Listing Maximal Independent Sets with Minimal Space and Bounded Delay

Alessio Conte[1], Roberto Grossi[1], Andrea Marino[1(✉)], Takeaki Uno[2], and Luca Versari[3]

[1] Università di Pisa, Pisa, Italy
{conte,grossi,marino}@di.unipi.it
[2] National Institute of Informatics, Tokyo, Japan
uno@nii.jp
[3] Scuola Normale Superiore, Pisa, Italy
luca.versari@sns.it

Abstract. An independent set is a set of nodes in a graph such that no two of them are adjacent. It is maximal if there is no node outside the independent set that may join it. Listing maximal independent sets in graphs can be applied, for example, to sample nodes belonging to different communities or clusters in network analysis and document clustering. The problem has a rich history as it is related to maximal cliques, dominance sets, vertex covers and 3-colorings in graphs. We are interested in reducing the delay, which is the worst-case time between any two consecutively output solutions, and the memory footprint, which is the additional working space behind the read-only input graph.

1 Introduction

Given an undirected graph $G = (V, E)$ with $|V| = n$ nodes and $|E| = m$ edges, a maximal independent set (MIS) $I \subseteq V$ does not contain any two nodes connected by an edge, and is maximal under inclusion (no other $I' \supset I$ has this property). We pose the question whether listing MISs can be achieved efficiently in small space and bounded delay.

Although this problem originated in graph theory, as MISs are related to dominance sets, vertex covers and 3-colorings in graphs, we observe that data is networked in information systems nowadays. The classical problem of looking at patterns in texts or sequences, or trees, can be translated into graphs.[1] Here the patterns are MISs, which can be seen as a way to build samples that are independent from each other, thus motivating the question.

One possible field of application is in networks analysis, such as social science, where a MIS identifies a group of persons from a tightly connected community

Work partially supported by University of Pisa under PRA_2017_44 project on Advanced Computational Methodologies for the Analysis of Biomedical Data.

[1] The algorithmic techniques are different, and even the simple query asking if a path occurs in a graph is NP-hard. Nevertheless, discovering patterns in sequences and patterns in graphs are quite similar tasks, and can share techniques in some cases.

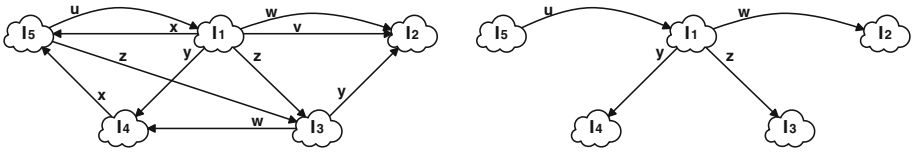

Fig. 1. Structure of the solution space defined by reverse search (left) and the solution tree defined by a PARENT function (right).

that are isolated from each other, or can be used as sample for communities, where each node of the MIS is a person from a different community.

MISs are a powerful tool for clustering: they can be used for clustering document collections (where two documents are linked if their content is similar), by using a MIS as a collection of starting points for the chosen clustering method or for clustering wireless networks to identify hierarchical structures [2]. Moreover, they are often used to build efficient indexes for answering shortest path or distance queries (see for instance [12]). MISs are applied for clustering purposes also in image segmentation, that aims at grouping image pixels into visually "meaningful" segments. In this case, the goal is to select the segments of an image that are distinct, and together partition the image area. In a graph where segments are nodes and edges correspond to the overlap of the segments, all the maximal independent sets correspond to all the non-overlapping segment partitions. [3] studied the maximum weighted independent set (MWIS) to get the maximally distinctive partitions by encoding a distinctiveness scoring of the segments into the nodes weights. This approach was also extended to clustering aggregation in general [15]. [21] modeled co-occurrences of words and documents in the web as a graph, and used MWIS's in this graph to find sets of important but distinct (i.e., rarely co-occurring) topics. However, the MWIS problem is NP-hard and hard to approximate. Listing all the MISs can also provide an exact solution for the latter problem, eventually testing different distinctiveness scoring systems.

Our Results. In this paper we describe an algorithm that lists all the MISs with $\tilde{O}(\min\{nd\Delta^2, mn\})$ delay—the $\tilde{O}$ notation ignores polylog factors—and $O(s)$ additional space, using the following parameters: d is the graph's degeneracy, that is, the minimum value d for which any induced subgraph has maximum node degree at most d; Δ is the maximum node degree; s is the maximum size of a MIS. We assume that the input graph is read-only, and the space complexity is the additional working space.

As it can be seen, the additional space is asymptotically minimal, and the delay can be as low as $\tilde{O}(n)$ (if d and Δ are $\tilde{O}(1)$), but never larger than the baseline of $O(nm)$ (ignoring logarithmic factors) given by Tskuiyama et al. [23]. We further reduce our time bound by providing a second algorithm with $\tilde{O}(\min\{d\Delta n, mn\})$ delay which increases the memory requirement to $O(n)$: this simultaneously improves both best known bounds for delay and space as $d\Delta$ is a pessimistic upper bound on the cost, which is smaller than m in practice.

Related Work. Listing MISs is a classical problem in enumeration which dates back at least to the 70s, with many results such as producing MISs in lexicographical order [16], experimentally or with guarantees [13], achieving $O(n^3)$ delay but using exponential space. Some results have been also proposed for particular classes of graphs: claw-free graphs [18], interval graphs, circular-arc graphs and chordal graphs [14,19,20], trees [14], permutation graphs [25].

In general, listing the MISs of a graph G is equivalent to listing the maximal cliques of its complement $\overline{G} = (V, \overline{E})$. However improved algorithms for maximal cliques, such as the space efficient solution that we presented in [9],[2] do not translate into improved bounds for listing MISs: the transformation from MISs to maximal cliques is not effective, especially in sparse graphs which have a dense complementary graph, but even in dense graphs, since their complementary graph can be dense too. These techniques mainly fall in the backtracking approach, as for [4], or in the reverse search paradigm introduced by [1].

