LISTING MAXIMAL SUBGRAPHS SATISFYING STRONGLY ACCESSIBLE PROPERTIES[*]

ALESSIO CONTE[†], ROBERTO GROSSI[‡], ANDREA MARINO[‡], AND LUCA VERSARI[‡]

Abstract. Algorithms for listing the subgraphs satisfying a given property (e.g., being a clique, a cut, a cycle, etc.) fall within the general framework of set systems. A set system $(\mathcal{U}, \mathcal{F})$ consists of a ground set $\mathcal{U}$ (e.g., a network's nodes) and a family $\mathcal{F} \subseteq 2^{\mathcal{U}}$ of subsets of $\mathcal{U}$ that have the required property. For the problem of listing all sets in $\mathcal{F}$ maximal under inclusion, the ambitious goal is to cover a large class of set systems, preserving at the same time the efficiency of the enumeration. Among the existing algorithms, the best-known ones list the maximal subsets in time proportional to their number but may require exponential space. In this paper we improve the state of the art in two directions by introducing an algorithmic framework based on reverse search that, under standard suitable conditions, simultaneously (i) extends the class of problems that can be solved efficiently to *strongly accessible* set systems, and (ii) reduces the additional space usage from exponential in $|\mathcal{U}|$ to *stateless*, i.e., with no additional memory usage other than that proportional to the solution size, thus accounting for just polynomial space.

Key words. Graph Enumeration, Set Systems, Maximal Common Subgraphs, Reverse Search

AMS subject classifications. 05C85, 68R10

1. Introduction.

Enumerating the solutions that satisfy certain conditions, such as subgraphs with desired combinatorial properties, is a common task in several areas such as network analysis [1, 19, 33, 37, 38, 48], bioinformatics [13, 27, 31, 36, 46] and graph databases [3, 15, 55]. Algorithms for subgraph enumeration have a long history: introduced in the 70s in the context of enumerative combinatorics and computational complexity [24, 32, 41, 44, 53], their interest has quickly broadened to a variety of other communities.

In this scenario graph enumeration has left the purely combinatorial border to meet the strict requirements of algorithm design [54]: not only a given listing problem must fit a given class of complexity, but its algorithms must be efficient in real-world applications too. Many papers address how to eliminate redundancy by listing only solutions which respect properties of closure [8] or maximality [14]. Other papers make a considerable effort to generalize the graph properties to be enumerated and unify the corresponding approaches [5, 12, 14, 32, 49]. These generalizations allow the same algorithm to solve many different problems without rediscovering the wheel each time.

The contribution of this paper goes in the above direction: efficient listing algorithms for inclusion-maximal solutions in large networks are within a framework proposed here, which models and solves several problems at the same time, leaving the algorithm designer in charge of few core tasks depending on the specific application. In particular, we focus on set systems [14, 32], defined in Section 1.1, as they can model interesting graph properties such as containing a clique, a cut, a cycle, a matching, and so on.

A main obstacle to the above goal is the trade-off between generality and efficiency, as a general output-sensitive algorithm for listing maximal solutions is impossible even for independence systems (unless P=NP) [32]. For this reason, Lawler et al. [32] and later Cohen et al. [14] have shown that the hardness of a listing problem can be

[*]Submitted to the editors October 16, 2017; accepted for publication January 8, 2019.

[†]National Institute of Informatics, Japan (conte@nii.ac.jp).

[‡]University of Pisa, Italy (grossi@di.unipi.it, marino@di.unipi.it, luca.versari@di.unipi.it).

linked to that of solving an easier core task, called *input-restricted problem* in [14]. We proceed in this direction, so that our framework can deal with a wide class of problems and, at the same time, improve from the currently known exponential space usage to polynomial (which means just a poly-logarithm of the output size [22]).

1.1. Set systems. Given a ground set of elements $\mathcal{U}$, called universe, a *set system* $(\mathcal{U}, \mathcal{F})$ is pair where $\mathcal{F} \subseteq 2^{\mathcal{U}}$ is a family of sets of elements that meet a given property, satisfied by the empty set (i.e. $\emptyset \in \mathcal{F}$) [32]. A solution X for $(\mathcal{U}, \mathcal{F})$ is any set of elements $X \subseteq \mathcal{U}$ such that $X \in \mathcal{F}$; moreover, it is *maximal* (short term for inclusion-maximal) if there exists no $Y \in \mathcal{F}$ such that $X \subset Y$. This is of interest for many applications in the literature (e.g. [12, 49, 26, 29]) where instances can have a number of maximal solutions that is much smaller than $|\mathcal{F}| \leq 2^{|\mathcal{U}|}$.

We will use the concept of property as a synonym for set system, when appropriate. As an example, consider the problem of listing maximal cliques in a graph $G = (V, E)$ (i.e., maximal sets of nodes verifying the property of forming a clique). It can be formulated as a set system with $\mathcal{U} = V$, where a node set X belongs to $\mathcal{F}$ if and only if the induced subgraph $G[X]$ is a clique.

In the rest of the paper we fix an arbitrary apriori order on the elements in $\mathcal{U}$, hereafter called *natural order*, and label each element $1, \dots |\mathcal{U}|$ according to its rank in this order. We also assume that the membership test for $\mathcal{F}$ is provided as an algorithm, i.e., we can check membership of X in $\mathcal{F}$ in time $\mathcal{M}_T = \text{poly}(|\mathcal{U}|)$ and space $\mathcal{M}_S = \text{poly}(|\mathcal{U}|)$ (e.g., we can check whether a set of nodes X is a clique in $\mathcal{M}_T = O(|X|^2)$ time, using $\mathcal{M}_S = O(m)$ space to store G). Moreover, we will denote by q the maximum size of a solution.

We focus on set systems which fulfill the following condition: $X, Y \in \mathcal{F}$ and $X \subset Y$ implies that there exists $z \in Y \setminus X$ such that $X \cup \{z\} \in \mathcal{F}$.

This class of set systems is called *strongly accessible* [4, 8], and was originally proposed to guarantee efficient enumeration of solutions respecting closure properties. However, it is also of particular interest when dealing with maximality because it allows a polynomial time procedure to check maximality or finding a maximal solution as long as $\mathcal{M}_T$ is polynomial: indeed a solution S of a strongly accessible set system $(\mathcal{U}, \mathcal{F})$ is maximal if and only if there is no $x \in \mathcal{U} \setminus S$ such that $S \cup \{x\} \in \mathcal{F}$. When this property is not satisfied, it may be hard to even recognize the maximality of a solution. For example, the more general class of *accessible* set systems [8] only requires that a solution $X \in \mathcal{F}$ has an element $z \in X$ such that $X \setminus \{z\} \in \mathcal{F}$. Here a solution may be non-extensible, i.e., no element can be added to it, but it may still be a subset of a larger solution. Subgraphs with lower bounded density are a member of this class, as we may always remove the lowest degree node from a graph without decreasing its density, but finding out whether X is maximal or a larger solution including X exists is shown to be NP-complete by Uno [52].

Furthermore, strongly accessible set systems include many known classes of set systems/properties, such as the following ones.

- Hereditary properties [32], i.e., properties that are still satisfied by any subset of a solution. These are also known as independence systems and include matroids.
- Connected-hereditary graph properties [14], i.e., graph properties that are still satisfied by connected subgraphs of a solution.
- Greedoids [30], as the exchange property implies the strongly accessible one.[1]

[1]Since $X \subset Y$ implies $|Y| > |X|$.

- Confluent properties [8], i.e., set systems where the union of two solutions is itself a solution whenever their intersection contains a nonempty solution.

As we mentioned, a key concept related to enumeration in set systems is the input-restricted problem, found in Lawler et al. [32] and formally named by Cohen et al. [14]:

Definition 1.1 (Input-restricted problem [32, 14]). *Given a set system* $(\mathcal{U}, \mathcal{F})$*, a maximal solution* $S \in \mathcal{F}$ *and an element* $u \in \mathcal{U} \setminus S$*, the* input-restricted problem $\textsc{restr}(S, u)$ *asks to list all the maximal solutions of the set system* $(S \cup \{u\}, \mathcal{F})$*.*

In this paper, we assume that we can enumerate the solutions of any input-restricted problem $\textsc{restr}(P, w)$ in $\mathcal{R}_T$ time, using $\mathcal{R}_S$ space; moreover, $\mathcal{R}_N$ denotes an upper bound on the number of solutions of the input-restricted problem. For example, for a maximal clique X and a node z not in that clique, an algorithm for $\textsc{restr}(X, z)$ lists all the maximal cliques in $G[X \cup \{z\}]$. Thus $\mathcal{R}_T = O(|X|) = O(q)$, $\mathcal{R}_S = O(|X|) = O(q)$, and $\mathcal{R}_N = 2$ for clique X and the clique obtained by taking z and its neighbors in X.

Finally, to analyze the complexity of enumeration algorithms, we will make use of the concepts of *delay*, an upper bound for the time required to output the i-th solution after the output of the $(i-1)$-th one (or since the start of the computation when $i = 1$), and *polynomial total time*, which refers to algorithms whose running time is polynomial in the size of both the input and the output [26, 24].

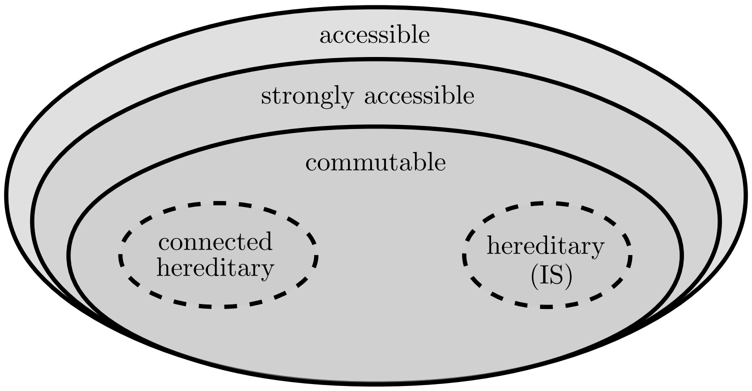

Fig. 1. *Relationships between some classes of set systems.*

1.2. Results. We define a new class of set systems called *commutable set systems*, that is a well populated subclass of strongly accessible set system, as it contains for example all the set systems induced by hereditary, connected-hereditary and confluent properties. We define the commutable set systems as strongly accessible set systems which also verify the *commutable property* given below.

Definition 1.2 (Commutable property). *A strongly accessible set system* $(\mathcal{U}, \mathcal{F})$ *is called* commutable *when the following holds for any nonempty* $X, Y \in \mathcal{F}$ *and* $a, b \in \mathcal{U}$*:* $X \cup \{a\} \in \mathcal{F}$*,* $X \cup \{b\} \in \mathcal{F}$ *and* $X \cup \{a, b\} \subseteq Y$ *implies* $X \cup \{a, b\} \in \mathcal{F}$*.*

We consider the commutable property only in combination with the strongly accessible one due to the fundamental implications that these latter has on testing maximality and finding a maximal solution. Furthermore, it is easy to see how commutable properties include hereditary, connected-hereditary, and confluent properties.[2] Figure 1 summarizes the inclusion relationships discussed so far.

We prove that any commutable set system admits polynomial delay and polynomial space enumeration of its maximal sets, based on the reverse search [5], provided that its input-restricted problem can be solved in polynomial time.[3]

Theorem 1.3. *The maximal solutions of any commutable set system $(\mathcal{U}, \mathcal{F})$ can be listed with $O(q^4|\mathcal{U}|^2\mathcal{M}_T\mathcal{R}_N + \mathcal{R}_T|\mathcal{U}|)$ delay, using $O(|\mathcal{U}| + \mathcal{M}_S + \mathcal{R}_S)$ space, where q is the maximum size of a solution, $\mathcal{R}_T$ and $\mathcal{R}_S$ are the time and space required to solve the input-restricted problem, $\mathcal{R}_N$ is the number of solutions it can yield, and $\mathcal{M}_T$ and $\mathcal{M}_S$ are the time and space required to check membership in $\mathcal{F}$.*

For example, with cliques, Theorem 1.3 gives $O(q^4n^2q^22 + qn) = O(q^6n^2)$ delay and $O(n + q + q) = O(n)$ space usage.

As the commutable property is satisfied by connected-hereditary properties, our framework gives polynomial space output-sensitive algorithms for connected-hereditary properties. This is the first polynomial space output-sensitive framework for this class of systems, and in particular solves the open problem in Cohen et al. [14]. As an application, our result implies the first output-sensitive polynomial space listing algorithm for maximal isomorphisms corresponding to maximal common connected induced subgraphs, using just $O(q)$ memory, as described in Section 3.

Our results improve the space usage of the algorithm in Berlowitz et al. [7] to enumerate connected k-plexes, making it polynomial space while increasing the delay by only a polynomial factor. We also give new polynomial total time, polynomial space algorithms for listing connected bipartite (induced) subgraphs and feedback vertex (arc) set on undirected (directed) graphs. See Section 4 for details.

Our approach also extends to strongly accessible set systems independently from the input-restricted problem (but still assuming $\mathcal{M}_T = \text{poly}(|\mathcal{U}|)$), with delay parameterized by the maximum solution size and arguably close to optimal, as formalized in Theorem 1.4.

