Università di Pisa
Computer Science Department
Master's degree in Computer Science

Master's thesis

A New Algorithmic Framework for Enumerating Commutable Set Properties

Supervisor:
Prof. *Roberto Grossi*

Author:
Luca Versari

Accademic Year 2016–2017

Abstract

This thesis considers a new algorithmic framework for listing maximal sets satisfying a given property (e.g. being a clique, a cut, a cycle, etc.), which fall within the general framework of set systems. A set system $\mathcal{F}$ over a ground set U (e.g. the network nodes) is a collection of subsets of U for which there exists some function that checks if an arbitrary subset of U belongs to $\mathcal{F}$. For all *maximal* subsets in $\mathcal{F}$ under inclusion to be listed, the ambitious goal is to cover a large class of set systems while preserving the efficiency of their enumeration algorithms at the same time. The best-known ones list the maximal subsets in time proportional to their number but may require exponential space. This thesis improves the state of the art in two directions by introducing an algorithmic framework that, under suitable conditions, simultaneously (i) extends the class that can be solved efficiently to *commutable set systems*, and (ii) reduces the additional space usage from exponential in $|E|$ to *stateless*, thus accounting for just $O(q)$ space, where $q \leqslant |E|$ is the largest size of a maximal set.

B

Contents

Chapter 1

Introduction

Algorithms for graph listing have a long history and even if they were born in the 70s in the context of enumerative combinatorics and computational complexity [17, 26, 32, 38, 44], the interest has quickly broadened to a variety of other communities in computer science and not, massively involving algorithm design techniques.

In network analysis discovering special communities corresponds to finding all the subgraphs with a given property [1, 15, 27, 29, 30, 42]. In bioinformatics, listing all the solutions is desirable, as single or few solutions may not be meaningful due to noise of the data, noise of the models, or unclear objectives [9, 21, 25, 28, 40]. In graph databases, graph structures are used for semantic queries, where nodes, edges, and properties are used to represent and store data; retrieving information corresponds to find all the suitable subgraphs in these databases [2, 11, 46]. When dealing with incomplete information, it may be impossible to completely satisfy a query. Subgraph listing algorithms can find answers that *maximally* satisfy a partial query; for instance, there is a one-to-one correspondence between the results of a join or full disjunction query and certain subgraphs of the *assignment graph*, a special graph obtained by combining the relational database with the given query. Moreover, the kind of subgraphs to look for depend not only on the database, but also on the query [11].

In this scenario, graph enumeration has left the theoretical border [24] to meet more stringent requirements: not only a given listing problem must fit a given class of complexity, but its algorithms must be efficient also in real-world applications. Algorithm design has made a big effort to generalize the graph properties to be enumerated and to unify the corresponding approaches [3, 8, 10, 26, 43]. These generalizations allow the same algorithm to solve many different problems.

This thesis presents some of the work that has been done so far and new results that fit into this line of research: on one side, we want to obtain efficient listing algorithms able to deal with large networks; on the other side, we aim at designing an algorithmic framework which solves simultaneously many problems and leave the designer in charge of few core tasks depending on the specific application. In particular, we focus on efficient enumeration algorithms for maximal subgraphs satisfying a given property (e.g. being a clique, a cut, a cycle, a matching, etc.), as they fall within the general framework of set systems [10, 26].

This thesis is organized as follows: Chapter 1 contains an overview of the thesis and some well-known definitions necessary to develop the content. Chapter 2 contains an introduction to set systems, that gives definitions, examples, known lower bounds and known algorithm for the enumeration of its maximal sets. Chapter 3 explains how the new algorithmic framework works, and Chapter 4 gives some applications of the framework to enumeration problems.

The original work in this thesis is mainly contained in Chapter 2 for the introduction of commutable set systems, Chapter 3 for the new algorithmic framework and Chapter 4 for its applications, except for Subsection 3.2.1 that gives an improved overview over already-existing work. Moreover, parts of the thesis build upon further work that was published in [12, 13].

1.1 Preliminaries

As graphs play a vital role in this thesis, we start by stating some standard definitions: an *undirected graph* G is a pair (V, E) with $E \subset V \times V$. The elements of V are called *nodes* of the graph, while the elements of E are called *edges*. The edge set of G must be symmetrical, i.e. if $(a, b) \in E$ then $(b, a) \in E$, and must not contain self-loops (i.e. $(a, a) \notin E$ for every a in V). We say that a node v is a *neighbour* of w if $(w, v) \in E$. Moreover, we denote with $N_G(v)$ the set of all neighbours of v in G. When the graph is unambiguous, we use $N(v)$.