In the former case, these approaches are not output sensitive for both cliques and MISs, in the sense that their guarantee on the running time is not related to the number of solutions. In the relatively recent work by Eppstein et al. [10] for cliques, the overall time $O(dn3^{d/3})$ becomes $O(n^2 \cdot 3^{n/3})$ to list all the MISs, as the degeneracy d can be $\Theta(n)$ in the complementary graph, while the delay remains exponential, as in the case of the algorithm in [22]. Moreover, the space usage, without storing the transformed graph, becomes $O(ns)$ for [22] and $O(n^2)$ for [10]. On the other hand, while adapting the reverse search for maximal cliques to MISs, the algorithms by Chiba and Nishizeki [6], by Makino and Uno [17] and Chang et al. [5] require $O(n^2 - m)$ space: recalling that arboricity, maximum degree, and degeneracy of the complementary graph can be linear, the delay bound becomes $O(n(n^2 - m))$ for [6], and $O(n^4)$ for [5,17]. Moreover, as shown next in Remark 1, the delay bound in [9] becomes $\tilde{O}(nm)$, which does not improve upon Tskuiyama et al. [23].

Also, since MISs can be considered a hereditary property or independence set system, they can be listed using the framework of Cohen et al. [7] but the resulting bounds still do not improve over those of [23]. For these reasons *ad hoc* algorithms for cliques and MISs have been proposed *separately* in the literature, and the best output sensitive bounds for MISs are $O(nm)$ delay with $O(n^2)$ space [23], or $O(n^{2.37})$ delay with $O(n^2)$ space by using matrix multiplication [17], or $O(n^{2.09})$ delay with $O(n^{4.27})$ space [8].

Our Approach. The algorithmic challenges addressed here are related to the reverse search, which is a powerful enumeration technique introduced by Avis and Fukuda [1]. Consider the graph-like structure shown in Fig. 1 (left), which we call the *solution digraph*: each "cloud", or node, corresponds to a distinct MIS, and an arrow from MIS I_i to MIS I_j with label v means that I_j can be computed from I_i through a node v, using a rule that is specific for the algorithm at hand. As in other techniques, such as divide and conquer, the algorithmic contribution is the efficient implementation of the generic step, for the problem at hand.

[2] This paper has been organized so as to highlight the novelties with respect to [9].

To list all the MISs, we use the rule to traverse the solution digraph and output each node/solution once. An easy way to do so is to keep track of all visited nodes. Even though such methods have been used, e.g. in [13], they are expensive as they require exponential memory. Reverse search avoids this issue by choosing a single *parent* I_i for each MIS I_j, such that $I_i < I_j$ for some given order (such as the lexicographic one), among all MISs leading to I_j. This way it induces a directed spanning forest on the solution digraph, as illustrated in Fig. 1 (right). Some MISs have no parent and are the *roots* of the spanning forest. Note that the solution digraph and directed spanning forest are for the purpose of explanation and never materialized. The roots can be easily identified and are at most n.

Traversing the solution digraph can be implicitly done by performing a DFS: each time we explore the possible children solutions and recur in just the ones whose parent is the current solution, following [17]. This visit can be made iterative, by avoiding the stack of the recursion. We can restore the state of the parent when returning from the call to a child. This strategy is particularly useful if we want to achieve sublinear additional memory, since we avoid using memory proportional to the height of the recursion tree, where a single bit per recursion level is too much. Here the techniques in [9] for maximal cliques do not translate smoothly into improved bounds for MISs, as discussed in Remark 1.

The complexity of the reverse search is dominated by the cost of applying the rule to the current MIS in the directed spanning forest. Computing the rule is expensive as the time spent checking not fruitful candidate children is completely charged on the delay of the algorithm. Thus we introduce a novel technique that allows us to apply the rule only to the children, rather than to *all* the out-neighbors in the solution digraph. We check a necessary condition, which is lighter to compute than the rule, so that the rule is actually applied to selected out-neighbors. During this task, we use a small amount of space.

2 Preliminaries

Let $G = (V, E)$ be an undirected and connected graph, represented as adjacency lists. In this work, we will use the following notation: $N(x)$ is the neighborhood of node x, and $\overline{N}(x) = V \setminus (N(x) \cup \{x\})$ the complementary-neighborhood; for a set of nodes A, $\overline{N}(A) = \bigcap_{x \in A} \overline{N}(x)$ (and we consider $\overline{N}(\emptyset) = V$).

We assume the nodes labeled as $v_1 < v_2 < \cdots < v_n$, in a *reversed degeneracy ordering*, i.e., so that $v_n < v_{n-1} < \cdots < v_1$ is a degeneracy ordering (see, e.g. [11]). It is easy to see that this ordering can be obtained with $O(1)$ additional space if it is not given.

We define $V_{<v_i}$ as $\{v_1, v_2, \ldots, v_{i-1}\}$. Given a set of nodes A, we then define $A_{<v_i}$ as respectively $A \cap V_{<v_i}$; for brevity, let $N_{<v_i}(v_i)$ be $N_<(v_i)$. $V_{>v_i}$, $A_{>v_i}$ and $N_>(v_i)$ are similarly defined. Note that in a degeneracy ordering $|N_>(v)| \leq d$, so as we are using a *reversed* degeneracy ordering, we have $|N_<(v)| \leq d$.

Given an independent set I and a vertex $v \in V \setminus I$, we denote as I'_v the set $I_{<v} \cap \overline{N}(v)$.

Given any two independent sets I_i and I_j, we say that $I_i < I_j$ if I_i is lexicographically smaller than I_j as node sets, thus inducing a lexicographic order on the independent sets. Given an independent set $I \subseteq V$, COMPLETE(I) is defined as the lexicographically smallest MIS that contains I, and can be computed by iteratively adding the smallest element that can be added to I, obtaining the following lemma.

Lemma 1. COMPLETE(I) *can be computed in* $\tilde{O}(m)$ *time.*

3 Listing MISs

Given an independent set I and a node $v \in V \setminus I$ such that $I_{<v} \neq \emptyset$, formula (1) generates a new maximal independent set.

$$F(I, v) = \text{COMPLETE}(I'_{<v} \cup \{v\}) \tag{1}$$

In the solution digraph illustrated in Fig. 1, there is an arrow $I_i \rightarrow I_j$ labeled with v if and only if $I_j = F(I_i, v)$. The parent-child relationship, which defines the directed spanning forest, is as follows.

$$\text{PARENT}(I) = \text{COMPLETE}(I_{<\text{PI}(I)}) \tag{2}$$

where PI(I), the *parent index* of I, is the smallest element $v \in I$ for which COMPLETE($I_{\leq v}$) $= I$. Note that if $I_i = \text{PARENT}(I_j)$ and $v = \text{PI}(I_j)$, then I_i, I_j, v satisfy formula (1).

