Algorithmica (2013) 66:173–186
DOI 10.1007/s00453-012-9632-8

Fast Maximal Cliques Enumeration in Sparse Graphs

Lijun Chang · Jeffrey Xu Yu · Lu Qin

Received: 10 June 2010 / Accepted: 22 February 2012 / Published online: 6 March 2012

Abstract In this paper, we consider the problem of generating all maximal cliques in a sparse graph in polynomial delay. Given a graph $G = (V, E)$ with n vertices and m edges, the latest and fastest polynomial delay algorithm for sparse graphs enumerates all maximal cliques in $O(\Delta^4)$ time delay, where Δ is the maximum degree of vertices. However, it requires an $O(n \cdot m)$ preprocessing time. We improve it in two aspects. First, our algorithm does not need preprocessing. Therefore, our algorithm is a truly polynomial delay algorithm. Second, our algorithm enumerates all maximal cliques in $O(\Delta \cdot H^3)$ time delay, where H is the so called H-value of a graph or equivalently it is the smallest integer satisfying $|\{v \in V \mid \delta(v) \geq H\}| \leq H$ given $\delta(v)$ as the degree of a vertex. In real-world network data, H usually is a small value and much smaller than Δ.

Keywords Maximal clique · Polynomial delay · Sparse graph · H-Partition · H-Value

1 Introduction

Maximal clique enumeration is a long-standing problem, studied not only in graph theory [1, 2], but also in bioinformatics [7, 12], data mining [3, 11], and web mining [9]. Finding all maximal cliques in a graph has been applied in a variety of applications, and enumeration algorithms have been extensively studied.

L. Chang (✉) · J.X. Yu · L. Qin
The Chinese University of Hong Kong, Hong Kong, China
e-mail: ljchang@se.cuhk.edu.hk

J.X. Yu
e-mail: yu@se.cuhk.edu.hk

L. Qin
e-mail: lqin@se.cuhk.edu.hk

1.1 Previous Works

The algorithms to generate all maximal cliques generally fall into two categories, namely, heuristic (greedy) algorithms and output-sensitive algorithms. First is the Bron-Kerbosch algorithm [2] and its sibling [1], which generate all maximal cliques based on branch-and-pruning search on the graph. A more recent algorithm proposed by Tomita et al. [13] generates all maximal cliques in worst-case $O(3^{n/3})$ time for any graph with n vertices and m edges. This time complexity is proved to be optimal [13], since there exist up to $3^{n/3}$ maximal cliques in an n-vertex graph.

Several complexity measures are considered in [8] to measure the performance of algorithms that may have exponential-size output, namely, *polynomial total time*, *incremental polynomial time*, and *polynomial delay*. An algorithm is in polynomial delay, then it is also in incremental polynomial time. An algorithm is in incremental polynomial time, then it is also in polynomial total time. Polynomial delay is of importance in many scenarios [5, 8], e.g., when users (or other programs) read the answers as they are delivered, or are only interested in a small portion of the answers. Tsukiyama et al. [14] first propose a polynomial delay algorithm with $O(n \cdot m)$ time delay, i.e., the computation time between any consecutive outputs, the time before outputting the first answer, and the time to terminate after outputting the last answer, are all bounded by $O(n \cdot m)$. Chiba and Nishizeki [4] reduce the time complexity to $O(\alpha(G) \cdot m)$, where $\alpha(G)$ is the arboricity of G with $m/(n-1) \leq \alpha(G) \leq m^{1/2}$. Makino and Uno [10] propose two algorithms with improved delay time complexity. One for general graphs, it enumerates all maximal cliques in $O(M(n))$ time delay and $O(n^2)$ space, where $M(n)$ is the time to multiply two $n \times n$ matrices. One for sparse graphs with low degrees, it enumerates all maximal cliques in $O(\Delta^4)$ time delay and $O(n+m)$ space, where Δ is the maximum degree of vertices in G, and it additionally requires $O(n \cdot m)$ time as a preprocessing before outputting the first maximal clique.

1.2 Our Contribution and Techniques

In this paper, we propose new algorithms to enumerate all maximal cliques in a sparse graph with improved delay time complexity. First, we use the concept of *H-value*, which is the smallest integer satisfying $|\{v \in V \mid \delta(v) \geq H\}| \leq H$ where $\delta(v)$ is the degree of vertex v, to our design and analysis of maximal cliques enumeration algorithms. Second, we remove the requirement of preprocessing, so that the time before outputting the first maximal clique is also bounded by the same polynomial function. Instead of defining enumeration tree, we define an enumeration forest, the roots of which are well-defined and can be found easily. Then, we propose two algorithms based on our enumeration forest: one is a recursive algorithm, and the other is a non-recursive algorithm with linear space. The general idea of our algorithms is to partition the original sparse graph into two parts: one part with few vertices of high degree, and the other large part with all remaining vertices of low degree. We treat vertices in the two parts differently when testing connection information between a vertex and a set of vertices. Equivalently, for the vertices that have high degree, we enumerate vertices; for the vertices that have low degree, we enumerate their neighbors. We introduce our algorithms in a general setting, where an arbitrary integer θ is

used to partition the vertices V into two sets: one set consists of vertices with degree no smaller than θ, and the other set consists of vertices with degree smaller than θ. We show that the time complexity achieves its optimal value when θ is chosen as H, which is the *H-value*. We call this partition as *H-partition*. Our recursive algorithm enumerates all maximal cliques in $O(\Delta \cdot H^3)$ time delay and $O(n \cdot \Delta \cdot H + m)$ space, and our non-recursive algorithm enumerates all maximal cliques in $O(\Delta \cdot H^3)$ time delay and $O(n + m)$ space. Although, in a sparse graph, the maximum degree Δ does not necessarily to be small, the H-value H usually is a small value for real-world networks [6], theoretical analyses of H for scale-free networks are also studied in [3, 6].