A *directed graph* G is a pair (V, A) with $A \subset V \times V$. The elements of V are called *nodes*, while the elements of A are called *arcs*. As with undirected graphs, G must not contain self-loops. A node v is a *in-neighbour* of w if $(v, w) \in A$. Similarly, it is an *out-neighbour* of w if $(w, v) \in A$. The sets of all in- and out-neighbours is denoted as $N^-(v)$ and $N^+(v)$, respectively.

When not further specified, *graph* means undirected graph.

A *path* in a graph G is a sequence of nodes $v_1, \dots, v_k$ such that (v_i, v_{i+1}) is an edge (or an arc) for every $i = 1 \dots k - 1$. We say that the *length* of this path is $k - 1$ (i.e. the number of edges or arcs involved in it). A subset of nodes of G is *connected* if, for every pair of nodes in it, there is at least a path that connects them that is made only of nodes of the subset. As being connected is an equivalence relation, G may be partitioned into equivalence classes that are called *connected components*.

A *subgraph* of a graph G is the graph that has as nodes a given subset V' of V and as edges some of the edges in G that have as endpoints two members of V'. An *induced subgraph* is a subgraph that has all possible edges of E for that choice of nodes.

A graph is *bipartite* if it has no odd cycles or, equivalently, if it can be partitioned into two sets A, B such that there is no edge between any pair of nodes in A nor in B.

We assume the nodes of a graph to be ordered, and denote them as $v_1, \dots, v_n$, where $n = |V|$.

We will also use the concept of partial order:

Definition 1.1. A **partial order** is a binary relation $\sqsubseteq$ on a set A that is

1. *reflexive*, i.e. $a \sqsubseteq a$ for every a in A,
2. *transitive*, i.e. if $a \sqsubseteq b$ and $b \sqsubseteq c$, then $a \sqsubseteq c$,
3. *antisymmetric*, i.e. if $a \sqsubseteq b$ and $b \sqsubseteq a$, then $a = b$.

If $\sqsubseteq$ also satisfies that $\forall a, b \in A$ either $a \sqsubseteq b$ or $b \sqsubseteq a$, we call $\sqsubseteq$ a **total order** on A.

An important consequence of the fact that orders are transitive and antisymmetric is that an order contains no cycles, i.e. it is impossible to have a set of distinct elements $a_1, \dots, a_n$ in A such that $a_1 \sqsubseteq \dots \sqsubseteq a_n \sqsubseteq a_1$.

If $\sqsubseteq$ is an order, we write $a \sqsubset b$ to mean that $a \sqsubseteq b$ and $a \neq b$.

Finally, we say that an element m is *maximal* if there is no other element v such that $m \sqsubset v$.

An interesting family of orders is the **lexicographical order**. If A is a set with an order $\sqsubseteq$, we define an order on the set of all the tuples of elements of A as follows. Let $P = (p_1, \dots, p_k)$ and $Q = (q_1, \dots, q_h)$ be two tuples of elements of A.

- If P is a prefix of Q (i.e. $k \leqslant h$ and $p_i = q_i$ for $1 \leqslant i \leqslant k$), then $P \prec Q$.
- Otherwise, let i be the first index such that $p_i \neq q_i$. Then, $P \prec Q$ if and only if $p_i \sqsubset q_i$.

We may also define a lexicographical order on the power set 2^A of A. More precisely, the order between any two subsets of A is the same as the order between the tuples that have as elements the same elements, in ascending order. As an example, take sets $B = \{9, 3, 7, 5\}$ and $C = \{7, 5, 3, 11\}$ in $\mathbb{N}$. The corresponding sorted tuples are $(3, 5, 7, 9)$ and $(3, 5, 7, 11)$; the former is lexicographically smaller than the latter, so we say that B is lexicographically smaller than C.

Another interesting order is the **inclusion order** on the power set of any set A: given $B, C \in 2^A$, we say that B is less than C if and only if $B \subset A$.

1.2 Enumeration complexity measures

We call α the number of solutions of the enumeration problem. Since for many enumeration problems α may be exponential in the size of the input, traditional complexity definitions are not enough to provide all the information on the running time of an enumeration algorithm. For example, it was proven that in a graph there may be up to

$3^{n/3}$ maximal cliques [41], so the worst-case complexity of any enumeration algorithm for maximal cliques may not be any lower than that.

Considering this fact, complexity classes for enumeration algorithms have been defined [20] in a way that takes into account the effective number of solutions of the given problem. We will report the most useful among these classes here.