The *roots*, which have no parent, can be found as COMPLETE($\{v\}$), for any $v \in V$ such that $\min\{\text{COMPLETE}(\{v\})\} = v$; the number of roots is at most n.

During the traversal of the solution digraph, in the current MIS I_j we compute its parent I_i and the parent index v, thus restoring the state of the traversal when returning from the call to I_j. Keeping this in mind, the delay per listed MIS is bounded by the maximum amount of time spent in the current MIS, say I, observing that this time is intermixed with the calls to its children.

The delay can be bounded by the cost of (*i*) computing I from its parent using formula (1); (*ii*) checking each candidate child to see if a call with formula (1) should be applied; finally (*iii*) if I is not a root, restoring the state to parent of I using formula (2). Indeed, once we get the costs (*i*)–(*iii*), we can employ the well-known alternative output technique in [24], so that the delay is bounded by the costs (*i*)–(*iii*) times a constant. We recall that in the alternative output technique, when the level (i.e. the distance from the root) is odd the output is done before exploring the children, otherwise it is done soon after.

Algorithm 1: BASE, Enumeration of Maximal Independent Sets

Assume $\min(\emptyset) = \texttt{null}$, and recall that $I'_v \equiv I_{<v} \cap \overline{N}(v)$.
For function CHILD-EXISTS refer to Algorithm 2

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

Function IS-ROOT (I)
    return PI(I) = min(I)
Function PARENT-STATE (I)
    v ← PI(I)
    return ⟨COMPLETE(I_<v), v⟩
Function GET-NEXT-CAND (I,v)
    return
    min{w ∈ ⋃_{u∈I} N̄_>(u) \ I  :
                                w > v}
Function COMPLETE (I)
    while N̄(I) ≠ ∅ do
        I ← I ∪ min{N̄(I)}
    return I
Function PI (I)
    return min{v ∈ I :
            COMPLETE(I_≤v) = I}
```

In the rest of the paper we give the details for our new two algorithms for listing MISs using the above notions. The first algorithm is presented in Sect. 4 and achieves $O(s)$ additional memory and $\tilde{O}(\min\{d\Delta^2 n, mn\})$ delay. The second algorithm is presented in Sect. 5 and reduces the delay to $\tilde{O}(\min\{d\Delta n, mn\})$, albeit using $O(n)$ space.

One of the core ideas for both algorithms is the efficient computation of the test in point (ii) above. For the sake of simplicity, this behavior is encapsulated by the function CHILD-EXISTS, which is executed for each possible candidate. Our algorithms minimize the space usage by implementing some efficient implicit iterators that avoid building sets explicitly. For instance, the set $I'_v = I_{<v} \cap \overline{N}(v)$ in formula (1) is never materialized, as its explicit computation is expensive both in terms of time and space.

Remark 1. Using the above ideas, a listing algorithm for MISs can be immediately obtained by adapting those for maximal cliques in [9] to MISs, using them on the complementary graph $\overline{G}$ explicitly (see Sect. 1) or implicitly by using the complementary neighborhood $\overline{N}()$ in place of the neighborhood $N()$ whenever the latter is needed. We refer to the resulting algorithm to list the MISs using this simple modification of [9] as BASE. This is shown in Algorithm 1 and uses the implementation of function CHILD-EXISTS provided by Algorithm 2.

We further remark that some important optimizations in [9] for cliques are not useful as they do not bring any benefit for BASE. Indeed, while in [9] we have $|N_>(x)| \leq d$ thanks to the degeneracy ordering and $|N(x)| \leq \Delta$, these bounds do not hold for complementary neighborhoods. Hence, the improvements done to check the existence of a child (see Algorithm 2 in [9]) does not improve upon the basic child conditions of Makino-Uno based approach [17], which is reported in Algorithm 2 (see also Algorithm 1 in [9]). Indeed, one of the most important

benefit of [9] was the speed up and the memory improvements of this function thanks to the definition of B_K (see Algorithm 2 in [9]), which can be stored in a $O(d)$ and allows to discard candidates. In the case of MISs, dealing with B_K can be costly, as its size can be $\Theta(n)$ for a generic K in $\overline{G}$. For this reason we need to use a more basic check and the function CHILD-EXISTS is replaced with the equivalent, more basic check used in Algorithm 1 in [9]. As a result, unfortunately, BASE has no benefit as shown next. The size of the CAND set increases to $\Theta(n)$, thus the cost per solution, that is the cost of a recursive call, is bounded by n times the cost of a COMPLETE call. As the cost of COMPLETE takes $\tilde{O}(m)$ (as shown in Lemma 1), this gives us a total time cost per solution $\tilde{O}(nm)$ which does not improve upon the one by Tsukiyama et al [23] (its additional space usage is $O(n)$).

Algorithm 2: CHILD-EXISTS check as in [9]

Input : A maximal independent set I and a node $v > \text{PI}(I)$
Output: *true* if $\text{COMPLETE}(I'_v \cup \{v\})$ is a child of I, *false* otherwise

