ in line 10 as follows. We enumerate all proper subsets of each $C_1 \in \mathcal{M}_B$ in line 7, while checking the conditions defined in $\mathcal{X}$ (see Eq. (1)) to prune the unqualified subsets. The checking of the last condition in $\mathcal{X}$ is similar to the maximality checking when constructing T_{B^*}. Note that the set $comNB(C)$ of a clique C can be easily obtained from T_{B^*} as the set of B-neighbor leaves of the paths containing C.

Given $\mathcal{X}$, we then compute $EXT(C_1)$ for each $C_1 \in \mathcal{X}$ (line 12). The maximality checking defined in $EXT(C_1)$ (see Eq. (2)) is done in the same way as that in $\mathcal{X}$. As for the computation of $maxCL(comNB(C_1))$, we use an existing in-memory MCE algorithm. Since $comNB(C_1)$ consists of common B-neighbors of all vertices in C_1, $comNB(C_1)$ is small and thus it is efficient to compute $maxCL(comNB(C_1))$.

However, in order to compute $maxCL(comNB(C_1))$, we need to know the induced subgraph $G_{comNB(C_1)}$. Note that once we get T_{B^*}, we remove G_{B^*} from main memory. Thus, we now have more space to keep partial $G_{B_{nb}}$. In order to avoid random access to G in the disk, we do the following. For all B-neighbor leaves in T_{B^*} ordered by the DFS traversal, we divide them into k partitions P_i $(1 \leq i \leq k)$ such that the adjacency lists of the B-neighbors in each P_i can fit into main memory. We then read G from the disk

ALGORITHM 4: *EmMCE*

Input: a graph G, recursive step k
Output: the set of max-cliques, $\mathcal{M}$, in G

1. Select a set of base vertices, B, and extract (and remove) G_{B^*} from G;
2. Construct T_{B^*} from G_{B^*} by an existing in-memory MCE algorithm;
3. Delete G_{B^*} from memory;
4. Compute-B^+-Max-Cliques (Alg. 3) from T_{B^*};
5. **if** $(k = 1)$ /* B^+-max-cliques are max-cliques by Theorem 5.5 */
6. Store the B^+-max-cliques in a hashtable X;
7. Output the B^+-max-cliques;
8. **else**
9. Check the maximality of the B^+-max-cliques using X;
10. Output the B^+-max-cliques that are globally maximal;
11. Update X;
12. **if** (G is not empty)
13. EmMCE(G, $k + 1$);

sequentially and for each $v \in B_{nb}$, we write $(nb(v) \backslash B)$ into the partition(s) v is in. We keep the first partition in main memory, while each of the other partitions is written into consecutive disk pages. In this way, we read a partition (partial $G_{B_{nb}}$) into main memory each time when computing $maxCL(comNB(C_1))$ and avoid random disk access.

6. OUR ALGORITHM: EMMCE

With the foundation established in Section 5, we now present our external-memory algorithm for maximal clique enumeration, *EmMCE*, as given in Algorithm 4.

As outlined in Section 3, EmMCE is a recursive algorithm. For each recursive step, the algorithm starts with the selection of a set of base vertices, B, and the extraction of G_{B^*} from G (line 1). While we extract G_{B^*} from G, we also remove G_{B^*} from G at the same time; that is, we remove all the B-vertices and all the edges connected to them (note that the B-neighbors are still retained in G for the completeness of maximal clique enumeration). We delay the discussion on how we select B to Section 7. But we note that the selection of B and the extraction of G_{B^*} from G can be done in one scan of G from disk and with linear CPU time complexity.

After extracting G_{B^*}, we construct T_{B^*} by an existing in-memory MCE algorithm as discussed in Section 5.1 (line 2). After obtaining T_{B^*}, we delete G_{B^*} to release the occupied memory (line 3). We then invoke Algorithm 3 to compute the set of B^+-max-cliques $\mathcal{M}_{B^+}$ from T_{B^*} (line 4), as discussed in Section 5.

Now, $\mathcal{M}_{B^+}$ is computed and outputted, and G_{B^*} and T_{B^*} are discarded. We are ready to move on to the next recursion to process the remaining part of G. This process continues until G becomes empty (lines 12–13).

To guarantee the maximality of the cliques, our algorithm performs the following checking. Consider the first and the second recursive steps. Let B and L be the set of base vertices at the first and the second recursive steps, respectively. If a clique C is maximal in G_{L^+}, then C may not be maximal globally in G. This is because there may exist a clique $C' \in \mathcal{M}_{B^+}$ such that $C' \supset C$ and $C'_{B_{nb}} = C$. We remark that if a clique C' is maximal in G_{B^+}, then C' is also maximal in G, because B only has connection with B_{nb}. This may not be true for a clique C in G_{L^+} though, because L may have connection with both B and L_{nb}.

We address this problem as follows. First at the first recursive step, for each $C \in \mathcal{M}_{B^+}$, if $|C_{B_{nb}}| > 1$, we keep $C_{B_{nb}}$ in a hashtable X (lines 5–7). Then for each of the following recursive steps (lines 8–11), let C' be a B^+-max-clique computed in the current recursive

step. If $|C'| = 1$, then $C' = \{v\}$ is globally maximal if and only if v is an isolated vertex in the original graph. If $|C'| > 1$, we hash C' to check if C' exists in the hashtable. If C' is not in the hashtable, then C' is globally maximal and we also add $C'_{L_{nb}}$ (if $|C'_{L_{nb}}| > 1$) into the hashtable for the maximality checking in subsequent recursive steps. Otherwise, C' is not globally maximal and we also remove C' from the hashtable, since C' will not be computed again in subsequent steps. We also control the number of cliques kept in the hashtable as follows. At each recursive step except the first one, we delete any record C currently in the hashtable if $\exists v \in C$ such that v is a B-vertex selected at the current recursive step, because all globally maximal cliques containing v must be generated at the end of this step (according to Theorem 5.5).

Finally, the set of all max-cliques in an input graph G can be computed by invoking EmMCE(G, 1). The following theorem proves the correctness of EmMCE.

THEOREM 6.1. *The set of max-cliques outputted by EmMCE is sound and complete with respect to the set of all max-cliques in G.*

PROOF. We first prove the soundness. At the first recursive step, the set of B^+-max-cliques is computed in line 4 and outputted directly in line 7 of Algorithm 4. The B^+-max-cliques are proved to be maximal in G in Theorem 5.5. Next, at each subsequent recursive step, the B^+-max-cliques are computed in line 4, and the maximality of the outputted B^+-max-cliques is ensured by the checking in line 9 of Algorithm 4.

We now prove the completeness. At the first step, the set of B^+-max-cliques is complete with respect to the max-cliques in G that contain at least one vertex in B (by Theorem 5.5). Let L be the set of base vertices at the second recursive step. The set of L^+-max-cliques are computed in the same way as at the first recursive step (line 7). This means that the set of L^+-max-cliques is complete with respect to the set of max-cliques in $(G \backslash G_{B^*})$ that contain at least one vertex in L. Combining with the B^+-max-cliques (computed from G_{B^*} at the first step) that contain at least one vertex in L, it gives a complete set of max-cliques in G that contain at least one vertex in L. Similarly by recursion, a complete set of max-cliques in G that contain at least one vertex in the corresponding L is given after each recursive step. Since the recursion terminates when the graph G becomes empty (i.e., all vertices have been considered to form max-cliques), the algorithm outputs a complete set of max-cliques in G. □

Complexity. Our algorithm needs $\mathcal{O}(|G|/|G_{B^*}|)$ recursions, assuming that the size of G_{B^*} selected at each step is approximately the same. At each recursion, we need to scan G (sequentially) twice, once for selecting B and extracting G_{B^*} from G (line 1 of Algorithm 4) and once for reading the edges of $G_{B_{nb}}$ when computing the B^+-max-cliques (line 4 of Algorithm 4). For both scans of G, we require I/O operations as we read the graph G from disk. Apart from this, we also require I/O operations to write the B^+-max-cliques to disk (lines 7 and 10 of Algorithm 4). The I/O operations are counted as the number of blocks being read/written from/to disk. Thus, the algorithm requires $\mathcal{O}((|G|/|G_{B^*}|)(|G|/b) + (|\mathcal{M}|/b))$ I/Os, where b is the block size and $\mathcal{O}(|\mathcal{M}|/b)$ is the I/O cost to write the result to disk. Note that the total cost to write/read the partial $G_{B_{nb}}$ for computing the B^+-max-cliques is also bounded by $\mathcal{O}(|\mathcal{M}|/b)$.

In line 1 of Algorithm 4, we also remove G_{B^*} from G. Removing G_{B^*} from G can be performed at the same time during the sequential reading of G from disk when we extract G_{B^*}. Meanwhile, we also need to write the newly updated G back to disk, which in total requires $\mathcal{O}(|G|/b)$ I/Os for each recursion of Algorithm 4. Thus, this write operation does not change the overall I/O complexity given before. We also note that the graph G after each recursion becomes smaller after removing G_{B^*}.

The memory space complexity of EmMCE is $\mathcal{O}(|G_{B^*}| + |T_{B^*}|)$. For the CPU time complexity, we compare EmMCE with an existing in-memory MCE algorithm A. Let

$A(G)$ denote the algorithm A when it is applied directly to the whole graph G. If we only consider the in-memory operations, the time required for the entire recursive steps in EmMCE is comparable to that of $A(G)$. This is because Algorithm 3 essentially expands those paths in T_{B^*} that would be generated by $A(G)$ as well, while the computation of each $maxCL(comNB(\cdot))$ corresponds to the construction of subtrees in the search tree created by $A(G)$.

7. SELECTION OF BASE VERTEX-SET

The discussions in Sections 5 and 6 assume that the base vertex-set is given. The base vertex-set can be randomly selected or selected according to a certain criterion. In this section, we give the details on the selection of the base vertex-set at each recursive step.