We say that an algorithm

- runs in *polynomial total time* if its running time is bounded by a polynomial in the input size and in α.
- runs in *incremental polynomial time* if the time needed to generate solution number $X \leqslant \alpha$ is bounded by a polynomial in the input size and in X.
- has *polynomial cost per solution* if its running time is bounded by a polinomial in the input size times α.
- produces *incremental output* if it outputs the first $X \leqslant \alpha$ within a time bounded by X times a polynomial of the input size.
- has *bounded delay* if the running time between any two consecutive produced solutions is bounded by a polynomial in the input size.

Chapter 2

Set systems

This chapter introduces the notion of set systems, discussing some useful classes of them and giving an overview on what is known about enumeration algoritms that list all their maximal elements. Section 2.2 gives some examples of families that form various kinds of set systems.

2.1 Definitions

One of the most well-known enumeration problem is that of enumerating all the maximal cliques in a graph. A clique is a complete subgraph of a given graph G (a more formal definition is given in Definition 2.7), and so it may be considered as a subset of the nodes of the graph. Families of solutions given by the subsets of a given set are a recurring situation in enumeration problems (as another example, feasible solutions of a knapsack problem may be considered subsets of a given set), so this situation has justified the following general definition.

Definition 2.1. A **set system** $\mathcal{F}$ over a *support set* U is a nonempty family of subsets of U, i.e. $\mathcal{F} \subset 2^U$ and $\mathcal{F} \neq \emptyset$. A member of $\mathcal{F}$ is called a *feasible set*.

This thesis will focus mostly on set systems built on graphs. In this case, unless otherwise noted, the support set of the set system will be the set of the nodes of the graph.

As set systems are way too generic to provide any meaningful result, the following subclasses have been studied so far (see for example [6]).

Definition 2.2. An **accessible set system** $\mathcal{F}$ over U is a set system that satisfies the following property: for any nonempty $X \in \mathcal{F}$, there is an element $x \in X$ such that $X \setminus \{x\} \in \mathcal{F}$.

Definition 2.3. A **strongly accessible set system** $\mathcal{F}$ over U is a set system that satisfies the following property: for any nonempty $X \in \mathcal{F}$ and any $Y \subset X$ that belongs to $\mathcal{F}$, there

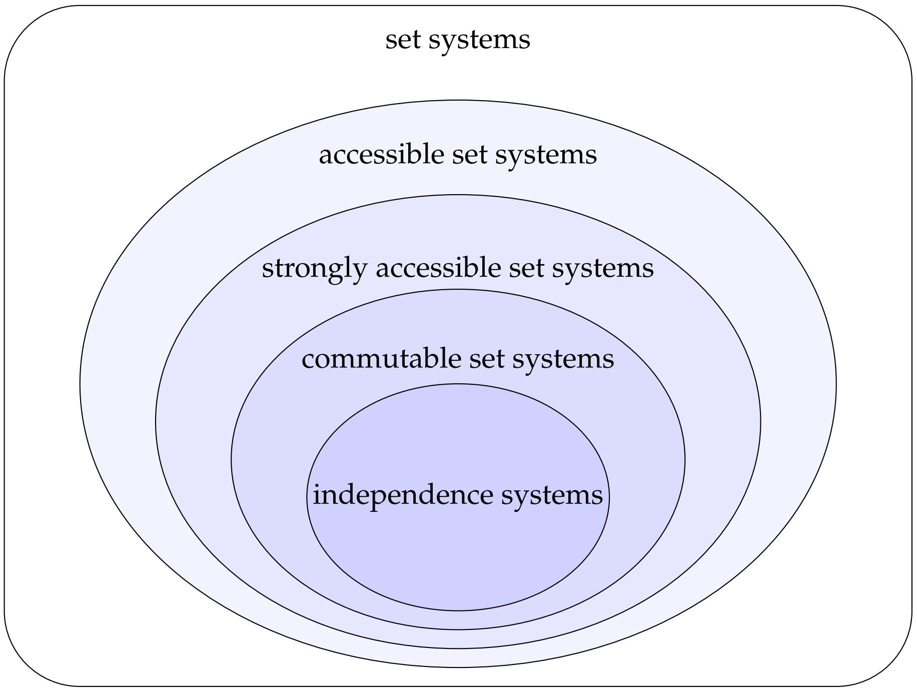


Figure 2.1: Relationships between the various kinds of set systems.

is an element $x \in X \setminus Y$ such that $X \setminus \{x\} \in \mathcal{F}$ or, equivalently, there is an element $y \in X \setminus Y$ such that $Y \cup \{y\} \in \mathcal{F}$.

Definition 2.4. An **independence system** $\mathcal{F}$ over U is a set system that satisfies the following property: for any nonempty $X \in \mathcal{F}$, every $Y \subseteq X$ belongs to $\mathcal{F}$ too.