The rest of the paper is organized as follows. In Sect. 2, we present some preliminaries and notations used in this paper. In Sect. 3, we explain the algorithm of Makino and Uno. We introduce our improved enumeration algorithm for sparse graphs in Sect. 4. Conclusions are given in Sect. 5.

2 Preliminary

We consider an undirected graph, $G = (V, E)$, with a vertex set $V = \{v_1, \ldots, v_n\}$ and an edge set $E = \{e_1, \ldots, e_m\}$. For an edge $e_i = (u, v)$, we say that u and v are connected (or adjacent) to each other. We assume that there is no self-loops, i.e., $(v, v) \notin E, \forall v \in V$. Every vertex $v_i \in V$ has an index which is i. We say vertex v_i is less than v_j, i.e., $v_i < v_j$, if the index of v_i is less than that of v_j, i.e., $i < j$. Let n and m denote the numbers of vertices and edges of G, respectively, i.e., $n = |V|$ and $m = |E|$. Through out this paper, we assume without loss of generality that G is connected, otherwise the algorithm can be applied to each connected component.

For a vertex $v \in V$, denote the set of *neighbors* of v as $\Gamma(v) = \{u \in V \mid (u, v) \in E\}$, and denote the *degree* of v as $\delta(v) = |\Gamma(v)|$. Let Δ denote the maximum degree of G, i.e., $\Delta = \max_{v \in V} \delta(v)$. Similarly, for a set of vertices $S \subset V$, $\Gamma(S) = \{u \in V \setminus S \mid \exists v \in S, (u, v) \in E\}$ and $\delta(S) = |\Gamma(S)|$. Intuitively, $\Gamma(S)$ is the set of vertices not in S but connected to at least one vertex in S. We also define $\Lambda(S)$ to be the set of common neighbors of vertices in S, i.e., $\Lambda(S) = \{v \in V \setminus S \mid \forall u \in S, (u, v) \in E\}$. Let $V_{\leq i}$ denote the set of vertices with indices less than or equal to i, i.e., $V_{\leq i} = \{v_1, \ldots, v_i\}$. For a vertex set S and an integer $i \in [1, n]$, $S_{\leq i} = S \cap V_{\leq i}$ which is the subset of S with vertex indices less than or equal to i. $\Gamma(S)_{\leq i}$ and $\Lambda(S)_{\leq i}$ are defined similarly.

A graph is a *clique* if it is a complete graph, i.e., there are edges between any two vertices. We also call a vertex set $K \subseteq V$ a clique if the subgraph induced by the vertex set K is a clique, and K is a *maximal clique* if there exists no proper superset of K that is also a clique. For two non-empty vertex sets X and Y, we say X is *lexicographically larger* than Y if the smallest vertex (i.e., the vertex with smallest index) in $(X \setminus Y) \cup (Y \setminus X)$ is contained in X. For a clique K, let $C(K)$ denote the maximal clique that is the lexicographically largest among all maximal cliques containing K, which can be found by recursively adding to it the smallest vertex that is adjacent to all vertices in the current clique. It has $K \subseteq C(K)$, and $C(K)$ is not lexicographically smaller than K. Note that, there may exist multiple maximal cliques

containing K, however, $C(K)$ is unique. Specifically, $C(\emptyset)$ is defined as the maximal clique that is the lexicographically largest among all the maximal cliques in G.

3 Enumeration Framework

The most recent algorithms to enumerate all maximal cliques with polynomial delay are introduced in [10]. For a given graph $G = (V, E)$, Makino and Uno [10] propose two algorithms: (1) one algorithm generates all maximal cliques in G in $O(M(n))$ time delay and $O(n^2)$ space; (2) another algorithm generates all maximal cliques in G in $O(\Delta^4)$ time delay and $O(n + m)$ space, where $O(nm)$ time is required as a preprocessing step. Both of these algorithms are based on the same framework of [8, 14]. We introduce the general enumeration framework below.

Let K_0 denote the maximal clique that is lexicographically largest among all maximal cliques, then $K_0 = C(\emptyset) = C(\{v_1\})$ and $v_1 \in K_0$. For a maximal clique $K (\neq K_0)$, the *parent* of K, $P(K)$, is defined by $C(K_{\leq i-1})$ such that i is the maximum index satisfying $C(K_{\leq i-1}) \neq K$, and such i is called the *parent index*, denoted as $p(K)$. $p(K_0)$ is trivially defined to be 1. Therefore, $P(K) = C(K_{\leq p(K)-1})$, and $P(K)$ is lexicographically larger than K. Note that, for any clique $K (\neq K_0)$, $P(K)$ exists, since $C(K_{\leq 0}) \neq K$. For each maximal clique K in G, except K_0, it has a unique parent $P(K)$, which is lexicographically larger than K. Then, the following lemma holds.

Lemma 3.1 [10] *The parent-child relation constructs an in-tree of all maximal cliques rooted by* K_0.

This in-tree is called an *enumeration tree* for maximal cliques in graph G. In order to enumerate maximal cliques based on the enumeration tree, it needs to generate children of a maximal clique. The following lemma characterizes the children for any maximal clique.

