Finding Maximal Cliques in Massive Networks by H*-graph

James Cheng
School of Computer Engineering
Nanyang Technological University, Singapore
j.cheng@acm.org

Yiping Ke
Department of Systems Engineering and Engineering Management
The Chinese University of Hong Kong
ypke@se.cuhk.edu.hk

Ada Wai-Chee Fu
Department of Computer Science and Engineering
The Chinese University of Hong Kong
adafu@cse.cuhk.edu.hk

Jeffrey Xu Yu
Department of Systems Engineering and Engineering Management
The Chinese University of Hong Kong
yu@se.cuhk.edu.hk

Linhong Zhu
School of Computer Engineering
Nanyang Technological University, Singapore
zhul0003@ntu.edu.sg

ABSTRACT

Maximal clique enumeration (**MCE**) 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 network graphs. Since MCE requires random access to different parts of a large graph, it is difficult 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. We propose a novel notion, *H^*-graph*, which defines the core of a network and extends to encompass the neighborhood of the core for MCE computation. We propose the first external-memory algorithm for MCE (**ExtMCE**) that uses the H^*-graph to bound the memory usage. We prove both the correctness and completeness of the result computed by ExtMCE. Extensive experiments verify that ExtMCE efficiently processes large networks that cannot be fit in the memory. We also show that the H^*-graph captures important properties of the network; thus, updating the maximal cliques in the H^*-graph retains the most essential information, with a low update cost, when it is infeasible to perform update on the entire network.

Categories and Subject Descriptors

G.2.2 [**DISCRETE MATHEMATICS**]: Graph Theory—*Graph algorithms*

General Terms

Algorithms, Experimentation, Performance

Keywords

Maximal Clique Enumeration, Massive Networks, Scale-Free Networks, H*-graph, h-index

1. INTRODUCTION

Maximal clique enumeration (**MCE**) [2, 7] 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) [28], graph coloring [8], maximal common induced subgraphs [19], maximal common edge subgraphs [19], 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 [15], hierarchy detection through email networks [10], study of structures in behavioral and cognitive networks [3], statistical analysis of financial networks [6], clustering in dynamic networks [26], the detection of emergent patterns in terrorist networks [4], as well as various applications in computational biology [1], protein-protein interaction complex detection [29], and clustering protein sequences [23].

MCE algorithms have been extensively studied [2, 7, 28, 20, 19, 21, 26, 1, 27, 9, 12, 25]. The worst-case time complexity of *in-memory algorithms* have proved to be optimal recently [27]; however, the best existing algorithms require space which 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 fast rate. For example, the Web graph has over 1 trillion webpages (Google), 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.

Despite the low cost of memory, applying in-memory algorithms is clearly infeasible on such massive data. For large graphs, *external memory algorithms* offer a possible recourse; however, designing such an algorithm is fraught with difficulties. MCE computations access vertices in a rather arbitrary manner. This potential random disk access requirement makes it difficult to divide the graph and process it 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 paper, we develop the first external-memory algorithm for MCE (**ExtMCE**) which operates on the broad class of *scale-free* graphs [24, 11]. Extensive studies [14, 24, 6, 5] have shown that scale-free graphs are prevalent in real-world applications. In particular, Dorogovtsev and Mendesand [11] show that a wide spectrum of real-world networks are scale-free, which include the WWW, citation networks, collaboration networks, neural networks, metabolic reaction networks, genome and protein networks, ecological and food webs, word web of human language, telephone call graph, mail networks, power grids and industrial networks, electronic circuits, nets of software components, and energy landscape networks.

Given a large input graph G, ExtMCE recursively computes a portion of G at a time, such that each portion can be fit into the main memory for MCE computation. Two questions arise: (1) What portion of G should be chosen at each recursive step and how? (2) How to ensure that the set of maximal cliques computed locally at recursive steps is sound and complete with respect to the set of maximal cliques globally in G?

To answer the first question, we propose the novel notion of H^*-*graph*. The key component of the H^*-graph is the *largest* set of h vertices in G that have degree at least h, called the h-*vertices*. This is inspired by the concept of h-*index* [16], which is the *maximum* h for a scientist who has h publications of citations at least h. The h-index is widely used to assess a scientist's publication productivity and quality. Putting into the context of a graph, the h-vertices correspond to the h publications that contribute to the h-index. The induced subgraph of G by the h-vertices constitutes the *core* of the H^*-graph. Then, we extend from this core to their neighborhood and thus form the H^*-graph.

The H^*-graph is the key component in the first step of ExtMCE and sets the limit on the memory usage for all subsequent steps. To this end, we need to first make sure that the H^*-graph is small enough to be kept in the memory. For a general graph, the H^*-graph can spread to cover a large part of the graph so that it may be too large to fit into the memory. However, we show that for scale-free graphs, the H^*-graph is only a small portion of the entire graph. Furthermore, we derive bounds on the size of the H^*-graph for scale-free networks, which are also bounds on the memory requirement of ExtMCE. We also devise a method to handle the case that even the H^*-graph cannot be fit into the memory.

Now we answer the second question. MCE computation involves random accesses to all parts of G; therefore, if we take any part of G and compute MCE on each part, then either the set of cliques computed may not be complete and may contain non-maximal ones, or the cost of merging the results from each part and ensuring completeness is substantially high.

It is challenging in linking the MCE computation from one part of G to the other parts while ensuring the correctness and completeness of the result. Let G_i be the part of G at the i-th recursive step of ExtMCE (thus, G_1 is the H^*-graph). We formulate G_i in such a way that it allows smooth transition from G_i to G_{i+1} so that we can compute a subset of maximal cliques in one part of G, and then move on to another part until we finish the entire G. We prove that the maximal cliques computed in each *local* part are indeed maximal *globally* in the entire graph. We then prove that the set of maximal cliques computed is also *complete*.

Real-world networks undergo frequent updates. However, there is only one known algorithm [26] that can be applied to update the set of maximal cliques when a network is updated. Their algorithm is impractical for large networks because the set of maximal cliques in a large network is too large to be kept in the memory, while keeping them on the disk results in extremely high update cost.

We take a new approach. We show that the H^*-graph captures many important properties of the original network, while the set of maximal cliques computed from the H^*-graph are those that consist of the most important vertices in the network. Therefore, we propose to update only the maximal cliques in the H^*-graph, which can be processed in the memory due to the much smaller number. Furthermore, we show that the portion of the updates in G that are related to the H^*-graph is very small and thus by focusing the update maintenance on the H^*-graph, we can significantly reduce the overall update cost. Given the up-to-date H^*-graph, we then re-compute the whole set of maximal cliques in G on demand.

Finally, we conduct experiments on a set of large real-world networks, with size up to about 10 million vertices and 80 million edges, collected from different domains. Our results verify that the H^*-graph represents a significant portion of the original network, and it is effective to use the H^*-graph to bound the memory usage in ExtMCE. We demonstrate that ExtMCE uses comparable time, but significantly less memory, as compared with the state-of-the-art in-memory MCE algorithm [27]. When the memory is not sufficient, ExtMCE still computes MCE efficiently with a bounded memory usage. Our results also verify that our approach of update maintenance is effective, and significantly more efficient than [26].

Contributions. We summarize our main contributions as follows.

- We propose ExtMCE, the first external-memory algorithm for MCE computation. We prove both the correctness and completeness of the result computed by ExtMCE.

- We propose the novel notion of the H^*-graph, which is used to bound the memory usage as well as to guide the recursive steps of ExtMCE. We derive bounds on the size of the H^*-graph.

- We propose the first feasible solution for update maintenance of MCE in large networks. We show that by updating the maximal cliques in the H^*-graph, we retain the essential information in the entire network, leading to a low update cost.

Organization. Section 2 formally defines the problem and gives the basic notations. Section 3 presents the H^*-graph. Section 4 details the ExtMCE algorithm. Section 5 discusses update maintenance. Section 6 reports the experimental results. Section 7 discusses the related work. Section 8 gives the conclusion.

2. PROBLEM DEFINITION

In this paper, we focus on large graphs whose degree distribution follows a *power law*, or called *scale-free networks* [14, 24]. Let $G = (V, E)$ be an undirected and unlabeled graph. We define $n = |V|$ and $m = |E|$. We define the *size* of G, denoted as $|G|$, as $|G| = m$. Given $S \subseteq V$, 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 by C is a complete graph in G. 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 paper, we solve the problem of MCE for large scale-free graphs that cannot be fit in the main memory.

