Another Time-Complexity Analysis for Maximal Clique Enumeration Algorithm CLIQUES

Etsuji TOMITA, Alessio CONTE

著者（英） | 著者（英）
--- | ---
Etsuji TOMITA | Alessio CONTE

著者（日） | 著者（日）
--- | ---
伊東延二 | アレッソ・コンテ

発行日 | 2020年5月

URL | http://id.nii.ac.jp/1438/00009571/
Another Time-Complexity Analysis for
Maximal Clique Enumeration Algorithm CLIQUES

Etsuji TOMITA† and Alessio CONTE††

† The Advanced Algorithms Research Laboratory, The University of Electro-Communications
Chofugaoka1–5–1, Chofu, Tokyo 182–8585, Japan
†† Department of Computer Science, University of Pisa
Largo Bruno Pontecorvo 3, 56127 Pisa, Italy
E-mail: †e.tomita@uec.ac.jp, ††conte@di.unipi.it

Abstract We revisit the maximal clique enumeration algorithm CLIQUES that appeared in Theoretical Computer Science 2006. It is proved to work in \(O(3^{n/3})\)-time in the worst-case for an \(n\) vertex graph. In this note, we extend the time-complexity analysis with respect to the number of maximal cliques, an issue that was left as an open problem since TCS 2006.

Key words maximal clique, maximal independent set, enumeration, algorithm, time-complexity, delay

1. Introduction

A clique is defined to be a subgraph in which every pair of vertices are adjacent. In particular, it is maximal if it is not contained in a properly larger clique. Given a graph, the enumeration of all its maximal cliques is a fundamental and important problem in graph theory. In addition, it has many applications as in clustering, data mining, bioinformatics, social networks, and many others. An independent set of a graph \(G\) is a clique of the complementary graph \(\overline{G}\).

Tsukiyama et al. was the first to give an algorithm for enumerating all maximal independent sets with theoretical time-complexity analysis. Given a graph \(G\) with \(n\) vertices and \(m\) edges, their algorithm takes \(\mu O(nm)\)-time for enumerating all maximal independent sets, where \(\mu\) is the number of maximal independent sets. This implies that their algorithm enumerates all maximal cliques in \(\alpha O(n(n^2 - m))\)-time for \(G\), where \(\alpha\) is the number of maximal cliques in \(G\). Their approach is based on so-called reverse search [1] approach. On the other hand, Tomita et al. [20], [21] presented another algorithm CLIQUES for enumerating all maximal cliques. CLIQUES is independently developed based on depth-first search algorithms for finding a maximum clique [11], [19], [23], but the pruning techniques in CLIQUES are found to be the same as in Bron and Kerbosch algorithms [4] for the same problem. CLIQUES is proved to enumerate all maximal cliques in \(O(3^{n/3})\)-time for an \(n\)-vertex graph in the worst-case. This is optimal as a function of \(n\) since there exist up to \(3^{n/3}\) maximal cliques as in Moon-Moser graphs [17]. CLIQUES is the first algorithm with theoretical time-complexity analysis in this approach.

In the reverse search approach after Tsukiyama et al., steady improvements have been made by Chiba and Nishizeki [6], Johnson et al. [13], Makino and Uno [15], Chang et al. [5], and Conte et al. [7], [9]. One of Conte et al.’s algorithms requires \(\alpha O(maxQ \cdot d\Delta)\)-time for enumerating all maximal cliques, where \(maxQ\) is the size of a maximum clique, \(d\) is the degeneracy of \(G\), that is defined as the smallest number such that every subgraph of \(G\) contains a vertex of degree at most \(d\). Based on CLIQUES, Eppstein et al. proposed an improved algorithm that runs in \(O(dn3^{d/3})\)-time, where degeneracy \(d\) is much smaller than \(n\) for sparse graphs. In general, it is experimentally confirmed that if the number of cliques is small then the running time for their enumeration is small. However, no theoretical time-complexity analysis with respect to the number of maximal cliques is
made after CLIQUES [21], where the problem is noted in [21] as an important open problem. This is in con- 
trast to the time-complexity analysis in reverse search approach. Example runs by two approaches for the same 
problem can be found in [22]. For early histories and ap- 
lications, see the excellent review articles [18], [2] where 
CLIQUES [20] is also included for review.

In this note, we are concerned with another time- 
complexity analysis of CLIQUES and others in the same 
approach other than that with respect to the number of 
vertices in the worst-case.