Theorem 1.4. *The maximal solutions of any strongly accessible set system $(\mathcal{U}, \mathcal{F})$ can be listed with $\tilde{O}(2^q)$ delay, where $\tilde{O}$ ignores factors polynomial in $\mathcal{U}$ and q is the maximum size of a solution. Moreover, there is no general listing algorithm for the problem with less than $\Omega(2^{q/2})$ delay unless the Strong Exponential Time Hypothesis [25] is false.*

1.3. Related work. Many papers in the literature focus their attention on listing maximal solutions of specific set systems, solving them with problem-specific techniques. For example, they focus on cliques [9, 21, 47, 35, 2], independent sets [26, 49], acyclic subgraphs [57, 42], matchings [23, 50], k-plexes [7, 56] which all correspond to hereditary properties (also known as independence systems [14]).

It is well known that a general output-sensitive algorithm for these kinds of problems would imply P=NP [32]. On the other hand, output-sensitive time and polynomial space general frameworks [14, 32] that require little problem-specific insights (such as being able to solve the input-restricted problem) have been developed in the

[2]To see the latter one, i.e., how the confluent property implies the commutable one, it suffices to observe that the two solutions $X' = X \cup \{a\}$ and $X'' = X \cup \{b\}$ have a nonempty solution (X) in their intersection, thus by the confluent property $X \cup \{a, b\} \in \mathcal{F}$.

[3]For completeness, this also proves that if the input-restricted problem is solvable in polynomial total time and polynomial space, the problem is solvable in polynomial total time and polynomial space, as an instance of the input-restricted problem cannot have more solutions than the original problem.

past. However, as shown in Section 3.2, a direct adaptation of these frameworks to commutable set systems is not possible unless P = NP.

These frameworks have been generalized to connected-hereditary properties in Cohen et al. [14] while still maintaining the same problem-specific requirements. This generalization achieves its goal by sacrificing space efficiency and obtaining an exponential-space algorithm.

On the other hand, enumeration for set systems has been studied in the very general setting of strongly accessible set system, but under different requirements on the sets. In particular, the frameworks by Boley et al. [8] and Arimura et al. [4] are able to list all the *closed* sets in a strongly accessible set system, for some closure operator that satisfies extensivity, monotonicity, and idempotence properties. However, maximality is not defined by a closure operator, so a different approach is needed.

Finally, low memory enumeration approaches have been proposed in the past to deal with huge data sets. In this setting, we require the enumeration algorithm to use a small amount of space on top of the input data. As an example, previous work is able to achieve efficient enumeration algorithms using polynomial space for hereditary properties [14], or even space proportional to the size of the largest solution produced on specific problems [5, 17, 16].

These techniques, unfortunately, cannot be applied outside of hereditary set systems. For example, [14, 17, 16] require the key operation of finding *lexicographically minimum* solution containing a given set of elements according to the natural order. This problem is solvable in polynomial time for hereditary properties, but we show that it is NP-hard for at least one connected-hereditary property (see Section 3.2).

Our proposed framework is based on reverse search [5] and circumvents this issue by relaxing this requirement to weaker ones, and fulfills these weaker requirements by means of a more sophisticated order, on more general set systems than just hereditary ones. This allows us to obtain algorithms with polynomial delay and space usage for set systems which are not hereditary, such as the one described in Section 3.

2. Algorithmic Framework.

This section will explain how our algorithmic framework works, for both strongly accessible set systems and (the more specific case of) commutable set systems.

First, in Section 2.1 we will define a new order, called prefix-closed order, that facilitates the design of an efficient function to obtain a maximal solution, called COMPLETE. Using these notions, we will describe an implicit arborescence forest among all the maximal solution, through the PARENT function, which defines the main structure of the enumeration. Section 2.2 will then prove the fundamental properties of the PARENT function that make our algorithm work, and Section 2.3 will provide the exponential-delay algorithm for any strongly accessible set system.

After that, Sections 2.4 and 2.5 will explain how to modify the above algorithm to get a polynomial delay one on commutable set systems. Finally, Sections 2.6 and 2.7 will give our main algorithm and its time complexity analysis, and Section 2.8 will explain the modifications needed to achieve polynomial space.

2.1. Beyond the natural order.

Our goal is to define a parent-child structure among maximal solutions, as common to reverse search-based approaches. We do so by means of a total order among maximal solutions, and given a maximal solution S, a function which allows us to find another maximal solution T that precedes it in this order.

Known reverse search-based approaches such as [14, 17, 16] use the natural order, and implement this task by taking subsets $S' \subset S$, then finding the lexicographically smallest solution T containing S'. Unfortunately this can be NP-hard in strongly accessible set systems (see again Section 3.2). We thus use a more sophisticated order among solutions, based on a *prefix-closed* internal order of the elements.

In the following we consider a strongly accessible set system $(\mathcal{U}, \mathcal{F})$, and use $Z = \{x \in \mathcal{U} : \{x\} \in \mathcal{F}\}$ to denote the set of singleton solutions. By definition of strongly accessible set systems, each nonempty solution $S \in \mathcal{F}$ contains at least one element of Z, as $\emptyset \in \mathcal{F}$ and clearly $\emptyset \subset S$. Given a solution $S \in \mathcal{F}$, we use $S^+ = \{a \in \mathcal{U} \setminus S : S \cup \{a\} \in \mathcal{F}\}$ to denote which elements of $\mathcal{U}$ can be added to S while preserving membership in $\mathcal{F}$.

We do not define a total order on $\mathcal{U}$ but rather a family of orders, which we call *prefix-closed*, using a function Π that yields an order of a subset of $\mathcal{U}$ depending on some given parameters. More formally:

DEFINITION 2.1 (Prefix-closed orders). *Let $\Pi(X, v)$ be a family of orders parameterized by $X \in \mathcal{F}$ and $v \in X \cap Z$ such that $\Pi(X, v)$ yields a permutation of $X \cup X^+$. For $X \in \mathcal{F}$ and $v \in X \cap Z$, let us denote by $x_1^v, ..., x_k^v$ the elements of X ordered according to $\Pi(X, v)$.*[4] *We call the family Π* prefix-closed *if for all $X \in \mathcal{F}$ and $v \in X \cap Z$, and $i \in \{1, ..., k-1\}$, the following properties hold*

(first) *The minimal element is u, i.e., $x_1^u = u$.*

(prefix) *The i-th prefix $X_i = \{x_1^u, ..., x_i^u\}$ of X is a solution, i.e., $X_i \in \mathcal{F}$.*

(greedy) *The element x_{i+1} is the minimal element of $X_i^+ \cap X$ with respect to the order $\Pi(X_i, v)$.*

The rationale of the above properties is based upon the fact that each subset X of a maximal solution S does not necessarily belong to $\mathcal{F}$ (as the set system is not hereditary). When discovering S, the *first* property indicates that we can start from an element $v \in S \cap Z$, whereas the *greedy* property indicates that we can iteratively expand $X = \{v\}$ by considering the elements of $X \cup X^+$ in a prefix-closed order, so that any prefix $\{x_1, \ldots, x_j\}$ thus found is in $\mathcal{F}$ for the *prefix* property.

We will discuss two prefix-closed orders, specifically in Section 2.3 (for strongly accessible properties) and Section 2.5 (for just commutable properties).

In the rest of the paper, for a given prefix-closed order, we use the shorthand notation $\prec_S^t$ for convenience:

for any two elements $a, b \in \mathcal{U}$ we say $a \prec_S^t b$ to say that a occurs before b in $\Pi(S, t)$. We thus remark that $\Pi(S, t)$ and $\prec_S^t$ correspond to two equivalent ways of referring to the same permutation. We will frequently use the latter as its notation is more convenient in our proofs.

Given a solution $S \in \mathcal{F}$ we define its *seed*, denoted $\textsc{seed}(S)$, as the minimum element (in natural order) in the nonempty intersection $S \cap Z$. For simplicity's sake, we will omit the superscript t when $t = \textsc{seed}(S)$, giving us the simplified notations $\prec_S$ (as this will frequently be the case). Moreover, observe that $S^+ = \emptyset$ when S is maximal, thus $\prec_S$ defines an order on the elements of $S \cup S^+ = S$. In this case, we will call *solution order* of the maximal solution S the permutation $s_1, \ldots, s_{|S|}$ induced by $\prec_S$.

For completeness, we show that the natural order is not sufficient as it is not prefix-closed for at least one connected-hereditary property.

Natural order is not prefix-closed for paths in a cycle. Consider the toy problem of listing all the subgraphs of a cycle $C = \{c_0, \ldots, c_{n-1}\}$, with $n > 3$, that are

[4]Note that $x_1 = v$ and that $x_i, x_{i+1} \in X$ are not necessarily consecutive in $\Pi(X, v)$ as some elements from X^+ can be interleaved with them.

paths. The problem is connected-hereditary since any connected subgraph of a path is a path. Furthermore, for any possible labelling of the nodes of C we must have a contiguous triplet c_i, c_{i+1}, c_{i+2} (indices taken modulo n) such that c_{i+1} is largest according to the natural order, i.e., $c_i < c_{i+1}$ and $c_{i+1} > c_{i+2}$. Then the prefix of the path $\{c_i, c_{i+1}, c_{i+2}\}$ containing its two smallest elements is $\{c_i, c_{i+2}\}$, that is not connected.

The prefix-closed order in Definition 2.1 allows us to define a greedy function, called COMPLETE, that deterministically produces a maximal solution containing X, for any given $X \in \mathcal{F}$.

COMPLETE(X): iteratively add to X the smallest element in X^+ (according to $\prec_X$), as long as X^+ is nonempty; the resulting solution is maximal by definition of strongly accessible set systems.

It should be noted that since X is expanded by COMPLETE, order $\prec_X$ is implicitly changed at each step of its computation (in particular, SEED(X) may also change). However, we can observe that as long as SEED(X) is not changed, COMPLETE(X) will tend to add elements coherently with the solution order of the resulting maximal solution, due to the *greedy property* of the prefix-closed order.

To close this section, we give two final notions. Firstly, given a maximal solution S, we denote by $S[j]$ the *prefix* $s_1, \ldots, s_j$ of its solution order (by $\prec_S$). Secondly, using the solution order, we can finally define our order $<$ among maximal solutions in $\mathcal{F}$: given any two maximal solutions $S \neq T$, consider their solution orders $s_1, \ldots, s_{|S|}$ and $t_1, \ldots, t_{|T|}$, respectively. When $s_1 \neq t_1$, we say that $S < T$ if and only if $s_1 < t_1$. Otherwise, let $i > 1$ be the smallest index for which $s_i \neq t_i$ (which exists as S and T are maximal and distinct). Then, we say that $S < T$ if and only if $s_i \prec_L t_i$, where $L = S[i-1] = T[i-1]$ is the longest common prefix of the two orders.[5]

2.2. Arborescence forest on the maximal solutions.

We now use COMPLETE to define the PARENT of a maximal solution: this induces a parent-child relationship on the set of all maximal solutions, which defines an arborescence forest (i.e., a union of arborescences). Thus, being able to generate the children of a solution will be enough to do a traversal on this arborescence forest, giving us all solutions. To this end, let us define the following concepts.

DEFINITION 2.2 (Parent, parent index, core, root). *Given a maximal solution S, let $s_1, \ldots, s_{|S|}$ be the solution order of S.*

If COMPLETE$(S[j]) = S$ *for all* $1 \leq j \leq |S|$, *we say that S is a* root.

Otherwise, let $j > 1$ be the smallest index such that COMPLETE$(S[j]) = S$*:*

- PI$(S) = s_j$ *is the* parent index*,*
- CORE$(S) = S[j-1]$ *is called* core *prefix,*
- PARENT$(S) =$ COMPLETE(CORE(S)) *is the maximal solution that is the* parent *of S.*

Furthermore, if $P =$ PARENT(S), we say that S is a *child* of P. The following key property is at the heart of our reverse search approach to obtain the desired arborescence forest.

LEMMA 2.3. PARENT$(S) < S$ *for every non-root maximal solution S.*

[5]It is straightforward to see that both s_i and t_i belong to L^+ and thus to the permutation $\Pi(L, \text{SEED}(L))$, and SEED$(L) =$ SEED$(S) =$ SEED(T), so this is equivalent to saying $s_i \prec_{S[i-1]} t_i$ or $s_i \prec_{T[i-1]} t_i$.

Proof. We show that $\textsc{complete}(S[j]) \leq S$ for any $1 \leq j \leq |S|$. Then, the lemma immediately follows by the definition of $\textsc{parent}$.

If $\textsc{complete}(S[j]) = S$, we have nothing to prove. Otherwise, let the solution orders of S and $T = \textsc{complete}(S[j])$ be $s_1, \dots, s_{|S|}$ and $t_1, \dots, t_{|T|}$, respectively. Since $s_1 \in S[j] \subseteq T$, if $t_1 \neq s_1$ then it must be that t_1 is smaller than s_1 in natural order (by definition of $\textsc{seed}(T)$), thus $T < S$.