Table 1 shows the notations used frequently in the paper.

Table 1: Notations

Symbol	Description
n	Number of vertices in graph $G = (V, E)$
m	Number of edges in graph $G = (V, E)$
$\lvert G\rvert$	Size of G, defined as $\lvert G\rvert = \lvert E\rvert = m$
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
H	The set of h-vertices in G; $\forall v \in H, d(v) \geq h$
H_{nb}	The set of h-neighbors in G (non-h-vertices)
H^+	$H \cup H_{nb}$
G_H ; G_{H+}	H-graph / H^+-graph; the induced subgraph of G by H / H^+
G_{H^*}	H^*-graph; $G_{H^*} = (H^+, E_{HH} \cup E_{HH_{nb}})$
$\mathcal{M}$	the set of max-cliques in the whole graph G
$\mathcal{M}_X$	the set of X-max-cliques in G_X, X can be H^*, H^+, or H
T_{H^*}	H^*-max-clique tree; a prefix-tree to keep $\mathcal{M}_{H^*}$
C=$(C_H \cup C_{H_{nb}})$	for a clique C in G_{H+}: C_H=$(C \cap H)$; $C_{H_{nb}}$=$(C \cap H_{nb})$
$HNB(X)$	the set of common h-neighbors of all vertices in X, where X is a clique in G_H
$maxCL(S)$	the set of all max-cliques in G_S
$\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3$	three disjoint subsets of $\mathcal{M}_{H+}$, defined in Lemmas 4-6
$\mathcal{X}$	a set of "H" parts used to form cliques in $\mathcal{M}_3$, see Eq. (10)
$EXT(C)$	a set of h-neighbors used to extend $C \in \mathcal{X}$, see Eq. (11)

3. THE H*-GRAPH

In this section, we introduce a novel concept of H^*-graph for real-world networks. The H^*-graph plays a crucial role in the first recursive step of our ExtMCE algorithm (details in Section 4).

3.1 The notion of the H*-graph

We first define the set of h-vertices which forms the core of the H^*-graph.

Definition 1 (h-VERTICES). *Given a graph $G = (V, E)$, the set of h-vertices of G, denoted as H, is defined as $H = \{v : v \in V, d(v) \geq h\}$ such that $|H| = h$, and $\forall v \in (V \backslash H)$, $d(v) \leq h$.*

Essentially, the set of h-vertices of G consists of h vertices in G that have a degree of at least h. From the h-vertices, we extend to the h-neighbors defined as follows. Note that h-neighbors are defined to be non-h-vertices.

Definition 2 (h-NEIGHBORS). *The set of h-neighbors, denoted as H_{nb}, is defined as $H_{nb} = \{v : v \in (nb(u)\backslash H), u \in H\}$.*

We use a notation H^+ to denote the union of the sets of h-vertices and h-neighbors. The "$+$" sign is used to indicate the extension from the h-vertices to the h-neighbors.

Definition 3. $H^+ = H \cup H_{nb}$.

With the set of h-vertices H, we define the concept of H-graph in G as follows.

Definition 4 (H-GRAPH). *The H-graph of a graph G, denoted as G_H, is defined as the induced subgraph of G by H.*

Similarly, we define the concept of H^+-graph.

Definition 5 (H^+-GRAPH). *The H^+-graph of a graph G, denoted as G_{H^+}, is defined as the induced subgraph of G by H^+.*

With the H-graph and the H^+-graph, we now define the notion of H^*-graph. Intuitively, H^*-graph is a graph that "lies" between H-graph and H^+-graph.

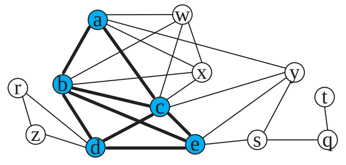


Figure 1: An Example Graph G

Definition 6 (H^*-GRAPH). *Given a graph $G = (V, E)$, the H^*-graph of G, denoted as G_{H^*}, is defined as $G_{H^*} = (H^+, E_{HH} \cup E_{HH_{nb}})$, where $E_{HH} = \{(u, v) : u, v \in H, (u, v) \in E\}$ and $E_{HH_{nb}} = \{(u, v) : u \in H, v \in H_{nb}, (u, v) \in E\}$.*

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

We use the following example to illustrate these basic concepts.

Example 1. Figure 1 gives an example graph G, which contains 13 vertices and 25 edges. The set of h-vertices in G is $H = \{a, b, c, d, e\}$, which means that $h = 5$. It can be easily checked in the figure that all the 5 vertices in H (shaded vertices) have degree at least 5 and all the remaining vertices in G have degree less than 5. The set of h-neighbors is given as $H_{nb} = \{r, s, w, x, y, z\}$. And $H^+ = \{a, b, c, d, e, r, s, w, x, y, z\}$. The two vertices q and t are not in H^+ since they are not incident to any vertex in H. The H-graph consists of the shaded vertices and bold edges in Figure 1, which is the induced subgraph of G by H. The H^+-graph contains all edges in G except for the two edges incident to q and t. Finally, the H^*-graph contains all edges in the H^+-graph except for the edges between h-neighbors, i.e., (w, x), (s, y), and (r, z). □

In the following two subsections, we analyze and justify why we use G_{H^*} in our ExtMCE algorithm.

3.2 Analysis of H*-graph

Our algorithm ExtMCE uses the H^*-graph to estimate and control the memory usage. To this end, we need to examine two important factors: the size of H and the size of G_{H^*}.

We first discuss the size of H. Faloutsos et al. [14] show that for real-world networks following a power law degree distribution:

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

In Eq. (1), $r(v)$ is the *degree rank* of a vertex v, i.e., v is the $(r(v))$-th highest degree vertex in G, and $\mathcal{R}$ is the *rank exponent*, where $\mathcal{R} < 0$. By Definition 1, H is a set of h vertices having degree at least h; in other words, the lowest-degree vertex v in H has a rank of h and its degree is at least h. Thus, by substituting $r(v)$ by h in Eq. (1) and $d(v)$ should be at least h, we have

$$d(v) = \frac{1}{n^{\mathcal{R}}}h^{\mathcal{R}} \geq h. \tag{2}$$

Solving the inequality, we have

$$h \leq n^{\frac{\mathcal{R}}{\mathcal{R}-1}}. \tag{3}$$

Faloutsos et al. [14] 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 value of $\mathcal{R}$ measured in [14] for three snapshots of the internet graph is between -0.8 and -0.7. For a graph of 1 million vertices, we have $h \leq 464$ and therefore $|H| \leq 464$ when $\mathcal{R} = -0.8$. The value of h 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. (1), we have the following upper bound for $|G_{H^*}|$.

$$|G_{H^*}| \quad \leq \quad \sum_{r=1}^{h} (\frac{r}{n})^{\mathcal{R}}. \tag{4}$$

The right-hand side of Eq. (4) 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. (4). The equality holds when there is no edge connecting two h-vertices; in this case, the H^*-graph consists of h "stars", each centered at an h-vertex.

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

$$|G_{H^*}| \quad \geq \quad \sum_{r=1}^{h} (\frac{r}{n})^{\mathcal{R}} - \frac{h(h-1)}{2}. \tag{5}$$

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.

$$|G| \quad = \quad \frac{1}{2} \sum_{r=1}^{n} (\frac{r}{n})^{\mathcal{R}}. \tag{6}$$

By Eq. (4)-(6), we have

$$\frac{2\sum_{r=1}^{h} r^{\mathcal{R}} - n^{\mathcal{R}} h(h-1)}{\sum_{r=1}^{n} r^{\mathcal{R}}} \leq \frac{|G_{H^*}|}{|G|} \leq \frac{2\sum_{r=1}^{h} r^{\mathcal{R}}}{\sum_{r=1}^{n} r^{\mathcal{R}}} \tag{7}$$

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. (7), 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. (3) 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.

3.3 Why H*-graph?

We examine why we use G_{H^*} instead of G_H or G_{H^+}. We first analyze $|G_H|$ as follows.

$$0 \leq |G_H| \leq \frac{h(h-1)}{2}. \tag{8}$$

Eq. (8) gives the lower and upper bounds of $|G_H|$. Since h 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.