2. Definitions and notation

(1) We are concerned with a simple undirected graph
G = (V, E) with a finite set V of vertices and a finite 
set E of unordered pairs (v, w) of distinct vertices, called 
edges. A pair of vertices v and w are said to be adjacent 
if (v, w) ∈ E.

(2) For a vertex v ∈ V, let Γ(v) be the set of all 
vertices that are adjacent to v in G = (V, E), i.e., 
Γ(v) = {w ∈ V | (v, w) ∈ E} ( ≠ v).

(3) For a subset W ⊆ V of vertices, G(W) = (W, E(W)) 
with E(W) = {(v, w) ∈ W × W | (v, w) ∈ E} is called 
a subgraph of G = (V, E) induced by W. For a set W 
of vertices, | W | denotes the number of elements in W.

(4) Given a subset Q ⊆ V of vertices, if (v, w) ∈ E for all 
v, w ∈ Q with v ≠ w then the induced subgraph G(Q) 
is said to be a clique. In this case, we may simply say 
that Q is a clique. If a clique is not a proper subgraph 
of another clique then it is called a maximal clique.

3. Maximal clique enumeration algorithm

CLIQUES

We revisit a depth-first search algorithm, CLIQUES [20], [21] for enumerating all maximal cliques of an undi- 
rected graph G = (V, E) with |V| = n. All maximal 
cliques found are output in a tree-like form. The basic 
framework of CLIQUES is almost the same as that for 
finding a maximum clique without the bounding condi- 
tion [11], [19], [23]. We maintain a global variable Q = 
{p1, p2, ..., pk} that consists of the vertices of a current 
clique, and let SUBG = V ∩ Γ(p1) ∩ Γ(p2) ∩ ... ∩ Γ(pk). 
We begin the algorithm by letting Q ← ∅ and SUBG ← 
V (the set of all vertices). We select a certain vertex 
p from SUBG and add p to Q. Then, we compute 
SUBG_p = SUBG ∩ Γ(p) as the new set of vertices 
in question. In particular, the initially selected vertex 
u ∈ SUBG is called a pivot. This CLIQUES() procedure 
is applied recursively while SUBG_p ≠ ∅.

We describe two methods to prune unnecessary parts 
of the searching, which happen to be the same as in the 
Bron-Kerbosch algorithms [4]. We regard the set SUBG 
(= V at the beginning) as an ordered set of vertices, and 
we continue to enumerate maximal cliques from vertices 
in SUBG step-by-step in this order

The first pruning method : Let FINI be a subset 
of vertices of SUBG that have already been processed 
by the algorithm (FINI is short for FINISHED). Then 
we denote by CAND the set of remaining CANDIDATE 
for expansion: CAND = SUBG \ FINI. Initially, 
FINI ← ∅ and CAND ← SUBG = V. In the subgraph 
G(SUBG_p) with SUBG_p = SUBG ∩ Γ(p), compute 
CAND_p = CAND ∩ Γ(p), 
FINI_p = FINI ∩ Γ(p).

Then only the vertices in CAND_p excluding FINI_p can 
be candidates for expanding the clique Q ∪ {p} to find 
new larger cliques.

The second pruning method : For the initially 
selected pivot u in SUBG, any maximal clique Q' in 
G(SUBG ∩ Γ(u)) is not maximal in G(SUBG), since 
Q' ∪ {u} is a larger clique in G(SUBG). Therefore, 
searching for maximal cliques from SUBG ∩ Γ(u) should 
be excluded.

Algorithm 1: Algorithm CLIQUES in [21]

<table>
<thead>
<tr>
<th>Input</th>
<th>A graph G = (V, E).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>All maximal cliques in G.</td>
</tr>
</tbody>
</table>

1 CLIQUES (∅, V, ∅)

2 Function CLIQUES(Q, CAND, FINI)

3 if CAND ∪ FINI = ∅ then

4 // Q is a maximal clique

5 print ("clique, ")

6 else

7 u ← a vertex in CAND ∪ FINI maximizing | CAND ∩ Γ(u)|

8 foreach p ∈ CAND \ Γ(u) do

9 print (p, ")

10 CLIQUES (Q ∪ {p}, CAND ∩ Γ(p), FINI ∩ Γ(p))

11 CAND ← CAND \ {p}

12 FINI ← FINI ∪ {p}

13 print ("back," )
Taking the previously described pruning method into consideration, the only search subtrees to be expanded are from vertices in \((\text{SUBG} \setminus (\text{SUBG} \cap \Gamma(u))) \setminus \text{FINI} = \text{CAND} \setminus \Gamma(u))\). Here, to minimize \(|\text{CAND} \setminus \Gamma(u)|\), we choose the pivot \(u \in \text{SUBG}\) that maximizes \(|\text{CAND} \cap \Gamma(u)|\), which is crucial to establish the optimality of the worst-case time-complexity of the algorithm.