We discuss two strategies of selecting the base vertex-set: one aiming at filling available main memory for maximal clique enumeration in static graphs, and the other considering the update maintenance of the set of max-cliques in dynamic graphs.

7.1. Base Vertex Selection by Available Memory

In this selection scheme, we sequentially scan the input graph G to select a vertex v to include in B, along with $nb(v, G)$ to construct G_{B^*}, until the available memory assigned for G_{B^*} is filled.

Sequentially scanning G to select B and extract G_{B^*} has the advantage that, for the whole process of maximal clique enumeration, we only need to scan G once instead of $\mathcal{O}(|G|/|G_{B^*}|)$ times. That is, we simply select the first k vertices in G as the set of B-vertices, together with their adjacent list which is essentially G_{B^*}. The number k is determined by the available memory. Then, we continue with the next batch of vertices in G at the next recursion of Algorithm 4. In addition, by sequentially selecting B and extracting G_{B^*}, we also do not need to remove G_{B^*} from G and write the newly updated G to disk after removing G_{B^*}, as described in the complexity analysis in Section 6.

More importantly, this selection scheme also naturally allows parallel computation of maximal clique enumeration. That is, we distribute a B^*-graph (instead of the entire graph as do in the existing parallel algorithms [Du et al. 2009; Schmidt et al. 2009]) to each computing element for maximal clique enumeration, and yet guaranteeing the completeness of result.

The size of G_{B^*} is known after we select B; however, the size of T_{B^*} requires estimation since we do not have T_{B^*} constructed yet at the stage of base vertex selection. We describe a method for estimating $|T_{B^*}|$ as follows.

We devise an estimation strategy that borrows the concept of Knuth's method [Knuth 1975] for estimating the size of a backtracking tree T. Let $n(T)$ be the number of vertices in T. The idea is to randomly probe a set of paths P in T and estimate $n(T) = AVG_{p\in P}(n(p))$, where $n(p)$ is the size of a tree with the same root as p and using p as a building path. Let $p = \langle v_1, v_2, \ldots, v_k\rangle$, then $n(p) = (1 + f_1 + f_1 f_2 + \cdots + (f_1 \cdots f_{k-2} f_{k-1}))$, where f_i is the number of children of v_i. In the simple case that T is a complete binary tree, this method correctly estimates $n(T)$ as $(2^k - 1)$. It is shown that Knuth's method is unbiased and effective in practice [Kilby et al. 2006].

However, Knuth's method assumes the presence of T so that one can perform random probing of paths, while T_{B^*} in our case is not yet constructed. We propose a new method of probing paths in T_{B^*} by utilizing its unique properties, without actually constructing T_{B^*}. Each time we randomly choose a vertex $u \in B$. We consider u as a child of λ and attempt to probe randomly a path p from u as follows: we randomly choose a vertex v from the set of vertices that can be used to potentially grow p from u, and then continue the process recursively from v until the path p cannot be expanded any more (i.e., p corresponds to a B^*-max-clique). Since the vertices are ordered and $nb(v)$ is available

for every $v \in B$, we can virtually probe a path even though T_{B^*} does not exist. Thus, we can compute $n(p)$ as we move along p. To control the number of probing paths in the process, we limit the total number of probing paths starting from u to be at most k. Finally, we estimate $n(T_{B^*})$ by averaging $n(p)$ of all the paths probed.

Our method is simple and yet does not violate the principle of random probing [Knuth 1975]. Our empirical study shows that it gives a good estimation in practice (see Table III in Section 10.1).

In the case when the available main memory, M, is smaller than $(n(T_{B^*}) + |G_{B^*}|)$, we remove $(((n(T_{B^*})+|G_{B^*}|-M)/(n(T_{B^*})+|G_{B^*}|))|B|)$ vertices from B, together with their incident edges from G_{B^*}. We reestimate $n(T_{B^*})$ for the smaller G_{B^*} until $(n(T_{B^*})+|G_{B^*}|) \leq M$.

7.2. Base Vertex Selection: H-Vertices

In this selection scheme, we select the base vertices by introducing a novel concept of H-vertices for real-world networks. The H^*-graph with respect to H-vertices is used for designing an efficient update strategy for the set of max-cliques in dynamic graphs (details in Section 8).

7.2.1. H-Vertices and H-Graph.* The concept of H-vertices is inspired by the concept of *h-index*, which is commonly used to measure the publication quality and productivity of a scientist, developed by Jorge E. Hirsch in 2005. The h-index is defined as the *maximum* h for a scientist who has h publications with at least h citations. These h publications with at least h citations are identified as the most representative and important research work of a scientist.

Putting into the context of a graph, it is the maximum h for a graph that has h vertices (corresponding to the h publications) with at least h edges (corresponding to at least h citations). We define the "h-index" for a graph as follows.

Definition 7.1 (*h-Index for a Graph*). Given a graph $G = (V, E)$, the *h-index* of G, denoted as h_{max}, is defined as the maximum h for G that has h vertices with degree of at least h. Formally, $h_{max} = max\{h : \exists S \subseteq V, \text{ s.t. } |S| = h \text{ and } \forall v \in S, d(v, G) \geq h\}\}$.

Analogous to the h-index for a scientist, it is easy to see that these h_{max} vertices with degree at least h_{max} represent the most important vertices in G. We put these vertices into a set H and call them *H-vertices*.

The set of H-vertices is formally defined as follows.

Definition 7.2 (*H-Vertices*). Given a graph $G = (V, E)$, the set of *H-vertices* of G, denoted as H, is defined as $H = \{v : v \in V, d(v, G) \geq h_{max}\}$ such that $|H| = h_{max}$, and $\forall v \in (V \backslash H)$, $d(v, G) \leq h_{max}$.

Note that there may be multiple vertices that have a degree of h_{max}. In this case, the tie is broken arbitrarily.

The following example explains the concept.

Example 7.3. Consider the example graph G given in Figure 2. We have $h_{max} = 5$ and the set of H-vertices is $H = \{a, b, c, d, e\}$. It can be easily checked in the figure that all the 5 vertices in H (shaded vertices) have degree of at least 5 and all the remaining vertices in G have degree of less than 5.

Since H contains the most important vertices in G, we set $B = H$ for selection of the base vertex-set. To indicate that H is used as the base vertex-set, we replace B by H in our notations in the discussion of the remainder of this section. Thus, the H^*-graph of G, that is, G_{H^*}, is essentially the B^*-graph G_{B^*} where $B = H$.

We now show that H and G_{H^*} can be computed by one scan of G. Algorithm 5 presents the algorithm for computing the set of H-vertices H, together with the set of

ALGORITHM 5: *Compute-H^*-Graph*

Input: $G = (V, E)$.
Output: The set of H-vertices of G, H, and the set of their neighbors, $NB_H = \{nb(v) : v \in H\}$.

1. Set $h \leftarrow 0$ and initialize an empty *min-heap*, Q;
2. Let $(d(v), v, nb(v))$ be an *element* in Q, where $d(v)$ is the *key*;
3. Denote the *minimum key* of Q by min;
4. **for each** $v \in V$ **do**
5. **if** ($h = 0$ or ($d(v) > h$ and $min > h$))
6. *insert* $(d(v), v, nb(v))$ into Q;
7. h++;
8. **else if** ($d(v) > h$ and $min = h$)
9. *delete-min* and *insert* $(d(v), v, nb(v))$ into Q;
10. **return** $H \leftarrow \{v : (d(v), v, nb(v)) \in Q\}$ and $NB_H \leftarrow \{nb(v) : (d(v), v, nb(v)) \in Q\}$;

their neighbors NB_H. A min-heap Q is used to keep the H-vertices with their neighbors using the vertex degree as the key. Lines 4–9 perform a scan on the vertices in the input graph G to check whether a vertex can be added to Q as a potential H-vertex. A vertex with degree larger than the current h is either directly inserted to Q in lines 5–7 (when h can still grow since the min-degree in Q is larger than h) or replace the min-degree vertex in Q in lines 8–9 (if h is incremented, the min-degree vertex no longer satisfies the degree requirement and is thus discarded). Finally, the set of vertices kept in Q is returned as H. After we obtain H and their neighbor sets NB_H (i.e., the adjacency lists), we essentially obtain the H^*-graph.

We prove the correctness of Algorithm 5 as follows.

THEOREM 7.4. *Algorithm 5 correctly computes H and G_{H^*} in $\mathcal{O}(h_{max} \cdot \log(h_{max}) + n)$ time and $\mathcal{O}(|G_{H^*}|)$ space, and uses $\mathcal{O}(|G|/b)$ I/Os, where b is the block size.*

PROOF. To prove the correctness, we need to show that: the h computed by Algorithm 5 is equal to h_{max}. Suppose to the contrary that $h < h_{max}$, which implies that there are h_{max} vertices with a degree of at least h_{max}. However, according to Algorithm 5, these h_{max} vertices must be inserted into Q at some point, since their degree is greater than h and the value of h is never decreasing in Algorithm 5. Therefore, h computed by Algorithm 5 should be at least h_{max} in this case. On the other hand, h cannot be larger than h_{max} since each increment of h (line 7 of Algorithm 5) follows the definition of H-vertex (line 5). Thus, we have $h = h_{max}$.

We have $\mathcal{O}(h_{max})$ insertions/updates, each taking $\mathcal{O}(\log(h_{max}))$ time, plus n comparisons between h_{max} and $d(v)$ for each $v \in V$. Space is needed to keep H-vertices and their adjacency lists, which takes $\mathcal{O}(|G_{H^*}|)$ space.

Since each vertex $v \in V$ is processed only once in the order of their occurrence in G, we only need to read G sequentially in blocks from disk once, which requires $\mathcal{O}(|G|/b)$ I/Os. □

When we set $B = H$ and $G_{B^*} = G_{H^*}$ in Algorithm 4, we need to invoke Algorithm 5 in line 1 of Algorithm 4 at every recursive call of EmMCE. However, as shown in Theorem 7.4, executing Algorithm 5 requires only one scan of G, or $\mathcal{O}(|G|/b)$ I/Os. Thus, extracting G_{H^*} by Algorithm 5 has the same I/O complexity as extracting G_{B^*} as described in the complexity analysis in Section 6, while the CPU overhead incurred by Algorithm 5 is small compared with the I/O cost. Removing G_{H^*} from G, however, requires another scan of G and to write the newly updated G to disk, but the overall I/O complexity still remains to be the same.