Lemma 3.2 [10] *Let K and K' be maximal cliques in G, then K' is a child of K if and only if $K' = C((K_{\leq i} \cap \Gamma(v_i)) \cup \{v_i\})$ holds for some i such that*

(a) $v_i \notin K$;
(b) $i > p(K)$;
(c) $K_{\leq i} \cap \Gamma(v_i) = C((K_{\leq i} \cap \Gamma(v_i)) \cup \{v_i\})_{\leq i-1}$;
(d) $K_{\leq i} = C(K_{\leq i} \cap \Gamma(v_i))_{\leq i}$.

Moreover, if an index i satisfies (a)–(d), *then i is the parent index of K', i.e., $p(K') = i$.*

Based on the above two lemmas, a recursive algorithm can generate maximal cliques based on a DFS-traversal of the enumeration tree. The children of a maximal clique can be generated in $O(nm)$ time, where there are n possible indices i and checking the conditions (a)–(d) takes $O(m)$ time. Optimizations of this naive enumeration are taken on two aspects: (1) reducing the number of possible indices i to

be checked; (2) reducing the time needed to check condition (a)–(d) for each possible index. Makino and Uno [10] show that, based on Lemma 3.2, any maximal clique K ($\neq K_0$) has at most Δ^2 children, and each checking of conditions (a)–(d) can be accomplished in $O(\Delta^2)$ time, recall that Δ is the maximum degree of vertices in G. However, the root K_0 still has $\Omega(n)$ children in general. In order to reduce the delay between outputting consecutive maximal cliques, Makino and Uno [10] precompute all children of K_0 in $O(nm)$ preprocessing time.

4 Our Solution

In this paper, we consider generating maximal cliques in sparse graphs G. In a sparse graph, most of its vertices are of low degree, with only a few vertices of high degree. The maximum degree Δ does not necessarily to be small, e.g., some hub nodes can have very high degrees. We use the concept of H-value.

Definition 4.1 The *H-value* of a graph G, H, is the smallest integer satisfying $|\{v \in V \mid \delta(v) \geq H\}| \leq H$.

The H-value of a graph usually is a small value for read-world networks [6], and is guaranteed to be smaller than Δ. Theoretical analyses of H for scale-free networks are also studied in [3, 6]. Based on the concept of H-value, the size of maximum clique in a graph with H-value, H, can be at most H, and it achieves the largest value when there is an edge between each pair of vertices with degree at least H.

We propose algorithms to enumerate all maximal cliques that improve the algorithm proposed by Makino and Uno [10] in two aspects: (1) avoiding the $O(nm)$ time preprocessing; (2) reducing the delay between outputting consecutive maximal cliques. Our algorithms enumerate maximal cliques in polynomial delay without preprocessing. Before introducing the algorithms, we first refine the parent-child relationship.

4.1 Revised Parent-Child Relationship

We redefine the parent of a maximal clique K as $C(K_{\leq i-1})$ such that i is the maximum index satisfying $K_{\leq i-1} \neq \emptyset$ and $C(K_{\leq i-1}) \neq K$. Based on this definition, it is possible that some maximal clique K ($\neq K_0$) has no parent. It is easy to verify that, if a maximal clique K has a parent, then $p(K) \geq 2$. Then, we define the parent index of those maximal cliques that do not have parents as 1.

Lemma 4.3 *Our revised parent-child relation constructs a forest $\mathcal{F}$ of all maximal cliques.*

Each maximal clique in G corresponds to one node in $\mathcal{F}$, and each node in $\mathcal{F}$ is a maximal clique. We call $\mathcal{F}$ as *enumeration forest*. Our goal is to traverse this enumeration forest efficiently to enumerate all maximal cliques.

Lemma 4.4 *There is a one-to-one correspondence between the set of roots of the trees in $\mathcal{F}$ and the set of maximal cliques $\{C(\{v_i\}) \mid v_i \in V, \Gamma(v_i) \cap V_{\leq i} = \emptyset\}$.*

Proof sketch Consider an arbitrary root in $\mathcal{F}$, $K = \{v_{i_1}, \ldots, v_{i_l}\}$. Without loss of generality, assume that $i_1 < i_2 < \cdots < i_l$. Because K does not have any parent, then $C(\{v_{i_1}\}) = K$. Therefore, $\Gamma(v_{i_1}) \cap V_{\leq i_1} = \emptyset$.

Consider an arbitrary $C(\{v_i\})$ with $\Gamma(v_i) \cap V_{\leq i} = \emptyset$. Assume $C(\{v_i\}) = \{v_{i_1}, \ldots, v_{i_l}\}$, with $i_1 < i_2 < \cdots < i_l$. It is easy to see that $v_{i_1} = v_i$. Based on the definition of $C(K)$, $C(\{v_{i_1}, \ldots, v_{i_j}\}) = C(\{v_i\})$, for any $1 \leq j \leq l$. Therefore $C(\{v_i\})$ has no parent, and it is a root in $\mathcal{F}$. □

From the above lemma, we can get all the roots of trees in $\mathcal{F}$ efficiently. We call the vertices satisfying Lemma 4.4 *starting vertices*, i.e., v_i is a starting vertex for $C(\{v_i\})$ if $\Gamma(v_i) \cap V_{\leq i} = \emptyset$. Following, we show a lemma similar to Lemma 3.2 that characterizes children of any maximal clique based on our revised definition of parent.