The algorithm CLIQUES [20], [21] is shown as Algorithm 1 in the previous page. It enumerates all maximal cliques in \(O(3^{n/3})\)-time in the worst-case based upon the above methods, where all maximal cliques enumerated are presented in a tree-like form [20], [21]. We can easily obtain a tree representation of all the maximal cliques from the output sequence, where a dummy root is added to form a tree (Fig.4 of [21]). The tree-like output format is also important practically, since it saves space in the output file.

**Search tree**

The process of enumerating all maximal cliques of \(G = (V, E)\) by CLIQUES \((\emptyset, V, \emptyset)\) is represented by a search tree:

- The root of the search tree is a newly introduced dummy root \(p_0 \notin V\) to form a tree.
- Every vertex in \(V\) is a child of the dummy root \(p_0\). The vertex of \(V\) is called a node in the search tree.
- Assume we have a path from the dummy root \(p_0\) to a certain node \(p_h\) in the search tree as a sequence of nodes \(p_0, p_1, p_2, ..., p_h\), and let \(\text{SUBG}_h = V \cap \Gamma(p_1) \cap \Gamma(p_2) \cap ... \cap \Gamma(p_h)\).

Then, every vertex in \(\text{SUBG}_h\) is a child of \(p_h\) in the search tree.

Suppose \(u\) with the maximum degree is chosen as a pivot in \(\text{SUBG}_h\) then every node in \(\Gamma(u)\) is a leaf since it should not be expanded by CLIQUES according to the second pruning method. Such a leaf in \(\Gamma(u)\) is called a black node.

![An example run of CLIQUES](image)

**Figure 1** An example run of CLIQUES
Suppose $\text{SUBG}_b = \text{FINI}_b \cup \text{CAND}_b, q_i, q_j \in \text{SUBG}_b \setminus \Gamma(u)$, where $i < j$ and $q_i, q_j$ are adjacent with $q_i \in \text{FINI}_b$.

Then $q_j \in \text{SUBG}_b$ has a child $q_i$ since $q_i$ and $q_j$ are adjacent in $\text{SUBG}_b$, but the child $q_i$ should not be expanded by CLIQUES according to the first pruning method, and hence it is a leaf of the search tree. Such a leaf $q_i$ is called a bad node or bad leaf. Note that if a bad node were expanded it could not lead to a new maximal clique.

- When the above $\text{SUBG}_b$ is a singleton $\{q_1\}$ then the $q_1$ is a leaf in the search tree.

The search tree consists of the nodes from $V \cup \{p_0\}$ and the parent-child relationship holds if and only if one of the above conditions holds.

Let $q_i$ be a child of $q_0$, then the set $\{p_1, p_2, ..., p_h, p_i\}$ constitutes a clique. This clique is called an accompanying clique and is denoted by $Q_{p_0:p_1:...:p_h}$, or simply $Q_q$, or $Q$ when it is clear.

See Figure 1 of a search tree for an example run, and Figure 2 of a general search tree.

### 4. Bron-Kerbosch algorithms

Let us recall the antecedent algorithms by Bron and Kerbosch for enumerating maximal cliques [4].

The main difference between theirs (Bron-Kerbosch) and ours (CLQUIES) is how the pivot node is chosen on Line 7.

We call BK the original Bron-Kerbosch algorithm without pivoting, which corresponds to entirely removing Line 7, and replacing Line 8 with "foreach $v \in \text{CAND}$ do".

Next, we call BKP the original Bron-Kerbosch algorithm with pivoting, where no specific pivot choice is required, i.e., Line 7 can be replaced with "$u \leftarrow$ an arbitrary vertex in $\text{CAND} \cup \text{FINI}$".