In the remainder of this section, we analyze and explain why we use H and G_{H^*}.

7.2.2. Analysis of H-Graph.* We first examine two important factors: the size of H and the size of G_{H^*}.

We first discuss the size of H. Faloutsos et al. [1999] show that for real-world networks following a power law degree-distribution, it holds that

$$d(v) = \frac{1}{n^{\mathcal{R}}}(r(v))^{\mathcal{R}}. \tag{3}$$

In Eq. (3), $r(v)$ is the *degree rank* of a vertex v, that is, v is the $(r(v))$-th highest degree vertex in G, and $\mathcal{R}$ is the *rank exponent*, where $\mathcal{R} < 0$. By Definition 7.2, H is a set of h_{max} vertices having degree at least h_{max}; in other words, the lowest-degree vertex v in H has a rank of h_{max} and its degree is at least h_{max}. Thus, by substituting $r(v)$ by h_{max} in Eq. (3) and $d(v)$ should be at least h_{max}, we have

$$d(v) = \frac{1}{n^{\mathcal{R}}}h_{max}^{\mathcal{R}} \geq h_{max}. \tag{4}$$

Solving the inequality, we have

$$h_{max} \leq n^{\frac{\mathcal{R}}{\mathcal{R}-1}}. \tag{5}$$

Faloutsos et al. [1999] show that $\mathcal{R}$ is a constant for most real-world networks, which can be easily measured by plotting the degree distribution of the networks. The typical value of $\mathcal{R}$ found by Faloutsos et al. [1999] is between -0.8 and -0.7. For a graph of 1 million vertices, we have $h_{max} \leq 464$ and therefore $|H| \leq 464$ when $\mathcal{R} = -0.8$. The value of h_{max} decreases to about 300 when $\mathcal{R} = -0.7$. This shows that the number of H-vertices in a large real-world network is small.

Next, we estimate the size of G_{H^*}. By Eq. (3), we have the following upper bound for $|G_{H^*}|$.

$$|G_{H^*}| \leq \sum_{r=1}^{h_{max}} \left(\frac{r}{n}\right)^{\mathcal{R}} \tag{6}$$

The right-hand side of Eq. (6) is the sum of degrees of all the H-vertices. Since the edges connecting two H-vertices (if there is any) are counted twice, we have the "$<$" sign in Eq. (6). The equality holds when there is no edge connecting two H-vertices; in this case, the H^*-graph consists of h_{max} "stars", each centered at an H-vertex.

We can also obtain a lower bound for $|G_{H^*}|$ as follows.

$$|G_{H^*}| \geq \sum_{r=1}^{h_{max}} \left(\frac{r}{n}\right)^{\mathcal{R}} - \frac{h_{max}(h_{max}-1)}{2} \tag{7}$$

The lower bound occurs when all H-vertices are pairwise connected. In this case, all edges connecting two H-vertices are double counted and hence deducting the number of these edges from the degree sum gives the lower bound of $|G_{H^*}|$.

Similarly, we also obtain the size of G, which is half of the degree sum of all vertices in V, since all edges are counted twice.

$$|G| = \frac{1}{2}\sum_{r=1}^{n} \left(\frac{r}{n}\right)^{\mathcal{R}} \tag{8}$$

By Eqs. (6)–(8), we have

$$\frac{2\sum_{r=1}^{h_{max}} r^{\mathcal{R}} - n^{\mathcal{R}} h_{max}(h_{max}-1)}{\sum_{r=1}^{n} r^{\mathcal{R}}} \leq \frac{|G_{H^*}|}{|G|} \leq \frac{2\sum_{r=1}^{h_{max}} r^{\mathcal{R}}}{\sum_{r=1}^{n} r^{\mathcal{R}}}. \tag{9}$$

For a network with $\mathcal{R} = -0.7$ and 1 million vertices, $|G_{H^*}|$ is within [12%, 15%] of the entire network, and the percentage lowers considerably when the network becomes larger: the ratio is in the range of [8%, 10%] when n increases to 10 million.

With the result of Eq. (9), the amount of memory required for keeping G_{H^*} is reasonable. Another desirable aspect of the H^*-graph is that the rank exponent in Eq. (5) is a constant for most real-world networks. This property allows us to even estimate the size of G_{H^*} when the network grows, so that we can predict the memory resource required at a certain point in the future. For many real-world networks, it is possible to predict the growth of the network based on its past growth pattern, and thus we can prepare in advance the memory resource required for our computation in the future.

7.2.3. Why H-Graph?* The use of H^*-graph has the following three advantages.

Why not G_H or G_{H^+}? We first analyze $|G_H|$ as follows.

$$0 \leq |G_H| \leq \frac{h_{max}(h_{max}-1)}{2}. \tag{10}$$

Eq. (10) gives the lower and upper bounds of $|G_H|$. Since h_{max} is small, if we use G_H as the in-memory partition, it leads to too many recursive steps in the max-clique computation and hence too many scans of G from the disk. In fact, the formulation of G_H is not applicable for max-clique computation.

As for $|G_{H^+}|$, let $s = \sum_{r=1}^{h_{max}} (\frac{r}{n})^{\mathcal{R}}$, that is, the degree sum of H-vertices. $|G_{H^+}|$ reaches its maximum when: (1) the number of H-neighbors is maximized (i.e., $|H_{nb}| = s$); (2) the degrees of H-neighbors rank top among non-H-vertices (i.e., the degree rank of H-neighbors is from $(h_{max}+1)$ to $(h_{max}+s)$ in G); and (3) all H-neighbors connect with only vertices in H^+ (i.e., all edges incident to H-neighbors are in G_{H^+}). Thus, the upper bound of $|G_{H^+}|$ is

$$\begin{aligned} |G_{H^+}| &\leq \frac{1}{2}\left(s + \sum_{r=1}^{s}\left(\frac{h_{max}+r}{n}\right)^{\mathcal{R}}\right) \\ &= \frac{1}{2}\sum_{r=1}^{h_{max}+s}\left(\frac{r}{n}\right)^{\mathcal{R}}. \end{aligned} \tag{11}$$

The lower bound of $|G_{H^+}|$ is simply $|G_{H^*}|$ since $G_{H^*} \subseteq G_{H^+}$. Eq. (11) shows that G_{H^+} is too large to be kept in main memory. For example, when $\mathcal{R} = -0.7$ and n is 1 million, G_{H^+} can be as large as 65% of the graph G.

From the semantic point of view, G_H only retains the very core of G and does not reveal much global information, while G_{H^+} may be giving too much general information and making it not much different from G. On the contrary, G_{H^*} gives the core of G as well as the relationship from the core to other parts of G. We examine empirically more properties of G_{H^*} in Section 10.3.

Using G_{H^*} for updates in dynamic networks. Let $\mathcal{M}$ and $\mathcal{M}_{H^*}$ be the set of all max-cliques in G and G_{H^*}, respectively. Real-world networks undergo frequent updates. When the network G is updated, $\mathcal{M}$ also needs to be updated. However, a simple edge insertion or deletion can cause a series of updates to the set of max-cliques, while each update of a clique can easily lead to a cascade of updates in $\mathcal{M}$. This is a very costly operation because $\mathcal{M}$ is very large and needs to be kept on disk, while the

updates require random disk accesses to read and write the cliques. For many dynamic networks, update maintenance of $\mathcal{M}$ is simply infeasible.

Updating $\mathcal{M}$ is infeasible; however, computing $\mathcal{M}$ from scratch is also too expensive for large networks. We propose a *semi-dynamic* update strategy: by updating $\mathcal{M}_{H^*}$ and computing $\mathcal{M}$ from $\mathcal{M}_{H^*}$ on demand. This approach is feasible for the following reasons. First, $\mathcal{M}_{H^*}$ is much smaller and can be kept in main memory of most commodity computers, thus enabling in-memory dynamic updates. Second, G_{H^*} keeps the most significant information of G and as a result, $\mathcal{M}_{H^*}$ also keeps the most essential part of $\mathcal{M}$.

Finally, we adopt G_{H^*} instead of G_{B^*} in the dynamic update framework because the set of maximal cliques computed from G_{H^*} are those that consist of the most important vertices in the graph (with respect to vertex degree). The subgraph consisting of those high-degree vertices is also the densest and maximal clique enumeration in this subgraph is more costly; thus updating from $\mathcal{M}_{H^*}$ can save this high cost of clique enumeration. If we select B instead of H as in Section 7.1, the vertex set B can be any vertices instead of the more important ones and hence the corresponding B^*-max-cliques may not be important as well.

Why not top-k? Can we simply choose the top-k highest-degree vertices instead of the H-vertices? The use of H-vertices is a good choice for the following reasons. First, the H-vertices are essentially the top-h_{max} highest-degree vertices. Second, the value of h_{max} is fixed for a given network, while it is difficult to choose the best k for different networks. Moreover, an analysis based on the importance of the vertices leads k to h_{max} as follows.

Let k be the number of vertices that have degree at least k'. A lower k' leads to a larger k, while a higher k' leads to a smaller k. If the degree of a vertex indicates its significance or quality in G (similar to "more citations indicate higher impact of a publication"), then the values of k and k' represent a trade-off between the quantity and the quality of the set of vertices to be selected from G. A balance point in the trade-off is $k = k'$ (as implied by h-index), which is exactly the definition of h_{max}.

In our framework, h_{max} also serves as a good balance point for another trade-off. Suppose that more memory is available for us to choose $k > h_{max}$ vertices. This will reduce the number of recursive steps in our computation and hence is more time efficient. However, a larger k also means a higher update maintenance cost, while update maintenance is an important issue in today's fast-growing networks. On the other hand, a smaller k allows faster update but the resultant graph is also much smaller and does not keep as rich information as G_{H^*}. It is also slower to compute the whole set of max-cliques when k is small.