Lemma 4.5 *Let K and K' be maximal cliques in G, then K' is a child of K if and only if $K' = C((K_{\leq i} \cap \Gamma(v_i)) \cup \{v_i\})$ holds for some i such that*

(a) $K_{\leq i} \cap \Gamma(v_i) \neq \emptyset$;
(b) $v_i \notin K$;
(c) $i > p(K)$;
(d) $K_{\leq i} \cap \Gamma(v_i) = C((K_{\leq i} \cap \Gamma(v_i)) \cup \{v_i\})_{\leq i-1}$;
(e) $K_{\leq i} = C(K_{\leq i} \cap \Gamma(v_i))_{\leq i}$.

Moreover, if an index i satisfies (a)–(e), *then i is the parent index of K', i.e., $p(K') = i$.*

Proof sketch Proof of the "only if" part. If K' is a child of K, then K' is also a child (as defined by Lemma 3.2) of K. Therefore, conditions (b)–(e) hold based on Lemma 3.2. We prove condition (a) by contradiction. Assume condition (a) does not hold, i.e., $K_{\leq i} \cap \Gamma(v_i) = \emptyset$. By condition (d), $C((k_{\leq i} \cap \Gamma(v_i)) \cup \{v_i\})_{\leq i-1} = \emptyset$, i.e., $C(\{v_i\})_{\leq i-1} = \emptyset$. Then, based on Lemma 4.4, K' is a root in $\mathcal{F}$, which contradicts the fact that K' is a child of K.

Proof of the "if" part. If conditions (b)–(e) hold, based on Lemma 3.2, i is the maximum index satisfying $C(K'_{\leq i-1}) \neq K'$, and $C(K'_{\leq i-1}) = K$. From conditions (d) and (a), $K'_{\leq i-1} = K_{\leq i} \cap \Gamma(v_i) \neq \emptyset$. Therefore, K is a parent of K', and the parent index of K' is i. □

Theorem 4.1 *Based on the definition of roots of $\mathcal{F}$ from Lemma 4.4 and the definition of children for any maximal clique from Lemma 4.5, it can generate all maximal cliques in G without duplication.*

Based on Lemma 4.4, we can find the set of starting vertices in $O(n)$ time, each of which corresponds to the root of one enumeration tree in $\mathcal{F}$. This avoids the $O(nm)$ time preprocessing of Makino and Uno's algorithm.

4.2 Recursive Algorithm

From the previous subsection, we can design a naive enumeration algorithm. However, the delay time will highly depend on Δ (as in [10]), which still can be large for sparse graph. In the following, we introduce a novel partition of G into two parts, so that the delay time is largely reduced. The motivation is that, in a sparse graph, the hardness is usually caused by the few vertices with high degrees. So, we give special treatments to the vertices with high degrees.

In a more general situation, for a specific integer $\theta \leq \Delta$, we partition the vertices into two sets, and call all the vertices with degree at least θ as *high-nodes* (*h-nodes*), and all the other vertices as *non-high-nodes* (*nh-nodes*). Without loss of generality, we assume that $\{v_1, \ldots, v_\beta\}$ is exactly the set of vertices with degree at least θ, i.e., $\delta(v_i) \geq \theta, \forall 1 \leq i \leq \beta$ and $\delta(v_i) < \theta, \forall \beta < i \leq n$. We use the adjacency list to store edges, since it is the most efficient data structure occupying only linear space, i.e., it stores the neighbors of v for each $v \in V$. We assume that the neighbors $\Gamma(v)$ are stored in an increasing index order. Let $\Gamma_j(v)$ denote the j-th (starting from 1) neighbor of v in this order. Note that the size of $\Gamma(v)$ is at least one, i.e., $|\Gamma(v)| \geq 1$, since G is connected.

In order to enumerate all maximal cliques in a graph, there are two tasks to be done: (1) find all the roots in $\mathcal{F}$ based on Lemma 4.4; (2) enumerate children for maximal cliques found so far based on Lemma 4.5. For the first task, using our adjacency list storage, we can find the set of starting vertices for roots in $\mathcal{F}$ in $O(n)$ time, denoted as Q, which is exactly the set of vertices $\{v_i \in V \mid v_i < \Gamma_1(v_i)\}$. However, this requires an $O(n)$ preprocessing time. We will show how to avoid this preprocessing time by generating Q incrementally.

The pseudocode of enumerating all maximal cliques in graph G is shown in Algorithm 1. The set of starting vertices Q is initialized as $\{v_1\}$ (Line 2), since v_1 is a starting vertex for $C(\{v_1\})$. *lastChecked* is initialized as 1 (Line 3), which indicates the set of vertices that have been checked for starting vertices. Then, while Q is not empty (Line 4), let v be the first vertex in Q (Line 5), we enumerate the maximal cliques in the tree rooted at $C(\{v\})$ in $\mathcal{F}$ (Line 6).

In order to enumerate all maximal cliques in an enumeration tree, through procedure GENERATE, we need to generate children of K. Let I be set of indices satisfying conditions (a)–(e) in Lemma 4.5, which is got from a procedure call of GENERATECHILD and will be discussed later. In the procedure GENERATE, we first compute $K \leftarrow C(X)$, which is the unique maximal clique determined by X (Line 8), then get I (Line 12), and enumerate the subtrees rooted at each child (Lines 13–14). Note that, in the recursive calls of GENERATE, it outputs the maximal clique K before its descendants if the depth d is odd (Lines 10–11), otherwise it outputs K after its descendants (Lines 15–16). This special treatment is needed to guarantee polynomial delay [10]. After generating a maximal clique K, at Line 9, we check the starting vertices Q incrementally. Each time, we check at most $|K|$ vertices in $O(|K|)$ time, this ensures the delay time and still guarantees the correctness.

