Sublinear-Space Bounded-Delay Enumeration for Massive Network Analytics: Maximal Cliques*

Alessio Conte[1], Roberto Grossi[2], Andrea Marino[3], and Luca Versari[4]

1 **Dipartimento di Informatica, Università di Pisa, Pisa, Italy**
`conte@di.unipi.it`
2 **Dipartimento di Informatica, Università di Pisa, Pisa, Italy**
`grossi@di.unipi.it`
3 **Dipartimento di Informatica, Università di Pisa, Pisa, Italy**
`marino@di.unipi.it`
4 **Scuola Normale Superiore, Pisa, Italy**
`luca.versari@sns.it`

Abstract

Due to the sheer size of real-world networks, delay and space become quite relevant measures for the cost of enumeration in network analytics. This paper presents efficient algorithms for listing maximum cliques in networks, providing the first sublinear-space bounds with guaranteed delay per enumerated clique, thus comparing favorably with the known literature.

1998 ACM Subject Classification E.1 Data Structures – Graphs and networks, G.2.2 Graph Theory – Graph algorithms

Keywords and phrases Enumeration algorithms, maximal cliques, network mining and analytics, reverse search, space efficiency

Digital Object Identifier 10.4230/LIPIcs.ICALP.2016.148

1 Introduction

The design of efficient algorithms for enumerating all possible solutions of a given problem dates back to the 1950s [5, 19, 35]. Enumeration algorithms have the purpose of either counting the number of solutions or listing the solutions one by one. Their study originated in the area of complexity and optimization [17, 23, 39], and then spread over several other application domains, including bioinformatics, machine learning, network analytics, and social analysis [1, 26, 33]. For instance, a number of papers described how to enumerate triangles [6, 3, 22, 31] and their generalizations such as cliques or other dense subgraphs [7, 9, 11, 12, 14, 15, 21, 25, 29, 34, 38]. Among the first problems attacked is the enumeration of maximal cliques [2, 5, 7, 18, 24, 28], where a maximal clique is a subset of pairwise connected vertices that is maximal under inclusion.

This paper focuses on two worst-case efficiency measures, namely, delay and space, that become relevant for enumeration in massive networks. The delay is the maximum latency between any two consecutively reported solutions. The space is the maximum amount of extra memory that should be allocated to enumerate all the solutions, besides the amount required by the input graph.

* Work partially supported by the Italian Ministry of Education, University, and Research (MIUR) under PRIN 2012C4E3KT research project AMANDA – Algorithmics for MAssive and Networked DAta.

43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016).
Editors: Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi;
Article No. 148; pp. 148:1–148:15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Motivation. Let $G(V, E)$ be an undirected connected graph, represented with ordered adjacency lists, where $n = |V|$ is the number of vertices and $m = |E|$ is the number of edges. Although it is known that the number α of maximal cliques in real-world networks is much smaller than the exponentially many possible subsets of vertices [32], it happens that α is still large for massive networks. In this scenario the notion of delay provides a guarantee on the maximum time that a postprocessing algorithm has to wait before the next enumerated maximal clique is produced: letting $t(n, m)$ be the delay, we observe that not only the total time is output-sensitive, i.e. $O(\alpha\, t(n, m))$ plus the setup cost, but we also guarantee that listing the next solution takes $O(t(n, m))$ time in the worst case. This is a stronger notion than average throughput (e.g., number of cliques per second), which is obtained by dividing α by the total time. The former implies the latter, but not vice versa. We observe that the delay has been already used in many papers, e.g. [9, 11, 12, 23, 25, 36], as a worst-case measure.

Space is another measure that has been considered when enumerating solutions, e.g. [13, 10, 41]. Modern CPUs have multiple cores, where each core has a very fast – but small – private cache, with a shared last-level cache and a shared slow random access memory. Consider multiple enumeration threads running simultaneously on the same massive graph in the shared memory: modern machines have very large shared memory compared to the private cores, so massive graphs can be stored in shared memory but not in the private cores. If the memory footprint of each thread is small and fits the private cache of a core, the shared memory access bottleneck is only caused by accessing the graph itself, and not the private data. To make a concrete example, the private cache for CPUs in today's commodity desktops is few hundreds of kilobytes while the random access memory can host several terabytes and networks contain millions of vertices and edges. For example, network EU-2005 (Section 7) contains $n = 850$ thousand vertices, $m = 16.1$ million edges, and $\alpha = 5.7$ million cliques: here, the space of known algorithms ranges from 3 to 164 megabytes, so their working set does not fit the private cache (Table 2).

We remark that delay and space are somehow related measures. An algorithm with good overall time can accumulate solutions in the shared memory or store them temporarily in a file for a subsequent phase of postprocessing. If space is limited (especially in fast memory), this approach cannot be taken into account.

Furthermore, as most current output-sensitive approaches have a recursive nature, they can easily reach $\Theta(n)$ nesting levels in the worst case even for sparse graphs, thus using the stack should be avoided: indeed, just storing one memory word per recursion level kills sublinearity. This motivates the search for enumeration algorithms that provably use sublinear space and have good delay bounds. Some space-efficient algorithms [13, 10, 41] work well in practice for real-world networks but cannot guarantee sublinear space.

We remark that this paper does not address cache-efficient or cache-oblivious algorithms in the parallel or distributed setting, which is worth investigating in future work. What is emphasized here is that small footprint enumeration algorithms have more chances to reduce memory contention and memory bandwidth issues when run on modern processors.

Our results. We provide the first algorithms with sublinear space and bounded delay for enumerating the maximal cliques when the vertices of G are provided in degeneracy order. This means that there exists an integer d, as small as possible, such that each vertex has d or fewer neighbors appearing later in the ordering: d is called the degeneracy of G, a well-known sparsity measure [13, 15, 40, 41], and is equivalently defined as the smallest integer such that every nonempty subgraph of G has at least one vertex of degree $\leq d$. The degeneracy

Table 1 Bounds for maximal clique enumeration, where $q-1 \leq d \leq \Delta \leq n-1 \leq m$, $d = O(\sqrt{m})$. †: it does not list cliques, but outputs a compressed representation. $^{+}$: h is the smallest integer such that $|\{v \in V : |N(v)| \geq h\}| \leq h$, where $d \leq h \leq \Delta$. *: it uses matrix multiplication. A lower bound in the column for the delay means that there exists a family of graphs with that delay.