8. UPDATE IN DYNAMIC NETWORKS

In this section, we discuss an effective way of updating the set of max-cliques when the input graph undergoes frequent updates. The effectiveness of our update approach relies on the selection of the base vertex-set as H.

To indicate that H is used as the base vertex-set, we replace B by H in our notations in this section. The update to G is to be reflected on the set of H^*-max-cliques $\mathcal{M}_{H^*}$ instead of the set of all max-cliques $\mathcal{M}$. In other words, we first compute and keep $\mathcal{M}_{H^*}$ in memory, but we do not compute $\mathcal{M}$ (since $\mathcal{M}$ is often too large to be kept in memory while updating $\mathcal{M}$ directly from disk is prohibitively expensive). When G is updated, we only perform update to $\mathcal{M}_{H^*}$ in memory. On the other hand, we can compute $\mathcal{M}$ periodically or on demand, which is done by invoking Algorithm 4 with $\mathcal{M}_{H^*}$ already computed in the first recursion (our experiments show that computing $\mathcal{M}$ from $\mathcal{M}_{H^*}$ is significantly faster than from scratch).

We consider two types of updates in G: edge insertion and edge deletion. Vertex insertion/deletion can be considered as a series of edge insertions/deletions preceded/followed by the insertion/deletion of an isolated vertex, which is a rather trivial operation.

8.1. Update of H-Vertices

Both the insertion and deletion of a single edge e may trigger a change in H: a vertex v should be added to H or removed from H. We maintain the degree sequence of the vertices in G; thus, it is easy to check whether H needs to be updated.

We first consider the case of adding v to H. We also need to update T_{H^*} accordingly. There are two cases: $v \notin H_{nb}$ or $v \in H_{nb}$. First, if $v \notin H_{nb}$, then we simply create v as the rightmost child of λ and create each $u \in nb(v)$ as a child of v. Second, if $v \in H_{nb}$, then v is a leaf in T_{H^*}. For every occurrence of v in T_{H^*}, let S be the set of siblings of v that are H-neighbors. We reconnect $(S \backslash nb(v))$ as right siblings of v and then $(S \cap nb(v))$ as the set of children of v. Then, if $\exists u \in (nb(v) \backslash H)$ such that u is not a child of any occurrence of v in T_{H^*}, then we also create v as the rightmost child of λ and create each such u as a child of v.

Next, we consider the case of removing a vertex v from H. If v now becomes an H-neighbor, then we reconnect v as leaves in T_{H^*}; otherwise, we remove v from T_{H^*}. Then, we need to check all affected paths in T_{H^*} to see if they still represent H^*-max-cliques and perform further update of T_{H^*} accordingly. This operation can be costly. Thus, we perform *lazy* updates instead. We simply retain v in H and T_{H^*} and perform no update. Keeping v in T_{H^*} does not violate the definition of T_{H^*} as far as we consider that v is still in H. We perform an active update periodically or only when T_{H^*} grows large.

The lazy update is feasible for the following reasons. First, according to the principle of "*the rich get richer*" or the model of *preferential attachment*, for the update in most real-world networks, the majority of H-vertices remain to be H-vertices. Second, since v was in H, v has a high degree compared with the vertices that are not in H. Thus, v is at the border even it is removed from H and it is possible that v may be qualified to become an H-vertex again soon. Third, the removal of a vertex from H is far more frequent when H shrinks, while H usually shrinks when G shrinks. However, most real-world networks keep growing rather than shrinking.

8.2. Update of H*-Max-Cliques

We first consider the insertion of a new edge $e = (u, v)$ and the possible updates to H^*-max-cliques. First, if $u, v \notin H$, we do not need to update H or T_{H^*}, unless u and/or v now becomes an H-vertex, which is handled in Section 8.1. Next, if $u \in H$ and/or $v \in H$, inserting e creates new H^*-max-clique(s). Let $NB_{uv} = nb(u) \cap nb(v)$ denote the set of common neighbors of u and v. We find the cliques that can form larger cliques with $\{u, v\}$ as $\mathcal{S} = \{C : C \subseteq (C' \cap NB_{uv}), C' \in \mathcal{M}_{H^*}, C \neq \emptyset\}$, which can be obtained easily by traversing T_{H^*}. To ensure the maximality, we take away nonmaximal cliques in $\mathcal{S}$ and get $\mathcal{S}_M = \{C : C \in \mathcal{S}, \nexists C' \in \mathcal{S}$ such that $C' \supset C\}$. Then, for each $C \in \mathcal{S}_M$, we insert $(C \cup \{u, v\})$ into T_{H^*}. We also remove $(C \cup \{u\})$ and/or $(C \cup \{v\})$ from T_{H^*} if they are originally in the tree. Note that if $\mathcal{S} = \emptyset$, then $\{u, v\}$ is maximal and we simply insert $\{u, v\}$ into T_{H^*}.

We now consider deleting an edge $e = (u, v)$. If $u, v \notin H$, there is no update needed for H and T_{H^*}. If $u \in H$ and/or $v \in H$, we need to remove from T_{H^*} all H^*-max-cliques containing both u and v. Thus, we need to find $\mathcal{S}' = \{C : u, v \in C, C \in \mathcal{M}_{H^*}\}$. Assume that $u \prec v$, we can obtain $\mathcal{S}'$ by finding all occurrences of v in the subtree rooted at each occurrence of u in T_{H^*}, and collecting the H^*-max-cliques containing both u and v by traversing the corresponding paths. We remove each $C \in \mathcal{S}'$ from T_{H^*}. We also insert $(C \backslash \{u\})$ and/or $(C \backslash \{v\})$ if they now become maximal.

ALGORITHM 6: *Maximum*

Input: a graph G
Output: the maximum clique C_{max}

1. $C_{max} \leftarrow \emptyset$;
2. **return** $subMaximum(C_{max}, G)$;

8.3. Complexity Analysis

We first analyze the cost of individual updates and then give an analysis on the frequency of the updates.

When a vertex v becomes an H-vertex: if v was not in H_{nb}, we need $\mathcal{O}(nb(v))$ time to insert v and the set of v's children $nb(v)$ into T_{H^*}; if v was in H_{nb}, we need $\mathcal{O}(N(v, T_{H^*}) * (nb(v) \cap H_{nb}))$ time to reconnect v and v's siblings in $(nb(v) \cap H_{nb})$ in T_{H^*}, where $N(v, T_{H^*})$ is the number of occurrences of v in T_{H^*}. Second, the cost of the lazy update is at most the cost of reconstructing T_{H^*}, but the update is rare and can be performed at idle time.

On edge insertion, the cost is $\mathcal{O}(|T_{H^*}| + |\mathcal{S}|^2 + \sum_{C \in \mathcal{S}_M}(|C \cup \{u, v\}| \log f_{avg}))$ time, where f_{avg} is the average number of children of a node in T_{H^*}. Computing $\mathcal{S}$ takes $\mathcal{O}(|T_{H^*}|)$ time. Computing $\mathcal{S}_M$ takes time less than $|\mathcal{S}|^2$ since we do not need to compare cliques with the same size, or those largest cliques in $\mathcal{S}$. In most cases, $|\mathcal{S}|$ is small because otherwise it implies that u and v are very closely related and hence the edge (u, v) is likely to already exist. Finally, inserting each $(C \cup \{u, v\})$ takes at most $\mathcal{O}(\log f_{avg})$ time at each level of T_{H^*}. On edge deletion, it takes $\mathcal{O}(|T_{H^*}| + \sum_{C \in \mathcal{S}'}(|C| \log f_{avg}))$ time to obtain $\mathcal{S}'$ and delete C (as well as to insert $(C \backslash \{u\})$ and/or $(C \backslash \{v\})$ if they are maximal).

Now we examine how frequently these updates are performed. Since we only perform updates related to the H^*-max-cliques, there is no update for the insertion or deletion of an edge (u, v), where $u, v \notin H$. As shown in Section 7.2.2, the size of H, that is, h_{max}, is usually very small compared to the total number of vertices in G. Therefore, the percentage of the updates in G that can "hit" an H-vertex and thus trigger an update in H^*-max-cliques is very low, which is also verified in our experimental studies.

9. FINDING MAXIMUM CLIQUE IN A MASSIVE NETWORK

In this section, we discuss an application of our EmMCE algorithm framework to find the maximum clique in a massive network. A *maximum clique* of a graph G is a clique in G with the largest number of vertices. Finding the maximum clique in a graph is also a long-standing problem in graph theory and has numerous important applications [Bomze et al. 1999]. Most existing algorithms on maximum clique finding are also in-memory ones and unable to handle massive networks.

Our EmMCE algorithm framework provides a general framework for designing an external-memory algorithm for maximum clique computation in large graphs. However, it is not efficient enough if we use a general B^*-graph in the framework, since it requires to enumerate all maximal cliques in order to find the one with the largest size. In the following, we show that with the help of the special version of the B^*-graph, H^*-graph, we are able to find the maximum clique in a more effective way.

We give our algorithm in Algorithm 6. The algorithm recursively invokes Procedure 7 to compute the H^*-graph G_{H^*} at each time. Then, we first check if the value of h_{max} computed from the current input graph G is smaller than the size of the maximum clique C_{max} computed so far. If this is the case, then C_{max} must be the maximum clique globally, because the size of the maximum clique in the current input graph is at most $(h_{max} + 1)$ (see Lemma 9.1). Otherwise, the algorithm computes the maximum clique C'_{max} from G_{H^*} as follows (lines 4–7 of Procedure 7).