```
1 Function CHILD-EXISTS (I, v)
2     I'_v ← I_<v ∩ N̄(v)
3     return COMPLETE(I'_v) = I and COMPLETE(I'_v ∪ {v})_<v = I'_v
```

In the following, in order to bound our costs, we will use the lemma below.

Lemma 2. $\text{PI}(I)$ *and* $\text{PARENT}(I)$ *can be computed in* $\tilde{O}(m)$ *time.*

Proof. Given a MIS I, with $x = \text{PI}(I)$, we have $\text{PARENT}(I) = \text{COMPLETE}(I_{<x})$. Furthermore, we know that $\text{COMPLETE}(I_{<y})$ is equal to I iff $y > x$. Thus by computing $\text{COMPLETE}(I_{<y})$ we know whether y is larger than x or not. We can thus look for x in a binary search-like fashion. We can thus find x by performing $O(\log |I|)$ times a COMPLETE call, to then compute $\text{COMPLETE}(I_{<x})$. The cost follows. □

4 Using Minimal Space $O(s)$

In this section we present our first algorithm, whose focus is minimizing the additional memory: the algorithm will only store $O(s)$ information on top of the input graph while keeping the performance competitive with state of the art approaches. This algorithm aims at improving the cost of CHILD-EXISTS of Algorithm 2 using $O(s)$ space. The improvements are due to two factors.

On one hand we identify stricter theoretical conditions to determine whether CHILD-EXISTS will succeed or not, which are used in place of CHILD-EXISTS; Lemmas 3 and 4 prove the correctness of these conditions, allowing us to prove the equivalence of Algorithms 2 and 3 in Lemma 5.

On the other hand, we provide non-trivial techniques which allow us to compute our theoretical conditions quickly and using only $O(s)$ additional space: in detail, we provide fast implicit iterators for sets which are too costly to compute, and simulate the behavior of COMPLETE, stopping it prematurely when suitable conditions are met.

Lemma 3. COMPLETE$(I'_v \cup \{v\})_{<v} = I'_v$ *is equivalent to* $\min\{\overline{N}(I'_v \cup \{v\})\} > v$.

Proof. Recalling its definition (see Sect. 2), we know that COMPLETE (I) adds nodes to I in increasing order: indeed, at a given step we select the smallest node x in $\overline{N}(I)$ and add it to I; in the following step $\overline{N}(I)$ will shrink because we added x to I, and its minimum cannot be smaller than (or equal to) x.

Let y be the first node selected by COMPLETE$(I'_v \cup \{v\})$, that is, $y = \min\{\overline{N}(I'_v \cup \{v\})\}$. If $y < v$, then COMPLETE$(I'_v \cup \{v\})_{<v} \neq I'_v$ as the earlier set contains y while the latter does not. Otherwise, if $y > v$ all other nodes selected by the COMPLETE function will too be greater than v; since $I'_v = I_{<v} \cap \overline{N}(v)$ we thus have COMPLETE$(I'_v \cup \{v\})_{<v} = I'_v$. □

Algorithm 3: Improved CHILD-EXISTS-MS check with $O(s)$ additional memory ($I'_v \equiv I_{<v} \cap \overline{N}(v)$ is actually not computed explicitly, see Lemma 9)

Input : A maximal independent set I and a node $v > \text{PI}(I)$
Output: *true* if COMPLETE$(I'_v \cup \{v\})$ is a child of I, *false* otherwise

1 **Function** CHILD-EXISTS-MS (I, v)
2 **if** *not* $\min\{\overline{N}(I'_v \cup \{v\})\} > v$ **then return** *false*
3 $A \leftarrow I'_v$
4 **while** $\overline{N}(A) \neq \emptyset$ **do**
5 $x \leftarrow \min\{\overline{N}(A)\}$
6 **if** $x \in I_{<v}$ **then** $A \leftarrow A \cup \{x\}$
7 **else return** $x \in I$

Lemma 4. *If* $v > \text{PI}(I)$*, Algorithm 3 line 7 returns true iff* COMPLETE$(I'_v) = I$.

Proof. Lines 4–7 of Algorithm 3 correspond to simulating COMPLETE(I'_v) until the node x that is selected to be added to I'_v is not in $I_{<v}$. Then two cases are possible: either $x \notin I_{>v}$ or $x \in I_{>v}$. In the first case, COMPLETE$(I'_v) \neq I$ since $x \in$ COMPLETE(I'_v) and $x \notin I$, so Algorithm 3 returns *false*. Otherwise, if $x \in I_{>v}$ we show that all nodes in $I_{<v}$ that were not in I'_v have been added to it: since I is an independent set, whose nodes are not adjacent to each other, any node in $I_{<v} \setminus I'_v$ must be in $\overline{N}(I'_v)$. As so far we only added nodes in $I_{<v} \setminus I'_v$ (line 6), any other node in $I_{<v}$ that was in $\overline{N}(I'_v)$, and was not added to I'_v, is still in it. However, we have that $x = \min\{\overline{N}(I'_v)\} > v$, thus all and only nodes in $I_{<v} \setminus I'_v$ have been added to I'_v, making the set equal to $I_{<v}$. As COMPLETE$(I_{<v}) = I$, it must be that COMPLETE$(I'_v) = I$ thus the algorithm returns *true*. □

We show that, as a result of Lemmas 3 and 4, we can conclude the following.

Lemma 5. CHILD-EXISTS-MS *in Algorithm 3 can be used in place of* CHILD-EXISTS *in* BASE.

Proof. By Lemma 3 we have that CHILD-EXISTS-MS will return *false* on line 2 iff COMPLETE$(I'_v \cup \{v\})_{<v} \neq I'_v$. Otherwise, by Lemma 4, CHILD-EXISTS-MS will return *true* if COMPLETE$(I'_v) = I$, and *false* otherwise. Thus CHILD-EXISTS-MS will return the same result as CHILD-EXISTS. □

Space and Time Cost of Algorithm 3

In the following we provide space and time bounds for Algorithm 3, by firstly fixing some useful properties in Lemmas 6, 7, and 8.