Algorithm	Time			Space
	Setup	Delay	Overall	
Bron-Kerbosch [7]	$O(m)$	unbounded	unbounded	$O(n+q\Delta)$
Tomita et al. [34]†	$O(m)$	$\Omega(n^3)$	$O(3^{n/3})$	$O(n+q\Delta)$
Eppstein et al. [16]	$O(m)$	$\Omega(3^{n/6})$	$O(d(n-d)3^{d/3})$	$O(n+d\Delta)$
Johnson et al. [23]	$O(mn)$	$O(mn)$	$\alpha\, O(mn)$	$O(\alpha n)$
Tsukiyama et al. [36]	$O(n^2)$	$O((n^2-m)n)$	$\alpha\, O((n^2-m)n)$	$O(n^2)$
Chiba-Nishizeki [11]	$O(m)$	$O(md)$	$\alpha\, O(md)$	$O(m)$
Makino-Uno [25]	$O(mn)$	$O(\Delta^4)$	$\alpha\, O(\Delta^4)$	$O(m)$
Chang et al. [9]$^{+}$	$O(m)$	$O(\Delta h^3)$	$\alpha\, O(\Delta h^3)$	$O(m)$
Makino-Uno [25]*	$O(n^2)$	$O(n^{2.37})$	$\alpha\, O(n^{2.37})$	$O(n^2)$
Comin-Rizzi [12]*	$O(n^{5.37})$	$O(n^{2.09})$	$\alpha\, O(n^{2.09})$	$O(n^{4.27})$
This paper	$\tilde{O}(m)$	$\tilde{O}(qd(\Delta+qd))$	$\alpha\, \tilde{O}(qd(\Delta+qd))$	$O(q)$
This paper	$\tilde{O}(m)$	$\tilde{O}(\min\{md, qd\Delta\})$	$\alpha\, \tilde{O}(\min\{md, qd\Delta\})$	$O(d)$

n = #vertices $\quad$ Δ = max degree $\quad$ α = #maximal cliques
m = #edges $\quad$ d = degeneracy $\quad$ q = largest clique size

ordering can be computed in $O(n+m)$ time by repeatedly removing the vertex of minimum degree from G. Also, it can be proved that $d = O(\sqrt{m})$, and that the size q of the maximum clique in G and the maximum degree in the graph Δ satisfy $q-1 \leq d \leq \Delta \leq n-1$.

The last two rows in Table 1 report our bounds in terms of the above parameters, where the $\tilde{O}()$ notation ignores $\log^{O(1)}(n+m)$ factors. We observe that our bounds are expressed in terms of the parameters q, d, Δ instead of m, n, whenever possible, as they are actually smaller than m or n. For example, network EU-2005 has $q = 387$, $d = 388$ and $\Delta = 68\,963$. Also, since both q and d are always $O(\sqrt{m})$, our space is provably sublinear (around 20 kilobytes for EU-2005!). Note that Δ is not always sublinear in the graph size as real-world networks are sparse and could have $\Delta = \Theta(n)$, as shown in Table 2 (see also [13]). Also, our $O(q)$ space is close to optimal as we have to single out a subset of q vertices from G.

The other rows in Table 1 report the bounds for the main results in the state of the art (see below for a discussion). The setup time is the preprocessing cost before starting to list the solutions, and ours is comparable to that of previous results. As for the delay, the cost in the last row is asymptotically smaller in many cases, except for dense graphs, where matrix multiplication based algorithms [12, 25] are preferable (but massive networks can hardly be processed by quadratic space algorithms). As for the overall time, we have a similar improvement for output-sensitive bounds where the α term appears. Moreover, we observe that [16] has great time performance in practice (see Section 7) but it is not output-sensitive as α does not appear in the complexity, and cannot guarantee sublinear space. Summing up, our algorithms can compete with the state of the art when suitably implemented, with the additional bonus of guaranteeing small space and bounded delay.

Related work. The idea of using small space in enumeration algorithms is not new, as witnessed by the notion of "compactness" introduced in Fukuda [17]: however, his goal is not sublinear space as in our paper, but polynomially bounded space in terms of the input size and the maximum output size of a solution. It is also worth mentioning that the class of CAT (constant amortized time) enumeration algorithms described in Ruskey's book [30] seems to be very promising but our algorithms cannot fall within this class as their amortized cost per solution is non-constant.

Our algorithms are based on the reverse search paradigm introduced by Avis and Fukuda [4] as it has been conceived to be space-efficient by its authors. Also, we reuse some of the machinery introduced by Tsukiyama et al. [36] and Makino and Uno [25]. In the reverse search the solutions are the nodes of a virtual digraph, for which a "successor" function is defined to jump from one solution to the other. (Gély et al. [20] study new combinatorial properties of this virtual digraph.) A spanning tree represents all the solutions, where the depth corresponds to the number of nested levels of the corresponding recursion. To achieve sublinear space, our algorithms employ the stateless reverse search to avoid using the stack, plus other properties that exploit the structure of the maximal clique enumeration problem.

Turning to the state of the art for the maximal clique enumeration problem, Table 1 summarizes the main results. The papers by Bron and Kerbosch [7] and Tsukiyama et al. [36] have defined the main lines of research for algorithms using polynomial space, and they are currently at the heart of many other algorithms. The Bron-Kerbosch algorithm relies on a backtracking scheme that is adopted in several efficient algorithms [16, 34, 41]. The algorithm by Tsukiyama et al. has been originally conceived for the enumeration of maximal independent sets, which is a problem equivalent to that of maximal cliques, and inspired at least in part the algorithm by Johnson et al. [23], which produces solutions in lexicographic order but requires exponential space (see also [20]). The approach has been subsequently adapted to clique enumeration by Chiba-Nishizeki [11], and Makino-Uno [25] has reinterpreted [36] in the paradigm of reverse search.

Bron-Kerbosch scheme. The Bron-Kerbosch based algorithms are a popular choice for enumerating cliques due to their simplicity and good performance in practice. The original version [7] does not provide any guarantee. The version in [34] guarantees a total running time of $O(3^{n/3})$, which is optimal for Moon-Moser graphs as they have $3^{n/3}$ cliques [8, 27]. The version in [16] further refines and improves the work for sparse graphs, which may have up to $(n-d)3^{d/3}$ cliques, by producing an algorithm with $O(d(n-d)3^{d/3})$ time. All of these approaches use $O(n+q\Delta)$ space to store sets of candidates and visited vertices. It is possible to modify these algorithms to decrease their space usage (e.g. by modifying the data structure in [16]) but, to the best of our knowledge, with state of the art techniques they would still require $\Omega(\Delta)$, which is not sublinear as Δ can be $\Theta(n)$ in sparse graphs. Algorithms that follow this scheme are characterized by their complexity being related to the worst-case number of cliques in a graph (instead of α), and give no guarantees on the cost per solution nor the delay: these can be shown to be both $\Omega(n^3)$ for [34], while [16] can have a delay of $\Omega(3^{n/6})$ for some families of graphs.