PROCEDURE 7: *subMaximum*(C_{max}, G)

```
1. Compute G_{H*} from G;
2. if(h_max < |C_max|)
3.     return C_max;
4. else
5.     Construct T_{B*} from G_{H*} by an existing in-memory MCE algorithm;
6.     Remove G_{H*} from G;
7.     Compute the maximum clique C'_max from T_{B*} in a similar way as Algorithm 3
       but apply pruning by branch-and-bound;
8. if(|C'_max| > |C_max|)
9.     return subMaximum(C'_max, G);
10. else
11.    return subMaximum(C_max, G);
```

The algorithm first constructs T_{B^*} from G_{H^*} by an existing in-memory MCE algorithm. Then, we compute the H^*-max-cliques from T_{B^*} in a similar way as described in Algorithm 3. However, during the process, we apply pruning by branch-and-bound. That is, for any H^*-max-clique to be computed, if the size of its candidate set is not larger than $|C_{max}|$ or the maximum clique C'_{max} computed at the current recursive step, then by the definition of the maximum clique we can safely prune the whole branch to be explored.

Finally, we continue the search in the remaining part of the graph after removing G_{H^*}, by passing the maximum clique we obtain so far to the next recursive step (lines 8–11 of Procedure 7).

In the following lemma we show the connection between the H^*-graph of a graph and the maximum clique in the graph.

LEMMA 9.1. *Given a graph G, let G_{H^*} be the H^*-graph of G and C_{max} be the maximum clique in G. Then, $|C_{max}| \leq (h_{max} + 1)$.*

PROOF. Suppose on the contrary that $|C_{max}| > (h_{max} + 1)$. Since the complete graph induced by C_{max} consists of at least $(h_{max} + 2)$ vertices that have a degree of at least $(h_{max} + 1)$, we have $|H| \geq (h_{max} + 1)$ which contradicts to the fact that $|H| = h_{max}$ according to the definition of G_{H^*}. Therefore, we have $|C_{max}| \leq (h + 1)$. □

With Lemma 9.1, we prove the correctness of Algorithm 6 as follows.

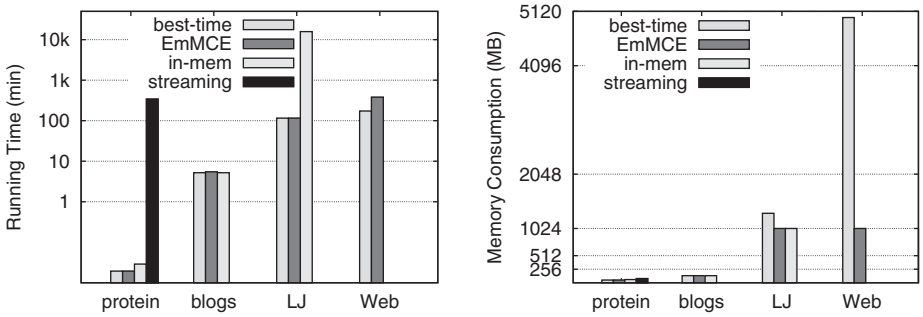
THEOREM 9.2. *Given a graph G, Algorithm* 6 *correctly computes the maximum clique C_{max} in G.*

PROOF. First, according to Lemma 9.1, the size of the maximum clique in the current input graph (i.e., the input graph at the current recursive step) is at most $(h_{max} + 1)$. Since the value of h_{max} with respect to the H^*-graph at the current recursive step is never greater than those at the previous recursive steps, the maximum clique C_{max} computed previously must be the maximum clique globally in the original input graph if $|C_{max}| > h_{max}$ as tested in line 2 of Procedure 7.

Next, lines 5–7 of Procedure 7 compute the maximum clique C'_{max} at the current recursive step. Then, lines 8–11 of Procedure 7 pass the larger clique to the next recursive step. Thus, the clique used for test in line 2 of Procedure 7 is ensured to be the maximum clique computed so far. Therefore, Algorithm 6 computes the maximum clique correctly. □

Table II. Datasets (K = 1,000 and M = 1,000,000)

	protein	*blogs*	*LJ*	*Web*
$n = \|V\|$	20K	1M	4.8M	52.9M
$m = \|E\|$	40K	6.5M	43M	274.8M
Storage size (MB)	1	186	1310	5004



(a) running time (b) memory consumption

Fig. 4. Performance comparison with existing algorithms.

10. EXPERIMENTAL EVALUATION

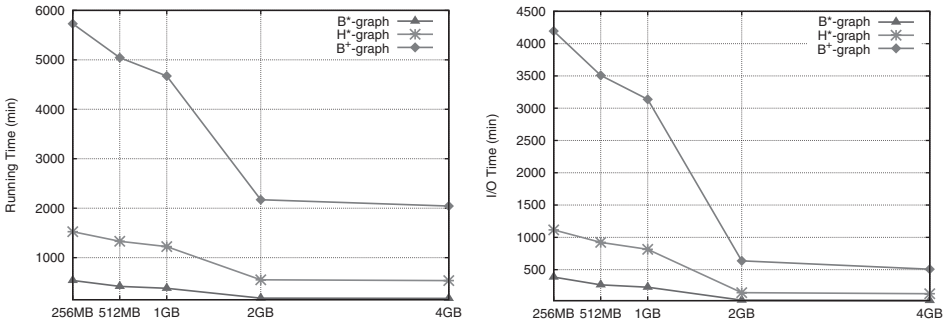
In this section, we evaluate the performance of our algorithm. We compare with the state-of-the-art in-memory algorithm [Tomita et al. 2006] for maximal clique enumeration, and the only existing streaming algorithm for maximal clique enumeration in dynamic networks [Stix 2004], which are denoted as *in-mem* and *streaming* in our experiments, respectively. We ran all experiments on a machine with an Intel Xeon 2.67 GHz CPU and 5GB RAM, running CentOS 5.4 (Linux).

Datasets. We use the following four datasets: *protein*, *blogs*, *LiveJournal* (*LJ*), and *Web*. *Protein* is a human protein interaction network from the Human Protein Database (www.hprd.org), in which vertices are proteins and edges are protein-protein interactions. The *blogs* network is collected from the top-15 popular queries published by Technorati (technorati.com) every three hours from Nov 2006 to Mar 2008. For each query, the top-50 results are retrieved. In the *blogs* network, vertices are blogs and edges indicate that two blogs appear in the search result of the same query. *LJ* is the free online community called LiveJournal, where vertices are members and edges represent friendship between members. LJ is the current largest network available from snap.stanford.edu. The *Web* graph is obtained from the YAHOO Web spam dataset (barcelona.research.yahoo.net/webspam), where vertices are Web pages and edges are hyperlinks. We give the details of each dataset (number of vertices and edges, physical storage size) in Table II.

Among the four datasets, *protein* and *blogs* are small graphs, while *LJ* and *Web* are two larger graphs. We use the smaller graphs to mainly assess how much CPU time our algorithm takes when comparing with an in-memory algorithm, while we use the larger graphs to evaluate the performance of our algorithm under different settings when memory is insufficient to hold the input graph.

10.1. Performance Comparison with Existing Algorithms

We first compare *EmMCE* with *in-mem* and *streaming*. We limit the available memory to 1GB to test the performance of the different algorithms with limited memory resource. For EmMCE, we select the base vertex-set B such that the corresponding B^*-graph fills the available memory as discussed in Section 7.1. Figure 4 reports the



(a) running time (b) I/O time

Fig. 5. Performance on different memory size and algorithm variation.

Table III. Size Estimation for $|T_{B^*}|$

	protein	*blogs*	*LJ*	*Web*
(*estimated* $|T_{B^*}|$)/$|T_{B^*}|$	1.00	1.01	0.93	0.97

total running time (wall-clock time) and peak memory consumption of finding the set of all max-cliques using *EmMCE*, *in-mem*, and *streaming*, respectively.

To fully assess whether *EmMCE* is indeed time efficient, we also report the best running time that we can obtain using any of the three algorithms, by allocating sufficient memory to hold the entire input graph in memory. This result is denoted by *best-time* in the figures.

First, on the smaller networks *protein* and *blogs*, *EmMCE* is as fast as *in-mem* and uses similar amount of memory. The result verifies our assertion in Section 6 that the complexity of *EmMCE* when running in memory is indeed comparable to that of an in-memory algorithm for maximal clique enumeration.

On the larger networks, the advantage of *EmMCE* over *in-mem* is immediately seen. As shown in Figure 4(a), *in-mem* is at least two orders of magnitude slower than *EmMCE* for computing max-cliques from the network *LJ*, when *in-mem* needs more than 1GB of memory. For the larger *Web* graph, we are not able to obtain the result for *in-mem*.

On the contrary, *EmMCE* computes the result for all the datasets with a bounded memory consumption. When the size of the available memory is larger than the size of the input graph, as in the case of *protein* and *blogs*, *EmMCE* only uses as much memory as the in-memory algorithm. When the input graph is larger than the size of the available memory, as in the case of *LJ* and *Web*, *EmMCE* uses all available memory and achieves competitive running time. For the *LJ* graph, the running time of *EmMCE* is comparable to that of *best-time*. For the *Web* graph, the running time of *EmMCE* is about twice that of *best-time*, but it uses only 1GB of memory while *best-time* uses 5GB of memory, and we note that the I/O time accounts for about 60% of the total time (see Figure 5).

We are only able to obtain the result of *streaming* for the smallest network, *protein*, which already takes many orders of magnitude more time to complete. The poor performance of *streaming* is mainly because it reads an edge at a time and updates the current set of max-cliques for each edge. We report this result to demonstrate that although *streaming* reads the graph only once, the time complexity of such a streaming algorithm for max-clique computation is extremely high.

Finally, Table III reports the result of our method of estimating $|T_{B^*}|$, as discussed in Section 7.1. We set $k = 100$, which is used to limit the total number of probing paths

Table IV. Running Time and Memory Consumption of Extracting G_{H^*}

	protein	*blogs*	*LJ*	*Web*
Running time (sec)	0.222	1.29	2.48	29
Memory consumption (MB)	1.2	8.5	27	388

starting from each randomly selected B-vertex. The number of probing paths for this k is about 18% of the total number of paths in T_{B^*} for the *protein* dataset (because its T_{B^*} is too small), but only 0.5% for the other datasets. The result shows that our method of estimating $|T_{B^*}|$ is highly accurate. This result, together with the memory consumption of *EmMCE* for larger datasets, shows that our base vertex selection method is effective and *EmMCE* is able to fully utilize the available memory.