As for $|G_{H^+}|$, let $s = \sum_{r=1}^{h} (\frac{r}{n})^{\mathcal{R}}$, i.e., 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 $(h+1)$ to $(h+s)$ 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

$$\begin{aligned} |G_{H^+}| \quad &\leq \quad \frac{1}{2}(s + \sum_{r=1}^{s} (\frac{h+r}{n})^{\mathcal{R}}) \\ &= \quad \frac{1}{2} \sum_{r=1}^{h+s} (\frac{r}{n})^{\mathcal{R}}. \end{aligned} \tag{9}$$

The lower bound of $|G_{H^+}|$ is simply $|G_{H^*}|$ since $G_{H^*} \subseteq G_{H^+}$. Eq. (9) shows that G_{H^+} is too large to be kept in memory. For example, when $\mathcal{R} = -0.7$ and n is 1 million, G_{H^+} can be as large as 65% of the whole 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 6.1.

Algorithm 1 *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 $NB_H \leftarrow \{nb(v) : (d(v), v, nb(v)) \in Q\}$;
```

3.4 Computing the H*-graph

Algorithm 1 presents the algorithm for computing the set of h-vertices H, together with the set of 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 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.

THEOREM 1. *Algorithm 1 correctly computes the set of h-vertices of G and the set of their neighbors in $\mathcal{O}(h \log h + n)$ time and $\mathcal{O}(|G_{H^*}|)$ space, with one scan of G.*

PROOF. To prove the correctness, we need to show that: let h_0 be the true value of h of G, then the h computed by Algorithm 1 is equal to h_0. Suppose to the contrary that $h < h_0$, which implies that there are $h_0 > h$ vertices with a degree greater than $h_0 > h$. However, according to Algorithm 1, these h_0 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 1. Therefore, h computed by Algorithm 1 should be at least h_0 in this case. On the other hand, h cannot be larger than h_0 since each increment of h (Line 7 of Algorithm 1) follows the definition of h-vertex (Line 5). Thus, we have $h = h_0$.

We have $\mathcal{O}(h)$ insertions/updates, each takes $\mathcal{O}(\log h)$ time, plus n comparisons between h 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, we only need one scan of G. $\square$

4. RECURSIVE CLIQUE COMPUTATION

In this section, we discuss our algorithm **ExtMCE**. We first give the framework of ExtMCE as follows.

- **The first step:** extract G_{H^*} from G, compute the set of *local* max-cliques in G_{H^*}, obtain and output a subset of *global* max-cliques from local max-cliques by linking to the remaining part of G, and update G by removing G_{H^*};
- **The i-th step:** extract another subgraph G_i (of similar structure as G_{H^*}), where $|G_i| \leq |G_{H^*}|$, from G, repeat the first step (by replacing G_{H^*} with G_i);

  The recursive step continues until G becomes empty.

4.1 H*-max-cliques and H*-max-clique tree

We start the first step by defining the notions of H^*-max-cliques and H^*-max-clique tree.

4.1.1 H*-max-cliques

We first define the notion of H^*-max-cliques.

Definition 7 (H^*-MAX-CLIQUE). *An* H^*-max-clique *is a max-clique in G_{H^*}. The set of all H^*-max-cliques is denoted as $\mathcal{M}_{H^*}$.*

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

LEMMA 1. *The following statements of H^*-max-clique are true:*

1. *An H^*-max-clique contains at least one h-vertex.*
2. *An H^*-max-clique contains at most one h-neighbor.*

PROOF. Since each h-neighbor in G_{H^*} is connected to at least one h-vertex and there is no edge between any two h-neighbors, an H^*-max-clique containing an h-neighbor must also contain at least an incident h-vertex, which proves the first statement. The second statement holds since there is no edge among h-neighbors. $\square$

4.1.2 H*-max-clique tree

We now present the data structure used to keep the set of H^*-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 8 (H^*-MAX-CLIQUE TREE). *Given G_{H^*} of a graph, define a total order $\prec$ on H and H_{nb}. Moreover, $\forall u \in H$ and $\forall v \in H_{nb}$, $u \prec v$. The* H^*-max-clique tree*, T_{H^*}, of G_{H^*} is a prefix tree defined as follows.*

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

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

By Definition 8, we have the following lemma.

LEMMA 2. *The following statements of T_{H^*} are true:*

1. *An h-neighbor can only be a leaf in T_{H^*}.*
2. *All children of λ are h-vertices.*

PROOF. Lemma 1 states that an H^*-max-clique contains at most one h-neighbor. By the definition of the order $\prec$ and the tree T_{H^*}, an h-neighbor can only be a leaf in T_{H^*}.

Similarly, all children of λ are h-vertices since an H^*-max-clique contains at least one h-vertex as stated in Lemma 1 and all h-vertices are ordered before h-neighbors in a path in T_{H^*}. $\square$

Most existing algorithms for computing max-cliques can be modified to construct T_{H^*} with small overhead, particularly the algorithms [7, 19, 27, 9, 22, 12, 25] that adopt a *backtracking search tree*, which is essentially an H^*-max-clique tree. It is not our focus to propose another *in-memory* MCE algorithm; however, we highlight two improvements that we can make by employing the unique properties of T_{H^*}.

Given a path $p = \langle \lambda, u, \dots, v \rangle$ in T_{H^*}, let S be the set of vertices that can be used to potentially grow p from v. If $S \subseteq H_{nb}$, we first test if $\{u, \dots, v\}$ (the corresponding clique of p), instead of $\{u, \dots, v, s\}$ for some $s \in S$, is maximal. If $\{u, \dots, v\}$ is maximal, by Statement 1 in Lemma 2, we directly create S as the set of children of v (we also mark v for a condition test in Line 7 of Algorithm 2). Second, unlike a normal prefix tree or a backtracking search tree, by Statement 2 in Lemma 2, we only need to construct the subtree rooted at an h-vertex that is a child of λ. These two improvements can considerably speed up the process as they save a lot of unnecessary checking and comparisons.

4.1.3 Size estimation of H*-max-clique tree

The first step of our algorithm is critical as it sets the memory limit for the subsequent recursive steps. In Section 3.2, we give bounds on $|G_{H^*}|$. However, there is another element, T_{H^*}, that may take much memory. We estimate $|T_{H^*}|$ here.

Unlike $|G_{H^*}|$, which can be estimated by properties of scale-free networks, $|T_{H^*}|$ is difficult to estimate because the number and sizes of max-cliques in G_{H^*} vary greatly for different networks. The best known upper bound on the number of max-cliques is exponential [27], which is obviously too loose to be used to estimate the memory usage.

We devise an estimation strategy that borrows the concept of Knuth's method [18] 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 $p = \langle v_1, v_2, \dots, v_k \rangle$, $n(p) = (1 + f_1 + f_1 f_2 + \cdots + (f_1 \cdots f_{k-2} f_{k-1}))$, 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 [17].

However, Knuth's method assumes the presence of T so that one can perform random probing of paths, while T_{H^*} in our case is not yet constructed when estimating its memory usage. We propose a new method of probing paths in T_{H^*} by utilizing its unique properties, without actually constructing T_{H^*}. Each time we randomly choose a vertex $u \in H$. 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 an H^*-max-clique). Since the vertices are ordered and $nb(v)$ is available for every $v \in H$, we can virtually probe a path even though T_{H^*} does not exist. Thus, we can compute $n(p)$ as we move along p. Then, we estimate $n(T_{H^*})$ by averaging $n(p)$ of all the paths probed.

Our method is simple and yet does not violate the principle of random probing [18]. Our empirical study shows that it gives a good estimation in practice (see Table 5 in Section 6.1).