Algorithm 1 RECURSIVE(G)

Input: A graph $G = (V, E)$, where edges are stored in adjacency list format.
Output: Enumerate all maximal cliques of G.

```
1: Assume v_1, ..., v_β are exactly the vertices with degree at least θ.
2: Initialize Q ← {v_1}.
3: Initialize lastChecked ← 1.
4: while Q ≠ ∅ do
5:     v ← Q.REMOVEFIRST().
6:     GENERATE({v}, 1).

7: Procedure GENERATE(X, d)
8: K ← C(X).
9: CHECKROOT(|K|).
10: if d is odd then
11:     Output K.
12: I ← GENERATECHILD(K).
13: for each i ∈ I do
14:     GENERATE((K ∩ Γ(v_i)) ∪ {v_i}, d + 1).
15: if d is even then
16:     Output K.

17: Procedure CHECKROOT(step)
18: while step > 0 and lastChecked < n do
19:     lastChecked ← lastChecked + 1.
20:     step ← step − 1.
21:     if v_lastChecked < Γ_1(v_lastChecked) then
22:         Q.ADDLAST(v_lastChecked).
```

4.2.1 Generating Children Indices

We show how to generate the children indices I for a maximal clique K efficiently. Recall that, I consists of indices i such that K is the parent of $C((K_{\leq i} \cap \Gamma(v_i)) \cup \{v_i\})$. Then, we can get children of K efficiently based on I.

Let c denote the size of maximum clique. From conditions (a) and (b) of Lemma 4.5, we can prove that $I \subseteq \Gamma(K)$. And $|\Gamma(K)| \leq \Delta \cdot c$, because $|K| \leq c$ and $\Gamma(v) \leq \Delta, \forall v \in V$. So, we can first compute $\Gamma(K)$, and then remove all the indices that do not satisfy either condition (c) or (d) or (e). Condition (c) is easy to check, we show how to check conditions (d) and (e) in the following. The pseudocode is shown in Algorithm 2 (GENERATECHILD). It initializes the candidate indices I' to be checked as $\Gamma(K)$ (Line 1). Then, for each $i \in I'$, add it to I if it satisfies conditions (c)–(e) (Lines 3–5). Line 2 finds v_x for K, which will be used when checking condition (e). Recall the two notations $\Gamma(S)$ and $\Lambda(S)$, where S is a set of vertices, which will be used in the following. $\Gamma(S)$ denote the set of vertices in V that are connected to any vertex in S, and $\Lambda(S)$ denote the set of vertices in V that are connected to all vertices in S. For a single vertex v, $\Gamma(v)$ is equal to $\Lambda(v)$.

Checking Condition (d) If we add $\{v_i\}$ on both side of condition (d), then it is equivalent to the following equation,

$$(K_{\leq i} \cap \Gamma(v_i)) \cup \{v_i\} = C((K_{\leq i} \cap \Gamma(v_i)) \cup \{v_i\})_{\leq i}$$

which is then equivalent to the condition that,

$$\Lambda((K_{\leq i} \cap \Gamma(v_i)) \cup \{v_i\})_{\leq i} = \emptyset. \tag{1}$$

Let $X \leftarrow (K_{\leq i} \cap \Gamma(v_i)) \cup \{v_i\}$, which can be computed in $O(\theta + \beta + c)$ time as follows: if $\delta(v_i) \geq \theta$, we compare only the first β neighbors of v_i with $K_{\leq i}$; otherwise, $\delta(v_i) < \theta$, we compare all the neighbors of v_i with $K_{\leq i}$.

Equation (1) is equivalent to checking whether there is any vertex $v_j (j \leq i)$ connecting to all vertices in X, i.e., $\Lambda(X)_{\leq i} = \emptyset$ where v_i is the vertex with largest index in X and the size of X is at most c. A naive method will need $O(c \cdot \Delta)$ time. We check it in two cases, i.e., whether $\delta(v_i)$ is smaller than θ or not, or equivalently whether there is an *nh-node* in X or not, where *nh-nodes* are the vertices with degree less than θ. If the degree of v_i is no smaller than θ, then we can check each $j \in [1, i]$, since there are at most β such indices. Each checking of connection between v_j and X can be done in $O(\beta + c)$ time by comparing $\Gamma(v_j)$ and X, where comparing two sorted lists can be done in a merge-sort fashion in linear time, since at most the first β neighbors of v_j need to be checked in this scenario and the size of X is bounded by $O(c)$.

Otherwise, the degree of v_i is smaller than θ. Let S denote the candidate indices j that need to be checked, X_H denote the set of *h-nodes* in X. Then $\Lambda(X)_{\leq i} = \Lambda(X \setminus X_H)_{\leq i} \cap \Lambda(X_H)_{\leq i}$ since $X = (X \setminus X_H) \cup X_H$. We have that, $X \setminus X_H \neq \emptyset$, and the size of S is less than θ, since $S \subseteq \Lambda(X \setminus X_H)_{\leq i}$. We can get $\Lambda(X \setminus X_H)_{\leq i}$ in $O(c \cdot \theta)$ time by intersecting the neighbors of *nh-nodes* in X, each of which is of size smaller than θ. Then, for each node in S, we can check whether it is connected to all nodes in X_H in $O(\beta + \theta + c)$ time. The pseudocode of procedure SatCondD is shown in Algorithm 2.