10.2. Performance on Different Memory Size and Algorithm Variation

In this experiment, we assess whether our external-memory algorithm is truly effective by testing the algorithm under a range of memory limits. We set the available memory to 256MB, 512MB, 1GB, 2GB, and 4GB, respectively. We use the largest *Web* dataset in this experiment.

We also test whether using the B^*-graph by filling the available memory (as discussed in Section 7.1) is indeed more effective for maximal clique enumeration on static graphs than using the H^*-graph and the B^+-graph, respectively. Note that by adopting the H^*-graph instead of the B^*-graph, the algorithm is essentially ExtMCE proposed in Cheng et al. [2010], which is a special case of EmMCE. Thus, in Figure 5(a) and Figure 5(b), the labels "B^*-graph" and "H^*-graph" represent EmMCE and ExtMCE [Cheng et al. 2010], respectively.

Figure 5(a) reports the total running time (wall-clock time) of the three variations of the algorithm under the different available memory settings. The result shows that with all the available memory settings, using the B^*-graph is approximately three times faster than using the H^*-graph, and is more than an order of magnitude faster than using the B^+-graph.

Figure 5(b) reports the total I/O time of the three variations of the algorithm. The result shows that when we reduce the size of the available memory, the I/O time increases significantly. When the available memory size is only 256MB, the I/O time is about 60% of the total running time using the B^*-graph; however, when the available memory size increases to 4GB, the I/O time is only 12% of the total running time. The result also shows that using the H^*-graph and the B^+-graph takes significantly more I/O time than using the B^*-graph, which explains why they are much slower than using the B^*-graph. Using the H^*-graph uses more I/Os mainly because it underutilizes memory at the recursive steps. Using the B^+-graph uses more I/Os because a small portion of B-vertices are processed at each recursive step and hence a significantly greater number of scans of the graph is needed, as discussed in Section 4.1.

In summary, this result not only shows that our current algorithm has significantly improved the first and only external-memory algorithm for max-clique enumeration [Cheng et al. 2010], but also demonstrates that the effectiveness of using the B^*-graph rather than the B^+-graph in our framework.

10.3. Evaluation of the H*-Graph

Before we evaluate the performance of update in dynamic networks using the H^*-graph, we first evaluate the quality of the H^*-graph. We set the available memory as 1GB for this set of experiments.

Table IV shows that it is very efficient to extract G_{H^*} from G. The memory consumption is also low.

Table V. Sizes of H, H_{nb}, G_H, G_{H^*} and G_{H^+}

	protein	*blogs*	*LJ*	*Web*
$\|H\|$	77	718	987	3447
$\|H_{nb}\|$	4K	192K	441K	5.6M
$\|G_H\|$	0.5K (1%)	37K (0.6%)	25K (0.06%)	22K (0.01%)
$\|G_{H^*}\|$	8.6K (22%)	840K (13%)	1.7M (4%)	15M (5.5%)
$\|G_{H^+}\|$	21K (54%)	4M (64%)	11M (25%)	28M (10.1%)

Table VI. Closeness, Reachability, and Number of Max-Cliques

	protein	*blogs*	*LJ*	*Web*
closeness (H-vertices)	3.1	3.4	4.3	6.6
reachability (H-vertices)	47%	56%	100%	37%
Number of max-cliques	25K	1.1M	173M	205M
(contain H-vertices)	239	4K	69K	3.0M
(contain H-neighbors)	12K	510K	43M	186M

Table V reports the sizes of H, H_{nb}, G_H, G_{H^*}, and G_{H^+}. We also give a better perception on the sizes of G_H, G_{H^*}, and G_{H^+} as their ratio to G (given in parentheses in the table). For all datasets, H is small but it extends to a much larger H-neighbor set H_{nb}. As a result, G_H is too small, thus requiring many disk scans for MCE computation, while G_{H^+} is too large, thus demanding too much memory. On the contrary, G_{H^*} is much smaller than G_{H^+} but is significantly greater than G_H, thus allowing efficient dynamic update with reasonable memory usage.

Table VI shows the average *closeness* of the H-vertices, the percentage of vertices in G that are reachable from the H-vertices (*reachability*), the number of max-cliques. The closeness of an H-vertex u is defined as $AVG_{v \in V, dist(u,v) \neq \infty}(dist(u, v))$, where $dist(u, v)$ is the length of the shortest path from u to v in G.

The closeness shows that from the H-vertices, we can reach other vertices in G within a few steps and we are able to reach the majority of the vertices in G. This result demonstrates that G_{H^*} represents a significant portion of G and that G_{H^*} also has a close relationship with the rest part of G.

Table VI also reports the number of all max-cliques, the number of those max-cliques containing H-vertices and H-neighbors. The result shows that the number of max-cliques containing H-vertices is significantly smaller than the number of all max-cliques. The result justifies the feasibility of our update strategy based on a much smaller set of cliques containing H-vertices since it is much more efficient. From the H-vertices we can extend to the H-neighbors, while the result shows that the set of max-cliques containing H-neighbors represents a large portion of the whole set of max-cliques.

10.4. Performance on Update in Dynamic Network

Table VII reports the results on the performance of update using the H^*-graph in dynamic graphs. We use the *blogs* network, for which we are able to obtain a timestamp for its edges. The initial network was created in Nov 2006 and the network grows from 347K edges to 6.5M edges over 12 months. We average the results for every two-month period, as represented by P1-P6 in Table VII. We set the available memory as 1GB for this set of experiments.

Table VII shows that the average time of processing an edge insertion that triggers an update in T_{H^*}, shown as "Avg. update time", is only slightly more than 1 millisecond. Although P1 requires 3 milliseconds, this is because the initial network is not large enough and hence T_{H^*} changes considerably during P1, which is also reflected by the rapid increase in the number of H-vertices from P1 to P2.

Table VII. Performance on Update in Dynamic Network

	P1	P2	P3	P4	P5	P6
Avg. update time (msec)	3.0	1.4	1.4	1.2	1.5	1.6
Number of updates in G_{H^*}	3K	11K	19K	25K	28K	28K
Number of updates in G	385K	457K	550K	461K	526K	670K
Number of H-vertices	294	425	508	566	614	696
% of H-vertices retained	92	92	95	96	94	96
Memory Consumption (MB)	418	427	436	443	451	463
MCE time w/ T_{H^*} (sec)	7.3	13.4	27.4	41.5	52.4	69.5
MCE time w/o T_{H^*} (sec)	22	37.6	63.6	86.6	107.8	137.9

Table VIII. Performance on Update Using T_{H^*} and T_{B^*}

	P1	P2	P3	P4	P5	P6
Avg. update time w/ T_{H^*} (msec)	3.0	1.4	1.4	1.2	1.5	1.6
Avg. update time w/ T_{B^*} (msec)	4.7	5.4	8.8	7.3	4.2	5.4
MCE time w/ T_{H^*} (sec)	7.3	13.4	27.4	41.5	52.4	69.5
MCE time w/ T_{B^*} (sec)	14.7	19.2	69.8	70.6	75.6	75.7

Table VII also shows "Number of updates in G_{H^*}", which is the number of edge insertions that trigger an update in T_{H^*}, and "Number of updates in G", which is the number of all edges inserted into the network. On average, the percentage of edges that trigger an update in T_{H^*} is only 3.8%, which is a very small portion of the total updates. Thus, updating only T_{H^*} is a feasible solution to handle frequent updates.

Among the existing algorithms, *streaming* is the only one that updates the set of max-cliques upon each edge insertion. However, *streaming* is three orders of magnitude slower than our algorithm on average. We do not report the result for *streaming* because it takes too long to complete all updates (it takes 190 hours to update only 40K edges).

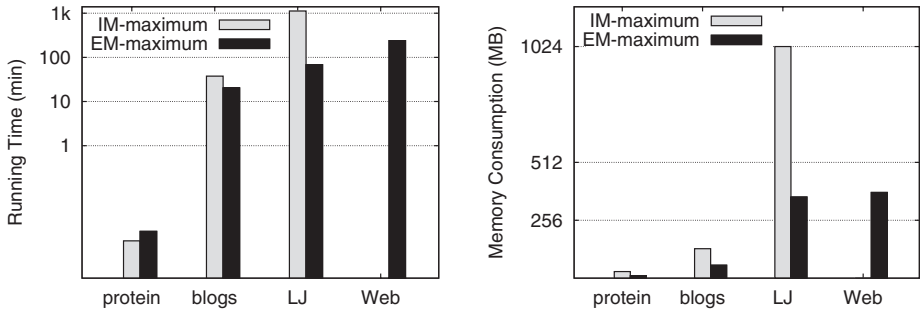
The number of H-vertices increases stably as the network increases, except the initial network which is relatively small and thus unstable. We also show % of H-vertices retained, that is, the percentage of H-vertices in P_i that are also in P_{i+1}. The result shows that the majority of the H-vertices remains to be H-vertices.

We also show the memory consumption, which increases as the network grows. The last two rows of Table VII report the time to compute the set of all max-cliques from the dynamically maintained T_{H^*} ("Time w/ T_{H^*}") and from scratch ("Time w/o T_{H^*}"), respectively. The result shows that it is much more efficient to compute the set of all max-cliques from the dynamically maintained T_{H^*} than from scratch from the network, thus demonstrating the benefit of update maintenance as well as the feasibility of maintaining $\mathcal{M}_{H^*}$ (i.e., T_{H^*}) for $\mathcal{M}$.