In the case when memory is very limited such that the available memory N is smaller than $n(T_{H^*})$, we remove the lowest-degree vertices from H. The number of vertices to be removed can be approximated as $\bar{h} = (1 - N/n(T_{H^*}))h$. Then, we use the remaining $(h - \bar{h})$ vertices as H and extract a smaller G_{H^*} accordingly. We re-estimate $n(T_{H^*})$ for the smaller G_{H^*} until $N > n(T_{H^*})$. The memory limit for the subsequent recursive steps is then set to the size of the smaller G_{H^*} and the corresponding T_{H^*}. We may lose some of the nice properties of using G_{H^*}, especially for dynamic update maintenance; however, when memory is scarce, this is a necessary compromise but importantly, our recursive algorithm also handles the case of limited memory resource.

4.2 From H*-max-cliques to global max-cliques

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

4.2.1 H^+-max-cliques: a subset of global max-cliques

We first define the notion of H^+-max-cliques as follows.

Definition 9 (H^+-MAX-CLIQUE). *An* H^+-max-clique *is a max-clique in* G_{H^+} *that* **consists of at least one h-vertex**. *The set of all H^+-max-clique is denoted as $\mathcal{M}_{H^+}$.*

LEMMA 3. *An H^+-max-clique is also a max-clique in G.*

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

With the result of Lemma 3, we have the following theorem.

THEOREM 2. *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 an h-vertex, i.e., $\mathcal{M}_0 = \{C : C \in \mathcal{M}, C \cap H \neq \emptyset\}$. Then, $\mathcal{M}_{H^+} = \mathcal{M}_0$.*

PROOF. First, Lemma 3 shows that $\mathcal{M}_{H^+} \subseteq \mathcal{M}_0$. Next, $\forall C \in \mathcal{M}_0$, $\exists u \in C$ such that $u \in H$. Since $\forall v \in C \backslash \{u\}$, $(u, v) \in E$, we have either $v \in H$ or $v \in H_{nb}$, implying that $C \in \mathcal{M}_{H^+}$ and hence $\mathcal{M}_0 \subseteq \mathcal{M}_{H^+}$. Thus, $\mathcal{M}_{H^+} = \mathcal{M}_0$. $\square$

Theorem 2 is important because it enables us to compute a subset of $\mathcal{M}$ separately on a portion of G, output it, and move on to computing another subset of $\mathcal{M}$ for another portion in the remaining of G, and so on recursively until we finish the whole graph.

4.2.2 Categorizing H^+-max-cliques

It is infeasible to compute $\mathcal{M}_{H^+}$ directly from G_{H^+}, since G_{H^+} is too large (see Eq. (9) and the analysis right after in Section 3.3). Instead, we compute $\mathcal{M}_{H^+}$ from T_{H^*}. We first define some notation used in the subsequent discussions.

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

We first identify three disjoint categories of H^+-max-cliques as follows. Let $C = (C_H \cup C_{H_{nb}})$ be an H^+-max-clique.

1. "$C_{H_{nb}} = \emptyset$": the set of H^+-max-cliques in this category is defined as $\mathcal{M}^1_{H^+} = \{C : C \in \mathcal{M}_{H^+}, C_{H_{nb}} = \emptyset\}$.

2. "$C_{H_{nb}} \neq \emptyset$ and $C_H \in \mathcal{M}_H$": the set of H^+-max-cliques in this category is defined as $\mathcal{M}^2_{H^+} = \{C : C \in \mathcal{M}_{H^+}, C_{H_{nb}} \neq \emptyset, C_H \in \mathcal{M}_H\}$.

3. "$C_{H_{nb}} \neq \emptyset$ and $C_H \notin \mathcal{M}_H$": the set of H^+-max-cliques in this category is defined as $\mathcal{M}^3_{H^+} = \{C : C \in \mathcal{M}_{H^+}, C_{H_{nb}} \neq \emptyset, C_H \notin \mathcal{M}_H\}$.

Recall that our objective in this subsection is *to obtain $\mathcal{M}_{H^+}$ from T_{H^*} or equivalently from $\mathcal{M}_{H^*}$*. 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 $\mathcal{M}_{H^*}$. We then prove that $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$ are sound and complete with respect to the above-defined three categories of H^+-max-cliques, respectively. We further prove that $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$ give the complete set of $\mathcal{M}_{H^+}$ in Theorem 3. Finally, we show how $\mathcal{M}_{H^+}$ can be computed from T_{H^*} in Theorem 4.

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

LEMMA 4. *Let $\mathcal{M}_1 = \mathcal{M}_H \cap \mathcal{M}_{H^*}$. Then, $\mathcal{M}_1 = \mathcal{M}^1_{H^+}$.*

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

(Prove $\mathcal{M}^1_{H^+} \subseteq \mathcal{M}_1$). $\forall C \in \mathcal{M}^1_{H^+}$, we have $C_{H_{nb}} = \emptyset$, which implies that $C = C_H$ and $C \in \mathcal{M}_H$. We have $C \in \mathcal{M}_{H^*}$ as well since $C_{H_{nb}} = \emptyset$. Thus, $C \in (\mathcal{M}_H \cap \mathcal{M}_{H^*}) = \mathcal{M}_1$. $\square$

Essentially, each $C \in \mathcal{M}_1$ corresponds to a root-to-leaf path in T_{H^*} where the leaf is an h-vertex. Thus, $\mathcal{M}_1$ can be readily obtained from T_{H^*}.

We now define $\mathcal{M}_2$. Intuitively, for each clique C in $\mathcal{M}_2$, its "H" part (i.e., C_H) is in $\mathcal{M}_H$; or equivalently, its "H" part is maximal in G_H.

LEMMA 5. *Let $\mathcal{M}_2 = \{C_1 \cup C_2 : C_1 \in (\mathcal{M}_H \backslash \mathcal{M}_1), C_2 \in maxCL(HNB(C_1))\}$. Then, $\mathcal{M}_2 = \mathcal{M}^2_{H^+}$.*

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

(Prove $\mathcal{M}_2 \subseteq \mathcal{M}^2_{H^+}$). $\forall C = (C_1 \cup C_2) \in \mathcal{M}_2$, we have $C_H = C_1$ and $C_{H_{nb}} = C_2$. Since $C_H \in \mathcal{M}_H$, C_H is maximal in G_H. Since $C_{H_{nb}} \in maxCL(HNB(C_H))$, $C_{H_{nb}}$ is also maximal in $G_{HNB(C_H)}$, which defines the h-neighborhood shared by all vertices in C_H. Thus, C is maximal in G_{H^+} (i.e., $C \in \mathcal{M}_{H^+}$). Since $C_H \notin \mathcal{M}_1$, we have $HNB(C_H) \neq \emptyset$ and thus $C_{H_{nb}} \neq \emptyset$. Since $C \in \mathcal{M}_{H^+}$, $C_H \in \mathcal{M}_H$, and $C_{H_{nb}} \neq \emptyset$, we have $C \in \mathcal{M}^2_{H^+}$.

(Prove $\mathcal{M}^2_{H^+} \subseteq \mathcal{M}_2$). $\forall C \in \mathcal{M}^2_{H^+}$, $C_{H_{nb}} \neq \emptyset$ and $C_H \in \mathcal{M}_H$. Thus, $C_H \in (\mathcal{M}_H \backslash \mathcal{M}_1)$. Since C is maximal in H^+, $C_{H_{nb}}$ must be maximal in $G_{HNB(C_H)}$, i.e., $C_{H_{nb}} \in maxCL(HNB(C_H))$. Let $C_1 = C_H$ and $C_2 = C_{H_{nb}}$, we have $C \in \mathcal{M}_2$. $\square$

Essentially, the C_H of each $C \in \mathcal{M}_2$ is locally maximal in G_H. According to Lemma 5, we can compute $\mathcal{M}_2$ as follows. For each path in T_{H^*} that corresponds to each $C_1 \in \mathcal{M}_H$ and has at least an h-neighbor leaf (since $C_1 \in (\mathcal{M}_H \backslash \mathcal{M}_1)$), compute $maxCL(HNB(C_1))$, and output $C = (C_1 \cup C_2)$ for each $C_2 \in maxCL(HNB(C_1))$. We will explain how to check whether a path in T_{H^*} corresponds to a clique in $\mathcal{M}_H$ later in this section.

Finally, we define $\mathcal{M}_3$. Intuitively, for each clique C in $\mathcal{M}_3$, its "H" part is no longer maximal in G_H but just some proper subset of a max-clique in G_H. The non-maximal "H" part then forms a max-clique in G_{H^+} by taking into account its h-neighborhood.

In order to define $\mathcal{M}_3$, we first need to define two notations: $\mathcal{X}$ and $EXT(\cdot)$. We enumerate the proper subsets of a max-clique in $\mathcal{M}_H$ that have at least one common h-neighbor as

$$\begin{aligned}\mathcal{X} = \{&C_1 : C_1 \subset C, C \in \mathcal{M}_H, C_1 \neq \emptyset, HNB(C_1) \neq \emptyset, \text{ and} \\ &\nexists C'_1 \subseteq C', C' \in \mathcal{M}_H, \text{ s.t. } C_1 \subset C'_1, HNB(C_1) = HNB(C'_1)\}. \quad (10)\end{aligned}$$

The last condition ensures that each $C_1 \in \mathcal{X}$ is not subsumed by its proper superset when forming a clique with its h-neighbors. Then, for each $C_1 \in \mathcal{X}$, we use $EXT(C_1)$ to denote the set of h-neighbors that can be used to extend C_1, defined as

$$\begin{aligned}EXT(C_1) = \{&C_2 : C_2 \in maxCL(HNB(C_1)), \text{ and} \\ &\nexists C' \in \mathcal{M}_2, \text{ s.t. } C' \supset (C_1 \cup C_2), \text{ and} \\ &\nexists C''_1 \in \mathcal{X}, \text{ s.t. } C''_1 \supset C_1,\ C_2 \in EXT(C''_1)\}. \quad (11)\end{aligned}$$