If $s_1 = t_1 = \textsc{seed}(S)$, let $i > 1$ be the smallest index such that $s_i \neq t_i$. Two cases are possible:

- If $i \leq j$, then $s_i \in S[j] \subseteq T$, and thus $s_i \in T$. Moreover, $T[i-1] = S[i-1]$, so as $s_i \in S[i-1]^+ = T[i-1]^+$, we have $s_i \in T[i-1]^+ \cap T$. By the *greedy* property (Definition 2.1) t_i is the minimal element of $T[i-1]^+ \cap T$ according to $\prec^{t_1}_{T[i-1]}$: Since s_i is in the scope of $\prec^{t_1}_{T[i-1]}$ (that is $Ti-1] \cup T[i-1]^+$), it follows that $t_i \prec_L s_i$ with $L = S[i-1] = T[i-1]$, i.e. $T < S$.
- If $i > j$, then $T[j] = S[j]$. Consider t_{j+1}: by the *greedy* property t_{j+1} is the minimal element of $T \setminus T[j]$ according to $\prec^{t_1}_{T[j]}$, and by the *prefix* property $t_{j+1} \in T[j]^+$. Since the first step of $\textsc{complete}(T[j])$ adds to $T[j]$ the minimal element of $T[j]^+$ according to $\prec^{t_1}_{T[j]}$, and the element added must be an element of T (as $T = \textsc{complete}(T[j])$), it follows that this element must be precisely t_{j+1}. The same applies to the following elements, meaning that $\textsc{complete}(T[j])$ adds the elements of $\{t_{j+1}, \dots, t_{|T|}\}$ exactly in the order given by $\prec^{t_1}_{T}$ (i.e., $\Pi(T, t_1)$). This holds in particular for $i \geq j+1$, as t_i is the element added to $T[i-1]$ by $\textsc{complete}(T[i-1])$, meaning that it is the minimal element of $T[i-1]^+$ according to $\prec^{t_1}_{T[i-1]}$. As $T[i-1] = S[i-1]$, we have $s_i \in T[i-1]^+$, and thus $t_i \prec_L s_i$ with $L = S[i-1] = T[i-1]$, meaning that $T < S$. ∎

Lemma 2.3 has a powerful implication, as indeed it allows us to find, for any non-root solution S, another one which is smaller according to $<$ for maximal solutions, without relying on the NP-hard problem of finding the lexicographically minimum solution containing a set of elements (used by some known approaches on the easier case of hereditary properties).

Furthermore, the following lemma allows us to identify all the solutions corresponding to the roots of the arborescences.

Lemma 2.4. *S is a root (i.e., $\textsc{complete}(S[j]) = S$ for all $1 \leq j \leq |S|$) iff $\textsc{complete}(S[1]) = S$.*

Proof. Let S be a maximal solution. If $\textsc{complete}(S[1]) \neq S$ then S is not a root by definition. Otherwise, assume $\textsc{complete}(S[1]) = S$. By the definition of $\textsc{complete}$, $\textsc{complete}(S[j])$ adds each time the minimum possible element of $S[j]^+$ according to $\prec_{S[j]}$, which by the *greedy* property of the solution order of S corresponds to s_{j+1}. This means that $\textsc{complete}(S[1])$ will add the elements in the solution order of S, and thus will consider all its prefixes $S[j]$, adding each time the $(j+1)$-th element. In turn, this means that $\textsc{complete}(S[j]) = S$ for all $1 \leq j \leq |S|$. ∎

Finally, we can state the following result.

Theorem 2.5. *The directed graph whose nodes are the maximal solutions of $\mathcal{F}$, and whose edges are $(\textsc{parent}(S), S)$ for each non-root maximal solution S, is an arborescence forest where each root S satisfies $\textsc{complete}(\{\textsc{seed}(S)\}) = S$.*

Proof. Since all edges are directed from the $\textsc{parent}$ of a solution to its child, and since $\textsc{parent}(S) < S$ by Lemma 2.3, the solution graph cannot contain cycles.

Moreover, since the indegree of each node is at most 1, it must be an arborescence forest. Finally, Lemma 2.4 gives a complete characterization of the roots of this graph, i.e. the nodes that have no parent. ∎

It is now clear that defining a family of prefix-closed orders Π is enough to define an arborescence forest among the maximal solutions of the given problem, whose roots can be identified in polynomial time. To finalize an enumeration algorithm for the maximal solution we just need to find the set $\textsc{children}(P) = \{S : P = \textsc{parent}(S)\}$, for a maximal solution P.

2.3. Listing maximal solutions in strongly accessible set systems. At this point it is interesting to make a quick detour on what can be achieved on strongly accessible set systems, not necessarily satisfying the commutable property, proving Theorem 1.4 (the reader interested only in the final algorithm may skip directly to Section 2.4 as remainder of the paper does not use the concepts defined in this section). By definition, we have that $\textsc{core}(S) \subsetneq \textsc{parent}(S)$. Thus, one way to generate all children of solution P is checking which sets $X \subseteq P$ are actually $\textsc{core}(S)$ for some child S of P. Specifically, setting $S = \textsc{complete}(X \cup \{v\})$, for each of the $2^{|P|}$ possible subsets $X \subseteq P$ and each $v \in \mathcal{U} \setminus X$, we have to check whether the conditions $P = \textsc{parent}(S)$, $X = \textsc{core}(S)$ and $v = \textsc{pi}(S)$ simultaneously hold.

We define a prefix-closed order $\overset{*}{\Pi}$ that satisfies Definition 2.1. Given $\{x_1, \dots, x_j\}$, let $\textsc{complete}^*(\{x_1, \dots, x_j\}, X)$ be a variation of $\textsc{complete}$ that chooses x_{j+1} simply as the minimum element (in natural order) in $\{x_1, \dots, x_j\}^+ \cap X$. Setting initially $x_1 = v$ gives $\overset{*}{\Pi}(X, v)$ and, consequently, the relation $\overset{*}{\prec}{}^v_X$ and an enumeration algorithm for strongly accessible set systems.

Definition 2.6 (prefix-closed order for strongly accessible set systems). *Given $X \in \mathcal{F}$ and $v \in X \cap Z$, the order $\overset{*}{\Pi}(X, v)$ is defined as: v first; then the remaining elements of X in the order in which they are added by $\textsc{complete}^*(\{v\}, X)$; then all the elements in X^+ (if any) in increasing natural order.*

We observe that $\overset{*}{\Pi}$ is a prefix-closed order, as it takes $X \in \mathcal{F}$ and $v \in X \cap Z$, and yields a permutation of $X \cup X^+$ satisfying all the conditions of Definition 2.1: the *first* property is trivially satisfied by definition; the *prefix* property is satisfied by definition of $\textsc{complete}^*()$, which only selects elements addible to the previous ones; finally, let $x_1, \dots, x_{|X|}$ be the order of the elements of X given by $\overset{*}{\Pi}(X, v)$, and $X[j] = \{x_1, \dots, x_j\}$ a prefix: the earliest element in $X[j]^+ \cap X$ according to $\overset{*}{\Pi}(X[j], v)$ is the minimum (in natural order) in $X[j]^+ \cap X$, which is precisely the one chosen for addition by $\textsc{complete}^*(X[j], X)$. Thus the *greedy* property is satisfied.

Within the scope of this paragraph, we set $\textsc{complete}(S) := \textsc{complete}^*(S, \mathcal{U})$. As we can compute $\textsc{complete}^*$ in polynomial time, we obtain an algorithm that takes $\tilde{O}(2^q)$ time per solution, where $\tilde{O}$ ignores polynomial factors in $|\mathcal{U}|$, and q is the maximum size of a solution: indeed, we can identify all roots in polynomial time by Lemma 2.4, then recursively find all children of each solution in time $\tilde{O}(2^q)$, as the number of possible $\textsc{core}(S)$ of a child S of P is bounded by $2^{|P|} \leq 2^q$. For each tentative child C, we can try all $|\mathcal{U}|$ possibilities for parent index p, and check whether the given $S = \textsc{complete}(C \cup \{p\})$ has $C = \textsc{core}(S)$ and $p = \textsc{pi}(S)$. While an algorithm with this running time may not be practical, it has interesting implications from a theoretical point of view: firstly, it follows that we can obtain polynomial space and delay listing algorithms for all strongly accessible set systems whose maximal solutions have size constant or logarithmic in $|\mathcal{U}|$, i.e., such that 2^q is polynomial.[6] Furthermore, we can prove this algorithm to be almost optimal in the general case, unless the popular Strong Exponential Time Hypothesis [25] (SETH hereafter) is false.[7] We do so exploiting a reduction by Lawler et al. [32].

LEMMA 2.7. *An algorithm that generates all the α maximal solutions of any strongly accessible set system has a worst case time complexity of $\Omega(\alpha 2^{q/2})$, unless SETH is false.*

Proof. Suppose by contradiction that an algorithm A exists to list all the maximal solutions in an independence system, which is a particular case of strongly accessible set systems, in $O(\alpha 2^{\frac{q}{k}})$ time for some $k > 2$. This implies the existence of an algorithm B for SAT that runs in $O(n 2^{\frac{2n}{k}})$ time, where n is the number of variables: Indeed, using the reduction in Lawler et al. [32], B first transforms a formula of SAT in a hereditary (thus also strongly accessible) set system, which has exactly n maximal solutions plus one maximal solution for each satisfying assignment of the input formula (i.e., exactly n maximal solutions iff the formula is not satisfiable). Note that the largest maximal solution in this set system has size $q = 2n$. At this point, B can execute A waiting $O(n\, 2^{\frac{2n}{k}})$ time: the formula is not satisfiable iff A finds n solutions and terminates within this upper bound. However, SETH implies that $k \leq 2$, contradicting the hypothesis that $k > 2$. ∎

This completes the proof of Theorem 1.4.

2.4. Generating children in subexponential time. As the exponential time complexity discussed in Section 2.3 holds for the general case, we will now proceed in the same direction as Lawler et al. [32] and Cohen et al. [14], studying which conditions permit to achieve subexponential time per solution. Thus we introduce additional techniques to link the running time of the algorithm to the *input-restricted problem*. This will result in polynomial time per solution algorithms for some classes of enumeration problems.

From now on, we will require a prefix-closed order to satisfy a further property, which we call *core* property. We introduce the shorthand $T^s[j]$ to indicate the first j elements of T according to the order $\Pi(T, s)$. We will use $T^s[j]$ interchangeably as a sequence or the set of elements it contains.

DEFINITION 2.8 (Core property). *A family of prefix-closed orders Π satisfies the* core property *if, given a maximal solution S, its seed $s = \text{SEED}(S)$, and a solution T with $\text{CORE}(S) \cup \{\text{PI}(S)\} \subseteq T$ (which implies $s \in T$), then $T^s[|\text{CORE}(S)| + 1] = \text{CORE}(S) \cup \text{PI}(S)$*

In other words, $\text{CORE}(S) \cup \{\text{PI}(S)\}$ is a prefix of T according to $\Pi(T, s)$. While possibly not intuitive, this property will be crucial to the correctness of our algorithm, as it will guarantee that we can find all children solutions from the solution of the *input-restricted problem* [32, 14] (see Definition 1.1).

We now show how to use Definitions 2.8 and 1.1. Consider a maximal solution S, with $P = \text{PARENT}(S)$ and $u = \text{PI}(S)$. Among the solutions of the input-restricted problem $\text{RESTR}(P, u)$, surely there is a solution R such that $\text{CORE}(S) \cup \{u\} \subseteq R$, since

[6]Note that maximal solutions may also trivially be generated in $O(n^q)$ total time by trying all subsets of size at most q, but this does not guarantee output sensitivity since, e.g., when $q = \Theta(\log n)$ with a constant number of solutions.

[7]The hypothesis essentially states that there is no algorithm for SAT which runs in $O(2^{cn})$ time for any $c < 1$, where n is the number of variables.

$\textsc{core}(S) \cup \{u\} \in \mathcal{F}$ and $\textsc{core}(S) \cup \{u\} \subseteq P \cup \{u\}$. We can use the *core* property to find $\textsc{core}(S) \cup \{u\}$ given R, since it will be a prefix of R according to the prefix-closed order $\Pi(R, \textsc{seed}(S))$.

Since $S = \textsc{complete}(\textsc{core}(S) \cup \{u\})$, this yields an algorithm for finding the set $\textsc{children}(P)$ in time proportional to that required to solve the input-restricted problem, times polynomial factors in $|\mathcal{U}|$. The only missing piece is defining a suitable prefix-closed order that satisfies the core property.

2.5. Layer order in commutable systems. For the improved version of our framework, we restrict our attention to *commutable* set systems, i.e., strongly accessible set systems which satisfy the commutable property (Definition 1.2). Recall that in a commutable set system $X \cup \{a\} \in \mathcal{F}$, $X \cup \{b\} \in \mathcal{F}$ and $X \cup \{a, b\} \subseteq Y \in \mathcal{F}$ imply $X \cup \{a, b\} \in \mathcal{F}$ for any $X, Y \in \mathcal{F}$ and $a, b \in \mathcal{U}$. We introduce the notion of layer.

Definition 2.9 (layer). *Given $X \in \mathcal{F}$ and a* starting element $v \in X \cap Z$, *define inductively B_i:*[8]

- $B_0 = \{v\}$
- $B_i = B_{i-1} \cup (B_{i-1}^+ \cap X)$, *for $i > 0$*

Then, for any $y \in X \cup X^+$, its layer *is defined as $\textsc{lay}_X^v(y) = 0$ if $y = v$, and $\textsc{lay}_X^v(y) = \min\{i : y \in B_{i-1}^+\}$ otherwise.*

Again, we omit the superscript v when $v = \textsc{seed}(X)$, and X when it is clear from the context, resulting in simplified notation such as $\textsc{lay}_X(y)$ or just $\textsc{lay}(y)$. We can now define our prefix-closed order $\mathring{\Pi}$ for commutable set systems, or equivalently its corresponding binary relation $\mathring{\prec}$.