Table VIII shows the performance on update if we adopt B (as discussed in Section 7.1) instead of H in the update framework. The result shows that the update time of using T_{B^*} is up to 6.4 times (average 4 times over P1-P6) longer than using T_{H^*}, because the update is now performed over a large scattered set of B-vertices rather than a small concentrated set of high-degree H-vertices. The result also shows that computing the set of all max-cliques from T_{B^*} is on average 1.7 times slower than from T_{H^*}. Thus, for update in dynamic networks, using T_{H^*} is preferred to using T_{B^*}.

10.5. Performance on Maximum Clique Computation

In this subsection, we report the performance of our algorithm on maximum clique computation. We name our algorithm as *EM-maximum* and compare with its in-memory counterpart, named as *IM-maximum*, which is a typical branch-and-bound algorithm for maximum clique computation [Tomita and Seki 2003]. We report the total



(a) running time (b) memory consumption

Fig. 6. Performance on maximum clique computation.

Table IX. Size of Maximum Clique

	protein	*blogs*	*LJ*	*Web*
Maximum clique size	11	49	13	478

running time (wall-clock time) and peak memory consumption of the two algorithms in Figure 6.

As shown in Figure 6, *EM-maximum* is at least two times faster than *IM-maximum* on the smaller networks *protein* and *blogs*, with only half of the memory consumption as that of *IM-maximum*. On the larger network *LJ*, *EM-maximum* is approximately 20 times faster than *IM-maximum* (note that Figure 6(a) is in log scale) and consumes significantly less memory. On the *Web* graph, we are not able to obtain the result for *IM-maximum* due to its high memory consumption, while *EM-maximum* finds the maximum clique efficiently using only 388MB of memory. This result demonstrates the effectiveness of our algorithm framework for finding even the maximum clique, in addition to the set of all maximal cliques.

Finally, we also report the size of the maximum clique in each graph in Table IX for reference.

11. RELATED WORK

There is a large literature on maximal clique enumeration. We discuss the more prominent and recent ones, while comprehensive reviews can be found in Bomze et al. [1999] and Cazals and Karande [2008]. The review on the problem of finding the maximum clique, including its applications, can also be found in Bomze et al. [1999].

The first significant improvement on maximal clique enumeration was the algorithms [Akkoyunlu 1973; Bron and Kerbosch 1973] that use the *backtracking* method. They take $\mathcal{O}(n^2)$ memory space. Further improvements [Koch 2001; Tomita et al. 2006; Cazals and Karande 2008] were made by selecting good *pivots* to prune the backtracking search tree. The optimal worst-case time of backtracking-based maximal clique computation was shown to be $\mathcal{O}(3^{n/3})$ [Tomita et al. 2006]. Recently, parallel algorithms [Du et al. 2009; Schmidt et al. 2009] were proposed to compute maximal cliques from different points of the search tree in parallel. However, all these studies did not focus on reducing the memory complexity and require $\mathcal{O}(m+n)$ memory space in the best case. Algorithm for output-sensitive maximal clique enumeration was also introduced [Tsukiyama et al. 1977] which is based on *reverse search*, and recent work [Makino and Uno 2004] used matrix multiplication to reduce the time delay to $\mathcal{O}(d_{max}^4)$ for sparse graphs (but with $\mathcal{O}(nm)$ preprocessing time), where d_{max} is the maximum degree of a graph. Other

algorithms, such as obtaining a k-clique by joining two $(k-1)$-cliques [Kose et al. 2001], by making use of triangles [Wan et al. 2006], by limiting the minimum size [Modani and Dey 2008], were also proposed. However, all these algorithms require memory space at least $\Omega(m+n)$. Stix [2004] proposed an algorithm that updates the set of max-cliques upon each edge insertion, and the graph is read only once.

Apart from the aforementioned in-memory algorithms, the first external-memory algorithm was proposed recently by Cheng et al. [2010]. However, the algorithm in Cheng et al. [2010] only allows the H^*-graph to be the subgraph being processed at each recursive step. In this article, on the contrary, we propose a new general framework that allows the flexibility of selecting the subgraph for the local max-clique computation. Thus, within our general framework, we can consider the H^*-graph as a special version of the B^*-graph, which is specifically applied to design an efficient update strategy for maintaining maximal cliques in dynamic graphs. However, in addition to the scheme for processing dynamic graphs, with the general framework we also devise a scheme to select a subgraph that fully utilizes the available memory at each recursive step, in order to minimize the I/O cost for computing maximal cliques in static graphs. Consequently, the new general framework allows us to incorporate the work of Cheng et al. [2010] and the work in this article to handle both the cases of static graphs and dynamic graphs. More extensive experimental results on larger graphs also show that our new algorithm achieves better performance. Furthermore, this article also applies the general framework, with the adoption of the H^*-graph, to design the first external-memory algorithm for computing the maximum clique in a large graph, an equally important problem with numerous applications [Bomze et al. 1999]. Finally, the proposal of a general algorithm framework has the potential to be adopted for designing external-memory algorithms for solving other related problems such as independent set and matching on massive graphs.

12. CONCLUSIONS

We present an external-memory algorithm, EmMCE, for maximal clique enumeration in large graphs. We evaluate the performance of EmMCE on large real networks of size up to 52.9 million vertices and 274.8 million edges, by comparing with the state-of-the-art in-memory algorithms. Our results demonstrate the effectiveness of our framework in the following four aspects. First, for maximal clique enumeration, EmMCE achieves comparable performance with the in-memory algorithm when the input graph can fit in memory, and proves to be highly efficient when the input graph cannot fit in memory and is too expensive to be processed by the in-memory algorithm. Second, EmMCE is I/O-efficient under different bounds on the available memory size and significantly outperforms the algorithm in Cheng et al. [2010]. Third, dynamic maintenance using the H^*-graph, a special version of the B^*-graph, is efficient in dynamic networks. Lastly, our external-memory algorithm for maximum clique computation is significantly more efficient than the in-memory counterpart, whether or not the input graph can fit in memory.

With the result in our article, we are able to process MCE even for very large networks that cannot be processed in most commodity PCs using the conventional in-memory algorithms. Moreover, our algorithm naturally allows parallel computation of MCE, which is useful for processing MCE in modern computing environments such as the cloud.

APPENDIX

This appendix gives the proofs to Lemmas 5.6, 5.8, and 5.10, which are introduced in Section 5.2.1.

LEMMA 5.6. *Let* $\mathcal{M}_1 = \mathcal{M}_B \cap \mathcal{M}_{B^*}$. *Then,* $\mathcal{M}_1 = \mathcal{M}^1_{B^+}$.

PROOF. (Prove $\mathcal{M}_1 \subseteq \mathcal{M}^1_{B^+}$). Let C be a clique in $\mathcal{M}_1$. Since $C \in (\mathcal{M}_B \cap \mathcal{M}_{B^*})$, C contains only B-vertices and is maximal in G_{B^*}, which means that the vertices in C do not have any common B-neighbors (i.e., $comNB(C) = \emptyset$ and thus $C_{B_{nb}} = \emptyset$). Since $B^+ = (B \cup B_{nb})$, C is also maximal in G_{B^+}. Since $C_{B_{nb}} = \emptyset$, we have $C \in \mathcal{M}^1_{B^+}$.

(Prove $\mathcal{M}^1_{B^+} \subseteq \mathcal{M}_1$). $\forall C \in \mathcal{M}^1_{B^+}$, we have $C_{B_{nb}} = \emptyset$, which implies that $C = C_B$ and $C \in \mathcal{M}_B$. We have $C \in \mathcal{M}_{B^*}$ as well since $C_{B_{nb}} = \emptyset$. Thus $C \in (\mathcal{M}_B \cap \mathcal{M}_{B^*}) = \mathcal{M}_1$. □

LEMMA 5.8. *Let* $\mathcal{M}_2 = \{C_1 \cup C_2 : C_1 \in (\mathcal{M}_B \backslash \mathcal{M}_1), C_2 \in maxCL(comNB(C_1))\}$. *Then,* $\mathcal{M}_2 = \mathcal{M}^2_{B^+}$.

PROOF. It is obvious that all elements in $\mathcal{M}_2$ are cliques by the definitions of $comNB(\cdot)$ and $maxCL(\cdot)$.

(Prove $\mathcal{M}_2 \subseteq \mathcal{M}^2_{B^+}$). $\forall C = (C_1 \cup C_2) \in \mathcal{M}_2$, we have $C_B = C_1$ and $C_{B_{nb}} = C_2$. Since $C_B \in \mathcal{M}_B$, C_B is maximal in G_B. Since $C_{B_{nb}} \in maxCL(comNB(C_B))$, $C_{B_{nb}}$ is also maximal in $G_{comNB(C_B)}$, which defines the B-neighborhood shared by all vertices in C_B. Thus, C is maximal in G_{B^+}, that is, $C \in \mathcal{M}_{B^+}$. Since $C_B \notin \mathcal{M}_1$, we have $comNB(C_B) \neq \emptyset$ and thus $C_{B_{nb}} \neq \emptyset$. Since $C \in \mathcal{M}_{B^+}$, $C_B \in \mathcal{M}_B$, and $C_{B_{nb}} \neq \emptyset$, we have $C \in \mathcal{M}^2_{B^+}$.

(Prove $\mathcal{M}^2_{B^+} \subseteq \mathcal{M}_2$). $\forall C \in \mathcal{M}^2_{B^+}$, $C_{B_{nb}} \neq \emptyset$ and $C_B \in \mathcal{M}_B$. Thus $C_B \in (\mathcal{M}_B \backslash \mathcal{M}_1)$. Since C is maximal in G_{B^+}, $C_{B_{nb}}$ must be maximal in $G_{comNB(C_B)}$, that is, $C_{B_{nb}} \in maxCL(comNB(C_B))$. Let $C_1 = C_B$ and $C_2 = C_{B_{nb}}$, we have $C = (C_1 \cup C_2) \in \mathcal{M}_2$. □

LEMMA 5.10. *Let* $\mathcal{M}_3 = \{C_1 \cup C_2 : C_1 \in \mathcal{X}, C_2 \in EXT(C_1)\}$. *Then,* $\mathcal{M}_3 = \mathcal{M}^3_{B^+}$.