Notice that $I_{<v} \setminus I'_v = N_<(v) \cap I_{<v}$. Since $\overline{N}(I'_v \cup \{v\}) \subseteq \overline{N}(I'_v)$, if there exists a node $x \in \overline{N}(I'_v \cup \{v\})$ smaller than v, then x satisfies the properties shown in Lemma 6. We show that this lemma follows from the definition of parent index and it is useful to efficiently perform the computation in line 2 in Algorithm 3.

Lemma 6. *Let I be a MIS, and v a node s.t. $v \notin I$ and $v > \text{PI}(I)$. Then $\overline{N}(I_{<v})_{<v} = \emptyset$, and for each node x in $\overline{N}(I'_v)_{<v}$ we have that either x is in $I \cap N_<(v)$ or x has a neighbor in $I \cap N_<(v)$.*

Proof. Since $v \notin I$ and $v > \text{PI}(I)$, we have that $\text{COMPLETE}(I_{<v}) = I$ by definition of PI. If $\overline{N}(I_{<v})$ does contain a node x smaller than v, we could use one of such nodes to extend $I_{<v}$ and we would have $\text{COMPLETE}(I_{<v}) \neq I$, a contradiction.

Consider now $\overline{N}(I'_v)_{<v}$: As $I'_v \subseteq I_{<v}$, we have $\overline{N}(I'_v)_{<v} \supseteq \overline{N}(I_{<v})_{<v}$. Since $\overline{N}(I_{<v})_{<v} = \emptyset$, however, we have $\forall x \in \overline{N}(I'_v)_{<v}, x \notin \overline{N}(I_{<v})$, thus either x is in $I_{<v}$ has a neighbor in it by definition of $\overline{N}()$. As $x \in \overline{N}(I'_v)_{<v}$, if x is in $I_{<v}$, it is actually in $I_{<v} \setminus I'_v \subseteq N_<(v)$. The statement follows. □

Hence, we have to verify the conditions of Lemma 6 for the nodes $x \in \overline{N}(I'_v \cup \{v\})$. However, it is worth noting that the cost of storing $\overline{N}(I'_v \cup \{v\})$ is $O(n)$ which exceeds our memory requirements. To overcome this issue, we show in the following lemma how to build a heap-based iterator, that iterates over $\overline{N}(I'_v \cup \{v\})$ without computing it.

Lemma 7. *Let $X \subseteq V$ be a set of nodes and $Y = \bigcup_{x \in X} N(x)$. We can iterate over every $y \in Y$ in increasing order (without explicitly storing Y) in $\tilde{O}(\min\{|X|\Delta, m\})$ time using $O(|X|)$ additional space.*

Proof. Allocate a heap and add to it, for each node x in X, its smallest neighbor y (saving the x responsible for its addition). We can use this heap to iterate in order all nodes with a neighbor in X as follows: iteratively remove the minimum element y of the heap, recover the node x responsible for the addition of y, and insert in the heap the smallest neighbor of x larger than y. This way the smallest neighbor that we did not extract yet will always be on top of the heap. It is possible that the same node is extracted more than once; however we can trivially ignore duplicates as they appear contiguously, since nodes are extracted in increasing order. Adding/removing an element to/from the heap costs $\tilde{O}(1)$, so the total cost is bounded by $\tilde{O}(1)$ times the sum of all degrees of nodes in X, that is $\tilde{O}(\min\{|X|\Delta, m\})$. □

By Lemma 6, to answer the check at line 2, we can consider nodes y belonging to $I \cap N_<(v)$ or $N(I \cap N_<(v))$. In particular, for each of them, we will have to check that $y \notin N(v)$ and $N(y) \cap I'_v \neq \emptyset$. To this aim, we use the following lemma.

Lemma 8. *Let I be a MIS, $v \notin I$ a node such that $v > \text{PI}(I)$, and y any node. We have $N(y) \cap I'_v \neq \emptyset$ iff there exists $z \in N(y)$ such that $z < v$ and $z \in I$ and $z \notin N_<(v)$.*

Proof. Recall that $I'_v = I_{<v} \cap \overline{N}(v) = I_{<v} \setminus N_<(v)$, so all and only nodes in I'_v are smaller than v, in I, and not in $N_<(v)$. As $N(y) \cap I'_v \neq \emptyset$ iff any node in $N(y)$ is in I'_v, and $z \in I'_v$ iff ($z < v$ *and* $z \in I$ *and* $z \notin N_<(v)$) the statement follows. □

We are now ready to prove the overall cost of Algorithm 3.

Lemma 9. CHILD-EXISTS-MS *can be computed in* $\tilde{O}(\min\{d\Delta^2, m\})$ *time with* $O(s)$ *space.*

Proof. Consider line 2: since $\overline{N}(I'_v \cup \{v\}) \subseteq \overline{N}(I'_v)$, by Lemma 6, if there exists a node $x \in \overline{N}(I'_v \cup \{v\})$ smaller than v, then x is a neighbor of a node in $I \cap N_<(v)$ (note that x cannot be in $I \cap N_<(v)$ since it is in $\overline{N}(I'_v \cup \{v\})$). Thus, instead of computing I'_v and its complementary-neighborhood, we iterate over node y which has a neighbor in the set $X = I \cap N_<(v)$ using Lemma 7. For each y, we have to check that $y \notin N(v)$ and $N(y) \cap I'_v \neq \emptyset$; for this latter check we use Lemma 8. If any y fails the check, then we return *false*, otherwise $\overline{N}(I'_v \cup \{v\})_{<v} = \emptyset$ and we can continue. We have $|I \cap N_<(v)| \leq d$ (due to the reversed degeneracy ordering), thus the neighbors y that we have to test can be at most $d\Delta$. It follows that the iteration will cost $\tilde{O}(\min\{d\Delta, m\})$ time by Lemma 7. Furthermore, testing each node y as in Lemma 8 takes $\tilde{O}(|N(y)|)$ time as we can perform binary searches on I, thus the total cost of testing is bounded by $\tilde{O}(\min\{d\Delta^2, m\})$.