Definition 2.10 (prefix-closed order for commutable systems). *Given a solution S, and elements $t \in S \cap Z$ and $a, b \in S \cup S^+$, we say that $a \mathring{\prec}_S^t b$ iff $\langle \textsc{lay}_S^t(a), a\rangle < \langle \textsc{lay}_S^t(b), b\rangle$, where elements a and b in the second components of the pairs are compared according to the natural order.*

We remark that this induces a permutation of the elements of $S \cup S^+$ which we denote as $\mathring{\Pi}(S, t)$. We have now to prove that $\mathring{\Pi}$ satisfies the required properties of a prefix-closed order family, as well as the core property.

Lemma 2.11. *$\mathring{\Pi}$ is a family of prefix-closed orders (Definition 2.1).*

Proof. Firstly, note that $\mathring{\prec}_S^t$ is indeed an ordering on the elements of $S \cup S^+$, for $S \in \mathcal{F}$ and $t \in S \cap Z$, as is required by the definition. The *first* property is satisfied as t is the only element with layer 0.

Moreover, let $s_1, \ldots, s_{|S|}$ be the order of the elements of S according to $\mathring{\prec}_S^t$. It follows from Definition 2.9 that for any x in $S[j]^+$, $\textsc{lay}_{S[j]}(x) = \textsc{lay}_S(x)$. By definition of $\textsc{complete}()$, s_{j+1} is the element which minimizes $\langle \textsc{lay}_S(s_{j+1}), s_{j+1}\rangle$ in $S[j]^+ \cap S$, so it also minimizes $\langle \textsc{lay}_{S[j]}(s_{j+1}), s_{j+1}\rangle$. Thus we have that the *greedy* property is satisfied. Finally the *prefix* property is satisfied by induction: $S[1] \in \mathcal{F}$ since $s_1 = t \in Z$, and $S[j] \in \mathcal{F}$ implies $S[j+1] \in \mathcal{F}$ since $s_{j+1} \in S[j]^+$. ∎

Lemma 2.12. *$\mathring{\Pi}$ satisfies the* core *property (Definition 2.8).*

Proof. We will prove this by contradiction. Let T be a solution which contains $S' = \textsc{core}(S) \cup \{\textsc{pi}(S)\}$, and let $s = \textsc{seed}(S)$. The core property implies $T^s[|S'|] = S'$ (recalling that $T^s[|S'|]$ corresponds to the first $|S'|$ elements in $\mathring{\Pi}(T, s)$). Thus suppose by contradiction $T^s[|S'|] \neq S'$. Let $T^s[i]$ be the smallest prefix of T which is not a prefix of S' (note that $i < |S'|$), and let $s_1, \ldots, s_{|S'|}$ be the order of the elements of

[8]Note that B_i is made only of elements from X whereas B_i^+ is made of elements from $X \cup X^+$.

Algorithm 1: Listing Maximal Solutions in Commutable Set Systems

```
Input  : Commutable set system $(\mathcal{U}, \mathcal{F})$
Output: All maximal $X \in \mathcal{F}$

foreach $S$ such that COMPLETE(SEED($S$)) $= S$ do
    SPAWN($S$)

Function CHILDREN($P, w$)
    foreach $R \in$ RESTR($P, w$) do
        foreach $s \in (R \cap Z) \setminus \{w\}$ do
            $prefix \leftarrow \{x \in R : x \preceq^s_R w\}$
            $S \leftarrow$ COMPLETE($prefix$)
            if $\langle$PARENT($S$), PI($S$), R($S$), SEED($S$)$\rangle =$
            $\langle P, w, R, s\rangle$ then yield $S$

Function SPAWN($X$)
    /* Output $X$ if depth is odd */
    foreach $w \in \mathcal{U}$ do
        foreach $S \in$ CHILDREN($X, w$)
        do
            SPAWN($S$)
    /* Output $X$ if depth is even */

Function COMPLETE($X$)
    while $X^+ \neq \emptyset$ do
        $x \leftarrow \operatorname{argmin}_{y \in X^+} \langle$LAY$_X(y), y\rangle$
        $X \leftarrow X \cup \{x\}$
    return $X$
```

S' as they appear in $\mathring{\Pi}(S', s)$, and $t_1, \ldots, t_{|T|}$ the order of the elements of T as they appear in $\mathring{\Pi}(T, s)$. Observe that $s_1 = t_1 = s = \textsc{seed}(S)$.

By the *greedy* property, and as $T^s[i-1] = S'[i-1]$, we have $t_i = \min(T^s[i-1]^+ \cap T) = \min(S'[i-1]^+ \cap T)$. However, since $s_i \in S' \subseteq T$ we have $s_i \in S'[i-1]^+ \cap T$, and since $s_i \neq t_i$, this implies $t_i \mathring{\prec}_{T^s[i-1]} s_i$. As $\textsc{lay}^s_{T[i-1]}(t_i) = \textsc{lay}^s_T(t_i)$ and $\textsc{lay}^s_{T[i-1]}(s_i) = \textsc{lay}^s_T(s_i)$, it follows that $t_i \mathring{\prec}^s_T s_i$, meaning that t_i immediately follows s_{i-1} in $\mathring{\Pi}(T, s)$ because $t_i = \min(T^s[i-1]^+ \cap T)$ (by the *greedy* property). In turn, since $\textsc{pi}(S) \notin S[i-1]$ this means that $t_i \mathring{\prec}^s_T \textsc{pi}(S)$.

We also have that $S' \cup \{t_i\} \in \mathcal{F}$: indeed $T^s[i-1] = S[i-1]$ is a subset of both $T^s[i]$ and S' so we can see that both t_i and s_i belong to $S[i-1]^+$, and as $(S[i-1] \cup \{s_i\} \cup \{t_i\}) \subseteq S' \cup \{t_i\} \subseteq T \in \mathcal{F}$, the commutable property gives us $(S[i-1] \cup \{s_i\} \cup \{t_i\}) \in \mathcal{F}$. By induction on $j = i \ldots |S'|$, again using the commutable property, we can see that $S'[j] \cup \{t_i\} \in \mathcal{F}$. Thus, $t_i \in S'^+$.

By definition of layers, the layer of t_i in $\textsc{core}(S)$ is the same as the layer of t_i in T (as elements with lower layer than t_i in T all belong to S'). On the other hand, the layer of $\textsc{pi}(S)$ in S' cannot be smaller than its layer in T (as $S' \subset T$). Since $t_i \mathring{\prec}^s_T \textsc{pi}(S)$, it follows that $t_i \mathring{\prec}^s_{S'} \textsc{pi}(S)$.

Since $\textsc{complete}(S') = \textsc{complete}(\textsc{core}(S) \cup \{\textsc{pi}(S)\}) = S$, the first node selected by the $\textsc{complete}(S')$ is the minimum element in S'^+ according to $\mathring{\prec}^s_{S'}$, and it will be an element of S since $S = \textsc{complete}(S')$. By the greedy property this is $s_{|S'|+1}$, i.e., the element of S following $s_{|S'|}$ in the solution order of S. We also recall that $s_{|S'|} = \textsc{pi}(S)$ as $S' = \textsc{core}(S) \cup \textsc{pi}(S)$, thus $s_{|S'|+1}$ (s_x for short hereafter) is the element following the parent index in the solution order of S, meaning that $\textsc{pi}(S) \mathring{\prec}_{S' \cup \{s_x\}} s_x$.

By the definition of $\textsc{complete}$, it must be that $s_x \mathring{\preceq}_{S'} t_i$ since $t_i \in S'^+$, but since $t_i \mathring{\prec}_{S'} \textsc{pi}(S)$ then $s_x \mathring{\prec}_{S'} \textsc{pi}(S)$. We also know that $\textsc{pi}(S) \mathring{\prec}_{S' \cup \{s_x\}} s_x$ by the definition of solution order. As adding s_x to S' cannot change either its layer or that of $\textsc{pi}(S)$, this also means $\textsc{pi}(S) \mathring{\prec}_{S'} s_x$, which contradicts $s_x \mathring{\prec}_{S'} \textsc{pi}(S)$. ∎

2.6. Listing maximal solutions in commutable set systems. We are finally ready to discuss the resulting algorithm for efficiently listing maximal solutions in commutable set systems.

The pseudocode is shown in Algorithm 1, where $\textsc{restr}(P, w)$ denotes the set of solutions of the input-restricted problem on P and w (see Definition 1.1), and $\textsc{r}(S)$

must return an arbitrary but deterministic solution of said input-restricted problem which contains $\text{CORE}(S) \cup \{w\}$, given S, P, and w. Here we define $\text{R}(S)$ as the solution obtained by computing a variant of $\text{COMPLETE}(\text{CORE}(S) \cup \{w\})$ which is only allowed to add nodes in $P \cup \{w\}$. Note that a suitable $\text{R}(S)$ always exists when $w = \text{PI}(S)$, since $\text{CORE}(S) \cup \text{PI}(S) \in \mathcal{F}$.

As shown in Section 2.7, the algorithm first identifies the *roots* of the arborescence forest structure defined among solutions, corresponding to all solutions S such that $\text{COMPLETE}(\text{SEED}(S)) = S$. For each of these, the algorithm traverses the corresponding arborescence with the $\text{SPAWN}()$ function. As we will show, $\text{CHILDREN}(P, w)$ identifies all solutions S such that $\text{PARENT}(S) = P$ and $\text{PI}(S) = w$, by iterating over the choices of $R \in \text{RESTR}(P, w)$ and $s \in (R \cap Z)$. It trivially follows that $\text{SPAWN}(X)$ finds all children of X, and thus this recursive traversal starting on all roots of the arborescence forest will output all maximal solutions.

Let us prove the correctness of Algorithm 1.

LEMMA 2.13. *Function* $\text{CHILDREN}(P, w)$ *in Algorithm 1 yields the maximal solutions, without duplication, in* $\{S : \text{PARENT}(S) = P \text{ and } \text{PI}(S) = w\}$.

Proof. Let S be any maximal solution with $\text{PARENT}(S) = P$ and $\text{PI}(S) = w$. We show that S is yielded. Consider the iteration of the outer *foreach* in which $R = \text{R}(S)$: since $\text{CORE}(S) \cup \{\text{PI}(S)\} \in \mathcal{F}$, there must exist a (maximal) solution of $\text{RESTR}(P, w)$ which contains $\text{CORE}(S) \cup \{\text{PI}(S)\}$, thus $\text{R}(S) \in \text{RESTR}(P, w)$, and this iteration is executed. Furthermore, consider the iteration of the inner *foreach* in which $s = \text{SEED}(S)$, which is also executed since $\text{SEED}(S) \in (\text{CORE}(S) \cap Z) \subseteq (\text{R}(S) \cap Z)$, and $\text{SEED}(S) \neq \text{PI}(S)$ by definition of root, as S is not a root.

When these conditions are met, by the fact that the layer-based order $\overset{\circ}{\prec}$ satisfies the *core* property (Lemma 2.12) we have that $R^s[|\text{CORE}(S)| + 1] = \text{CORE}(S) \cup \text{PI}(S)$. Moreover, as this means $R^s[|\text{CORE}(S)|+1] = S[|\text{CORE}(S)|+1]$, we have that $\text{PI}(S) = w$ will be the last element of this prefix. We thus have that *prefix* $= \{x \in R : x \overset{\circ}{\preceq}{}^s_R\, w\} = \text{CORE}(S) \cup \{w\}$, i.e., the elements in R that come earlier than w in $\overset{\circ}{\Pi}(R, s)$ are all and only those in $\text{CORE}(S)$.

By Definition 2.2, it follows that $\text{COMPLETE}(\textit{prefix}) = \text{COMPLETE}(\text{CORE}(S) \cup \text{PI}(S)) = S$. At this point we have by assumption $\text{PARENT}(S) = P$, $\text{PI}(S) = w$, $\text{R}(S) = R$ and $\text{SEED}(S) = s$, thus the condition of the *if* is satisfied meaning that S is indeed yielded by the function.

Furthermore, on any other iteration of the outer/inner loops, we must have that either $R \neq \text{R}(S)$ or $s \neq \text{SEED}(S)$, meaning that if S is computed again the *if* check would fail, thus S is only yielded once. Finally, the *if* check also fails whenever $\text{PARENT}(S) \neq S$ or $\text{PI}(S) \neq w$, thus the solutions yielded will be all and only those claimed in the statement, which completes the proof. ∎

LEMMA 2.14. *Given a commutable set system* $(\mathcal{U}, \mathcal{F})$*, Algorithm 1 outputs all the maximal solutions in* $\mathcal{F}$ *without duplication.*

Proof. By Theorem 2.5, Algorithm 1 correctly identifies all roots of the arborescence forest structure among solutions, and recurs on them using $\text{SPAWN}()$.