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

LEMMA 6. *Let* $\mathcal{M}_3 = \{C_1 \cup C_2 : C_1 \in \mathcal{X}, C_2 \in EXT(C_1)\}$. *Then,* $\mathcal{M}_3 = \mathcal{M}^3_{H^+}$.

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

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

(Prove $\mathcal{M}^3_{H^+} \subseteq \mathcal{M}_3$). $\forall C \in \mathcal{M}^3_{H^+}$, $C_H \notin \mathcal{M}_H$ and $C_{H_{nb}} \neq \emptyset$. First, C_H must be a proper subset of some $C' \in \mathcal{M}_H$. Assume that $C_H \notin \mathcal{X}$, then C_H must be subsumed by some element in $\mathcal{X}$, contradicting to the maximality of C. Thus, $C_H \in \mathcal{X}$. We further have $C_{H_{nb}} \in EXT(C_H)$ since $C_{H_{nb}}$ is maximal in $G_{HNB(C_H)}$ and the maximality of C must be ensured. Thus, $C \in \mathcal{M}_3$. $\square$

Essentially, each $C \in \mathcal{M}_3$ differs from a clique in $\mathcal{M}_1$ or $\mathcal{M}_2$ in that C_H is not maximal in G_H. This category of H^+-max-cliques is not as straightforward to compute as the first two categories. We discuss the details in Section 4.2.3.

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

THEOREM 3. $\mathcal{M}_{H^+} = (\mathcal{M}_1 \cup \mathcal{M}_2 \cup \mathcal{M}_3)$, *where* $\mathcal{M}_1$, $\mathcal{M}_2$ *and* $\mathcal{M}_3$ *are defined in Lemmas 4-6.*

PROOF. By the categorization, $\mathcal{M}^1_{H^+}$, $\mathcal{M}^2_{H^+}$ and $\mathcal{M}^3_{H^+}$ are disjoint and $(\mathcal{M}^1_{H^+} \cup \mathcal{M}^2_{H^+} \cup \mathcal{M}^3_{H^+})$ gives exactly $\mathcal{M}_{H^+}$. By Lemmas 4-6, we have $(\mathcal{M}_1 \cup \mathcal{M}_2 \cup \mathcal{M}_3) = \mathcal{M}_{H^+}$. $\square$

Now we show that T_{H^*} (plus the knowledge of the edges between the h-neighbors) is sufficient to compute $\mathcal{M}_{H^+}$. We only need partial $G_{H_{nb}}$ but do not keep $G_{H_{nb}}$ in memory (see details in Section 4.2.3). Before discussing the computation of $\mathcal{M}_{H^+}$ from T_{H^*}, we first show that $\mathcal{M}_H$ can be obtained from T_{H^*} by the following lemma.

LEMMA 7. $\forall C \in \mathcal{M}_H$, *there exists a path* $p \in T_{H^*}$ *such that the set of* h*-vertices in* p *equals to* C.

PROOF. For each $C \in \mathcal{M}_H$, we sort the vertices in C by $\prec$ as $C = \{v_1, \ldots, v_k\}$. First, if $HNB(C) = \emptyset$, since C is maximal in G_H, there must exist a root-to-leaf path $p = \langle v_1, \ldots v_k \rangle$ in T_{H^*}. Next, if $HNB(C) \neq \emptyset$, i.e., $\exists u \in HNB(C)$, then $p = \langle v_1, \ldots, v_k, u \rangle$ must be a root-to-leaf path in T_{H^*} (since C is maximal in G_H). $\square$

THEOREM 4. $\mathcal{M}_{H^+}$ *can be computed from* T_{H^*} *and* $G_{H_{nb}}$.

PROOF. By Lemma 7, every $C \in \mathcal{M}_H$ exists in T_{H^*}. Therefore, $\mathcal{M}_H$ can be computed from T_{H^*} by removing all h-neighbor leaves and checking the maximality of all remaining paths (This can be incorporated into the maximality checking when constructing T_{H^*} without any extra cost). Thus, $\forall C \in \mathcal{M}_{H^+}$, C_H can be obtained from T_{H^*}. On the other hand, the set of common h-neighbors, $HNB(C_H)$, can be obtained from the h-neighbor leaves in T_{H^*}, from which the corresponding $C_{H_{nb}}$ can be computed if we know the part of $G_{H_{nb}}$ that gives $G_{HNB(C_H)}$. $\square$

Example 2. Figure 2 gives the H^*-max-clique tree T_{H^*} (with h-vertices shaded) computed from the G_{H^*} of the example graph G in Figure 1. Each root-to-leaf path in T_{H^*} represents an H^*-max-clique and thus there are totally eight H^*-max-cliques. The $\mathcal{M}_H$ consists of only two cliques $\{a, b, c\}$ and $\{b, c, d, e\}$ ($\mathcal{M}_H = \{abc, bcde\}$ for short), which can be obtained from T_{H^*} too.

The set of H^+-max-cliques obtained from the G_{H^+} in Figure 1 is $\mathcal{M}_{H^+} = \{abcwx, acy, bcde, cey, drz, esy\}$. We now compute $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$ from T_{H^*} (and partial $G_{H_{nb}}$). First, by Lemma 4, $\mathcal{M}_1 = (\mathcal{M}_H \cap \mathcal{M}_{H^*}) = \{bcde\}$, which is the only root-to-leaf path in T_{H^*} with a non-h-neighbor leaf. Next, by Lemma 5, we have $(\mathcal{M}_H \backslash \mathcal{M}_1) = \{abc\}$. Therefore, the C_1 in

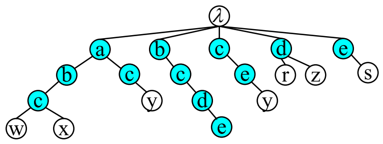


Figure 2: T_{H^*} of G in Figure 1

$\mathcal{M}_2$ can only be abc. Then $HNB(abc) = \{w, x\}$, which are the common h-neighbor leaves of paths in T_{H^*} containing abc. Since w and x are connected in G, we have $maxCL(HNB(abc)) = \{wx\}$. And thus $\mathcal{M}_2 = \{abcwx\}$. Finally for $\mathcal{M}_3$, we have $\mathcal{X} = \{ac, ce, d, e\}$. Essentially we should enumerate all proper subsets of a clique in $\mathcal{M}_H$. However, many of them are subsumed by their proper supersets in $\mathcal{X}$ or $\mathcal{M}_H$. For example, a is subsumed by ac since $HNB(a) = HNB(ac) = \{w, x, y\}$. Then for each $C_1 \in \mathcal{X}$, we compute $EXT(C_1)$. For example, considering ac, $maxCL(HNB(ac)) = \{wx, y\}$ but $EXT(ac) = \{y\}$. Note that $wx \in maxCL(HNB(ac))$ is excluded from $EXT(ac)$ because $acwx$ is checked to be non-maximal wrt. $abcwx \in \mathcal{M}_2$. Similarly, we have $EXT(ce) = \{y\}$, $EXT(d) = \{rz\}$, and $EXT(e) = \{sy\}$. Thus, by Lemma 6, $\mathcal{M}_3 = \{acy, cey, drz, esy\}$. It is easy to see that $(\mathcal{M}_1 \cup \mathcal{M}_2 \cup \mathcal{M}_3)$ gives exactly $\mathcal{M}_{H^+}$. □

4.2.3 *Computing H$^+$-max-cliques from H*-max-cliques*

We now discuss the algorithm to compute $\mathcal{M}_1$, $\mathcal{M}_2$ and $\mathcal{M}_3$, as shown in Algorithm 2.

It is straightforward to obtain both $\mathcal{M}_1$ and $\mathcal{M}_2$, by performing a depth-first search (DFS) on T_{H^*} (Lines 2-9). We do not store explicitly the set $\mathcal{M}_H$ and search it to check whether $C_1 \in \mathcal{M}_H$ (Line 7). Instead, we mark each vertex u whose root-to-u path forms a clique in $\mathcal{M}_H$ when we construct T_{H^*} (see the last paragraph of Section 4.1.2). 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 $C_1 \in \mathcal{M}_H$ in Line 7. We then check the conditions defined in $\mathcal{X}$ (see Eq. (10)) to prune the unqualified subsets. The checking of the last condition in $\mathcal{X}$ is similar to the maximality checking when constructing T_{H^*}. Note that the set $HNB(C)$ of a clique C can be easily obtained from T_{H^*} as the set of h-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. (11)) is done in the same way as that in $\mathcal{X}$. As for the computation of $maxCL(HNB(C'_1))$, we use an existing in-memory MCE algorithm. Since $HNB(C'_1)$ consists of common h-neighbors of all vertices in C'_1, $HNB(C'_1)$ is small and thus it is efficient to compute $maxCL(HNB(C'_1))$.

However, in order to compute $maxCL(HNB(C'_1))$, we need to know the induced subgraph $G_{HNB(C'_1)}$. Note that once we get T_{H^*}, we remove G_{H^*} from the memory. Thus, we now have more space to keep partial $G_{H_{nb}}$. In order to avoid random access to G in the disk, we do the following. Let N be the available memory. For all h-neighbor leaves in T_{H^*} ordered by the DFS traversal, we divide them into k partitions P_i ($1 \le i \le k$) such that the adjacency lists of the h-neighbors in each P_i can fit into N. We then read G from the disk sequentially and for each $v \in H_{nb}$, we write $(nb(v) \backslash H)$ into the partition(s) v is in. We keep the first partition in the memory, while each of the other partitions is written into consecutive disk pages. In this way, we read a partition (partial $G_{H_{nb}}$) into

Algorithm 2 *Compute-H^+-max-cliques*

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