Tsukiyama et al. scheme. The motivation behind algorithms in this class is to achieve an output-sensitive cost that is proportional to the number α of maximal cliques times a function that depends on the graph parameters. The original algorithm by Tsukiyama et al. [36] is a backtracking procedure that enumerates maximal independent sets in $O(mn)$ time per solution. As a maximal independent set is a maximal clique in the complementary

graph, this gives an algorithm for enumerating maximal cliques in $O((n^2 - m)n)$ time per solution. The adaptation of the algorithm to maximal cliques has been improved by Chiba and Nishizeki [11], who bring the delay down to $O(md)$:[1] this algorithm is still based on a stateful backtracking procedure in which the recursion tree has depth $\Omega(n)$, making the required space $\Omega(n)$. Makino and Uno [25] take a step towards statelessness by adapting [36] and [23] to the paradigm of reverse search. The algorithm is still a stateful recursive approach, which takes $\Omega(n)$ space (the depth of the recursion tree). Differently from its predecessors, each node of the tree corresponds to a unique maximal clique whose children can be computed as a function of the graph and the clique itself. The Makino-Uno algorithm is provided in two versions, one combinatorial with delay $O(\Delta^4)$ and one based on matrix multiplication with delay $O(n^{2.37})$. The former has been improved to $O(\Delta h^3)$ by Chang et al. [9], where h is the smallest integer such that $|\{v \in V : |N(v)| \geq h\}| \leq h$: since the reverse search is not stateless, here the space remains $\Omega(n)$. The latter has been improved to $O(n^{2.09})$ by Comin and Rizzi [12] using matrix multiplication but requiring higher space usage and setup time.

2 Preliminaries

Let $G(V, E)$ be an undirected graph with $|V| = n$ and $|E| = m$. We assume that G is connected and represented using ordered adjacency lists. A clique is a subset $K \subseteq V$ of pairwise connected vertices: we will use K to denote both this subset of vertices and the subgraph induced by them. Given G, let Δ be the maximum vertex degree, d the degeneracy, and q be the maximum clique size, where $q - 1 \leq d \leq \Delta \leq n - 1 \leq m$ and $d = O(\sqrt{m})$; also, let $v_1 \dots v_n$ be the vertices of V labelled in a degeneracy ordering (see the introduction). We denote by $V_{\leq i}$ the set of vertices $v_1 \dots v_i$. Let us define $N(v)$ as the set of neighbors of v and $N_>(v)$ as $\{x \in N(v) : x > v\}$. We define $N_<(v)$ analogously. Note that $|N_>(v)| \leq d$ as the graph is labelled in degeneracy ordering. Given a set of vertices $A \subset V$, we define $N(A) = \bigcap_{v \in A} N(v)$ as the set of neighbors common to all the vertices in A. We call *heads* the vertices v such that $N_<(v) = \emptyset$. Moreover, for any $v \in V$, $A_{\leq v} = A \cap V_{\leq v}$ and $A_{<v} = A_{\leq v} \setminus \{v\}$. Given two set of vertices A and B, we say that $A < B$ if A is lexicographically smaller than B. Given a set $X \subseteq V$, we define $\textsc{complete}(X)$ as the lexicographically smallest maximal clique that contains X.

▶ **Lemma 1.** *If $X_1 \subseteq X_2$, then $\textsc{complete}(X_1) \leq \textsc{complete}(X_2)$.*

Proof. Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be the set of maximal cliques containing respectively X_1 and X_2. Clearly $\mathcal{C}_1 \supseteq \mathcal{C}_2$, and as $\textsc{complete}(X)$ returns the lexicographically smallest maximal clique that contains X we have $\textsc{complete}(X_1) = \min(\mathcal{C}_1) \leq \min(\mathcal{C}_2) = \textsc{complete}(X_2)$. ◀

3 Reverse Search Revisited

We sketch a reverse search algorithm for the maximal clique enumeration that revises and simplifies the algorithm by Makino and Uno [25], which is itself a reinterpretation of the works of Tsukiyama et al. in [36] and Johnson et al. in [23] for maximal independent sets.

The rationale of the algorithm can be summarized as follows. Suppose we iteratively add the vertices of G to a graph G', which is initially empty. Each time a vertex v is added, the new maximal cliques can be computed by looking at the maximal cliques of G': an existing clique $K \subseteq G'$ can be extended partially or totally by v. Of course computing all

[1] The work actually exploits the arboricity, but we use d for simplicity as the arboricity is $\Theta(d)$.

Algorithm 1: Enumerate all maximal cliques (a.k.a. revisited Makino-Uno)

Input : Graph $G(V, E)$ where vertices are labeled in degeneracy ordering
Output : Maximal cliques in G

Let $v_1, v_2, \dots v_n \in V$ be the vertices labeled in degeneracy ordering, such that $v_1, \dots, v_j$ are heads, where j is the number of heads ($1 \leq j \leq n$).

```
for i ∈ {1, ..., j} do
    R_i ← COMPLETE({v_i})
    SPAWN(R_i)
Function COMPLETE(K)
    choose any v ∈ K
    foreach increasing w ∈ N(v) do
        if K ⊆ N(w) then
            K ← K ∪ {w}
    return K
Function SPAWN(K)
    CAND ← { w ∈ ⋃_{u∈K} N_>(u) \ K : w > PI(K) }
    foreach v ∈ CAND do
        K'_v ← K_<v ∩ N(v)
        D ← COMPLETE(K'_v ∪ {v})
        if COMPLETE(K'_v) = K and D_<v = K'_v then
            SPAWN(D)
```

the cliques of G in this way requires keeping most of the cliques of the graph in memory (as for independent sets in [23]), which takes exponential space. In order to avoid to store the cliques, we address the following question: given the clique K, maximal for G', which vertices in $G \setminus G'$ expand K and which cliques do they produce in G? This "production" relationship gives us the "successor" function of the reverse search paradigm. Technicalities are needed to avoid expanding K with candidates leading to the same clique more than once.

It is worth observing that, differently from [25], our algorithm assumes that the vertices of G are given in degeneracy ordering and begins the recursion from a set of root cliques (instead of the lexicographically minimum one), where each root R_i corresponds to COMPLETE run on the head v_i. Moreover, we alter the given degeneracy ordering in such a way that the heads are at the beginning. This is always possible: as heads have no backward edges, when moved backwards they will not change the number of forward edges of any vertex (see Section 6).

The pseudocode is shown in Algorithm 1. The function SPAWN makes use of the notion of parent index, borrowed from [25]:

▶ **Definition 2** (Parent Index). Given a maximal clique K, $\text{PI}(K)$ is the smallest x such that $\text{COMPLETE}(K_{\leq x}) = K$.