In addition, CLIQUES outputs all maximal cliques in a tree-like format in order to avoid the time for outputting a maximal clique every time it is found that is proportional to the size of a maximal clique found.

The pivot selection of the maximum-degree and the tree-like outputting are crucial in CLIQUES so that it accomplishes the worst-case optimal $O(3^n/3)$-time complexity.

The original Bron-Kerbosch algorithm BKP (with arbitrary pivot choice) should take $O(n3^n/3)$-time in the worst-case, while BK (without pivoting at all) runs in $\Theta(n2^n)$ on a complete graph, as it essentially generates all subsets of every clique.

### 5. The time-complexity of CLIQUES

We analyze the time-complexity of CLIQUES with respect to the output.

#### 5.1 The overall time-complexity of CLIQUES with respect to the output

##### 5.1.1 Size of the search tree

In this section we aim at bounding the number of nodes in the search tree with respect to the number of solutions.

Suppose we have a path from the dummy root $p_0$ to a certain node $p_h$ in the search tree as a sequence of nodes $p_0, p_1, p_2, ..., p_h$. Then the set $\{p_1, p_2, ..., p_h\}$ constitutes an accompanying clique.

We are interested in the accompanying clique $Q$ across different search tree nodes, and in particular let us observe the following:

**Lemma 1.** The accompanying clique $Q$ is distinct in any internal (non-leaf) node of a search tree of CLIQUES.

**Proof.** Let $x$ and $y$ be any pair of non-root internal nodes in the search tree of CLIQUES, where $x$ is generated before $y$ in the search tree. Let the nearest common ancestor of $x$ and $y$ be $p_h$, and let the path from the dummy root $p_0$ to node $p_h$ be $p_0, p_1, p_2, ..., p_h$. We then let the path from the dummy root $p_0$ to node $p_h$ be $p_0, p_1, p_2, ..., p_h$ with the accompanying clique $Q_{p_h} = \{p_1, p_2, ..., p_h\}$. In addition,
let the path from \( p_h \) to \( x \) be \( p_h, q_1, \ldots, x \) and that from \( p_h \) to \( y \) be \( p_h, q_1, \ldots, y \), respectively, and let \( \text{SUBG}_h = \Gamma(p_1) \cap \Gamma(p_2) \cap \ldots \cap \Gamma(p_h) = \{ q_1, q_2, \ldots, q_{i-1}, \ldots, q_j, \ldots, q_m \} \) \((i < j)\). See Figures 1 and 2. So, \( Q_x = Q_{p_h} \cup \{ q_i, \ldots, x \} \) and \( Q_y = Q_{p_h} \cup \{ q_j, \ldots, y \} \). When we visit node \( q_j \) we see that node \( q_j \) is in \( \text{FINI}_h = \text{SUBG}_h \setminus \text{CAND}_h \) where \( \text{CAND}_h = \{ q_j, \ldots, q_m \} \) at that moment. If \( q_i \notin \Gamma(q_j) \) then it clearly follows that \( q_i \notin Q_y \). When \( q_i \in \Gamma(q_j) \), \( q_i \) in the descendants of \( q_j \) is a bad node (leaf) since the previous node \( q_j \) is in \( \text{FINI}_h \). Moreover, the bad node \( q_i \) is not in the path \( q_j, \ldots, y \) since all nodes \( q_j, \ldots, y \) are internal nodes by the assumption. In any case, it follows that \( q_i \in Q_x \setminus Q_y \) and the lemma is proved.

We observe that the accompanying clique \( Q \) is always a subset of a maximal clique. Let \( \text{maxQ} \) be the size of a maximum clique and \( \mu \) the number of maximal cliques in \( G \).

**Lemma 2.** The number of nodes of a search tree of CLIQUES is at most \((1 + \Delta)\mu(2^{\text{maxQ}} - 1)\).

**Proof.** The above observation immediately implies that the number of internal nodes is bounded by the number of distinct non-maximal cliques of \( G \). Each maximal clique has at most \( 2^{\text{maxQ}} - 1 \) distinct subsets whose size is less than or equal to \( \text{maxQ} - 1 \). (The subset of size \( 0 \) corresponds to the dummy root \( p_0 \) and the subset of size \( \text{maxQ} \) is excluded.) So, the number of internal nodes is at most \( \mu(2^{\text{maxQ}} - 1) \) by Lemma 1. Accordingly, the number of leaves of the search tree is at most \( \Delta \mu(2^{\text{maxQ}} - 1) \). Summarizing the above, the number of nodes of a search tree is at most \((1 + \Delta)\mu(2^{\text{maxQ}} - 1)\).