Lemma 4.6 *The procedure* SatCondD *successfully checks condition* (d) *in* $O((\beta + \theta) \cdot (\beta + \theta + c))$ *time.*

Checking Condition (e) Condition (e) is equivalent to the following condition that,

$$\nexists j < i, v_j \text{ is adjacent to all vertices in } (K_{\leq i} \cap \Gamma(v_i)) \cup K_{\leq j}. \tag{2}$$

Let k be the maximum (vertex) index in $K_{\leq i} \cap \Gamma(v_i)$, it always exists because $K_{\leq i} \cap \Gamma(v_i) \neq \emptyset$ is ensured by condition (a). Let S denote the set of indices j that need to be checked for (2). We divide S into two sets, $S_{\leq k}$ and $S_{>k}$, and process them differently, i.e., $S_{\leq k} = \{1 \leq j \leq k \mid v_j \in \Lambda((K_{\leq i} \cap \Gamma(v_i)) \cup K_{\leq j})\}$, and $S_{>k} = \{k < j < i \mid v_j \in \Lambda((K_{\leq i} \cap \Gamma(v_i)) \cup K_{\leq j})\}$. Condition (e) is satisfied if and only if $S_{\leq k} = \emptyset$ and $S_{>k} = \emptyset$.

For $S_{\leq k}$, we can prove that $S_{\leq k} \subseteq \Lambda(K_{\leq i} \cap \Gamma(v_i))_{\leq k}$. First, we can generate $\Lambda(K_{\leq i} \cap \Gamma(v_i))_{\leq k}$ in $O((\beta + \theta) \cdot (\beta + \theta + c))$ time using a similar procedure as

Algorithm 2 GENERATECHILD(K)

Input: A maximal clique K of graph G.
Output: Indices for its children.

Let $I' \leftarrow \Gamma(K)$, and $I \leftarrow \emptyset$.
Let v_x be the smallest vertex with $\delta(v_x) < \theta$, and adjacent to all vertices in $K_{\leq x}$.
for each $i \in I'$ **do**
  if $i > p(K)$ **and** SATCONDD(i) **and** SATCONDE(i) **then**
    $I \leftarrow I \cup \{i\}$.

Procedure SATCONDD(i)
Let $X \leftarrow (K_{\leq i} \cap \Gamma(v_i)) \cup \{v_i\}$.
if $i < \beta$ **then**
  for $j \leftarrow 1$ **to** i **do**
    if v_j is connected to all vertices in X **then**
      return false.
else
  Let $X_H \leftarrow \{v \in X \mid \delta(v) \geq \theta\}$.
  for each $v \in \Lambda(X \setminus X_H)_{\leq i}$ **do**
    if v is connected to all vertices in X_H **then**
      return false.
return true.

Procedure SATCONDE(i)
Let v_k be maximum vertex in $K_{\leq i} \cap \Gamma(v_i)$.
for each $j \in \Lambda(K_{\leq i} \cap \Gamma(v_i))_{\leq k}$ **do**
  if v_j is connected to all vertices in $K_{\leq j}$ **then**
    return false.
if $k < x$ **then**
  for $j \leftarrow k+1$ **to** $\min\{\beta, i\}$ **do**
    if v_j is connected to all vertices in $K_{\leq j}$ **then**
      return false.
  if $x < i$ **then**
    return false.
else
  for each $j \in \Gamma(v_k)_{>k}$ **do**
    if v_j is connected to all vertices in $K_{\leq j}$ **then**
      return false.
return true.

SATCONDD. The modification is as follow: at Line 11 and Line 16, we add the corresponding j into $S_{\leq k}$ instead of returning false. The size of $S_{\leq k}$ is at most $\max\{\beta, \theta\}$. Then, we can check each $j \in S_{\leq k}$ whether it is connected to all vertices in $K_{\leq j}$ or not in $O(\beta + \theta + c)$ time. If $j \leq \beta$, we compare the first (at most) β neighbors of v_j with $K_{\leq j}$, otherwise, we compare the (at most) θ neighbors of v_j with $K_{\leq j}$. The total time complexity to check $S_{\leq k}$ is $O((\beta + \theta) \cdot (\beta + \theta + c))$.

For $S_{>k}$, it is equivalent to check whether v_j is connected to all vertices in $K_{\leq j}$, i.e., $S_{>k} = \{k < j < i \mid v_j \in \Lambda(K_{\leq j})\}$, since $(K_{\leq i} \cap \Gamma(v_i)) \subseteq K_{\leq j}$ is ensured by the definition of k. Note that, we do not need to compute the exact set of $S_{>k}$, but

only need to check whether $S_{>K} = \emptyset$ which is simpler. Let v_x denote the smallest vertex with degree smaller than θ that is connected to all vertices in $K_{\leq x}$, v_x can be assigned as v_{n+1} if no such a vertex exists. Note that, v_x is computed only once for each K, as shown in Line 2 of Algorithm 2. We consider the checking problem in two cases. (1) If $k < x$, then the candidate $S_{>k}$ can be divided into two ranges, $(k, \min\{\beta, i\}]$ and $[x, i]$, and all the candidates $\beta < j < x$ are not possible as checked by the computation of v_x. Note that, one or both of these two ranges can be empty, e.g., $[x, i]$ is empty if $x > i$. For the first range, we can check each j satisfying $k < j \leq \min\{\beta, i\}$ whether v_j is connected to all vertices in $K_{\leq j}$ or not in $O(\beta + c)$ time. For range $[x, i]$, if $x < i$, then v_x is adjacent to all vertices in $(K_{\leq i} \cap \Gamma(v_i)) \cup K_{\leq x}$, therefore condition (e) is not satisfied, otherwise condition (e) is satisfied. The correctness of this case can be verified by discussing it in two cases depending on the result of the comparison between β and i. (2) If $k \geq x$, then $S_{>k} \subseteq \Gamma(v_k)_{>k}$ and $\delta(v_k) < \theta$. We can check each $j \in \Gamma(v_k)_{>k}$ in $O(\theta + c)$ time. The time complexity to check $S_{>k}$ is $O((\beta + \theta) \cdot (\beta + \theta + c))$. The pseudocode of procedure SATCONDE is shown in Algorithm 2.