The CAND set contains the vertices that partially extend K and that are greater than $\text{PI}(K)$. For each vertex in CAND we try to generate a child clique; the check in SPAWN corresponds to conditions (c) and (d) from Lemma 2 in [25], which guarantees that each child clique is only generated once. We remark that each call of SPAWN returns at least a clique.

Space usage and delay. Referring to Algorithm 1, we observe that the COMPLETE function only needs to store the set K, whose size is $O(q)$, and performs $O(\Delta)$ iterations that take $\tilde{O}(q)$ time each. The cost of an iteration on vertex w is also bounded by $\tilde{O}(|N(w)|)$, so the cost of all the iterations is bounded by $\tilde{O}(m)$. Since PI can be computed by running COMPLETE q times, we can conclude the following:

▶ **Lemma 3.** *Function* COMPLETE *in Algorithm 1 takes $O(q)$ space and $\tilde{O}(\min\{q\Delta, m\})$ time. Moreover, computing* PI *takes $O(q)$ space and $\tilde{O}(q \min\{q\Delta, m\})$ time.*

For the sake of clarity, we will refer to the *vertices* in G and to the *nodes* in the recursive tree induced by our approach. The space requirement of each recursive node is bounded by

the size of CAND, as all other lines take $O(q)$ space. As CAND $\subseteq V$ is the union of up to q sets of vertices each of size at most d, we have $|\text{CAND}| \leq \min\{qd, n\}$. The recursion depth of Algorithm 1 is $O(n)$, for a total space requirement of $O(n \min\{qd, n\})$.

Consider now the delay of Algorithm 1. Note that selecting the next head takes constant time as heads are contiguous. By making use of alternative output [37], as we have no dead ends, the delay of the algorithm is bounded by the cost of $O(1)$ recursive nodes. The function SPAWN performs $|\text{CAND}|$ iterations, whose cost is dominated by COMPLETE. The total cost of the loop is thus $\tilde{O}(\min\{qd, n\} \min\{q\Delta, m\})$, which corresponds to the total cost of the recursion node as it dominates the cost of computing CAND. As a consequence we have some new interesting bounds shown in Lemma 4, which we improve in the rest of the paper.

▶ **Lemma 4.** *Algorithm 1 uses $O(n \min\{qd, n\})$ space and has $\tilde{O}(\min\{qd, n\} \min\{q\Delta, m\})$ delay.*

4 Improved Algorithm

Algorithm 1 does not meet our space requirements; still, it is the starting point to build our space efficient scheme. The first issue is its recursive nature: in function SPAWN, each time a recursive call is performed the *status* of the current call needs to be saved. As a result, we require the space needed by each recursive node multiplied by the height of the recursion tree. Note that the standard stack-based transformation of recursive programs into iterative ones, as done in [9], does not solve the issue, as the stack size would be $\Omega(n)$. We will therefore navigate implicitly the recursion tree induced by the reverse search, without using a stack. The other important issue to achieve sublinear space is the CAND set, whose size can be $\min\{qd, n\}$. Hence, we need to traverse CAND without materializing it.

To address the first issue, we represent the state of the computation with the pair:

- the current clique K
- the "bookmark" vertex v in CAND for K (where initially $v = \text{PI}(K)$).

We remark that the bookmark vertex allows us to resume the computation in the parent without the need of storing information for each recursive level.

▶ **Fact 1.** *The state $\langle K, v\rangle$ requires $O(q)$ space.*

Furthermore, we implicitly traverse the recursion tree using these navigation primitives:

- IS-ROOT(K) checks if K is the root of current recursion tree.
- PARENT-STATE(K) returns the state for the parent node of K in the recursion tree.
- GET-NEXT-CAND(K, v) finds the candidate following v in CAND for the current K.
- CHILD-EXISTS(K, v) checks whether the current state will lead to a child maximal clique.

Algorithm 2 shows how to implement and use the above primitives, and Lemmas 5, 6, 7, 10 prove their correctness.

▶ **Lemma 5.** *Let K be any maximal clique examined in Algorithm 1: K is a root iff $\text{PI}(K) = \min(K)$. As a corollary, roots have no parent.*

Proof. Let v_i be the head of the recursion tree containing K and $R_i = \text{COMPLETE}(v_i)$ the relative root. By definition we have that v_i is a head iff $N_<(v_i) = \emptyset$, so a head must be the smallest vertex of any clique. We then have $v_i = \min(R_i)$, and since $\min(D) \in K$ for any child D of K, there cannot be heads other than v_i in the current recursion tree. From this it immediately follows that no other roots are found in the subtree of the root R_i, and hence roots have no parent.

Algorithm 2: Improved enumeration of maximal cliques

Assume $\min(\emptyset) = \texttt{null}$, and adopt the following shorthands for maximal clique K: sub-clique $K'_v \equiv K_{<v} \cap N(v)$ and vertex set $B_K \equiv \{u \in V \setminus K : K_{<u} \subseteq N(u)\}$.

```
Function IMPROVED-SPAWN(K)
    v ← PI(K)
    while true do
        childless ← true
        while v ← GET-NEXT-CAND(K, v) ≠null do
            if CHILD-EXISTS(K, v) then
                K ← COMPLETE(K'_v ∪ {v})
                childless ← false
                break
        if childless then
            if IS-ROOT(K) then return
            else
                ⟨K, v⟩ ← PARENT-STATE(K)

Function IS-ROOT(K)
    return PI(K) = min(K)
Function CHILD-EXISTS(K, v)
    return N(K'_v) ∩ (B_K ∪ N_<(v)) = ∅
Function GET-NEXT-CAND(K,v)
    return min{w ∈ ⋃_{u∈K} N_>(u) \ K :
                w > v}
Function PARENT-STATE(K)
    v ← PI(K)
    return ⟨COMPLETE(K_<v), v⟩
```

We now show that $N_<(\min(K)) = \emptyset$ and $K = \text{COMPLETE}(\{\min(K)\})$ iff $\text{PI}(K) = \min(K)$. This concludes the proof as the first condition correspond to the definition of a root ($\min(K)$ being the corresponding head). Since $\{\min(K)\} = K_{\leq \min(K)}$, we have $\text{PI}(K) = \min(K)$ iff $K = \text{COMPLETE}(\{\min(K)\})$ by definition of PI. Moreover, $K = \text{COMPLETE}(\{\min(K)\})$ implies $N_<(\min(K)) = \emptyset$: indeed, if there was $w \in N_<(\min(K))$, COMPLETE would add w to $\{\min(K)\}$, making it different from K. ◀

The next lemma states that given a clique D, the parent index allows us to identify the clique that generated D and the vertex in CAND used to produce D.

▶ **Lemma 6.** *Let K, D, and v be defined as in* SPAWN *when the recursive call is performed on D. Then* $\text{PARENT-STATE}(D) = \langle K, v\rangle$ *in* IMPROVED-SPAWN.