5.1.2 The overall time-complexity of CLIQUES

**Theorem 1.** Since each node can be executed in \( O(n^2) \)-time [21], the running time of CLIQUES is \( O((1 + \Delta)\mu(2^{\text{maxQ}} - 1)\text{poly}(n)) \) for some polynomial \( \text{poly}(n) \).

Consider an algorithm \( A \) that enumerates all solutions in the given problem where the number of solutions in the problem is \( M \). If algorithm \( A \) requires \( O(M \cdot \text{poly}(n)) \)-time for some polynomial \( \text{poly}(n) \) to enumerate all solutions then the complexity of algorithm \( A \) is called amortized polynomial time.

**Corollary 1.** If the size of a maximum clique \( \text{maxQ} = O(\text{lg} \, n) \) then the running time of CLIQUES is amortized polynomial.

**Proof.** We have \( \text{maxQ} \leq c \text{lg} \, n \) for some constant \( c \) from the assumption. Therefore, \((1 + \Delta)\mu(2^{\text{maxQ}} - 1) \leq (1 + \Delta)\mu(2^{c \text{lg} \, n} - 1) = (1 + \Delta)(n^c - 1) \cdot \mu. \)

Hence the result.

Note that the properties in this Section 5.1 also hold for BK and BKP.

5.2 Delay of CLIQUES

Next we consider the delay of the algorithm, that is, the maximum time which can elapse between two consecutive outputs of a solution.

Delay of BK and BKP is easy to analyze and is \( \Omega(3^{n/3}) \)-time, that is described separately later in Section 6.1.

As for CLIQUES, the situation is challenging. The maximum-degree pivot choice is in practice very effective in pruning search subtrees that lead to no solutions, and we were unable to build ad-hoc counterexamples where this choice leads to an exponentially long time between two outputs.

We, however, manage to show a weaker, but essentially similar, result: we prove that if CLIQUES has polynomial delay, then \( P = NP \).

It is worth observing that a recursive call of Algorithm 1 (ignoring the cost of its children calls) takes polynomial time, namely \( O(n^2) \), so exponential delay means the algorithm must encounter an exponentially long sequence of consecutive recursive calls that perform no output. Furthermore, as the depth of the tree is \( O(\text{maxQ}) \), a leaf is always reached in polynomial time, so exponential delay means the algorithm must encounter an exponentially long sequence of consecutive leaves that perform no output (which we call bad leaves).

5.2.1 Hardness of the extension problem

Firstly, let us define what is called the extension problem for maximal cliques, and prove that it is NP-complete.

**Problem 1** (Extension Problem, EXT-P(\( G(V, E), X \))).

Given a graph \( G(V, E) \) and \( X \subset V \), does \( G \) have a maximal clique \( Q \) that does not intersect \( X \)?
This problem is a common building block of enumeration algorithms based on binary partition, although typically challenging when dealing with maximal solutions [3], [8], [14]. Looking at a recursive call of CLIQUES, setting $X \equiv \text{FINI}$ and $V \setminus X \equiv \text{CAND}$, we can observe how it essentially corresponds to asking “will a maximal clique be output in its recursion subtree?” Answering this question would allow excellent pruning, as we could identify all useless recursive calls, resulting in no bad leaves and thus polynomial delay.

Unfortunately this problem is NP-complete, as we show by a reduction from CNFSAT.

**Theorem 2.** The extension problem for maximal cliques (EXT-P($G(V,E), X$)) is NP-complete.

**Proof.** Let $\mathcal{F}$ be a CNF boolean formula on $h$ variables $v_1, \ldots, v_h$ and $l$ clauses $c_1, \ldots, c_l$, and let the positive and negative literals of the variable $v_i$ be represented by $v_i$ and $\neg v_i$.

We build a graph $G$, with a suitable vertex set $X$, such that $\mathcal{F}$ can be satisfied if and only if $G$ has a maximal clique not intersecting $X$.