Furthermore, Lemma 2.13 implies $\bigcup_{w \in \mathcal{U}} \text{CHILDREN}(P, w) = \text{CHILDREN}(P)$, thus the algorithm will recursively visit all children, and thus all descendants of the solutions found in the arborescence forest structure. In other words, Algorithm 1 will find all solutions.

Finally, still by Lemma 2.13, $\text{CHILDREN}(P, w)$ is free from duplication, and as each solution S has only one parent index it may appear in only one $\text{CHILDREN}(P, w)$

(i.e., when $w = \textsc{pi}(S)$). Thus each solution that is a root is found exactly once in the beginning of the algorithm, and each solution that is not a root is found exactly once when calling the function $\textsc{spawn}()$ on its parent. ∎

2.7. Running time. We here give complexity bounds for the running time of our algorithm, with the goal of showing that the algorithm runs with polynomial delay when $\mathcal{M}_T$ and $\mathcal{R}_T$ are polynomial (the space analysis will be completed in the next section). As previously mentioned in Section 1.1, $\mathcal{M}_T$ and $\mathcal{M}_S$ refer to the time and space required for checking membership in $\mathcal{F}$, $\mathcal{R}_T$ and $\mathcal{R}_S$ to the time and space required for computing all solutions of an input-restricted problem, and $\mathcal{R}_N$ to the maximum number of its solutions, noting that $\mathcal{R}_N \leq \mathcal{R}_T$ as we need at least $O(1)$ time to return each solution.

LEMMA 2.15. *$\textsc{complete}(S)$ takes $O(q^2|\mathcal{U}|\mathcal{M}_T)$ time and $O(|\mathcal{U}| + \mathcal{M}_S)$ space.*

Proof. Given a solution X, we can compute X^+ in $O(|\mathcal{U}|\mathcal{M}_T)$ time by using the definition of X^+. Thus, by computing $S[i]^+$ for any $i = 1 \dots |S|$, we can compute the levels of all the elements in S^+ in $O(|S| \cdot |\mathcal{U}|\mathcal{M}_T) = O(q|\mathcal{U}|\mathcal{M}_T)$ time (note that $S[1]$ is always known, and by the *greedy* property we can identify s_{i+1}, and thus $S[i+1]$ from $\mathring{\Pi}(S[i], s_1)$).

As we have all the levels of S^+, determining the next element to add to S is trivial. Moreover, if $\textsc{seed}(S)$ remains unchanged, we may again update the levels and compute the next S^+ in $O(|\mathcal{U}|\mathcal{M}_T)$ time. If, on the other hand, $\textsc{seed}(S)$ is changed by selecting an element smaller than the current $\textsc{seed}$ in natural order, we need to compute again the layers of all S^+ from scratch, starting from $S[1]$, in $O(q|\mathcal{U}|\mathcal{M}_T)$ time.

Since the total number of elements added to S is at most q and thus there are at most q seed changes, we get the claimed running time as a worst case bound. As for the space, we only need to store the element of S and S^+ at any time, plus $\mathcal{M}_S$ for checking membership in $\mathcal{F}$, thus the cost follows. ∎

LEMMA 2.16. *Function $\textsc{children}(P, w)$ in Algorithm 1 takes $O(q^3|\mathcal{U}|\mathcal{R}_N\mathcal{M}_T + \mathcal{R}_T)$ time and $O(|\mathcal{U}| + \mathcal{R}_S + \mathcal{M}_S)$ space.*

Proof. Firstly, the function takes time $\mathcal{R}_T$ to compute $\textsc{restr}(P, w)$, and generates $\mathcal{R}_N$ iterations of the outer for loop, and for each the inner for loop generates at most $q-1$ iterations.

In every iteration of the inner loop, computing *prefix* requires computing the solution order of R, which takes the time of a $\textsc{complete}()$ call, i.e., $O(q|\mathcal{U}|\mathcal{M}_T)$ by Lemma 2.15. The same is true for the following line of the pseudo code. Finally, we need to compute $\textsc{parent}(S)$, $\textsc{pi}(S)$, $\textsc{r}(S)$ and $\textsc{seed}(S)$, all of which also take the time needed for a $\textsc{complete}()$ call.

The total time taken by $\textsc{children}(P, w)$ will thus be $O(q^2|\mathcal{U}|\mathcal{R}_N\mathcal{M}_T + \mathcal{R}_T)$.

The space used is that required by $\textsc{restr}(P, w)$, $\textsc{complete}()$ and checking membership to $\mathcal{F}$, i.e., $O(|\mathcal{U}| + \mathcal{R}_S + \mathcal{M}_S)$. ∎

We remark that for every solution we run $\textsc{children}()$ a total of $|\mathcal{U}|$ times, so this will be the cost per solution of the algorithm. By using *alternative output* [51], i.e., output solutions at the beginning or end of a recursive call depending on the parity of the recursion depth, the delay becomes the same as the cost per solution.[9]

[9]Other strategies exist to suitably bound the delay of output-sensitive algorithms. E.g., [24] remarks that any *cumulative polynomial delay* algorithm (meaning the amortized cost per solution found is polynomial at any time during the execution) can be turned into a polynomial delay one.

We also should take into account the time required to find the root solutions: By Lemma 2.4, it is sufficient to identify all $x \in \mathcal{U}$ such that $x = \textsc{seed}(\textsc{complete}(\{x\}))$, which can be done in $O(|\mathcal{U}| \cdot q^2|\mathcal{U}|\mathcal{M}_T)$ time using Lemma 2.15. This is negligible as it is dominated by the delay of the algorithm. Furthermore, it requires just $O(|\mathcal{U}|)$ space if we simply store the set of all suitable x, recomputing $\textsc{complete}(\{x\})$ when $\textsc{stateless-spawn}(\textsc{complete}(\{x\})$ is executed (a cost that is already accounted for by the complexity analysis). We thus obtain the main complexity result.

Theorem 2.17. *Algorithm 1 lists all maximal solutions of a commutable set system with $O(q^4|\mathcal{U}|^2\mathcal{M}_T\mathcal{R}_N + \mathcal{R}_T|\mathcal{U}|)$ delay.*

We thus have that the algorithm has polynomial delay whenever $\mathcal{M}_T$ and $\mathcal{R}_T$ are polynomial (note that $\mathcal{R}_N \leq \mathcal{R}_T$).

Furthermore, it follows from Cohen et al. [14] that if R and R' are two maximal solutions of $\textsc{restr}(S, x)$ (for some maximal solution S and some $x \in \mathcal{U} \setminus S$), then $\textsc{complete}(R) \neq \textsc{complete}(R')$. The number of solutions of $\textsc{restr}(S, x)$ can thus be at most α, that of the general problem. This implies that a polynomial total time algorithm for the input-restricted problem also yields a polynomial total time algorithm for the general problem.

We also remark that problem-specific fine tuning is likely to yield better running times than the bound given by Theorem 2.17 (by, e.g., lowering the cost of $\textsc{complete}()$, or proving that only a limited set of elements may be the parent index of a child solution). An example of this is the example problem studied in Section 3.

Finally, in the following section, we make some modifications to the algorithm in order to guarantee that the space usage is polynomial.

2.8. Polynomial space via stateless visit. The recursive version of our algorithms do not yet guarantee polynomial space: indeed, there is no guarantee that the depth of the recursion will be polynomial, and a recursive implementation needs at least $O(1)$ memory for each nesting level. Moreover, Algorithm 1 stores the solutions of the input-restricted problem, which may be non-polynomial in number.

We address the first of these problems by removing the explicit recursion. The state of the computation inside a certain recursive call is fully determined by the variables P, w, R, s in Algorithm 1. Moreover, when a recursive call is made, the conditions written in the code imply that we can easily (and cheaply) compute the state variables using only information about the child. It is thus easy to modify these two algorithms to simulate the recursion avoiding an explicit stack.

With regard to the input-restricted problem, note that we can iterate over the solutions of the input-restricted problem using $\mathcal{R}_T$ time and $\mathcal{R}_S$ space: we can restart the iteration whenever we backtrack in the (simulated) recursion tree, as this does not impact the delay.

As the only other space requirements are $O(|\mathcal{U}| + \mathcal{M}_S)$ due to $\textsc{complete}$, and $O(|\mathcal{U}|)$ to store all $x \in \mathcal{U}$ such that $x = \textsc{seed}(\textsc{complete}(\{x\}))$ (which correspond to the root solutions, as remarked before), we can state the following bounds.

Theorem 2.18. *Algorithm 2, the stateless version of Algorithm 1, achieves the same delay $O(q^4|\mathcal{U}|^2\mathcal{M}_T\mathcal{R}_N + \mathcal{R}_T|\mathcal{U}|)$ and takes $O(|\mathcal{U}| + \mathcal{M}_S + \mathcal{R}_S)$ space.*

Our main result, Theorem 1.3, as well as Theorem 1.4, are the union of the statements of the theorems proved so far in Sections 2.8-2.2.

3. Maximal Common Connected Induced Subgraphs (MCCIS). The following problem is an example showing how to obtain an output-sensitive and poly-

Algorithm 2: Stateless framework with minimal memory

```
foreach $S$ such that COMPLETE(SEED($S$)) $= S$ do
    STATELESS-SPAWN($S$)
Function STATELESS-SPAWN($X$)
    $P \leftarrow X$
    $S \leftarrow$ null
    $w \leftarrow$ NEXT-NODE(null)
    $R \leftarrow$ NEXT-R($P, w,$ null)
    do
        do
            do
                if $S \leftarrow$ NEXT-CHILD($P, w, R, S$) $\neq$ null then
                    $\langle P, S, w, R \rangle \leftarrow \langle S,$ null, null, null$\rangle$ /* recur in child */
                    break
            while $R \leftarrow$ NEXT-R($P, w, R$) $\neq$ null
        while $w \leftarrow$ NEXT-NODE($w$) $\neq$ null
        if IS-ROOT($P$) then return
        else $\langle P, S, w, R \rangle \leftarrow \langle$PARENT($P$)$, P,$ PI($P$), R($P$)$\rangle$ /* backtrack */
    while true
Function IS-ROOT($X$)
    return COMPLETE(SEED($X$)) $= X$
Function NEXT-NODE($w$)
    return $\min\{v \in \mathcal{U} : v > w\}$
Function NEXT-R($P, w, R$)
    return the solution succeeding $R$ in RESTR($P, w$) (or null if $R$ is the last)
Function NEXT-CHILD($P, w, R, S$)
    foreach $y \in R : y >$ SEED($S$) do
        prefix $\leftarrow \{x \in R : x \preceq_R^y w\}$
        $D \leftarrow$ COMPLETE(prefix)
        if $\langle$PARENT($D$), PI($D$), R($D$), SEED($D$)$\rangle = \langle P, w, R, x \rangle$ then
            return $D$
    return null
```

nomial space algorithm, using our framework with some extra optimization that is problem-dependent.[10] For any two given input graphs H and F, a subgraph S of H is *in common* with F if S is isomorphic to a subgraph of F: it is maximal if there is no other common subgraph that strictly contains it, and maximum if it is the largest. The *maximum* common subgraph problem asks for the maximum ones, or simply for their size.[11] The *maximal* common subgraph (MCS) problem further requires discovering all the MCS's of H and F. The MCS problem can be constrained to *connected* and *induced* subgraphs (MCCIS) [11, 10, 28, 29], where the latter means that all the edges of H between nodes in the MCS are mapped to edges of F, and vice versa. The connectivity constraint is important to remove redundant solutions corresponding to

[10] We presented the ideas described in this section in the conference [18].

[11] As it is clear, maximal and maximum subgraphs are inherently different problems: listing *all* maximal ones can potentially find an exponential number of solutions, while finding the maximum connected ones corresponds to just the *single* largest one, and is in practice much faster (e.g. [45]). As pointed out in Koch et al. [28, 29], however, a maximum common subgraph does *not* always contain all the relevant/large common structures, which motivates the MCCIS problem.

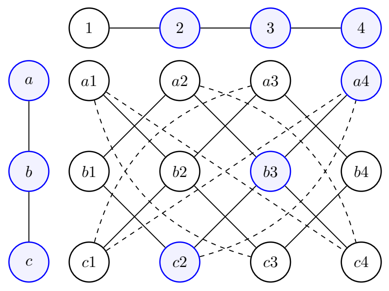

FIG. 2. *An example of* MCCIS *($\{a, b, c\}$ to $\{4, 3, 2\}$, in this order), with the corresponding* BC-*clique $\{a4, b3, c2\}$. White edges are represented as dashed lines.*

compatible combinations of disjoint connected subgraphs [28, 29], and the induced constraint is used to reduce the search space while still preserving the significance of the result [10, 11].

The problem of listing common subgraphs is the following: given any two graphs H and F, list all (isomorphisms corresponding to) maximal common connected induced subgraphs (MCCISs) between H and F in polynomial time per solution and total polynomial space. It has been introduced and investigated in the practical setting of proteins [29], e.g. for protein function annotation [40], and can be employed to mine significant information in many domains, for example identifying compound similarity and structural relationships among biological molecules [20].

This problem is at least as hard as the graph isomorphism problem [6]. For this reason a weaker form is considered, where the individual isomorphisms are listed (noting that several isomorphisms can correspond to the same MCCIS). Surprisingly, no output-sensitive listing algorithm with polynomial space is known for even this version, while several papers [10, 28, 29] adapt existing techniques for maximal clique enumeration without any guarantee.