```
1. Initilize $\mathcal{M}_1 = \mathcal{M}_2 = \mathcal{M}_3 = \emptyset$;
2. **for each** path $p = \langle v_1, \ldots, v_k \rangle$ in $T_{H^*}$ **do**
3.     **if** ($v_k \in H$)
4.         $\mathcal{M}_1 \leftarrow \mathcal{M}_1 \cup \{(v_1, \ldots, v_k)\}$;   /* by Lemma 4 */
5.     **else**
6.         $C_1 \leftarrow (v_1, \ldots, v_{k-1})$;
7.         **if** ($C_1 \in \mathcal{M}_H$)   /* by Lemma 5 */
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. (10) */
11. **for each** $C'_1 \in \mathcal{X}$ **do**        /* by Lemma 6 */
12.     Compute $EXT(C'_1)$;   /* see Eq. (11) */
13.     $\mathcal{M}_3 \leftarrow \mathcal{M}_3 \cup \{C'_1 \cup C'_2 : C'_2 \in EXT(C'_1)\}$;
14. **return** $\mathcal{M}_{H^+} \leftarrow (\mathcal{M}_1 \cup \mathcal{M}_2 \cup \mathcal{M}_3)$;
```

the memory each time when computing $maxCL(HNB(C'_1))$ and avoid random access in the disk.

4.3 Recursive steps

Now, $\mathcal{M}_{H^+}$ is computed and outputted, and G_{H^*} and T_{H^*} are discarded. We are ready to move on to the remaining part of G.

Let G' be the remaining part of G after removing G_{H^*}, which means that we delete all h-vertices and their incident edges from G to give G'. The first step is to extract a subgraph of G' that we can compute the max-cliques from it in the memory. Can we extract another H^*-graph G'_{H^*} from G' (wrt. another h') in the same way as we extract G_{H^*} from G? For any vertex v in G', $d(v) \le h$ (otherwise v should be in G_{H^*} instead). Thus, $h' \le h$ and $|G'_{H^*}| \le h^2$. In this case, G'_{H^*} is too small and we will need to scan G many times to compute $\mathcal{M}$, which is not desirable.

We propose to extract a subgraph from G' with a similar size to G_{H^*} as follows.

Definition 10 (L^*-GRAPH). *Let L be a set of vertices randomly selected from $G' = (V', E')$ such that*

$$\sum_{v \in L} d(v, G') \simeq |G_{H^*}|. \tag{12}$$

We define $G_{L^*} = (L \cup L_{nb}, E_{LL} \cup E_{LL_{nb}})$, where $L_{nb} = \{v : u \in L, v \notin L, (u, v) \in E'\}$, $E_{LL} = \{(u, v) : u, v \in L, (u, v) \in E'\}$, and $E_{LL_{nb}} = \{(u, v) : u \in L, v \in L_{nb}, (u, v) \in E'\}$.

Note that G_{L^*} is defined based on L in the same way as G_{H^*} is on H. Therefore, we can apply the method developed for G_{H^*} in Sections 4.1 and 4.2 to compute the max-cliques from G_{L^*}, by simply replacing H with L. After that, G' is updated by removing G_{L^*}. This process continues recursively until G' becomes empty.

There is a small problem in the transition from H to L. Consider the 2nd recursive step, i.e., the step right after we compute $\mathcal{M}_{H^+}$. A clique C may be maximal in G_{L^+}, but C may not be maximal in G, because it is possible that $\exists C' \in \mathcal{M}_{H^+}$ such that $C' \supset C$ and $C'_{H_{nb}} = C$. We remark that if C is maximal in G_{H^+}, then C is also maximal in G, because H only has connection with H_{nb} while L may have connection with both H and L_{nb}.

We address this problem as follows. For each $C \in \mathcal{M}_{H^+}$, if $|C_{H_{nb}}| > 1$, we keep $C_{H_{nb}}$ in a hashtable. Let C' be a max-clique in G_{L^+}. If $|C'| = 1$, then $C' = \{v\}$ is maximal in G only if $d(v) = 0$. If $|C'| > 1$, we hash C' to check if C' exists in the hashtable. If C' is not in the hashtable, then C' is maximal in G and we also add $C'_{L_{nb}}$ (if $|C'_{L_{nb}}| > 1$) into the hashtable for the

Algorithm 3 *ExtMCE*

Input: G, recursive depth k, size bound b
Output: $\mathcal{M}$