Lemma 4.7 *The procedure* SATCONDE *successfully checks condition* (e) *in* $O((\beta + \theta) \cdot (\beta + \theta + c))$ *time, excluding the time to compute* v_x.

Theorem 4.2 *The algorithm* GENERATECHILD *successfully finds all children indices for a maximal clique in* $O(\Delta \cdot c \cdot (\beta + \theta) \cdot (\beta + \theta + c))$ *time.*

Proof sketch The correctness of GENERATECHILD direct follows from Lemma 4.5, Lemma 4.6, and Lemma 4.7.

At Line 1 of algorithm GENERATECHILD, it computes $\Gamma(K)$ in $O(\Delta \cdot c^2)$ time, and the size of I' is upper bounded by $\Delta \cdot c$. At Line 2, it computes v_x in $O(\Delta \cdot c)$ time. Then, for each $j \in I'$, it checks conditions (c)–(e) in $O((\beta + \theta) \cdot (\beta + \theta + c))$ time. Therefore, the total time complexity of GENERATECHILD is $O(\Delta \cdot c \cdot (\beta + \theta) \cdot (\beta + \theta + c))$. □

Note that, for a graph G, Δ and c are determined by the graph, while θ is chosen by a user (or a program). Recall that, for a specific θ, β is the number of vertices with degree at least θ. Therefore, β decreases when θ increasing, and β increases when θ decreasing. The time complexity $O(\Delta \cdot c \cdot (\beta + \theta) \cdot (\beta + \theta + c))$ achieves its optimal value when $\beta = \theta$, which is equal to the *H-value*, H, of the graph. We call this as *H-partition* when $\beta = \theta$. Note that, the size of maximum clique in a graph with H-value H is at most H.

Theorem 4.3 *Our algorithm* RECURSIVE *achieves its optimal time complexity when* $\beta = \theta = H$, *then it enumerates all maximal cliques in* $O(\Delta \cdot H^3)$ *delay and* $O(n \cdot \Delta \cdot H + m)$ *space.*

The recursive algorithm (Algorithm 1) needs to store one maximal clique K and its candidate children indices I at each level of the recursive call of the procedure GENERATE, where I is of size $\Delta \cdot H$ and the maximum level of recursive calls can

be $O(n)$. So, the total space complexity is $O(n \cdot \Delta \cdot H + m)$. Note that, this space complexity analysis is very loose.

4.3 Linear Space Algorithm

In this section, we introduce a non-recursive algorithm, whose space complexity is strictly bounded by $O(n + m)$. The general idea is the same as RECURSIVE. We show LINEARGENERATE (see Algorithm 3) as a replacement of GENERATE of Algorithm 1. First, we use a stack to simulate the DFS-traversal on the enumeration forest. When processing a node (corresponding to one maximal clique) in the enumeration forest, the stack stores all the nodes in the path from root to that node.

In order to achieve the linear space constraint, we need to avoid (1) the storage of maximal cliques (2) and the children indices I, for all maximal cliques on the enumeration path. In our scenario, the parent of a maximal clique K, $P(K)$, can be computed as $C(K_{\leq p(K)-1})$. For all the maximal cliques in the enumeration path to K, we do not need to store the actual maximal cliques, but can store only the parent indices. The actual maximal clique can be computed from one of its children. For the children indices I, we do not compute and store it in one unit, but compute each child index on-demand.

The pseudocode of LINEARGENERATE is shown in Algorithm 3. A stack $\mathcal{S}$ is used to simulate the recursive call of GENERATE. Each entry of $\mathcal{S}$ consists of three fields, $(p(K), x, i)$, where $p(K)$ is used to compute the parent of K, x is computed as Line 2 in Algorithm 2 to reduce the delay of generating children indices, and i is an integer which indicates which subtrees of K have been enumerated. Initially, the root K is pushed into $\mathcal{S}$ (Line 3). While $\mathcal{S}$ is not empty (Line 5), it processes the top entry of $\mathcal{S}$, denote it as $(p(K), x, i)$ (Line 6). At this step, all the subtrees rooted at $C((K_{\leq j-1} \cap \Gamma(v_j)) \cup \{v_j\})$ with $j \leq i$ (and j is a child index of K) have been enumerated. We find the next child of K by calling NEXTCHILD (Line 7). If there is no more child of K (Line 8), then we go back to its parent $P(K)$ (Line 12). Otherwise, we update the child index of the top entry of $\mathcal{S}$ (Line 14), and push the newly generated child to $\mathcal{S}$ (Line 16). NEXTCHILD can be implemented in a similar way as Algorithm 2, where I' is initialized as $\Gamma(K)_{>i}$ at Line 1 and it returns only the first index in I. Note that, considering the loop consists of Lines 6–18 of LINEARGENERATE, it outputs at least one maximal clique for each two consecutive execution of the loop.

Theorem 4.4 *The algorithm* LINEARGENERATE *enumerates all maximal cliques in* $O(\Delta \cdot H^3)$ *delay and* $O(n + m)$ *space.*

4.4 A Remark on Makino and Uno's Algorithm

We give a remark on Makino and Uno's Algorithm based on the H-value H of a graph.