3.1. Converting the MCCIS problem to a maximal clique problem. Clique-based methods are widely employed to transform common subgraphs of H and F into maximal cliques in a compatibility graph [34]. As in [28], we define the *product graph* G between H and F as follows. (i) Any pair of nodes $(x, i) \in H \times F$ is a node of G iff they have the same label; (ii) there is a *black edge* between (x, i) and (y, j) iff $(x, y) \in E(H)$ and $(i, j) \in E(F)$; (iii) there is a *white edge* between (x, i) and (y, j) iff $x \neq y$, $i \neq j$, $(x, y) \notin E(H)$ and $(i, j) \notin E(F)$, where $E(\cdot)$ is the edge set.

The key property is that MCCISs between H and F correspond to maximal cliques in G spanned by black edges [28], which we will call BC-*cliques*. An example is shown in Fig. 2. Let G_B be the edge subgraph of G containing only black edges. For each isomorphism corresponding to MCCISs of H and F, there is a maximal clique in G connected by black edges in G_B (i.e. a BC-clique), and vice versa. Hence, in general, given a graph G whose edges are either black or white, we will show how to list all the maximal BC-cliques in G in an output-sensitive fashion without storing all the solutions (see Figure 3).

3.2. Obstacles with BC-cliques. A number of obstacles appear along the road to list BC-cliques. Cao et al. [11] observe that materializing the product graph G can be expensive memory-wise, so we do *not* want to materialize G: we navigate the huge solution space of the BC-cliques by navigating G implicitly using H and F, just requiring $O(q)$ additional space. However, directly adapting the state-of-the-art

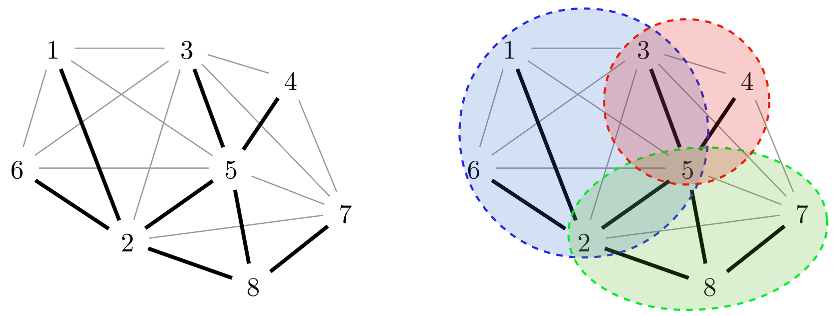

FIG. 3. *A graph with black and white edges, and its* BC*-cliques*

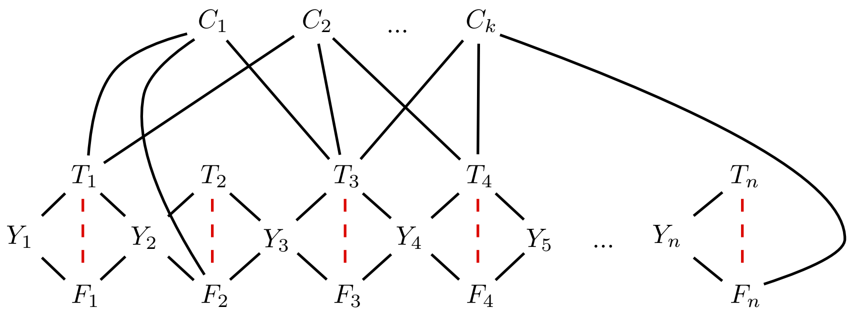

FIG. 4. *A graph with black and white edges. The dashed red edges between T_1-F_1, T_2-F_2 and so on symbolize that there is no white edge among those nodes, all other pairs of nodes (except the ones already connected by a black edge) are connected by a white edge. Computing the lexicographically minimum* BC*-clique containing $X = \{Y_1\}$ is* NP*-hard.*

approach [14] for low-memory enumeration of maximal sets satisfying hereditary properties also requires us to have a COMPLETE function returning the lexicographically smallest solution containing a given BC-clique, which we show below to be NP-hard. We are able to circumvent this problem by using our algorithmic framework, in which the COMPLETE function does not require the lexicographically smallest solution.

LEMMA 3.1. *Given a graph G whose edges are either black or white and any* BC*-clique X of G, it is* NP*-hard to find the lexicographically minimum among the maximal* BC*-cliques containing X.*

Proof. We prove that returning the lexicographically minimum BC-clique containing X can be used to solve SAT problems in polynomial time, by building a graph with nodes linear in the amount of the clauses and variables in the formula.

Given a SAT formula with n variables $x_1 \dots x_n$ and k clauses $d_1 \dots d_k$, we build the graph in Figure 4, whose nodes are $C_1 \dots C_k$, $T_1 \dots T_n$, $F_1 \dots F_n$ and $Y_1 \dots Y_n$, labelled increasingly in this order (i.e., $C_1 < C_2 < \dots < C_k$, with all other nodes having larger label than C_k). Each Y_i is connected with a *black* edge to T_i and F_i, and also with T_{i-1} and F_{i-1} (except for Y_1). The nodes T_i and F_i correspond to the "literal gadgets", with T_i corresponding to the literal x_i and F_i to $\neg x_i$, respectively.

The nodes of the form C_i correspond instead to the "clause gadgets", with each C_i representing the clause d_i. In particular, C_i is connected with a *black* edge to T_j iff d_i contains x_j, and to F_j iff d_i contains $\neg x_j$. Hence, nodes in $C_1 \dots C_k$ are connected with black edge to an arbitrary amount of T_i and F_i nodes, but not to any Y_i node.

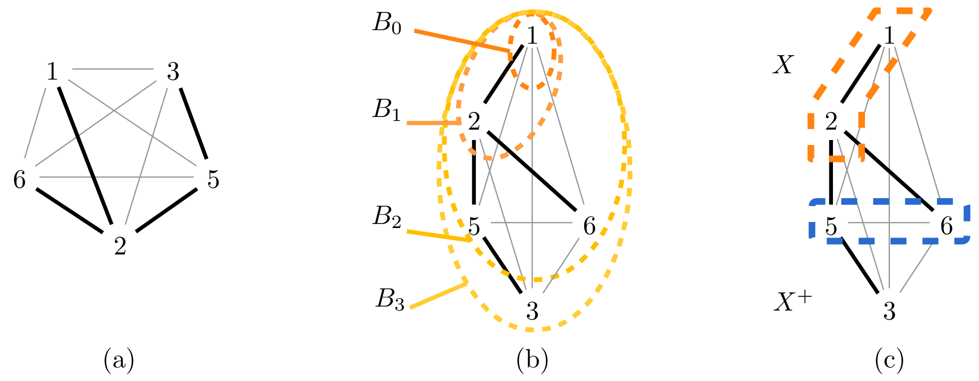

FIG. 5. *(a) A maximal* BC*-clique* S_1 *(from the graph in Figure 3) and (b) its solution order* $1, 2, 5, 6, 3$ *(from Section 2.5), where* B_i *denotes the layer* i*. (c) A non-maximal* BC*-clique* $X \subset S_1$ *and its set* X^+*.*

All pairs of nodes which are not already connected with a black edge, are connected with a *white* edge, except for the pairs (T_i,F_i) (illustrated by the dashed red edge in Figure 4). It is straightforward to see that any maximal BC-clique in this graph will contain exactly one between any T_i,F_i, and that any maximal BC-clique containing *all* nodes $C_1 \dots C_k$ will be lexicographically smaller than any other one that does not contain all of them (as these have the smallest labels).

Consider $X = \{Y_1\}$, and observe that any BC-clique containing Y_1 and all C_i nodes represents a satisfying assignment for the formula at hand: Indeed, in order for each C_i node to be reachable from Y_1 with black edges, at least one of the T_j or F_j nodes connected to C_i must be in the BC-clique. As the nodes connected to C_i correspond to the literal that satisfy it, and as we cannot have the pair T_j-F_j in the same BC-clique since they are not connected by any edge, the set of T_i and F_i nodes in the BC-clique will thus give us a set of literals $x_i/\neg x_i$ which satisfy the input formula. Hence, the lexicographically minimum BC-clique containing Y_1 contains all C_i nodes if and only if the input formula is satisfiable. This means that we can check satisfiability by computing the lexicographically minimum BC-clique containing $X = \{Y_1\}$, and checking whether this contains all C_i nodes. ☐

3.3. Applying the framework to BC-cliques. First, we remark that the input-restricted problem RESTR(P, w) for BC-cliques is easy. Indeed, the only solution other than P is the black connected component of $\{w\} \cup (P \cap N(w))$ containing w: any solution other than P not containing w would not be maximal, and any solution containing w is contained in the one stated above.

To ease the understanding of our approach, we will now give examples of the concepts defined in Section 2 for commutable set systems on BC-cliques. Consider the graph G in Figure 3, whose BC-cliques are $S_1 = \{1, 2, 3, 5, 6\}$, $S_2 = \{3, 4, 5\}$ and $S_3 = \{2, 5, 7, 8\}$. We focus on S_1, as shown in Figure 5, and begin with its layers. The elements $1, 2, 3, 5, 6$ have layers, respectively, $0, 1, 3, 2, 2$ (relatively to S_1 and from $t = \text{SEED}(S_1) = 1$, e.g. $\text{LAY}_{S_1}(2) = 1$ and $\text{LAY}_{S_1}(5) = 2$). A detailed view is shown in Figure 5. By sorting these elements according to the definition of $\mathring{\prec}^1_{S_1}$, we obtain the solution order $1 \mid 2 \mid 5, 6 \mid 3$, with layers in increasing order separated by "|". Also, we observe that S_1 is a root as COMPLETE applied to each prefix $S[j]$ (i.e. $S[1] = 1$; $S[2] = 1, 2; \dots ; S[5] = 1, 2, 5, 6, 3$) always gives S_1.

Consider now the other two BC-cliques S_2 and S_3. Their solution order is $3 \mid$

$5 \mid 4$ and $2 \mid 5, 8 \mid 7$. Also, S_1 is their parent: for example, $S_1 = \textsc{parent}(S_2) = \textsc{complete}(\{3, 5\})$ and thus $\textsc{core}(S_2) = \{3, 5\}$ and $\textsc{pi}(S_2) = 4$. We observe that during the execution of COMPLETE, the SEED changes (and so the solution order changes significantly): $\{3 \mid 5\} \to \{2 \mid 5 \mid 3\} \to \{1 \mid 2 \mid 5 \mid 3\} \to \{1 \mid 2 \mid 5, 6 \mid 3\}$. Also, $\textsc{core}(S_3) = \{2, 5\}$ and $\textsc{pi}(S_3) = 8$.

It follows that S_2 and S_3 are the children of S_1 (as there are no other maximal BC-cliques in the graph). Thus, S_1 is the only solution that is a root, and the arborescence forest only contains a single arborescence with three nodes and height 1.

3.4. Complexity with the implicit product graph. We give the complexity of our framework, taking into account that the product graph G is not a generic graph with white and black edges, but an implicit product graph between H and F that we do not want to materialize, whose size and features depend on H and F.

Recall that each node of G corresponds to a mapping between two nodes of H and F. For any given $v \in V(G)$, let these nodes be respectively $v_H \in V(H)$ and $v_F \in V(F)$. Also Δ_H and Δ_F are the maximum node degree in H and F, while Δ_B is the maximum degree in G_B. By construction of the product graph we have $\Delta_B \leq \Delta_H \, \Delta_F$. For brevity, we define Δ as $\Delta_H + \Delta_F$. These parameters are all significantly smaller than the size of G, which has $|V(H)| \cdot |V(F)|$ nodes, and $O(|V(H)|^2 \cdot |V(F)|^2)$ edges, either black or white.

Let X be a BC-clique in G. We denote as X_H and X_F respectively the set of nodes of H and F mapped in X. We keep a dictionary between the nodes of X and those of X_H and X_F, allowing us to retrieve v_H and v_F from v, or vice versa, in $O(1)$ time.[12]

LEMMA 3.2. *Let X be a BC-clique in G and v a node in $V(G)$. Testing whether $X \cup \{v\} \in \mathcal{F}$, e.g. it is a BC-clique, takes $O(\min(|X|, \Delta))$ time and $O(|X|)$ space.*

Proof. As X is a BC-clique in G, in order to check that $X \cup \{v\}$ is a BC-clique in G we need to check that $\{v\}$ is connected to a node in X through a black edge, and to all the others through either white or black edges. This can trivially be done in $O(|X|)$ time by checking adjacency with the nodes of X one by one.

However, a faster solution is possible if we focus on the edges that are *not* in G: for a given node $x \in X$, corresponding to a mapping between $x_H \in V(H)$ and $x_F \in V(F)$, there is *no* edge in G between v and x if either $\{v_H, x_H\} \in E(H)$ and $\{v_F, x_F\} \notin E(F)$, or $\{v_H, x_H\} \notin E(H)$ and $\{v_F, x_F\} \in E(F)$. Otherwise, there is either a black or white edge between v and x.