Proof. We prove that, given a maximal clique D which is not a root, the parent of D in the computational tree is $\text{COMPLETE}(D_{<\text{PI}(D)})$ and the vertex used by the algorithm SPAWN to produce D is $\text{PI}(D)$. Since K is the parent of D, we have $D = \text{COMPLETE}(K'_v \cup \{v\})$. From the conditions tested on K'_v and D, we have $\text{COMPLETE}(D_{<v}) = \text{COMPLETE}(K'_v) = K$. Now we only need to prove that $v = \text{PI}(D)$. We have that $D = \text{COMPLETE}(K'_v \cup \{v\}) = \text{COMPLETE}(D_{\leq v}) > \text{COMPLETE}(D_{<v}) = K$, applying Lemma 1. From this it follows that for any $w < v$, $\text{COMPLETE}(D_{\leq w}) \leq \text{COMPLETE}(D_{<v}) = K < D$, thus $v = \text{PI}(D)$. ◀

Since the candidates are examined in increasing order, when returning from the current child of K, we are able to provide all the remaining children of K. Indeed let v be $\text{PI}(D)$: as a consequence of Lemma 6 we know that D was generated from K using candidate v and that $K = \text{COMPLETE}(D_{<v})$. By using the next lemma, we can provide and test all the remaining candidates, i.e. the ones greater than v.

▶ **Lemma 7.** *For a given maximal clique K, let* $\text{CAND} = \{w \in \bigcup_{x\in K} N_>(x) \setminus K : w > \text{PI}(K)\}$ *in* SPAWN *and let $z_1, \ldots, z_r$ be the sequence of vertices generated by* GET-NEXT-CAND *in* IMPROVED-SPAWN. *Then* $\text{CAND} = \{z_1, \ldots, z_r\}$.

Proof. We give the proof by induction. The first time GET-NEXT-CAND is invoked v is $\text{PI}(K)$, meaning that z_1 is the minimum element of CAND. Let z_j be the last vertex generated by

GET-NEXT-CAND and let us assume that $z_1, \ldots, z_j$ are the first j vertices of CAND. Then $\min\{w \in \bigcup_{u \in K} N_>(u) \setminus K \ : \ w > z_j\}$ is the the $(j+1)$-th element of CAND if $|\text{CAND}| > j$, `null` otherwise. ◀

The check done in CHILD-EXISTS(K, v) is equivalent to the check done in SPAWN. We will prove this in Lemma 10 using Lemmas 8 and 9.

▶ **Lemma 8.** *$N(K'_v) \cap B_K = \emptyset$ is equivalent to* COMPLETE$(K'_v) = K$*, where* $B_K = \{u \in V \setminus K : K_{<u} \subseteq N(u)\}$.

Proof. Let us first prove that if $\exists w \in N(K'_v) \cap B_K$ then COMPLETE$(K'_v) \neq K$. Note that w is adjacent to all the vertices in $K_{<w}$ and in K'_v. COMPLETE(K'_v) will iteratively add to K'_v the smallest vertex z that is a neighbor of all the vertices in K'_v, so clearly $z \leq w$. If $z \notin K$ we have COMPLETE$(K'_v) \neq K$; if $z \in K$ then $z \in K_{<w}$, thus $z \in N(w)$ and w is still a candidate for the COMPLETE procedure. As w remains a candidate when $z \in K$, the process will eventually add either w or another $z \notin K$. Hence, COMPLETE$(K'_v) \neq K$.

We now prove that if COMPLETE$(K'_v) \neq K$ then $N(K'_v) \cap B_K \neq \emptyset$. Let z be the first vertex not in K selected by COMPLETE(K'_v). Since all vertices in $K_{<z}$ were added to K'_v before z, we have $K_{<z} \subseteq N(z)$ and hence $z \in B_K$. Moreover, since z has been selected by the COMPLETE procedure, $K'_v \subseteq N(z)$, which implies $z \in N(K'_v)$. It follows that $N(K'_v) \cap B \supseteq \{z\} \neq \emptyset$. ◀

▶ **Lemma 9.** *$N(K'_v) \cap N_<(v) = \emptyset$ is equivalent to* COMPLETE$(K'_v \cup \{v\})_{<v} = K'_v$.

Proof. We prove that if $N(K'_v) \cap N_<(v) \neq \emptyset$ then COMPLETE$(K'_v \cup \{v\})_{<v} \neq K'_v$. Let z be the smallest vertex in $N(K'_v) \cap N_<(v)$. Note that $z \notin K'_v$, since $z \in N(K'_v)$. The first iteration of COMPLETE$(K'_v \cup \{v\})$ selects the smallest vertex in $N(K'_v \cup \{v\})$, which is z, since $N(K'_v \cup \{v\}) = N(K'_v) \cap N(v)$. Since $z < v$, we have $z \in$ COMPLETE$(K'_v \cup \{v\})_{<v}$, which implies COMPLETE$(K'_v \cup \{v\})_{<v} \neq K'_v$.

We now prove that if COMPLETE$(K'_v \cup \{v\})_{<v} \neq K'_v$ then $N(K'_v) \cap N_<(v) \neq \emptyset$. Note that $K'_v \subseteq$ COMPLETE$(K'_v \cup \{v\})_{<v}$ since $K'_v \subseteq V_{<v}$. Let $z = \min(\text{COMPLETE}(K'_v \cup \{v\})_{<v} \setminus K'_v)$. Since z has been selected by function COMPLETE, we have $z \in N(K'_v \cup \{v\})$, that is $z \in N(K'_v) \cap N(v)$. Since $z < v$, we have $N(K'_v) \cap N_<(v) \supseteq \{z\} \neq \emptyset$. ◀

▶ **Lemma 10.** *$N(K'_v) \cap (B_K \cup N_<(v)) = \emptyset$ iff* COMPLETE$(K'_v) = K$ *and* COMPLETE$(K'_v \cup \{v\})_{<v} = K'_v$

Using Lemmas 5, 6, 7 and 10, we finally obtain the following result:

▶ **Lemma 11.** *Function* SPAWN *in Algorithm 1 and function* IMPROVED-SPAWN *in Algorithm 2 are equivalent.*

Proof. Setting $D =$ COMPLETE$(K'_v \cup \{v\})$ we observe that IMPROVED-SPAWN(K) simulates the preorder traversal of the recursion tree induced by SPAWN(K). When all the children of the current clique K have been explored during the traversal, *childless* is set to true. In this case, if K is the root of the current tree there are no more maximal cliques to be generated with the given head v_i; otherwise, the state of the parent is restored. ◀

5 Analysis

Analogously to Section 3, the delay of Algorithm 2 is the sum of the running times of all the iterations corresponding to the same K. Specifically, the number of iterations of the while loop is $|\text{CAND}| \leq \min\{qd, n\}$ (see Lemma 7). The space usage of IMPROVED-SPAWN, thanks

to its statelessness, corresponds to that of a single iteration. In order to give space and time bounds, let us analyze the costs of our navigation primitives.