```
1. if ($G$ is empty)   return;
2. if ($k = 1$)              /* 1st-step: compute $G_{H^*}$ */
3.     Compute-$H^*$-graph (Alg. 1);
4. else                  /* recursive steps: compute $G_{L^*}$ */
5.     Compute-$L^*$-graph with size bounded by $b$ (Def. 10);
6. Construct $T_{H^*}$ (or $T_{L^*}$) by an existing MCE algorithm A;
7. Compute-$H^+$(or $L^+$)-max-cliques (Alg. 2);
8. if ($k = 1$)        /* $H^+$-max-cliques are max-cliques by Theorem 2 */
9.     Build a global hashtable $X$ (see the last paragraph of Section 4.3);
10.    Output $H^+$-max-cliques;
11. else
12.    Check the maximality of $L^+$-max-cliques by $X$;
13.    Output $L^+$-max-cliques that are globally maximal;
14.    Update $X$ (see the last paragraph of Section 4.3);
15. Remove $G_{H^*}$ (or $G_{L^*}$) from $G$;
16. ExtMCE($G$, $k+1$, $|G_{H^*}|$);
```

maximality checking in subsequent recursive steps. Otherwise, C' is not 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. After each round of max-clique computation from G_{L^*}, we delete all C in the hashtable if $\exists v \in C$ such that $v \in L$, because all max-cliques containing v are generated after we finish G_{L^*} (by Theorem 2).

4.4 Overall Algorithm: ExtMCE

The overall recursive algorithm *ExtMCE* is presented in Algorithm 3. The set of all max-cliques in G can be computed by invoking ExtMCE(G, 1, 0). The second parameter k of the algorithm specifies the depth of the recursive process and the last parameter b sets a size bound on the portion of G that is under max-clique computation in each recursive step. b is set as 0 initially for the first step and as $|G_{H^*}|$ for the following recursive steps.

We state the correctness of ExtMCE in the following theorem.

THEOREM 5. *The results returned by ExtMCE 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 H^+-max-cliques is computed in Line 7 and outputted directly in Line 10 of ExtMCE. The H^+-max-cliques are proved to be maximal in G in Theorem 2. Next, at each subsequent recursive step, L^+-max-cliques are computed (Line 7). The maximality of the outputted L^+-max-cliques is ensured by the checking in Line 12 of ExtMCE.

We now prove the completeness. At the first step, the set of H^+-max-cliques is complete with respect to the max-cliques in G that contain at least one vertex in H (by Theorem 2). At the second recursive step, L^+-max-cliques are computed in the same way as H^+-max-cliques (Line 7). This means that the set of L^+-max-cliques is complete with respect to the set of max-cliques in $(G \backslash G_{H^*})$ that contain at least one vertex in L. Combining with the H^+-max-cliques (computed from G_{H^*}) 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 gives a complete set of max-cliques in G. □

Complexity. The memory space complexity of ExtMCE is $\mathcal{O}(|G_{H^*}| + |T_{H^*}|)$. We need $\mathcal{O}(|G|/|G_{H^*}|)$ scans of G for the entire process. We now analyze the time complexity of ExtMCE when compared with an 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 ExtMCE is comparable to that of $A(G)$. This is because Algorithm 2 essentially expands those paths in T_{H^*} (or T_{L^*}) that would be generated by $A(G)$ as well, while the computation of each $maxCL(HNB(\cdot))$ is also necessary in $A(G)$. Thus, if the memory is big enough to hold the whole graph G, ExtMCE performs comparably to $A(G)$. However, if the memory is insufficient (a typical case for a massive graph G), $A(G)$ would incur many random disk accesses to G, while ExtMCE has a bounded number of scans of G, which is much more efficient. These conclusions are empirically verified by our experiments.

5. UPDATE IN DYNAMIC NETWORKS

We consider two types of updates: edge insertion and edge deletion. Vertex insertion/deletion can be considered as a series of edge insertions/deletions proceded/followed by the insertion/deletion of an isolated vertex, which is a rather trivial operation. Note that G is also updated, but we focus our discussion on updates directly related to the H^*-max-cliques. In other words, we only maintain the H^*-max-cliques to be up-to-date, while we compute the set of all max-cliques $\mathcal{M}$ periodically or on demand.

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. 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 non-maximal cliques in $\mathcal{S}$ and get $\mathcal{S}_M = \{C : C \in \mathcal{S}, \nexists C' \in \mathcal{S}$ s.t. $C' \supset C\}$. 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.

We give an analysis on the cost of the updates as well as on the frequency of the updates.

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 frequent 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 3.2, the size of H, i.e., h, 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.

6. EXPERIMENTAL EVALUATION

We evaluate the performance of our method, comparing with the state-of-the-art *in-memory* MCE algorithm [27] and the only existing *streaming* MCE algorithm for dynamic networks [26], denoted as **in-mem** and **streaming** in our experiments. We ran all experiments on a machine with a 3.0GHz Pentium 4 CPU and 2GB RAM, running Windows XP.

Datasets. We use 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. In the *blogs* network, vertices are blogs and edges indicate that two blogs appear in the same search result. *LJ* is the free on-line community called Livejournal (www.livejournal.com), where vertices are members and edges represent friendship between members. The *Web* graph is obtained from the YAHOO webspam dataset (barcelona.research.yahoo.net/webspam), where vertices are pages and edges are hyperlinks. We give the details of each dataset (number of vertices and edges, physical storage size) as follows.

Table 2: Datasets (K = 1,000 and M = 1,000,000)

	protein	*blogs*	*LJ*	*Web*
$n = \|V\|$	20K	1M	4.8M	10M
$m = \|E\|$	40K	6.5M	43M	80M
Storage size (MB)	1	186	1310	2613

6.1 Evaluation of the H*-graph

Table 3 shows that it is very efficient to extract G_{H^*} from G. The majority of the time is used to read the graph from the disk, which is an inevitable cost.

Table 3: Time and memory usage of extracting G_{H^*}

	protein	*blogs*	*LJ*	*Web*
Total time (sec)	0.3	38	243	524
Disk-read time (sec)	0.2	31	199	405
Memory (MB)	1.2	8.5	27	140

Table 4 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 parenthese 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 more efficient MCE computation with reasonable memory usage. As seen from the size ratio, LJ has a relatively smaller G_{H^*} and G_{H+} than other datasets, which is mainly because the vertices in LJ are less densely connected as indicated by its small $|H|$ wrt. $|V|$.

Table 4: Sizes of H, H_{nb}, G_H, G_{H^*} and G_{H+}

	protein	*blogs*	*LJ*	*Web*
$\|H\|$	77	718	987	2982
$\|H_{nb}\|$	4K	192K	441K	4.4M
$\|G_H\|$	0.5K (1%)	37K (0.6%)	25K (0.06%)	29K (0.04%)
$\|G_{H^*}\|$	8.6K (22%)	840K (13%)	1.7M (4%)	25M (31%)
$\|G_{H+}\|$	21K (54%)	4M (64%)	11M (25%)	54M (68%)

Table 5 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, and the accuracy of estimating $|T_{H^*}|$. 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.

Table 5: Closeness, reachability, # of max-cliques, and $|T_{H^*}|$

	protein	*blogs*	*LJ*	*Web*
closeness (h-vertices)	3.1	3.4	4.3	7.1
reachability (h-vertices)	47%	56%	100%	73%
# of max-cliques	25K	1.1M	173M	267M
(contain h-vertices)	239	4K	69K	7.8M
(contain h-neighbors)	12K	510K	43M	146M
(*estimated* $\|T_{H^*}\|$)/$\|T_{H^*}\|$	1.00	1.01	0.93	0.97

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 5 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 that of all max-cliques. The result justifies the feasibility of our update maintenance 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.

Finally, Table 5 shows that our method of estimating $|T_{H^*}|$ is highly accurate. Thus, the result verifies the effectiveness of setting memory-bound at the first step of *ExtMCE*.

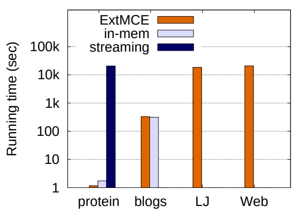
6.2 Performance of ExtMCE

Figure 3 reports the total running time and peak memory consumption of finding the set of all max-cliques using *ExtMCE*, *in-mem*, and *streaming*, respectively.

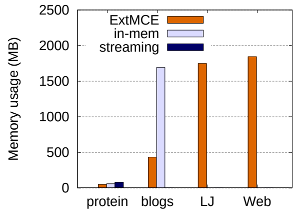
First, on the smaller networks *protein* and *blogs*, *ExtMCE* is as fast as *in-mem*, but with only 1 quarter of the memory usage of *in-mem*. The result verifies our assertion in Section 4.4 that the time complexity of *ExtMCE* is indeed comparable to that of an in-memory MCE algorithm.

On the larger networks, the advantage of *ExtMCE* over *in-mem* is immediately seen. As shown in Figure 3(b), *in-mem* runs out of memory, while *ExtMCE* computes the result for all the networks with a bounded memory consumption. The corresponding running time for *in-mem* is thus not shown in Figure 3(a) since *in-mem* does not complete the MCE task.

We are only able to obtain the result of *streaming* for the smallest *protein* network, which already takes many orders of magnitude more time to complete. The result is because *streaming* reads an



(a) Running Time



(b) Memory Consumption

Figure 3: Performance of *ExtMCE*

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 MCE computation is extremely high. On the contrary, *ExtMCE* reads the graph $\mathcal{O}(|G|/|G_{H^*}|)$ times, but is able to compute MCE efficiently with bounded memory usage.

We further analyze *ExtMCE* by showing the number of recursions it requires for each dataset. As shown in Table 6, the number of recursions actually performed by *ExtMCE* is very close to the estimated number, $|G|/|G_{H^*}|$. *LJ* has a higher number of recursions since its H^*-graph is relatively smaller as shown in Table 4.