Consider now lines 4–7: Again, using Lemma 6 we know that all nodes $x \in \overline{N}(I'_v)_{<v}$ are either in $I \cap N_<(v)$, or have a neighbor in it. We can rewrite this condition as: x has a neighbor in $X' = (I \cap N_<(v)) \cup \{v\}$.

In order to compute x in line 5, since $\overline{N}(A)_{<v} \subseteq \overline{N}(I'_v)_{<v}$, we use Lemma 7 to iterate over all the neighbors of nodes in X'; this iteration will yield in increasing order all nodes at any point in $\overline{N}(A)_{<v}$. We actually do not store A, but only the nodes that are added to A during the while, which we will here call A'. Thus to check that a node x belongs to $\overline{N}(A)$, we check $N(x) \cap I'_v = \emptyset$ and $N(x) \cap A' = \emptyset$; the earlier part can be done with Lemma 8, while the latter in $\tilde{O}(A')$ time by using binary searches.

Once we found $x = \min\{\overline{N}(A)_{<v}\}$, if it passes the check on line 6 we add it to A and repeat the loop, otherwise we return the result of the check.

Note that, as we are only iterating over $\overline{N}(I'_v)_{<v}$, we will not find among them any node in $I_{>v}$. However, this is easily fixed by saving the node $\min\{I_{>v}\}$: if at any point we have $x > \min I_{>v}$, or we finish the iteration on X', we return *true* since in both cases the candidate to be added to A would have been $\min\{I_{>v}\}$.

As $|X'| \leq \min\{d+1, s\}$, and the number of nodes with a neighbor in X are bounded by $O(\min\{d\Delta, n\})$, similarly to above we can bound the cost of the iteration with $\tilde{O}(\min\{d\Delta, m\})$, and the total cost of testing with $\tilde{O}(\min\{d\Delta^2, m\})$. Furthermore, note that the condition in line 6 can only succeed up to $\min\{d, s\}$ times, as each time x is in $I_{<v} \setminus I'_v$, and $|I_{<v} \setminus I'_v| = |I \cap N_<(v)| \leq \min\{d, s\}$; this means that $|A'| \leq \min\{d, s\}$.

Thus the total cost of CHILD-EXISTS-MS is $\tilde{O}(\min\{d\Delta^2, m\})$, using additional space $O(|X| + |X'| + |A'|) = O(\min\{d, s\})$ by Lemma 7. □

By using Lemma 9, we are now able to prove the following result.

Theorem 1. *There exists an algorithm that enumerates all maximal independent sets with* $\tilde{O}(\min\{nd\Delta^2, mn\})$ *delay and* $O(s)$ *additional space.*

Proof. By using the structure described in Sect. 3, we can create an algorithm that enumerates all MISs (that is Algorithm 1 where CHILD-EXISTS is replaced by CHILD-EXISTS-MS), that is complete and correct by Lemma 5. Its delay is bounded by the costs of (i) the generation function $F(I, v) =$ COMPLETE$((I_{<v} \cap \overline{N}(v)) \cup \{v\})$ (ii) testing each candidate with CHILD-EXISTS-MS, and (iii) PARENT-STATE to return to the parent solution. These costs are respectively (i) $\tilde{O}(m)$ (as shown in Lemma 1), (ii) $\tilde{O}(n \cdot \min\{d\Delta^2, m\})$ as we apply Lemma 9 to each of the $O(n)$ candidates, and (iii) $\tilde{O}(m)$ (as shown in Lemma 2). Since $m \leq n\Delta$, this gives a total cost of $\tilde{O}(m + \min\{nd\Delta^2, mn\}) = \tilde{O}(\min\{nd\Delta^2, mn\})$.

As we have no recursion stack, the additional space is simply storing I and v, and the space required by CHILD-EXISTS-MS, that is $O(s)$. □

5 Faster Version Using $O(n)$ Additional Memory

In this section, we propose a new algorithm which achieves a smaller time cost per solution by exploiting properties of the search space and an additional data structure of size $O(n)$, which mainly stores the amount of neighbors in $I_{<v}$ of each node in the graph. We use a function CHILD-EXISTS-FAST which improves the time cost of Algorithm 3, by constructing and maintaining this data structure. This is fundamental to improve the running time since it allows us to reduce the search space of the nodes considered by CHILD-EXISTS (and the corresponding iterations), as just nodes with zero neighbors in $I_{<v}$ need to be considered. Since v varies among all the possible candidates, even the ones not leading to a solution, this data structure cannot be rebuilt from scratch each time $I_{<v}$ changes, but needs to be properly updated and restored wherever possible. We will prove that we can cover these costs.

For the sake of completeness, the final pseudo-code is shown in Algorithm 4. The functions IS-ROOT and PARENT-STATE are the same as in BASE (see Algorithm 1). We will now analyze the difference between BASE and Algorithm 4, to show that they are equivalent, and that, hence, Algorithm 4 is correct.

The first difference we can notice is that we use a new function GET-NEXT-CAND with respect to the one in BASE. The new one is faster to compute, since the sum of the costs of all the calls done with the same I takes just $O(n)$ time but returns a superset of the one of BASE. This fact increases the number of candidate nodes to test but, on the other hand, testing them will be faster here due to an improved version of CHILD-EXISTS (see Lemma 12).

Algorithm 4: Fast enumeration of maximal independent sets