PROOF. By the definitions of $\mathcal{X}$ and $EXT(C_1)$, an element $C \in \mathcal{M}_3$ must be a clique.

(Prove $\mathcal{M}_3 \subseteq \mathcal{M}^3_{B^+}$). We first prove $\mathcal{M}_3 \subseteq \mathcal{M}_{B^+}$ by contradiction. Suppose $\exists C = (C_1 \cup C_2) \in \mathcal{M}_3$ such that $C \notin \mathcal{M}_{B^+}$, that is, $\exists C' = (C'_B \cup C'_{B_{nb}}) \in \mathcal{M}_{B^+}$ such that $C' \supset C$. We have $C'_B \supseteq C_B = C_1$ and $C'_{B_{nb}} \supseteq C_{B_{nb}} = C_2$. Assume that $C'_B = C_B$, then $C'_{B_{nb}} = C_{B_{nb}}$ since $C_{B_{nb}}$ is maximal in $G_{comNB(C_B)} (= G_{comNB(C'_B)})$ as defined in $EXT(C_B)$. This implies that $C' = C$ and contradicts to $C' \supset C$. Thus, we are left with the option that $C'_B \supset C_B$, which implies that $comNB(C'_B) \subseteq comNB(C_B)$. Since $C'_{B_{nb}} \supseteq C_{B_{nb}}$ and they are maximal respectively in $G_{comNB(C'_B)}$ and $G_{comNB(C_B)}$, but $comNB(C'_B) \subseteq comNB(C_B)$, we have $C'_{B_{nb}} = C_{B_{nb}}$ and $comNB(C'_B) = comNB(C_B)$. Since $C'_B \supset C_B$ and $comNB(C'_B) = comNB(C_B)$, we have $C'_B \notin \mathcal{X}$ (otherwise, $C_B = C_1 \notin \mathcal{X}$ since C_B is subsumed by C'_B, but $C_1 \in \mathcal{X}$ by the definition of $\mathcal{M}_3$). Therefore, C'_B can only be in $\mathcal{M}_B$. Since $C'_{B_{nb}} = C_{B_{nb}} \neq \emptyset$, we have $C' \in \mathcal{M}_2$. This contradicts $C_{B_{nb}} \in EXT(C_B)$ since there exists $C' \in \mathcal{M}_2$ such that $C'_B \supset C_B$ and $C'_{B_{nb}} = C_{B_{nb}}$, that is, $C' \supset (C_B \cup C_{B_{nb}})$. Thus, we prove $\mathcal{M}_3 \subseteq \mathcal{M}_{B^+}$. Finally, $\forall C \in \mathcal{M}_3$, we have $C_{B_{nb}} \neq \emptyset$ since $\emptyset \notin EXT(C_B)$, and $C_B \notin \mathcal{M}_B$ since $C_B \in \mathcal{X}$ and C_B is a proper subset of some $C'' \in \mathcal{M}_B$. Thus we further have $\mathcal{M}_3 \subseteq \mathcal{M}^3_{B^+}$.

(Prove $\mathcal{M}^3_{B^+} \subseteq \mathcal{M}_3$). $\forall C \in \mathcal{M}^3_{B^+}$, $C_B \notin \mathcal{M}_B$ and $C_{B_{nb}} \neq \emptyset$. First, C_B must be a proper subset of some $C' \in \mathcal{M}_B$. Assume that $C_B \notin \mathcal{X}$, then there must exist some $C'_B \in \mathcal{X}$ such that $C'_B \supset C_B$ and $comNB(C'_B) = comNB(C_B)$, which leads to the formation of a clique that is a proper superset of C and contradicts to the maximality of C. Thus $C_B \in \mathcal{X}$. We further have $C_{B_{nb}} \in EXT(C_B)$ since $C_{B_{nb}}$ is maximal in $G_{comNB(C_B)}$ by the maximality requirement of C. Thus $C \in \mathcal{M}_3$. □

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their constructive comments.

REFERENCES

ABU-KHZAM, F. N., BALDWIN, N. E., LANGSTON, M. A., AND SAMATOVA, N. F. 2005. On the relative efficiency of maximal clique enumeration algorithms, with applications to high-throughput computational biology. In *Proceedings of the International Conference on Research Trends in Science and Technology*.

AKKOYUNLU, E. A. 1973. The enumeration of maximal cliques of large graphs. *SIAM J. Comput. 2,* 1, 1–6.

BERNARD, H. R., KILLWORTH, P. D., AND SAILER, L. 1979. Informant accuracy in social network data iv: a comparison of clique-level structure in behavioral and cognitive network data. *Social Netw. 2,* 3, 191–218.

BERRY, N. M., KO, T. H., MOY, T., SMRCKA, J., TURNLEY, J., AND WU, B. 2004. Emergent clique formation in terrorist recruitment. In *Proceedings of the AAAI-04 Workshop on Agent Organizations: Theory and Practice*.

BOGINSKI, V., BUTENKO, S., AND PARDALOS, P. M. 2005. Statistical analysis of financial networks. *Comput. Statist. Data Anal. 48,* 2, 431–443.

BOMZE, I. M., BUDINICH, M., PARDALOS, P. M., AND PELILLO, M. 1999. The maximum clique problem. In *Handbook of Combinatorial Optimization*. Kluwer Academic Publishers, 1–74.

BRON, C. AND KERBOSCH, J. 1973. Algorithm 457: finding all cliques of an undirected graph. *Comm. ACM 16,* 9, 575–577.

BYSKOV, J. M. 2003. Algorithms for k-colouring and finding maximal independent sets. In *Proceedings of the Symposium on Discrete Algorithms (SODA)*. 456–457.

CAZALS, F. AND KARANDE, C. 2008. A note on the problem of reporting maximal cliques. *Theor. Comput. Sci. 407,* 1-3, 564–568.

CHENG, J., KE, Y., FU, A. W.-C., YU, J. X., AND ZHU, L. 2010. Finding maximal cliques in massive networks by h*-graph. In *Proceedings of the SIGMOD International Conference on Management of Data*. 447–458.

CREAMER, G., ROWE, R., HERSHKOP, S., AND STOLFO, S. J. 2007. Segmentation and automated social hierarchy detection through email network analysis. In *Proceedings of WebKDD/SNA-KDD*. 40–58.

DOROGOVTSEV, S. N. AND MENDESAND, J. F. F. 2003. *Evolution of Networks: From Biological Nets to the Internet and www.* Oxford University Press.

DU, N., WU, B., XU, L., WANG, B., AND XIN, P. 2009. Parallel algorithm for enumerating maximal cliques in complex network. In *Mining Complex Data*, 207–221.

FALOUTSOS, M., FALOUTSOS, P., AND FALOUTSOS, C. 1999. On power-law relationships of the internet topology. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures and Protocols for Computer Communications (SIGCOMM)*. 251–262.

FAUST, K. AND WASSERMAN, S. 1995. *Social Network Analysis: Methods and Applications*. Cambridge University Press.

GOUDA, K. AND ZAKI, M. J. 2001. Efficiently mining maximal frequent itemsets. In *Proceedings of the International Conference on Data Mining (ICDM)*. 163–170.

KILBY, P., SLANEY, J. K., THIÉBAUX, S., AND WALSH, T. 2006. Estimating search tree size. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*.

KNUTH, D. E. 1975. Estimating the efficiency of backtrack programs. *Math. Comput. 29,* 129, 121–136.

KOCH, I. 2001. Enumerating all connected maximal common subgraphs in two graphs. *Theor. Comput. Sci. 250,* 1-2, 1–30.

KOSE, F., WECKWERTH, W., LINKE, T., AND FIEHN, O. 2001. Visualizing plant metabolomic correlation networks using clique-metabolite matrices. *Bioinf. 17,* 12, 1198–1208.

MAKINO, K. AND UNO, T. 2004. New algorithms for enumerating all maximal cliques. In *Proceedings of the Scandinavian Workshop on Algorithms Theory (SWAT)*. 260–272.

MODANI, N. AND DEY, K. 2008. Large maximal cliques enumeration in sparse graphs. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*. 1377–1378.

MOHSENI-ZADEH, S., BRÉZELLEC, P., AND RISLER, J.-L. 2004. Cluster-c, an algorithm for the large-scale clustering of protein sequences based on the extraction of maximal cliques. *Comput. Biol. Chemist. 28,* 3, 211–218.

NEWMAN, M. E. J. 2003. The structure and function of complex networks. *SIAM Rev. 45*, 167–256.

SCHMIDT, M. C., SAMATOVA, N. F., THOMAS, K., AND PARK, B.-H. 2009. A scalable, parallel algorithm for maximal clique enumeration. *J. Parall. Distrib. Comput. 69,* 4, 417–428.

STIX, V. 2004. Finding all maximal cliques in dynamic graphs. *Comput. Optimiz. Appl. 27*, 173–186.

TOMITA, E. AND SEKI, T. 2003. An efficient branch-and-bound algorithm for finding a maximum clique. *Discr. Math. Theoret. Comput. Sci. 2731*, 278–289.

TOMITA, E., TANAKA, A., AND TAKAHASHI, H. 2006. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci. 363,* 1, 28–42.

TSUKIYAMA, S., IDE, M., ARIYOSHI, H., AND SHIRAKAWA, I. 1977. A new algorithm for generating all the maximal independent sets. *SIAM J. Comput. 6,* 3, 505–517.

WAN, L., WU, B., DU, N., YE, Q., AND CHEN, P. 2006. A new algorithm for enumerating all maximal cliques in complex network. In *Proceedings of the International Conference on Advanced Data Mining and Applications (ADMA)*. 606–617.

ZHANG, B., PARK, B.-H., KARPINETS, T. V., AND SAMATOVA, N. F. 2008. From pull-down data to protein interaction networks and complexes with biological relevance. *Bioinf. 24,* 7, 979–986.

Received October 2010; revised February 2011; accepted April 2011