Since computing $\text{PI}(K)$ dominates the space and time costs of functions IS-ROOT and PARENT-STATE, we use the bounds in Lemma 3. The next candidate can be obtained from K and the bookmark v as follows: for each element $x \in K$, perform a binary search in $N(x)$ to obtain the smallest vertex greater than v; the minimum of these vertices is the next candidate. Thus we obtain the bounds below:

▶ **Lemma 12.** IS-ROOT(K) *and* PARENT-STATE(K) *take* $\tilde{O}(q \min\{q\Delta, m\})$ *time and* $O(q)$ *space.* GET-NEXT-CAND *uses* $\tilde{O}(q)$ *time and constant space.*

Next we give a careful analysis of CHILD-EXISTS, the dominating cost of IMPROVED-SPAWN.

Cost of testing if a child exists. Function CHILD-EXISTS makes use of the sets B_K and $N(K'_v)$. Due to our memory constraints, we cannot store $N(K'_v)$ explicitly, since it would take $\Omega(\Delta)$ space. For this reason, we need to iterate over $N(K'_v)$ without materializing it.

Analogously, we cannot store $B_K = \{v \in V \setminus K : K_{<v} \subseteq N(v)\}$, whose size can be $\Omega(n)$. The following lemma will allow us to overcome this issue:

▶ **Lemma 13.** *The set* B_K *is equal to* $V_{<\min(K)} \cup \{u \in N_>(\min(K)) \setminus K : K_{<u} \subseteq N(u)\}$.

Proof. vertices in $V_{<\min(K)}$ are in B_K as $K_{<\min(K)} = \emptyset$, thus they can be stored implicitly. All other vertices must clearly be forward neighbors of $\min(K)$. ◀

We denote the non-trivial part of B_K, i.e. $\{u \in N_>(\min(K)) \setminus K : K_{<u} \subseteq N(u)\}$, as B'_K. We can compute it by testing $|N_>(\min(K))| \leq d$ candidates in $O(|K|) = O(q)$ time.

▶ **Lemma 14.** *Computing (or iterating over)* B'_K *can be done in* $\tilde{O}(qd)$ *time.*

We will now analyze the cost of iterating over $N(K'_v)$, and two possible ways of managing B_K (storing it or iterating it implicitly) leading to two different bounds.

Iterating over $N(K'_v)$. Iterating over $N(K'_v)$ using constant space can be done as follows. Let w be the vertex of K'_v with lowest degree: iterate over the vertices $z \in N(w)$ and for each check whether $K'_v \subseteq N(z)$. This costs $\tilde{O}(|N(w)||K'_v|) = \tilde{O}(\sum_{y \in K'_v} |N(y)|)$, since w is the lowest degree element of K'_v.

▶ **Lemma 15.** *The total cost of iterating over all of the sets* $N(K'_v)$ *for* v *in* $\bigcup_{x \in K} N_>(x)$ *is* $\tilde{O}(d \min\{q\Delta, m\})$ *time.*

Proof. Let $C_K = \bigcup_{x \in K} N_>(x)$. Since the cost of iterating over a given $N(K'_v)$ is bounded by $\tilde{O}(\sum_{y \in K'_v} |N(y)|)$, the following steps prove the lemma:

$$\begin{aligned}
\sum_{v \in C_K} \sum_{y \in K \cap N_<(v)} |N(y)| &= \sum_{v \in C_K} \sum_{y \in K} |N(y)| I_{\{y \in N_<(v)\}} \\
&= \sum_{y \in K} \sum_{v \in C_K} |N(y)| I_{\{v \in N_>(y)\}} \\
&= \sum_{y \in K} \sum_{v \in N_>(y)} |N(y)| \\
&= \sum_{y \in K} |N_>(y)| \cdot |N(y)| \\
&\leq d \sum_{y \in K} |N(y)| \leq d \min\{q\Delta, m\}
\end{aligned}$$

◀

Algorithm 3: In-place algorithm for moving heads to the beginning of V

Input : $G(V, E)$, with V in degeneracy ordering and E as ordered adjacency lists $L_1 \dots L_n$.
Output : $G(V, E)$ with V relabeled in degeneracy ordering so that the j heads are $v_1, \dots, v_j$.

```
1 s, e ← n + 1
2 foreach s in decreasing order from n to 1 do                    // Reorder, relabel N_>(v)
3     if s is not a head then
4         e ← e − 1
5         foreach v < s in L_s do replace s with e in L_v
6         replace all v < s with 0 in L_s
7         swap L_s and L_e
8 foreach v in decreasing order from n to 1 do                    // Relabel N_<(v)
9     foreach x ≠ 0 in L_v do replace the rightmost 0 in L_x with v
```

Storing B_K. By applying Lemma 13, since we can check in constant time whether a vertex is in $V_{<\min(K)}$, we only need to store B'_K, whose size is $O(d)$ by definition. The computation of B'_K, which costs $\tilde{O}(qd)$ time applying Lemma 14, is done once for K and once for each child. Hence its cost is payed at most twice for each clique.

Observe that the cost of CHILD-EXISTS is dominated by that of the computation of $N(K'_v)$, since testing $N(K'_v) \cap B_K = \emptyset$ and $N(K'_v) \cap N_<(v) = \emptyset$ can be done in $\tilde{O}(1)$ time for each element of $N(K'_v)$. Applying Lemma 15, we can conclude that:

▶ **Lemma 16.** *Function* CHILD-EXISTS *can be implemented such that the cumulative cost of all the calls to* CHILD-EXISTS(K, v) *for a fixed K is $O(d)$ space and $\tilde{O}(d \min\{q\Delta, m\})$ time.*

Iterating over B_K. In order to avoid storing B'_K, we can compute the intersection between $N(K'_v)$ and B_K iterating over both sets. The iterator for B'_K works as described in Lemma 14. Since the elements of both $N(K'_v)$ and B'_K are iterated in increasing order, the cost of computing their intersection for each call of CHILD-EXISTS is the sum of the costs of the two iterations. Consider the sum of these costs over all the calls of CHILD-EXISTS: applying Lemma 14 and Lemma 15, we obtain a total cost of $\tilde{O}(d \min\{q\Delta, m\} + qd \min\{qd, n\})$, since $|\text{CAND}| \leq \min\{qd, n\}$. As a result, we have the following lemma:

▶ **Lemma 17.** *Function* CHILD-EXISTS *can be implemented such that the cumulative cost of all the calls to* CHILD-EXISTS(K, v) *for a fixed K is $O(q)$ space and $\tilde{O}(d \min\{q\Delta, m\} + qd \min\{qd, n\})$ time.*

The final algorithm corresponds to plugging Algorithm 2 into Algorithm 1, i.e. replacing SPAWN with IMPROVED-SPAWN. In order to show that the final algorithm takes $O(q)$ or $O(d)$ space as well, we should also perform the setup described in Section 6. Using Algorithm 3 as setup, and applying Fact 1 and Lemmas 12, 16, and 17, we obtain our final result (recall that $q - 1 \leq d \leq \Delta \leq n - 1 \leq m$ and $d = O(\sqrt{m})$).