Let $V$ be as follows:

- A vertex $p_i$ for each positive literal $v_i$ in $\mathcal{F}$
- A vertex $n_i$ for each negative literal $\neg v_i$ in $\mathcal{F}$
- A vertex $c_i$ for each clause of $\mathcal{F}$

Let $E(G)$ be as follows:

- For all distinct $i$ and $j$, $E(G)$ contains $(p_i, p_j)$, $(p_i, n_j)$, $(n_i, p_j)$, and $(n_i, n_j)$, i.e., all literals are connected to all others (positive and negative) except their own negation.

- $E(G)$ contains $(c_i, p_i)$ if literal $v_i$ does not appear in $c_i$. Similarly, $(c_i, n_i) \in E(G)$ if $\neg v_i$ does not appear in $c_i$, i.e., each clause is connected to all literals except those that satisfy it.

An example is shown in Figure 3.

Now, let $X$ be the set of all vertices $c_i$ corresponding to clauses. Note that $V \setminus X$ is the set of all vertices corresponding to literals. Furthermore, observe how a clique cannot contain both $p_i$ and $n_i$, and any set $S \subseteq V \setminus X$ is a clique if and only if it does not contain both a literal and its negation: it follows that cliques in $V \setminus X$ correspond exactly to valid truth assignments to the literals.

Furthermore, observe that if $S$ does not contain any literal satisfying a clause $c_i$, then $c_i$ is adjacent to all literals in $S$ (due to how edges are placed in $E$), thus $S$ is not maximal because $c_i$ can be added to it. So if the literals in $S$ do not satisfy some clause $c_i$, then $S$ is not maximal.

Conversely, if $S$ is maximal and spans all variables (i.e., either $p_i$ or $n_i$ is in $S$ for any $i$), it means that each $c_i$ is not adjacent to some vertex in $S$: this means $c_i$ is satisfied by the corresponding literal, since clauses are adjacent to all literals that do not satisfy them. It follows that $S \subseteq V \setminus X$ is a maximal clique if and only if its corresponding literals are a satisfying assignment of $\mathcal{F}$ that assigns a truth value to all variables.

Finally, any CNF formula that has a satisfying assignment, trivially also has a satisfying assignment that assigns truth values to all variables (one can just choose an arbitrary value to each missing variable). It follows that EXT-P($G(V,E), X$) has positive answer if and only if $\mathcal{F}$ can be satisfied.

It only remains to show that EXT-P($G(V,E), X$) is NP: indeed testing a solution corresponds to verifying the maximality of a given clique, that can trivially be done in $O(n^2)$ time. The proof is complete. □

### 5.2.2 CLIQUES has polynomial delay only if $P=NP$

Now, given $G(V,E)$ and $X$, we build a graph $G'(V', E')$ such that, if CLIQUES runs on $G'(V', E')$ with polynomial delay, then we can solve the NP-complete problem (EXT-P($G(V,E), X$)) in polynomial time.

$G'(V', E')$ is built as follows. In particular, $V' = Q \cup X \cup \{x\} \cup P$, where:

- $Q = \{q_1, \ldots, q_Q\}$ is a clique of size $|Q| > |X| + |V| + 1$ (large enough so that one vertex is chosen as a pivot in the first recursive call of CLIQUES)
- $X = \{x_1, \ldots, x_{|X|}\}$ are the vertices of $X$ from

![Figure 3 Example construction of G from the F formula for a clause c1 = (v1, ¬v2, ¬v_h)](image-url)
EXT-P(G(V, E), X).

- v is a vertex connected to all of X and P (but not to Q)
- P = {p_1, ..., p_p} are the vertices of V \ X from EXT-P(G(V, E), X).

Where the vertices from G(V, E) are connected as in G.

If we run CLIQUES on G', it will first choose q_1 as a pivot without loss of generality, and q_2, ..., q_Q will become black nodes and be skipped.

The algorithm will find the maximal clique Q in O(|Q|^3)-time, then will start processing the remaining vertices, i.e., X \ {v} \ P. By processing we mean they are considered in the foreach loop of the root recursive call of Algorithm 1. As the algorithm does not specify in which order the vertices are processed, we will assume this is in the same order as they appear in Figure 4, i.e., first X, then v, then P.

Take the moment when v is processed:

We have that CAND \ \Gamma(v) is exactly P, while FINI \ \Gamma(v) is X because we already processed vertices of X.

If CLIQUES has polynomial delay, it must either find a new maximal clique or terminate, in polynomial time: as any maximal clique containing vertices of X has already been found, this process will output a new maximal clique if and only if there is a maximal clique in P that cannot be extended with vertices of X, i.e., since P corresponds to V \ X, there is a maximal clique in G(V, E) that does not intersect X.