Theorem 4.5 *With a preprocessing time of* $O(nm)$, *Makino and Uno's algorithm can enumerate maximal cliques in* $O(\Delta^2 \cdot H^2)$ *time delay.*

Algorithm 3 LINEARGENERATE(X)

Input: A graph $G = (V, E)$, where edges are stored in adjacency list format.
Output: Enumerate all maximal cliques of G.

```
1: K ← C(X).
2: Initialize stack S to be empty.
3: AddStack(S, K, 1).
4: Output K.
5: while not S.empty() do
6:    (p(K), x, i) ← S.top().
7:    i ← NextChild(K, i).
8:    if i = null then
9:       if |S| is even then
10:         Output K.
11:      S.pop().
12:      K ← P(K).
13:   else
14:      S.updateTop(p(K), x, i).
15:      K ← C((K_{≤i} ∩ Γ(v_i)) ∪ {v_i}).
16:      AddStack(S, K, i).
17:      if |S| is odd then
18:         Output K.

19: Procedure AddStack(S, K, p(K))
20: CheckRoot(|K|).
21: Let v_x be the smallest vertex with δ(v_x) < θ, and adjacent to all vertices in K_{≤x}.
22: S.push(p(K), x, p(K)).
```

Proof sketch As shown previously, for a graph with H-value H, the size of a maximal clique can be at most H. Then each maximal clique (excluding K_0) has at most $O(\Delta \cdot H)$ children, since the degree of every vertex is no larger than Δ. For each of the $O(\Delta \cdot H)$ candidate indices, the conditions (a)–(d) can be checked in $O(\Delta \cdot H)$ time, since the size of $(K_{\leq i} \cap \Gamma(v_i)) \cup \{v_i\}$ and $K_{\leq i} \cap \Gamma(v_i)$ in the right hand sides of conditions (c) and (d) are both at most H. □

Our algorithms do not need preprocessing which is $O(nm)$ in Makino and Uno's algorithm. Our time complexity is $O(\Delta \cdot H^3)$, whereas Makino and Uno's algorithm is $O(\Delta^2 \cdot H^2)$. It is worth noting that $H \ll \Delta$ in practice in real-world large sparse graphs.

5 Conclusion

In this paper, we studied the problem of enumerating all maximal cliques in a sparse graph G in polynomial delay and linear space. We first proposed a revised parent/child relationship between maximal cliques, based on which our algorithms avoid preprocessing and achieve polynomial delay. Then, we proposed a novel partition of G based on its vertex degrees, H-partition. Base on the H-partition, we proposed

efficient algorithms to generate children maximal cliques for an arbitrary maximal clique K. By combining these two techniques, our algorithm enumerates all maximal cliques in G without duplication in $O(\Delta \cdot H^3)$ time delay and $O(n+m)$ space.

Acknowledgement The work was supported by grants of the Research Grants Council of the Hong Kong SAR, China No. 419008 and 419109.

References

1. Akkoyunlu, E.A.: The enumeration of maximal cliques of large graphs. SIAM J. Comput. **2**(1), 1–6 (1973)
2. Bron, C., Kerbosch, J.: Finding all cliques of an undirected graph (algorithm 457). Commun. ACM **16**(9), 575–576 (1973)
3. Cheng, J., Ke, Y., Fu, A.W.-C., Yu, J.X., Zhu, L.: Finding maximal cliques in massive networks by h*-graph. In: SIGMOD, pp. 447–458 (2010)
4. Chiba, N., Nishizeki, T.: Arboricity and subgraph listing algorithms. SIAM J. Comput. **14**(1), 210–223 (1985)
5. Cohen, S., Fadida, I., Kanza, Y., Kimelfeld, B., Sagiv, Y.: Full disjunctions: Polynomial-delay iterators in action. In: VLDB, pp. 739–750 (2006)
6. Eppstein, D., Spiro, E.S.: The h-index of a graph and its application to dynamic subgraph statistics. In: WADS, pp. 278–289 (2009)
7. Harley, E., Bonner, A.J., Goodman, N.: Uniform integration of genome mapping data using intersection graphs. Bioinformatics **17**(6), 487–494 (2001)
8. Johnson, D.S., Papadimitriou, C.H., Yannakakis, M.: On generating all maximal independent sets. Inf. Process. Lett. **27**(3), 119–123 (1988)
9. Kumar, R., Raghavan, P., Rajagopalan, S., Tomkins, A.: Trawling the web for emerging cyber-communities. Comput. Netw. **31**(11–16), 1481–1493 (1999)
10. Makino, K., Uno, T.: New algorithms for enumerating all maximal cliques. In: SWAT, pp. 260–272 (2004)
11. Modani, N., Dey, K.: Large maximal cliques enumeration in sparse graphs. In: CIKM, pp. 1377–1378 (2008)
12. Mohseni-Zadeh, S., Brézellec, P., Risler, J.-L.: Cluster-c, an algorithm for the large-scale clustering of protein sequences based on the extraction of maximal cliques. Comput. Biol. Chem. **28**(3), 211–218 (2004)
13. Tomita, E., Tanaka, A., Takahashi, H.: The worst-case time complexity for generating all maximal cliques and computational experiments. Theor. Comput. Sci. **363**(1), 28–42 (2006)
14. Tsukiyama, S., Ide, M., Ariyoshi, H., Shirakawa, I.: A new algorithm for generating all the maximal independent sets. SIAM J. Comput. **6**(3), 505–517 (1977)