▶ **Theorem 18.** *Let G be an undirected connected graph with n vertices and m edges, whose adjacency lists are ordered and whose vertices are labeled in degeneracy ordering. Let Δ be the maximum degree of its vertices, q the size of its largest clique, and d its degeneracy. Then there exists an algorithm that lists all the maximal cliques in G that has $\tilde{O}(m)$ setup time, $O(q)$ space usage, and $\tilde{O}(qd(\Delta + qd))$ delay. The latter bound improves to $\tilde{O}(qd\Delta)$ if space usage is increased to $O(d)$. Space is always sublinear in the size of G.*

Table 2 Experimental results of our comparison with output-sensitive algorithms in the state of the art. For the graph statistics (upper part), we refer to Table 1. For the comparison (lower part) we have considered the total time (TIME), the delay (DELAY), and the space (MEM).

	DBLP-2008			AMAZON-0505			IN-2004			EU-2005		
n	511 163			410 236			1 353 703			862 664		
m	1 871 070			2 439 436			13 126 172			16 138 468		
α	447 563			1 034 135			3 384 922			5 727 256		
Δ	576			2 760			21 869			68 963		
d	114			10			488			388		
q	115			11			489			387		
ALGO.	TIME *sec*	DELAY *ms*	MEM *MiB*	TIME *sec*	DELAY *ms*	MEM *MiB*	TIME *sec*	DELAY *ms*	MEM *MiB*	TIME *sec*	DELAY *ms*	MEM *MiB*
CN	>2h	475	97.56	>2h	636	78.26	>2h	5 285	258.21	>2h	4 077	164.54
MU	39.5	18	3.01	16.9	22	3.01	6 102.7	3 691	52.62	>2h	11 395	30.52
CXQ	21.2	4	0.11	60.7	12	0.25	>2h	6 004	1.02	>2h	621	3.15
RALG2	1.8	0.6	1.57	3.1	0.8	0.81	66.6	401	6.46	237.4	245	3.16
ALG2	2.3	15	0.01	4.4	0.9	0.01	100.7	410	0.03	363.8	277	0.02

CN: Chiba-Nishizeki [11] MU: Sparse graph Makino-Uno [25] CXQ: Chang et al. [9]
RALG2: Recursive Algorithm 2 ALG2: Algorithm 2

6 Setup

Algorithm 3 gives some details on how to modify the degeneracy ordering of the vertices so as to place the heads at the beginning in $\tilde{O}(m)$ time and $O(1)$ space. As already noted, the resulting order is still a degeneracy ordering. The main idea is to maintain a *sliding window* over the list of adjacency lists. The window incrementally collects the heads while moving from the last vertex to the first one in the ordering, so that its final position is the beginning of the ordering. The window is moved by swapping the greatest vertex before the window, which is called s and the greatest vertex in the window, which is called e. In case s is a head, the window is not shifted but simply enlarged to include s. While moving the window, the occurrences of the labels of the swapped vertices s and e must be updated accordingly. To this aim just the occurrences of s in $N_>(v)$ are relabeled, for each v neighbor of s. At the end, the backward neighborhoods can be updated by looking at the forward ones.

It is straightforward to see that the time needed by Algorithm 3 is $\tilde{O}(m)$. Indeed, in the loop in line 2, each adjacency list is iterated at most once. The loop in line 8 has the same time bound, i.e. $\tilde{O}(m)$: it traverses the adjacency lists of all vertices performing replace operations, which can be done in logarithmic time on ordered lists. The constant space usage follows from the fact that just $O(1)$ variables are stored.

7 Experimental Evaluation

In this section, we report some preliminary experiments on some large real-world networks that have been taken from LASAGNE (`lasagne-unifi.sourceforge.net/`). Our computing platform is a 24-core machine with Intel(R) Xeon(R) CPU E5-2620 v3 at 2.40GHz, with 128GB of shared memory. The operating system is a Ubuntu 14.04.2 LTS, with Linux kernel version 3.16.0-30. The code has been written in `C++` and forced to run on a single core.

Table 2 reports the results of the comparison between our algorithm and existing algorithms which use at most linear space and are output-sensitive, i.e. the ones providing a bounded delay or cost per solution, namely CN, MU, and CXQ (see Table 1). We limited the running time to two hours. The algorithm RALG2 refers to a recursive version of Algorithm 2. ALG2 refers to Algorithm 2 which takes $O(q)$ memory by using Lemma 17 (similar results

can be found for the iterative version which uses Lemma 16). For each algorithm and for each graph, we report the total time, the maximum delay found while running, and the memory usage (excluding the input size). Note that both RALG2 and ALG2 are significantly faster than CN, MU, and CXQ. It is worth observing that the results highlight how the delay of RALG2 and ALG2 depends on d and q as shown in Theorem 18.

The running times of ALG2 are in general higher than RALG2, even though its performance is competitive. On the other hand, ALG2 uses the smallest amount of memory, namely always less than 0.03 MiB even when these graphs have hundreds of thousands of vertices and millions of edges. The most striking result is for EU-2005, having more than 850 thousands of vertices and more than 16 millions of edges, where our algorithm uses just 0.02 MiB. Finally, observe that the memory seems to be proportional to d and q, since IN-2004 and EU-2005 have relatively higher memory consumption.

For the sake of completeness, we also considered the state of the art for Bron-Kerbosch based algorithms. In particular, we ran the algorithm in [15, 16] using the code provided by the authors (we are grateful to them). Even though its cost per solution can be higher than ours and its delay can be exponential, this algorithm is on the average 3.7 times faster than ALG2. On the other hand, as this algorithm uses linear memory, its memory consumption is on the average 878.9 times larger than the memory of ALG2.

References

1 Nesreen K. Ahmed, Jennifer Neville, Ryan A. Rossi, and Nick G. Duffield. Efficient graphlet counting for large networks. In *2015 IEEE International Conference on Data Mining, ICDM 2015*, pages 1–10. IEEE, 2015.