```
1  Assume min(∅) = null.
2  Function ITERATIVE-SPAWN(I)
3      v ← PI(I);  prev ← v
4      BUILD(WS, I_<v)
5      while true do
6          childless ← true
7          while v ← GET-NEXT-CAND(I, v) ≠null do
8              UPDATE(WS, (I_<v \ I_<prev))
9              prev ← v
10             if CHILD-EXISTS-FAST(I, v, WS) then
11                 I ← COMPLETE(I'_v ∪ {v})
12                 childless ← false
13                 BUILD(WS, I_<v)
14                 break
15         if childless then
16             if IS-ROOT(I) then return
17             else
18                 ⟨I, v⟩ ← PARENT-STATE(I)
19                 prev ← v
20                 BUILD(WS, I_<v)
```

Algorithm 5: Fast check of the existence of a child and other auxiliary functions

```
1  Function CHILD-EXISTS-FAST(I, v, WS)
2      for x ∈ I ∩ N_<(v) do
3          for y ∈ N(x) do WS[y] − = 1
4      C ← {x : WS[x] = 0}
5      if not min{N̄(I'_v ∪ {v})} > v then return false
6      while C not empty do
7          c ← min{C}
8          if c ∈ (I \ I'_v) then
9              C ← C \ (N(c) ∪ {c})
10         else
11             UPDATE(WS, I ∩ N_<(v))
12             return c ∈ I
13 Function BUILD(WS, A)
14     ∀i ∈ V : WS[i] ← |N(i) ∩ A|
15 Function UPDATE(WS, A)
16     ∀i ∈ V : WS[i] + = |N(i) ∩ A|
17 Function GET-NEXT-CAND (I,v)
18     return min{(V \ I)_>v}
```

Lemma 10. *We can use function* GET-NEXT-CAND *of Algorithm 5 in place of* GET-NEXT-CAND *in* BASE.

Proof. GET-NEXT-CAND of Algorithm 5 iterates over $(V \setminus I)_{>v}$, while GET-NEXT-CAND in BASE iterates over $\{w \in \bigcup_{u \in I} \overline{N}(u) \setminus I \ : w > v\}$. As the latter is a subset of the former, Algorithm 5 will iterate over all the candidates that will lead to a child. Furthermore, nodes in $(V \setminus I)_{>v}$ are still greater than $\text{PI}(I)$, thus the conditions for CHILD-EXISTS are met, and the nodes that do not lead to a child will fail the check. □

Notice that the candidate set size is $\Theta(n)$, since a single complementary neighborhood has size $\Theta(n)$. Another difference with respect to BASE, is that we use function CHILD-EXISTS-FAST instead of CHILD-EXISTS function to improve its computational time. To this aim, we use an additional data structure WS (for *weights*), which we keep suitably updated in order to satisfy the following invariant.

Lemma 11. *When* CHILD-EXISTS-FAST(I, v, WS) *is called in Algorithm 4, for each node* $i \in V$*, we have* $\text{WS}[i] = |N(i) \cap I_{<v}|$.

Proof. We first remark that for any A, B s.t. $A \cap B = \emptyset$, if $\forall i \in V$ $\text{WS}[i] = |N(i) \cap A|$ and we call UPDATE(WS, B) we obtain $\forall i \in V$ $\text{WS}[i] = |N(i) \cap (A \cup B)|$. To prove the lemma, it is thus sufficient to show that just before line 8 is executed, $\forall i \in V$ $\text{WS}[i] = |N(i) \cap I_{<prev}|$ (for the value of *prev* at that point).

Let us consider when WS was last modified when line 8 is executed: If this is the first time that the while loop in line 7 is executed, then WS was last modified in either lines 4, 13 or 20 by calling BUILD$(\text{WS}, I_{<v})$. Indeed, we have $\forall i \in V$ $\text{WS}[i] = |N(i) \cap I_{<prev}|$ by definition of BUILD, as we set $prev = v$ in lines 3, 9, and 20 respectively, and *prev* remained unchanged until line 8. Otherwise, note that CHILD-EXISTS-FAST leaves WS unchanged (the changes at Line 2 are canceled out by Line 11), thus WS was last modified by the previous execution of line 8 in the while loop (line 7). We prove this case by induction: Let us refer to the values of v and *prev* at line 8 in the j-th iteration of the loop as v_j and $prev_j$. Assume that $\forall i \in V$ $\text{WS}[i] = |N(i) \cap I_{<prev_j}|$ was true at line 8 in the j-th iteration. As $v_j = prev_{j+1}$ (see line 9), after the line is executed, by the remark at the beginning of the proof, we have $\forall i \in V$ $\text{WS}[i] = |N(i) \cap I_{<v_j}| = |N(i) \cap I_{<prev_{j+1}}|$. Since the condition is true for the first iteration, it is true for any iteration, thus the statement holds in each case. □

By Lemma 11, the hypothesis on the WS data structure in the following lemmas are met, so that we can use the new CHILD-EXISTS-FAST instead of CHILD-EXISTS and CHILD-EXISTS-MS.

Lemma 12. *Suppose that* $\text{WS}[x] = |I_{<v} \cap N(x)|$ *for each* $x \in V$*. Then* CHILD-EXISTS-FAST *(I,v,*WS*) in Algorithm 5 can be used instead of* CHILD-EXISTS *(I,v) and* CHILD-EXISTS-MS *(I,v).*

Proof. Line 5 in Algorithm 5 is the same as line 2 in Algorithm 3. The loop at line 2 decrements WS[y] once for each neighbor of y in $I \cap N_<(v)$. After the loop we have that for each $x \in V$, WS[x] $= |(I_{<v} \setminus N_<(v)) \cap N(x)| = |I'_v \cap N(x)|$, thus $x \in \overline{N}(I'_v)$ iff WS[x] $= 0$. It follows that C is initialized to exactly $\overline{N}(I'_v)$. Thus, the loop in lines 6–10 will have the same outcome as the corresponding loop in lines 4–7 of Algorithm 3: when a node c is selected in line 8, C is updated as if c was added to I'_v, thus the following iterations will select the same nodes that would be selected in Algorithm 3, until finally line 10 is executed, which will give the same outcome as line 7 in Algorithm 3. □

The function ITERATIVE-SPAWN has been modified only with the addiction of the function UPDATE, which has effect only on the variable WS which is used by GET-NEXT-CAND. Thus since the functions GET-NEXT-CAND and CHILD-EXISTS-FAST used by Algorithm 4 are equivalent to, respectively, GET-NEXT-CAND and CHILD-EXISTS of BASE by Lemmas 10, 11, and 12, then the function SPAWN too is equivalent to the one of BASE, obtaining the following lemma.

Lemma 13. *Algorithm 4 correctly computes all maximal independent sets.*

Space and Time Cost of Algorithm 4

In the following, we analyze the complexity of Algorithm 4. We have already discussed the cost of the new GET-NEXT-CAND function. In particular, we analyze the cost of maintaining the counters and the cost of CHILD-EXISTS-FAST.

Lemma 14. *For any set $A \subseteq V$,* UPDATE(WS, A) *takes* $O(\min\{|A|\Delta, m\})$ *time.*

Proof. We can obtain the cost above by iterating over all nodes in A and for each node x incrementing by 1 the counter of its neighbors. This is bounded by $O(|A|\Delta)$, and by $O(m)$ too as it is a sum of the degrees of distinct nodes. □

The cost for BUILD is simply $O(n + \min\{|A|\Delta, m\})$, since we can set to zero each WS[x] for each $x \in V$ and then apply UPDATE(WS, A).

Lemma 15. CHILD-EXISTS-FAST *takes* $O(d\Delta)$ *time and* $O(n)$ *space.*

Proof. Recall from the proof of Lemma 9 that, as $v \notin I$ and $v >$ PI(I), we have $\min\{\overline{N}(I_{<v})\} > v$, thus $\forall x \in (V_{<v} \setminus I)$ we have WS[x] > 0.