Finally, since CLIQUES may spend exponential time before processing v, we want to skip this time: we can set up CLIQUES directly with arbitrary CAND and FINI by running

CLIQUES([v], P, X)

This way CLIQUES starts from the desired moment, and if it has polynomial delay, it will find the solution to EXT-P(G(V, E), X) in polynomial time. More formally, we can state that:

**Theorem 3.** If Algorithm CLIQUES [21] has polynomial delay, then P = NP. ☐

In addition, the following is a corollary to Theorem 1.

**Corollary 2.** If maxQ = O(lg n) then the running time of CLIQUES is polynomial delay.

**Proof.** Let p_l and p_r be any pair of consecutive outputs of maximal cliques in the search tree. (Consider Figure 2 where the internal node x is replaced by a leaf p_l and the internal node y by a leaf p_r.) Then we can show that the number of nodes between p_l and p_r in the search tree is at most 2(1 + \Delta)(2^{maxQ} - 1) as in the proof of Theorem 1.

If maxQ = O(lg n) then 2(1 + \Delta)(2^{maxQ} - 1) \leq 2(1 + \Delta)(n^c - 1).

Hence the result. ☐

**6. Delay of the other algorithms**

**6.1 Delay of Bron-Kerbosch Algorithms**

For BK and BKP the delay can be easily assessed. Consider a graph G built as follows: a clique A of constant size, a Moon-Moser graph M on k vertices [17], a vertex x adjacent to all of M, and another clique B of constant size. A graphical representation is given in Figure 5. Observe that the total number of vertices in G is n = k + O(1). Furthermore, all maximal cliques in G except A and B contain x. Consider now BKP on G: in the root recursive call, assume a vertex a of A is chosen as the first pivot (thus recursion on all other vertices of A is prevented by the pivot rule), assume that vertices are processed in the following order (again, by processing we mean they are considered in the foreach loop): First a, then x, then vertices of M, then finally vertices of B. All maximal cliques involving x are found

**Figure 5** The graph built in Section 6.1.
in the recursion subtree when processing \( x \). Once the algorithm backtracks and starts visiting vertices of \( M \), assume that \( x \) is never selected as pivot (an acceptable assumption if the pivot choice is arbitrary). Then by [21] we know BKP will take \( \Omega(3^{k/3}) = \Omega(3^{n/3}) \) time to process the Moon-Moser graph \( M \), but no new solution is found in this time, because all cliques in \( M \) can be extended with \( x \), finally the algorithm processes vertices of \( B \) and outputs \( B \), giving us a delay of \( \Omega(3^{n/3}) \)-time. The same bound holds for BK, since its recursion tree includes that of BKP. We obtain the following:

**Theorem 4.** The BK and BKP algorithms have delay \( \Omega(3^{n/3}) \)-time. □

### 6.2 Delay of Eppstein et al. algorithm

It is worth observing how Eppstein et al. [10] developed a variation of CLIQUES tailored for sparse graphs, running in \( O(n3^{d/3}) \) time, where \( d \) is the degeneracy of the input graph. The degeneracy \( d \) is typically small on real-world sparse graphs. This parameter is used to define a specific ordering of the vertices which creates smaller subgraphs to process (provided that \( d \) is small), using CLIQUES.

In a worst-case scenario, however, \( d \) can be \( \Omega(n) \), and Conte et al. [9] proves, by building suitable ad-hoc graphs, how the delay of Eppstein et al. algorithm is \( \Omega(3^{n/6}) \).

### Concluding remarks

In the midst of our present work, we noticed an article [16] claiming that “The Bron-Kerbosch algorithm with vertex ordering is output-sensitive.” But Conte and Versari pointed out a bug in its main lemma (Lemma 11 of [16], confirmed by the author in private communications). Therefore the question of the delay is now settled, but that of the amortized cost per solution remains open.

### Acknowledgments

The authors would like to thank L. Versari and E. Harley who joined in useful discussions. This work is supported in part by JSPS KAKENHI Grant Number 17K00006.

### References

http://id.nii.ac.jp/1438/00001899/
http://id.nii.ac.jp/1438/00001898/
https://doi.org/10.1007/978-3-642-27848-8_725-2
http://id.nii.ac.jp/1438/00008519/