Table 6: Actual/estimated number of recursions

	protein	*blogs*	*LJ*	*Web*
# of recursions	5	9	25	7
$\lvert G\rvert/\lvert G_{H^*}\rvert$	4.5	7.7	24.6	3.2
Time (1st recursion)	67%	36%	2%	34%

Table 6 also shows that the percentage of the total running time used for the first recursive step, i.e., *ExtMCE* operates on G_{H^*}. It shows that a large portion of the time is spent on computing the max-cliques at the first step (except *LJ*), which also justifies the choice of G_{H^*} for dynamic update maintenance. We also find that the peak memory consumption indeed occurs at the first recursive step, which verifies the correctness of $\mathcal{O}(|G_{H^*}| + |T_{H^*}|)$ as the memory bound for *ExtMCE*.

6.3 Performance on Update Maintenance

Table 7 reports the results for update maintenance. We use the *blogs* network, whose edges are associated with a timestamp, spanning over 12 months. We average the results for every two month-period, shown as P1-P6 in Table 7. The network grows from 347K edges to 6.5M edges.

Table 7 shows that the average time of processing an edge insertion that triggers an update in T_{H^*}, shown as "Avg. update time", is only 2 to 3 msec. The exception is P1 which requires 10 msec. 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 7 also shows "# of updates in G_{H^*}", which is the number of edge insertions that trigger an update in T_{H^*}, and "# 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 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 has taken 190 hours to update only 40K edges at the time of writing).

Table 7: Results for update maintenance

	P1	P2	P3	P4	P5	P6
Avg. update time (msec)	10	3	2	2	2	3
# of updates in G_{H^*}	3K	11K	19K	25K	28K	28K
# of updates in G	385K	457K	550K	461K	526K	670K
# of h-vertices	294	425	508	566	614	696
% of h-vertices retained	92	92	95	96	94	96
Memory (MB)	418	427	436	443	451	463
Time w/ T_{H^*} (sec)	12	22	45	68	86	114
Time w/o T_{H^*} (sec)	36	62	104	142	177	226

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. Note that the memory consumption is the same amount of memory needed for computing the set of all max-cliques by *ExtMCE*, since $\mathcal{O}(|G_{H^*}| + |T_{H^*}|)$ sets the bound for the memory usage of *ExtMCE*.

Finally, the last two rows of Table 7 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 $\mathcal{M}$.

7. RELATED WORK

There is a large literature on MCE. We discuss the more prominent and recent ones, a comprehensive review can be found in [9]. The first significant improvement on MCE was the algorithms [2, 7] that use the *backtracking* method. They take $\mathcal{O}(n^2)$ memory space. Further improvements [19, 27, 9] were made by selecting good *pivots* to prune the backtracking search tree. The optimal worst-case time of backtracking-based MCE was shown to be $\mathcal{O}(3^{n/3})$ [27]. Recently, parallel algorithms [12, 25] were proposed to enumerate max-cliques from different points of the search tree in parallel. However, all these works did not focus on reducing the memory complexity and require $\mathcal{O}(m + n)$ memory space in the best case. Output-sensitive MCE algorithm was also introduced [28] which is based on *reverse search*, and recent work [21] 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. There is also algorithm that obtains a k-clique by joining two $(k-1)$-cliques [20]. However, all these algorithms require memory space at least $\Omega(m + n)$. Stix [26] proposed an algorithm that updates the set of max-cliques upon each edge insertion, and the graph is read only once. Finally, we are aware of a recent work that adopts the concept of h-index for triangle counting [13]. Their work does not address the memory issue and takes $\mathcal{O}(m+n)$ memory, while the problem of MCE is also substantially more difficult than that of triangle counting.

8. CONCLUSIONS

We propose ExtMCE, the first external-memory algorithm for

MCE computation on large real-world networks. ExtMCE recursively processes a small part of a large graph at a time, while ensuring that the set of max-cliques computed in the local steps is correct and complete in the whole graph. ExtMCE bounds the memory usage by the H^*-graph, a novel concept defined based on the notion of h-index. We test ExtMCE on large networks of up to 10 million vertices and 80 million edges and verify that the effectiveness of using the H^*-graph for bounding memory usage. Our experimental results show that ExtMCE achieves comparable running time compared with the state-of-the-art in-memory MCE algorithm [27], but uses significantly less memory. For the larger networks, the in-memory algorithm does not work while ExtMCE still computes MCE efficiently with bounded memory usage. We also compare with a streaming MCE algorithm [26] and show that ExtMCE is orders of magnitude more efficient. We also verify the feasibility of update maintenance on large networks based on the H^*-graph.

9. ACKNOWLEDGMENTS

This research is supported in part by the AcRF Tier-1 Grant (M52020092) from Ministry of Education of Singapore, the Chinese University of Hong Kong (CUHK) Direct Grant No. 2050474, the RGC Research Direct Grant of the CUHK Projects 2050421 and 2150472, the CUHK Postdoctoral Fellowship Grant 2008-2009, and the RGC of the Hong Kong SAR, CUHK No. 419008.

10. REFERENCES

[1] F. N. Abu-Khzam, N. E. Baldwin, M. A. Langston, and N. F. Samatova. On the relative efficiency of maximal clique enumeration algorithms, with applications to high throughput computational biology. In *International Conference on Research Trends in Science and Technology*, 2005.

[2] E. A. Akkoyunlu. The enumeration of maximal cliques of large graphs. *SIAM J. Comput.*, 2(1):1–6, 1973.

[3] H. R. Bernard, P. D. Killworth, and L. Sailer. Informant accuracy in social network data iv: a comparison of clique-level structure in behavioral and cognitive network data. *Social Networks*, 2(3):191–218, 1979.

[4] N. M. Berry, T. H. Ko, T. Moy, J. Smrcka, J. Turnley, and B. Wu. Emergent clique formation in terrorist recruitment. In *The AAAI-04 Workshop on Agent Organizations: Theory and Practice*, 2004.

[5] G. Bianconi and M. Marsili. Emergence of large cliques in random scale-free networks. *Europhysics Letters*, 74(4):740–746, 2006.

[6] V. Boginski, S. Butenko, and P. M. Pardalos. Statistical analysis of financial networks. *Computational Statistics & Data Analysis*, 48(2):431–443, 2005.

[7] C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, 1973.

[8] J. M. Byskov. Algorithms for k-colouring and finding maximal independent sets. In *SODA*, pages 456–457, 2003.

[9] F. Cazals and C. Karande. A note on the problem of reporting maximal cliques. *Theor. Comput. Sci.*, 407(1-3):564–568, 2008.

[10] G. Creamer, R. Rowe, S. Hershkop, and S. J. Stolfo. Segmentation and automated social hierarchy detection through email network analysis. In *WebKDD/SNA-KDD*, pages 40–58, 2007.

[11] S. N. Dorogovtsev and J. F. F. Mendesand. Evolution of networks: From biological nets to the internet and www. *Oxford University Press*, 2003.

[12] N. Du, B. Wu, L. Xu, B. Wang, and P. Xin. Parallel algorithm for enumerating maximal cliques in complex network. In *Mining Complex Data*, pages 207–221. 2009.

[13] D. Eppstein and E. S. Spiro. The *h*-index of a graph and its application to dynamic subgraph statistics. In *WADS*, pages 278–289, 2009.

[14] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM*, pages 251–262, 1999.

[15] K. Faust and S. Wasserman. Social network analysis: Methods and applications. *Cambridge University Press*, 1995.

[16] J. E. Hirsch. An index to quantify an individual's scientific research output. *Proceedings of the National Academy of Sciences of the United States of America*, 102(46):16569–16572, 2005.

[17] P. Kilby, J. K. Slaney, S. Thiébaux, and T. Walsh. Estimating search tree size. In *AAAI*, 2006.

[18] D. E. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29(129):121–136, 1975.

[19] I. Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theor. Comput. Sci.*, 250(1-2):1–30, 2001.

[20] F. Kose, W. Weckwerth, T. Linke, and O. Fiehn. Visualizing plant metabolomic correlation networks using clique-metabolite matrices. *Bioinformatics*, 17(12):1198–1208, 2001.

[21] K. Makino and T. Uno. New algorithms for enumerating all maximal cliques. In *SWAT*, pages 260–272, 2004.

[22] N. Modani and K. Dey. Large maximal cliques enumeration in sparse graphs. In *CIKM*, pages 1377–1378, 2008.

[23] S. Mohseni-Zadeh, P. Brézellec, and J.-L. Risler. Cluster-c, an algorithm for the large-scale clustering of protein sequences based on the extraction of maximal cliques. *Computational Biology and Chemistry*, 28(3):211–218, 2004.

[24] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45:167–256, 2003.

[25] M. C. Schmidt, N. F. Samatova, K. Thomas, and B.-H. Park. A scalable, parallel algorithm for maximal clique enumeration. *J. Parallel Distrib. Comput.*, 69(4):417–428, 2009.

[26] V. Stix. Finding all maximal cliques in dynamic graphs. *Computational Optimization and applications*, 27:173–186, 2004.

[27] E. Tomita, A. Tanaka, and H. Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.*, 363(1):28–42, 2006.

[28] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM J. Comput.*, 6(3):505–517, 1977.

[29] B. Zhang, B.-H. Park, T. V. Karpinets, and N. F. Samatova. From pull-down data to protein interaction networks and complexes with biological relevance. *Bioinformatics*, 24(7):979–986, 2008.