Thus we compute C by simply adding to it any node y whose counter WS[y] is set to 0 during the loop at line 2. This is only guaranteed to add nodes which are smaller than v, but this will be enough. Recalling the proof of Lemma 12, we thus have that $C = \overline{N}(I'_v)_{<v}$. This costs us $O(\min\{d\Delta, m\})$ time, as it is the sum of the degrees of $|I \cap N_<(v)| \leq d$ distinct nodes. Line 5 takes $O(\Delta)$. Indeed, we can compute $\overline{N}(I'_v \cup \{v\})_{<v}$ as $C \setminus N(v)$. If this set is not empty, then the check fails and we return *false*.

Let us now consider the cost of the loop at line 6. Keeping C in a dynamic dictionary, lines 7 and 8 take both $\tilde{O}(1)$ time and 9 takes $\tilde{O}(|N(c)|)$ time. As the loop is executed a maximum of $|I_{<v} \setminus I'_v| + 1 \leq |N_<(v)| + 1 \leq d + 1$ times and c is different every time, this takes $\tilde{O}(\min\{d\Delta, m\})$ time. As in Lemma 9 we should

store $\min\{I_{>v}\}$: since initially C only contains $\overline{N}(I'_v)_{<v}$ rather than $\overline{N}(I'_v)$, we return *true* if C actually becomes empty or if $c > \min\{I_{>v}\}$, as in both these cases $\min\{I_{>v}\}$ would have been the candidate actually selected in line 9, that would result in the algorithm returning *true*. Finally, before returning the result, we restore the data structure WS by calling UPDATE(WS, $I \cap N_<(v)$). This takes $O(\min\{d\Delta, m\})$ time by Lemma 14.

Since the additional space used is $O(|C|) = O(\min\{d\Delta, n\})$, CHILD-EXISTS-FAST can be computed in $\tilde{O}(\min\{d\Delta, m\})$ time, using $O(n)$ space. □

By plugging the results of Lemmas 14 and 15 into the analysis of Theorem 1, we conclude the following.

Theorem 2. *Algorithm 4 lists all the maximal independent sets with a delay of* $\tilde{O}(\min\{nd\Delta, nm\})$*, and* $O(n)$ *additional space.*

Proof. Using Theorem 1 with the costs of CHILD-EXISTS-FAST given by Lemma 15 we obtain a total cost of $\tilde{O}(\min\{nd\Delta, nm\})$. We however need to add to steps (i) and (iii) the cost of BUILD(WS, $I_{<v}$) which takes $\tilde{O}(\min\{n\Delta, m\})$. Furthermore, we add to step (ii) the cost of UPDATE(WS, $(I_{<v} \setminus I_{<prev})$ for each candidate v. We argue that for any specific I, any node $i \in I$ is in $I_{<v} \setminus I_{<prev}$ only once, since v is never decreasing for I and *prev* keeps track of the previous value of v. The total cost is thus the same as UPDATE(WS, I), i.e., $\tilde{O}(\min\{|I|\Delta, m\})$ by Lemma 14. As $|I| \leq n$, neither of these additions affects the total cost of $\tilde{O}(\min\{nd\Delta, m\})$.

Space usage is given by the size of WS and C stored in CHILD-EXISTS-FAST, i.e., $O(n)$. □

6 Conclusions

In this paper we studied the enumeration of maximal independent sets (MISs) in graphs, introducing new ideas to check efficiently which neighbors in the reverse search should be explored, as this task is time- and space-consuming. For a read-only input graph, our results are the first algorithm with minimal additional space, proportional to the size of the largest MIS, and an algorithm which improves both delay and space usage of known approaches. We remark that a MIS can indeed have linear size: in this case, due to the modular nature of our algorithms, the execution of the minimal space version can switch on-the-fly to the faster version which uses $O(n)$ space without increasing the asymptotic space usage.

References

1. Avis, D., Fukuda, K.: Reverse search for enumeration. Discrete Appl. Math. **65**(1–3), 21–46 (1996)
2. Basagni, S.: Finding a maximal weighted independent set in wireless networks. Telecommun. Syst. **18**(1), 155–168 (2001)

3. Brendel, W., Todorovic, S.: Segmentation as maximum-weight independent set. In: Advances in Neural Information Processing Systems, pp. 307–315 (2010)
4. Bron, C., Kerbosch, J.: Finding all cliques of an undirected graph (algorithm 457). Commun. ACM **16**(9), 575–576 (1973)
5. Chang, L., Yu, J.X., Qin, L.: Fast maximal cliques enumeration in sparse graphs. Algorithmica **66**(1), 173–186 (2013)
6. Chiba, N., Nishizeki, T.: Arboricity and subgraph listing algorithms. SIAM J. Comput. **14**(1), 210–223 (1985)
7. Cohen, S., Kimelfeld, B., Sagiv, Y.: Generating all maximal induced subgraphs for hereditary and connected-hereditary graph properties. JCSS **74**(7), 1147–1159 (2008)
8. Comin, C., Rizzi, R.: An improved upper bound on maximal clique listing via rectangular fast matrix multiplication. CoRR, abs/1506.01082 (2015)
9. Conte, A., Grossi, R., Marino, A., Versari, L.: Sublinear-space bounded-delay enumeration for massive network analytics: maximal cliques. In: ICALP, vol. 148, pp. 1–15 (2016)
10. Eppstein, D., Löffler, M., Strash, D.: Listing all maximal cliques in large sparse real-world graphs. ACM J. Exp. Algorithmics **18** (2013). Article No. 3.1
11. Eppstein, D., Strash, D.: Listing all maximal cliques in large sparse real-world graphs. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 364–375. Springer, Heidelberg (2011). doi:10.1007/978-3-642-20662-7_31
12. Fu, A.W.-C., Wu, H., Cheng, J., Wong, R.C.-W.: IS-LABEL: an independent-set based labeling scheme for point-to-point distance querying. Proc. VLDB Endow. **6**(6), 457–468 (2013)
13. Johnson, D.S., Yannakakis, M., Papadimitriou, C.H.: On generating all maximal independent sets. Inf. Proc. Lett. **27**(3), 119–123 (1988)
14. Leung, J.Y.-T.: Fast algorithms for generating all maximal independent sets of interval, circular-arc and chordal graphs. J. Algorithms **5**(1), 22–35 (1984)
15. Li, N., Latecki, L.J.: Clustering aggregation as maximum-weight independent set. In: Advances in Neural Information Processing Systems, pp. 782–790 (2012)
16. Loukakis, E., Tsouros, C.: A depth first search algorithm to generate the family of maximal independent sets of a graph lexicographically. Computing **27**(4), 349–366 (1981)
17. Makino, K., Uno, T.: New algorithms for enumerating all maximal cliques. In: Hagerup, T., Katajainen, J. (eds.) SWAT 2004. LNCS, vol. 3111, pp. 260–272. Springer, Heidelberg (2004). doi:10.1007/978-3-540-27810-8_23
18. Minty, G.J.: On maximal independent sets of vertices in claw-free graphs. J. Comb. Theor. Ser. B **28**(3), 284–304 (1980)
19. Okamoto, Y., Uno, T., Uehara, R.: Linear-time counting algorithms for independent sets in chordal graphs. In: Kratsch, D. (ed.) WG 2005. LNCS, vol. 3787, pp. 433–444. Springer, Heidelberg (2005). doi:10.1007/11604686_38
20. Okamoto, Y., Uno, T., Uehara, R.: Counting the number of independent sets in chordal graphs. J. Discrete Algorithms **6**(2), 229–242 (2008)
21. Olteanu, A., Castillo, C., Diaz, F., Vieweg, S.: CrisisLex: a lexicon for collecting and filtering microblogged communications in crises. In: ICWSM (2014)
22. Tomita, E., Tanaka, A., Takahashi, H.: The worst-case time complexity for generating all maximal cliques and computational experiments. TCS **363**(1), 28–42 (2006)
23. 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)

24. Uno, T.: Two general methods to reduce delay and change of enumeration algorithms. National Institute of Informatics (in Japan) (2003). TR E, 4
25. Yu, C.-W., Chen, G.H.: Generate all maximal independent sets in permutation graphs. Int. J. Comput. Math. **47**(1–2), 1–8 (1993)