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

3 Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *J. ACM*, 42(4):844–856, 1995.

4 David Avis and Komei Fukuda. Reverse search for enumeration. *Discrete Applied Mathematics*, 65(1-3):21–46, 1996.

5 Edward Bierstone. Cliques and generalized cliques in a finite linear graph. *Unpublished report*, 1960.

6 Andreas Björklund, Rasmus Pagh, Virginia Vassilevska Williams, and Uri Zwick. Listing triangles. In *Automata, Languages, and Programming – 41st International Colloquium, ICALP 2014, Proceedings, Part I*, pages 223–234, 2014.

7 Coenraad Bron and Joep Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM*, 16(9):575–576, 1973.

8 Frédéric Cazals and Chinmay Karande. A note on the problem of reporting maximal cliques. *Theor. Comput. Sci.*, 407(1-3):564–568, 2008.

9 Lijun Chang, Jeffrey Xu Yu, and Lu Qin. Fast maximal cliques enumeration in sparse graphs. *Algorithmica*, 66(1):173–186, 2013.

10 James Cheng, Linhong Zhu, Yiping Ke, and Shumo Chu. Fast algorithms for maximal clique enumeration with limited memory. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD, pages 1240–1248, 2012.

11 Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on Computing*, 14(1):210–223, 1985.

12 Carlo Comin and Romeo Rizzi. An improved upper bound on maximal clique listing via rectangular fast matrix multiplication. *CoRR*, abs/1506.01082, 2015. URL: `http://arxiv.org/abs/1506.01082`.

13 Alessio Conte, Roberto De Virgilio, Antonio Maccioni, Maurizio Patrignani, and Riccardo Torlone. Finding all maximal cliques in very large social networks. In *Proceedings of*

the 19th International Conference on Extending Database Technology, EDBT 2016., pages 173–184, 2016.

14 Nan Du, Bin Wu, Liutong Xu, Bai Wang, and Pei Xin. Parallel algorithm for enumerating maximal cliques in complex network. In *Mining Complex Data*, pages 207–221. Springer, 2009.

15 David Eppstein, Maarten Löffler, and Darren Strash. Listing all maximal cliques in sparse graphs in near-optimal time. In *ISAAC (1)*, pages 403–414, 2010.

16 David Eppstein, Maarten Löffler, and Darren Strash. Listing all maximal cliques in large sparse real-world graphs. *ACM Journal of Experimental Algorithmics*, 18, 2013.

17 Komei Fukuda. Note on new complexity classes ENP, EP and CEP. `https://www.inf.ethz.ch/personal/fukudak/old/ENP_home/ENP_note.html`, 1996. Accessed: 2016-02-17.

18 Leonhard Gerhards and Wolfgang Lindenberg. Clique detection for nondirected graphs: Two new algorithms. *Computing*, 21(4):295–322, 1979.

19 Eo N. Gilbert. Enumeration of labelled graphs. *Canad. J. Math*, 8(1):05–411, 1956.

20 Alain Gély, Lhouari Nourine, and Bachir Sadi. Enumeration aspects of maximal cliques and bicliques. *Discrete Applied Mathematics*, 157(7):1447–1459, 2009.

21 Christopher J. Henry and Sheela Ramanna. *Transactions on Rough Sets XVI*, chapter Maximal Clique Enumeration in Finding Near Neighbourhoods, pages 103–124. Springer, 2013.

22 Alon Itai and Michael Rodeh. Finding a minimum circuit in a graph. *SIAM J. Comput.*, 7(4):413–423, 1978.

23 David S. Johnson, Mihalis Yannakakis, and Christos H. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27(3):119–123, 1988. `doi:10.1016/0020-0190(88)90065-8`.

24 H. C. Johnston. Cliques of a graph-variations on the Bron-Kerbosch algorithm. *International Journal of Parallel Programming*, 5(3):209–238, 1976.

25 Kazuhisa Makino and Takeaki Uno. New algorithms for enumerating all maximal cliques. In *Algorithm Theory-SWAT 2004*, pages 260–272. Springer, 2004.

26 R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: simple building blocks of complex networks. *Science*, 298:824–827, 2002.

27 John W. Moon and Leo Moser. On cliques in graphs. *Israel journal of Mathematics*, 3(1):23–28, 1965.

28 Gordon D. Mulligan and Derek G. Corneil. Corrections to Bierstone's algorithm for generating cliques. *J. ACM*, 19(2):244–247, 1972.

29 Long Pan and Eunice E. Santos. An anytime-anywhere approach for maximal clique enumeration in social network analysis. In *SMC*, pages 3529–3535. IEEE, 2008.

30 Frank Ruskey. *Combinatorial Generation*. 2003.

31 Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Experimental and Efficient Algorithms, 4th InternationalWorkshop, WEA 2005, Proceedings*, pages 606–609, 2005.

32 Matthew C. Schmidt, Nagiza F. Samatova, Kevin Thomas, and Byung-Hoon Park. A scalable, parallel algorithm for maximal clique enumeration. *Journal of Parallel and Distributed Computing*, 69(4):417–428, 2009.

33 Nino Shervashidze, S V N Vishwanathan, Tobias Petri, Kurt Mehlhorn, and Karsten M. Borgwardt. Efficient graphlet kernels for large graph comparison. In *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics, AISTATS 2009*, volume 5 of *JMLR Proceedings*, pages 488–495, 2009.

34 Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.*, 363(1):28–42, 2006.

35 Horace M. Trent. A note on the enumeration and listing of all possible trees in a connected linear graph. *Proceedings of the National Academy of Sciences*, 40(10):1004–1007, 1954.

36 Shuji Tsukiyama, Mikio Ide, Hiromu Ariyoshi, and Isao Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM J. Comput.*, 6(3):505–517, 1977.

37 Takeaki Uno. Two general methods to reduce delay and change of enumeration algorithms. *National Institute of Informatics (in Japan) Technical Report E*, 4:2003, 2003.

38 Takeaki Uno. An efficient algorithm for solving pseudo clique enumeration problem. *Algorithmica*, 56(1):3–16, 2008.

39 Leslie G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189–201, 1979.

40 Kunihiro Wasa, Hiroki Arimura, and Takeaki Uno. Efficient enumeration of induced subtrees in a k-degenerate graph. In Hee-Kap Ahn and Chan-Su Shin, editors, *Algorithms and Computation: 25th International Symposium, ISAAC 2014, Proceedings*, pages 94–102, Cham, 2014. Springer International Publishing.

41 Yanyan Xu, James Cheng, Ada Wai-Chee Fu, and Yingyi Bu. Distributed maximal clique computation. In *Big Data (BigData Congress), 2014 IEEE International Congress on*, pages 160–167. IEEE, 2014.