To check the presence of missing edges between v and nodes of X we can iterate over all $x_H \in N_H(v_H) \cap X_H$, and check that each is mapped by X in a node $x_F \in N_F(v_F) \cap X_F$. Then, similarly, iterate over all $x_F \in N_F(v_F) \cap X_F$ and check that they are mapped in some $x_H \in N_H(v_H) \cap X_F$. This can be done in $O(|N_H(v_H)| + |N_F(v_F)|) = O(\Delta)$ time. If no missing edge exists then $X \cup \{v\}$ is a clique in G. As a byproduct, this process finds all black edges between v and X, thus we may check at the same time that there is at least one, and thus that $X \cup \{v\}$ is a BC-clique. ☐

LEMMA 3.3. *For any BC-clique X in G, computing $\textsc{lay}_X(v)$ for all $v \in X$ takes $O(|X| \min(|X|, \Delta_H, \Delta_F))$ time and $O(|X|)$ space.*

Proof. The values of $\textsc{lay}_X(v)$ correspond to their distance from the $x = \textsc{seed}(X)$ in $G_B[X]$. This can be done via a BFS of $G_B[X]$ rooted at x. As $G_B[X]$ has $|X|$

[12]This data structure will be built at the beginning of a COMPLETE call. As building it takes $O(|X|)$ time and space, it will not affect the final complexity.

nodes, the trivial bound for this traversal is $|X|^2$. Once again, we can exploit the fact that G is the product graph of H and F: indeed, each node v of X corresponds to a mapping of a node v_H of H into *one* node v_F of F. For this reason, while v can have up to Δ_B neighbors in G_B, v_H may have at most $|N_H(v_H)|$ neighbors in X_H.

We can thus iterate on the black neighborhood of v in X in $O(\min(\Delta_H, \Delta_F))$ time by iterating on the neighbors of either v_H in H or v_F in F and then retrieve the corresponding nodes in X. In total, we process $|X|$ nodes, each in $O(\min(\Delta_H, \Delta_F))$ time. The cost follows. ∎

LEMMA 3.4. COMPLETE(X) *takes* $O(q(q + \Delta_B)\Delta) = O(q^2\Delta + q\Delta_B\Delta)$ *time and* $O(q)$ *space.*

Proof. In order to perform COMPLETE(X), we iterate over all nodes that can be added to X, adding the smallest, with respect to X and its seed x, first. For each node v in X (including those that are added during the procedure), we keep an iterator which will scan in increasing order its black neighbors. Clearly, each node must be considered after the smallest ones, and once it is considered it is either added to X or discarded, thus it does not need to be considered as a candidate anymore.

Given a node $c \notin X$, that has a black neighbor in X, we can see that $\text{LAY}_X(c) = \text{LAY}_X(v) + 1$, where v is the black neighbor of c in X that minimizes this value. Hence, to select the lexicographically smallest node, we must first consider the black neighbors of the nodes v that minimize $\text{LAY}_X(v)$. We thus order the nodes in a priority queue by value of $\text{LAY}_X(v)$, breaking ties by the value of the smallest black neighbor yet to consider, so that the first node in the priority queue is the smallest candidate to consider for addition to X.

As X will contain $|X| = O(q)$ nodes, and we will iterate on the $O(\Delta_B)$ black neighbors of each node exactly once, the total cost of this iteration is $O(q\Delta_B)$ time, and will yield up to $q\Delta_B$ nodes. Since by Lemma 3.2 testing a candidate takes $O(\min(q, \Delta))$ time, the total cost is $O(q\Delta_B \min(q, \Delta))$.

Furthermore, we need to account for the cost of changing SEED: after we add a node x to X, this becomes the new SEED of X if its label is smaller than that of the previous SEED. In this case, we need to update both the values of $\text{LAY}_X(v)$, and the priority queue of candidate nodes. By Lemma 3.3, this can be done in $O(q \min(q, \Delta_H, \Delta_F))$ time. We pay this cost at most q times as we add up to q nodes, for a cost of $O(q^2 \min(q, \Delta_H, \Delta_F))$ which is upper bounded by $O(q^2\Delta)$.

The total cost is thus $O(q\Delta_B \min(q, \Delta) + q^2\Delta) = O(q(q + \Delta_B)\Delta)$ time. ∎

Let CAND(P) be the set of nodes that do not belong to P, but are neighbors of some node of P in G_B, noting that these are at most $|P|\Delta_B \le q\Delta_B$. We observe that for any child S of P, we must have $\text{CORE}(S) \cup \{\text{PI}(S)\} \in \mathcal{F}$, meaning that PI$(S)$ must be neighbor in G_B of some node in P, i.e., it must be in CAND(P). In other words, when considering CHILDREN(P, w), we can restrict our attention to just $w \in \text{CAND}(P)$.

LEMMA 3.5. CHILDREN(P, w) *for all* $w \in \text{CAND}(P)$ *takes overall* $O(q^4\Delta_B\Delta + q^3\Delta_B^2\Delta) = O(q^4\Delta_H^2\Delta_F^2 + q^3\Delta_H^3\Delta_F^3) = O(q^4\Delta_H^3\Delta_F^3)$ *time and* $O(q)$ *space.*

Proof. For a given P and w, note that the corresponding input-restricted problem RESTR(P, w) only has two solutions. The cost of CHILDREN(P, w) is thus bounded by that of executing $O(q)$ times the last two lines in its pseudocode of Algorithm 1, which is bounded by the cost of a COMPLETE() call, i.e., $O(q(q + \Delta_B)\Delta)$ time. Since the number of nodes in CAND(S) is at most $q\Delta_B$, and for each we execute COMPLETE() $O(q)$ times, the total cost will be $O(q^3\Delta_B\Delta(q + \Delta_B))$, and since $\Delta_B = O(\Delta_H \Delta_F)$ and $\Delta = O(\Delta_H + \Delta_F)$, the cost is also bounded by $O(q^4\Delta_H^3 \Delta_F^3)$. ∎

GRAPH H			GRAPH F			σ	q	Koch [28]		ours		ours parallel	
n	m	Δ_H	n	m	Δ_F			TIME	#sol	TIME	#sol	TIME	#sol
200	235	5	200	234	7	5	12	28s	6691	0.2s	6691	0.04s	6691
100	122	7	100	119	5	4	22	11s	3654	0.6s	3654	0.1s	3654
2763	9488	14	2629	9059	12	12	68	2h	1998	2h	33874	2h	887293

TABLE 1
Comparison of polynomial space algorithms: running time of Koch's algorithm [28] vs ours and its parallel implementation. The first two rows are two pairs of random Erdos-Renyi graphs, and the last one a pair of graphs representing proteins from the Protein Data Bank (`1ald` and `1gox`) with a time limit of two hours. Parameters n, m, and σ are the number of nodes, edges, and node labels of the graphs, Δ_H and Δ_F their maximum degrees, and q the size of the largest found MCCIS.

Looking at Algorithm 1, we can see that the delay is bounded by the cost of the function CHILDREN(P), and the preprocessing by calling COMPLETE() for each node in G, i.e., at most $|V_H| \cdot |V_F|$ times. Furthermore, as shown in Lemmas 3.2, 3.3, and 3.4, the space required is always $O(q)$. We can thus state the main result of this section.

THEOREM 3.6. *Given two graphs H and F, each respectively of maximum degree Δ_H and Δ_F, their (isomorphisms corresponding to)* MCCIS*s can be listed in $O(q^4\Delta_H^3\Delta_F^3)$ delay using $O(q)$ space, where q is the number of nodes in the largest* MCCIS*, after a preprocessing of $O(q^2|V_H|\Delta_H^2|V_F|\Delta_F^2)$ time.*

As a final remark, the result in Theorem 3.6 can be implemented efficiently. Table 1 reports the running time of a sequential and parallel implementation[13] in `C++`, compared to the state-of-the-art algorithm by Koch [28]. Experiments were executed on a 12-core machine with two Intel Xeon E5-2620 CPUs and 128 gigabytes of RAM, with a time limit of two hours, showing that on top of giving theoretical guarantees, our algorithm is also fast in practice.

4. Other Application Examples.

This section shows some more examples of results obtained by applying our framework.

4.1. Connected k-plexes.

Berlowitz et al. [7] give an exponential space algorithm for listing maximal connected k-plexes, a popular pseudo-clique model. Given a graph $G = (V, E)$ with $|V| = n$ nodes and $|E| = m$ edges, a k-plex S is a set of nodes such that each node of S has at least $|S| - k$ neighbors in S.

The delay of the algorithm is parameterized in k (in particular is polynomial when $k = O(1)$), by essentially applying [14] and efficiently solving the input-restricted problem.

It immediately follows that we may apply our framework to the problem by plugging the subroutine for solving the input-restricted problem [7], obtaining an algorithm that uses polynomial space, and whose delay is larger than [7] by a polynomial factor, i.e., also parameterized in k. Indeed, our framework runs with space $O(|\mathcal{U}|+\mathcal{M}_S+\mathcal{R}_S)$ and delay $O(q^4|\mathcal{U}|^2\mathcal{M}_T\mathcal{R}_N+\mathcal{R}_T|\mathcal{U}|)$, that is also $O(q^4|\mathcal{U}|^2\mathcal{M}_T\mathcal{R}_T)$ by observing that $\mathcal{R}_N \leq \mathcal{R}_T$ (as at least $O(1)$ time is required to return each solution).

In the case of connected k-plexes we have $|\mathcal{U}| = n$, the maximum size of a solution q is $O(n)$, and $\mathcal{M}_T = O(n^2)$ as we can check that S is a k-plex by checking for each node in S how many nodes of S are adjacent to it. We thus obtain an algorithm with delay $O(n^8\mathcal{R}_T)$, where $\mathcal{R}_T$ is the time required by [7] to solve the input-restricted

[13]Code available at https://github.com/veluca93/parallel_enum/tree/bccliques as part of a parallel enumeration framework.

problem (this is not stated explicitly in [7], but is proven to be polynomial when $k = O(1)$ and in general parameterized in k). As for the space, $\mathcal{M}_S = O(m)$ as we only need to store the input graph, and $\mathcal{R}_S$ is surely polynomial when $\mathcal{R}_T$ is polynomial, thus for $k = O(1)$ our framework yields the first polynomial space and polynomial delay algorithm for listing maximal connected k-plexes.

Theorem 4.1. *The maximal connected k-plexes of a graph can be listed using polynomial space, in polynomial delay when $k = O(1)$, or space and delay parameterized in k, when k is unbounded.*

4.2. Connected bipartite (induced) subgraphs. Given a graph $G = (V, E)$, a subset $S \subseteq E$ of its edges is a bipartite subgraph if the edges form no cycle of odd length. Similarly, a subset $S \subseteq V$ of its nodes is an induced bipartite subgraph if the subgraph $G[S]$ induced by the nodes of S contains no cycle of odd length. A bipartite (sub)graph can be equivalently seen as a graph that is two-colorable (it is possible to assign one out of two colors to each node such that no pair of adjacent nodes has the same color).

We will first consider non-induced subgraphs. In this case, the instance of the input-restricted problem consists of a bipartite graph $S \subseteq E$ plus one extra edge e that connects two nodes x and y of the same color. Any subset of the edges in the input-restricted problem that gives a maximal solution different from S must be missing enough edges to disconnect these two nodes in the original bipartite graph, i.e. the set of removed edges must be a cut in the bipartite graph: indeed, any connection between x and y in S must have been of *even* length, thus it forms an odd cycle when adding e. Moreover, since we are interested in maximal solutions, the set of edges forming the cut must be inclusion minimal. Note that minimal x, y-cuts divide the graph into two connected components one containing x and the other y, thus adding e to the graph always yields a connected graph.

We can thus solve the input-restricted problem by listing minimal cuts, which can be done in polynomial total time as shown in Provan et al. [39]. As we remarked at the end of Section 2.7, this gives us a polynomial space and polynomial total time algorithm for maximal bipartite subgraphs.

As for induced subgraphs, we have a bipartite subgraph $S \subseteq V$, and the extra node v having some neighbors of one color and some of the other. We will call these two sets of neighbors B and W respectively. As before, we need to break the odd cycles involving v. To do so, we need to remove one or more nodes from the original bipartite graph so that there is no path from any node in B to any node in W.

This can be done by enumerating the minimal node-cuts between two dummy nodes b and w, where b is connected to all nodes in B and w to all those in W. As before, removing minimal node-cuts leaves the graph split into two connected components, and since v is the node obtained by identifying b with w, we have that the resulting graph is connected. Moreover, minimal node-cuts can be enumerated in polynomial total time, as shown in Shen et al. [43]. We thus obtain the result below.

Theorem 4.2. *The maximal connected bipartite (induced) subgraphs of a graph can be listed in polynomial total time, using polynomial space.*

4.3. Minimal feedback vertex and arc sets. A feedback vertex set of a graph $G = (V, E)$ is a set of nodes $S \subseteq V$ such that removing S from G makes the graph acyclic. A feedback arc set is a similarly defined set of edges. In this case, we can observe that the *complements* of the solutions, corresponding to maximal acyclic (induced) subgraphs, form a hereditary set system. Listing minimal feedback vertex sets and arc sets has been solved in polynomial delay by [42], albeit using exponential space.

As with bipartite subgraphs, the input-restricted problem involves a maximal solution S and an extra single node or a single arc to break all the cycles that are formed in a graph with no cycles. Thus, it may be solved by enumerating maximal cuts or node-cuts in the original graph. As seen in Section 4.2, node cuts in undirected graphs may be enumerated in polynomial total time (see [43]), and edge cuts may be enumerated in polynomial total time both in undirected and directed graphs (see [39]). The following result can be achieved using either Cohen et al. [14] or our framework as a further example of application.

THEOREM 4.3. *All the minimal feedback vertex sets in an undirected graph may be enumerated in polynomial total time, using only polynomial space. Moreover, all the minimal feedback arc sets in both directed and undirected graphs may be enumerated in polynomial total time, using only polynomial space.*

5. Conclusions.

This paper described an algorithmic framework that can be used to design time- and space-efficient listing algorithms for maximal solutions in strongly accessible set systems. Both positive and negatives results are provided in the area of enumeration: on one hand we enlarge the classes of listing problems which allow for polynomial delay and space algorithms, and on the other hand we show conditional lower bounds which prevent us from obtaining significantly lower running times on all strongly accessible set systems.

Specifically, one version of our framework has complexity $O(\alpha 2^q)$ time, where α is the number of maximal solutions and q the maximum size of one. This is, to the best of our knowledge, the first non-trivial general bound for this class of enumeration problems, that approaches our conditional lower bound of $\Omega(\alpha 2^{q/2})$ time. Furthermore, we solve the open problem left by Cohen et al. [14] by giving an improved version of our framework which links its delay to the input-restricted problem, but still uses polynomial space, for commutable set systems (which include the connected-hereditary graph properties considered in [14]). As a case study, we apply our framework to obtain the first polynomial-delay and polynomial-space algorithm for listing maximal connected subgraph isomorphisms between any two graphs, along with its implementation.

Future work is aimed at applying this framework to a variety of problems. Furthermore, an interesting point would be to achieve comparable complexity bounds in a wider class of problems, e.g., without relying on the commutable property or strong accessibility.

Acknowledgements. We thank the anonymous referees for the valuable and detailed comments that allowed us to greatly improve the structure and presentation of the paper. This work has been partially supported by JST CREST, grant number JPMJCR1401, Japan, and by MIUR, Italy.

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, IEEE, 2015.

[2] Eralp Abdurrahim Akkoyunlu. The enumeration of maximal cliques of large graphs. *SIAM Journal on Computing*, 2(1):1–6, March 1973.

[3] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1, 2008.

[4] Hiroki Arimura and Takeaki Uno. Polynomial-delay and polynomial-space algorithms for mining closed sequences, graphs, and pictures in accessible set systems. In *Proceed-*

ings of the SIAM International Conference on Data Mining, SDM 2009, April 30 - May 2, 2009, Sparks, Nevada, USA, pages 1088–1099, 2009.

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

[6] László Babai. Graph isomorphism in quasipolynomial time [extended abstract]. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 684–697. ACM, 2016.

[7] Devora Berlowitz, Sara Cohen, and Benny Kimelfeld. Efficient enumeration of maximal k-plexes. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 431–444, New York, NY, USA, 2015. ACM.

[8] Mario Boley, Tamás Horváth, Axel Poigné, and Stefan Wrobel. Listing closed sets of strongly accessible set systems with applications to data mining. *Theoretical Computer Science*, 411(3):691–700, 2010.

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

[10] Yiqun Cao, Anna Charisi, Li-Chang Cheng, Tao Jiang, and Thomas Girke. ChemMineR: a compound mining framework for R. *Bioinformatics*, 24(15):1733–1734, 2008.

[11] Yiqun Cao, Tao Jiang, and Thomas Girke. A maximum common substructure-based algorithm for searching and predicting drug-like compounds. *Bioinformatics*, 24(13):i366–i374, 2008.

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

[13] Giovanni Ciriello and Concettina Guerra. A review on models and algorithms for motif discovery in protein–protein interaction networks. *Brief Funct genomics & proteomics*, 7(2):147–156, 2008.

[14] Sara Cohen, Benny Kimelfeld, and Yehoshua Sagiv. Generating all maximal induced subgraphs for hereditary and connected-hereditary graph properties. *Journal of Computer and System Sciences*, 74(7):1147 – 1159, 2008.

[15] Sara Cohen and Yehoshua Sagiv. An abstract framework for generating maximal answers to queries. In *Proceedings of the 10th International Conference on Database Theory*, ICDT'05, pages 129–143, Berlin, Heidelberg, 2005. Springer-Verlag.

[16] Alessio Conte, Roberto Grossi, Andrea Marino, Takeaki Uno, and Luca Versari. Listing maximal independent sets with minimal space and bounded delay. In *String Processing and Information Retrieval: 24th International Symposium, SPIRE 2017*, pages 144–160, Cham, 2017. Springer International Publishing.

[17] Alessio Conte, Roberto Grossi, Andrea Marino, and Luca Versari. Sublinear-space bounded-delay enumeration for massive network analytics: Maximal cliques. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, pages 148:1–148:15, 2016.

[18] Alessio Conte, Roberto Grossi, Andrea Marino, and Luca Versari. Finding maximal common subgraphs via time-space efficient reverse search. In *Computing and Combinatorics - 24th International Conference, COCOON 2018, Qing Dao, China, July 2-4, 2018, Proceedings*, pages 328–340, 2018.

[19] Nan Du, Bin Wu, Xin Pei, Bai Wang, and Liutong Xu. Community detection in large-scale social networks. In *Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis*, pages 16–25. ACM, 2007.

[20] HC Ehrlich and M Rarey. Maximum common subgraph isomorphism algorithms and their applications in molecular science: a review. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 1(1):68–79, 2011.

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

[22] 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. [Online; accessed February 2016].

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

[24] Leslie Ann Goldberg. *Efficient algorithms for listing combinatorial structures.* PhD thesis, The University of Edinburgh, 1991.

[25] Russell Impagliazzo and Ramamohan Paturi. Complexity of k-sat. In *14th Annual IEEE Conference on Computational Complexity*, pages 237–240. IEEE, 1999.

[26] David S. Johnson, Mihalis Yannakakis, and Christos H. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27(3):119 – 123, 1988.

[27] Cecilia Klein, Andrea Marino, Marie-France Sagot, Paulo Vieira Milreu, and Matteo Brilli. Structural and dynamical analysis of biological networks. *Brief Funct Genomics*, 11(6):420–433, 2012.

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

[29] Ina Koch, Thomas Lengauer, and Egon Wanke. An algorithm for finding maximal common subtopologies in a set of protein structures. *Journal of Computational Biology*, 3(2):289–306, 1996.

[30] Bernhard Korte and László Lovász. Mathematical structures underlying greedy algorithms. In *Fundamentals of Computation Theory, FCT'81, Proceedings of the 1981 International FCT-Conference, Szeged, Hungary, August 24-28*, pages 205–209, 1981.

[31] Vincent Lacroix, Ludovic Cottret, Patricia Thébault, and Marie-France Sagot. An introduction to metabolic networks and their structural analysis. *IEEE ACM T Comput Bi*, 5(4):594–617, 2008.

[32] Eugene L. Lawler, Jan Karel Lenstra, and AHG Rinnooy Kan. Generating all maximal independent sets: NP-hardness and polynomial-time algorithms. *SIAM Journal on Computing*, 9(3):558–565, 1980.

[33] Victor E. Lee, Ning Ruan, Ruoming Jin, and Charu Aggarwal. A survey of algorithms for dense subgraph discovery. In *Managing and Mining Graph Data*, pages 303–336. Springer, 2010.

[34] Giorgio Levi. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *CALCOLO*, 9(4):341–352, 1973.

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

[36] Paulo Vieira Milreu, Cecilia Coimbra Klein, Ludovic Cottret, Vicente Acuña, Etienne Birmelé, Michele Borassi, Christophe Junot, Alberto Marchetti-Spaccamela, Andrea Marino, Leen Stougie, et al. Telling metabolic stories to explore metabolomics data: a case study on the yeast response to cadmium exposure. *Bioinformatics*, 30(1):61–70, 2014.

[37] Natwar Modani and Kuntal Dey. Large maximal cliques enumeration in sparse graphs. In *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 1377–1378. ACM, 2008.

[38] Robert J. Mokken. Cliques, clubs and clans. *Quality & Quantity*, 13(2):161–173, 1979.

[39] J. S. Provan and D. R. Shier. A paradigm for listing (s, t)-cuts in graphs. *Algorithmica*, 15(4):351–372, Apr 1996.

[40] Predrag Radivojac, Wyatt T Clark, Tal Ronnen Oron, Alexandra M Schnoes, Tobias Wittkop, Artem Sokolov, Kiley Graim, Christopher Funk, Karin Verspoor, Asa Ben-Hur, Gaurav Pandey, Jeffrey M Yunes, Ameet S Talwalkar, Susanna Repo, Michael L Souza, Damiano Piovesan, Rita Casadio, Zheng Wang, Jianlin Cheng, Hai Fang, Julian Gough, Patrik Koskinen, Petri Törönen, Jussi Nokso-Koivisto, Liisa Holm, Domenico Cozzetto, Daniel W A Buchan, Kevin Bryson, David T Jones, Bhakti Limaye, Harshal Inamdar, Avik Datta, Sunitha K Manjari, Rajendra Joshi, Meghana Chitale, Daisuke Kihara, Andreas M Lisewski, Serkan Erdin, Eric Venner, Olivier Lichtarge, Robert Rentzsch, Haixuan Yang, Alfonso E Romero, Prajwal Bhat, Alberto Paccanaro, Tobias Hamp, Rebecca Kaßner, Stefan Seemayer, Esmeralda Vicedo, Christian Schaefer, Do-

minik Achten, Florian Auer, Ariane Boehm, Tatjana Braun, Maximilian Hecht, Mark Heron, Peter Hönigschmid, Thomas A Hopf, Stefanie Kaufmann, Michael Kiening, Denis Krompass, Cedric Landerer, Yannick Mahlich, Manfred Roos, Jari Björne, Tapio Salakoski, Andrew Wong, Hagit Shatkay, Fanny Gatzmann, Ingolf Sommer, Mark N Wass, Michael J E Sternberg, Nives Škunca, Fran Supek, Matko Bošnjak, Panče Panov, Sašo Džeroski, Tomislav Šmuc, Yiannis A I Kourmpetis, Aalt D J van Dijk, Cajo J F ter Braak, Yuanpeng Zhou, Qingtian Gong, Xinran Dong, Weidong Tian, Marco Falda, Paolo Fontana, Enrico Lavezzo, Barbara Di Camillo, Stefano Toppo, Liang Lan, Nemanja Djuric, Yuhong Guo, Slobodan Vucetic, Amos Bairoch, Michal Linial, Patricia C Babbitt, Steven E Brenner, Christine Orengo, Burkhard Rost, Sean D Mooney, and Iddo Friedberg. A large-scale evaluation of computational protein function prediction. *Nature Methods*, 10:221–227, 01 2013.

[41] Ronald C. Read. A survey of graph generation techniques. In *Combinatorial mathematics VIII*, pages 77–89. Springer, 1981.

[42] Benno Schwikowski and Ewald Speckenmeyer. On enumerating all minimal solutions of feedback problems. *Discrete Applied Mathematics*, 117(1):253–265, 2002.

[43] Hong Shen and Weifa Liang. Efficient enumeration of all minimal separators in a graph. *Theoretical Computer Science*, 180(1):169 – 180, 1997.

[44] Dennis Stanton and Dennis White. *Constructive combinatorics*. Undergraduate Texts in Mathematics (Springer-Verlag), 1986.

[45] W. Henry Suters, Faisal N. Abu-Khzam, Yun Zhang, Christopher T. Symons, Nagiza F. Samatova, and Michael A. Langston. A new approach and faster exact methods for the maximum common subgraph problem. In Lusheng Wang, editor, *Computing and Combinatorics*, pages 717–727, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[46] Amos Tanay, Roded Sharan, and Ron Shamir. Discovering statistically significant biclusters in gene expression data. *Bioinformatics*, 18(suppl 1):S136–S144, 2002.

[47] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical Computer Science*, 363(1):28–42, 2006.

[48] Charalampos Tsourakakis, Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Maria Tsiarli. Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 104–112. ACM, 2013.

[49] Shuji Tsukiyama, Mikio Ide, Hiromu Ariyoshi, and Isao Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM Journal on Computing*, 6(3):505–517, 1977.

[50] Takeaki Uno. A fast algorithm for enumerating bipartite perfect matchings. In *Algorithms and Computation, 12th International Symposium, ISAAC 2001, Christchurch, New Zealand, December 19-21, 2001, Proceedings*, pages 367–379, 2001.

[51] Takeaki Uno. Two general methods to reduce delay and change of enumeration algorithms, 2003. NII Technical Report NII-2003-004E, Tokyo, Japan.

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

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

[54] Kunihiro Wasa. Enumeration of enumeration algorithms. *CoRR*, abs/1605.05102, 2016.

[55] David W Williams, Jun Huan, and Wei Wang. Graph database indexing using structured graph decomposition. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 976–985. IEEE, 2007.

[56] Bin Wu and Xin Pei. A parallel algorithm for enumerating all the maximal k-plexes. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 476–483. Springer, 2007.

[57] S Yau. Generation of all hamiltonian circuits, paths, and centers of a graph, and related problems. *IEEE Transactions on Circuit Theory*, 14(1):79–81, 1967.