From the definitions, it clearly follows that any independence system is also a strongly accessible set system, and any strongly accessible set system is also an accessible set system. Our main focus will be on a new class that sits in the middle of independence systems and strongly accessible set systems. These relationships are represented in Figure 2.1.

Definition 2.5. A **commutable set system** $\mathcal{F}$ over U is a strongly accessible system that satisfies the following property: for any $X \in \mathcal{F}$ and any $Y \subseteq X$ that belongs to $\mathcal{F}$, if $x, y \in X \setminus Y$ are such that $Y \cup \{x\} \in \mathcal{F}$ and $Y \cup \{y\} \in \mathcal{F}$ then $Y \cup \{x, y\} \in \mathcal{F}$.

Another possible definition is the following one:

Definition 2.6. A **commutable set system** $\mathcal{F}$ over U is a strongly accessible system that satisfies the following property: for any $X \in \mathcal{F}$ and any $Y \subseteq X$ that belongs to $\mathcal{F}$, if $A, B \subseteq X$ are members of $\mathcal{F}$ that properly contain Y, then $A \cup B \in \mathcal{F}$.

Lemma 2.1. *Definition 2.5 and Definition 2.6 are equivalent.*

Proof. Definition 2.6 trivially implies Definition 2.5.

If $|A \setminus X| = 0$ then there is nothing to prove. We will first prove the equivalence in the case in which $A = X \cup \{a\}$.

If $B = X \cup \{b\}$, then the thesis is exactly Definition 2.5. Otherwise let $b \in B \setminus X$ be such that $X \cup \{b\} \in \mathcal{F}$ (such a b exists because $\mathcal{F}$ is strongly accessible). Then we have that $X \cup \{a, b\} \in \mathcal{F}$ because of Definition 2.5. We may repeat this reasoning to the pair of sets $X \cup \{a, b\} = A \cup \{b\}$ and B, both of which contain $X \cup \{b\}$ and whose union is still $A \cup B$. Since now the number of elements that should be added to the common part of the two sets to obtain B is reduced, this completes the proof by induction.

Let us now consider the general case, and let $a \in A \setminus X$ be chosen such that $X \cup \{a\} \in \mathcal{F}$. Thanks to what we just proved, we know that $B \cup \{a\} \in \mathcal{F}$. As before, we may repeat this process on A and $B \cup \{a\}$, both of which contain $X \cup \{a\}$ and whose union is still $A \cup B$. By induction, this completes the proof.

□

As a set system may contain an exponential number of members, we will assume that the set system is not given explicitely, but through a **membership oracle**, a function $f_{\mathcal{F}}$ that takes a subset of U and returns true if that subset is a member of $\mathcal{F}$.

We will focus on the task of enumerating all the *maximal* (under inclusion) members of $\mathcal{F}$. This is a good compromise between the need of knowing the whole $\mathcal{F}$ and the issue of its size: all members of $\mathcal{F}$ are included in a maximal one and, for set systems that are at least strongly accessible, any element of $\mathcal{F}$ may be found from a maximal one by iteratively removing an element.

Notation. Throughout the thesis, n will denote the cardinality of U, q will denote the maximum size of any member of $\mathcal{F}$ and α will denote the number of maximal elements in $\mathcal{F}$.

2.2 Some accessible set systems

One of the most studied examples of independent systems are maximal cliques. This problem has been studied in a lot of contexts, including bioinformatics [14, 35], computational chemistry [33, 23, 4] and social network analysis [18, 45]. The family may be described as follows:

Definition 2.7 (Clique)**.** Given a graph $G = (V, E)$, a subset C of its vertices is called a clique if, for every $a \neq b \in C$ there is an edge between a and b.

Another studied example is the k-plex, which is a generalization of a clique that relaxes the condition that all pairs of nodes must be connected:

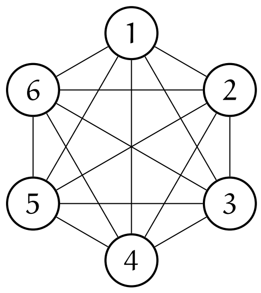
Definition 2.8 (k-plex)**.** Given a graph $G = (V, E)$, a subset C of its vertices is called a k-plex if, for every a in C,

$$|C \cap N(a)| \geqslant |C| - k$$

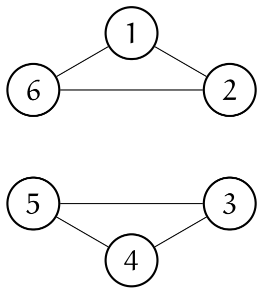
holds.[1]

This definition of k-plex has some practical issues, such as every set of size at most k being a k-plex. To reduce the impact of this issue, a minor variant of this family has been considered [5]:

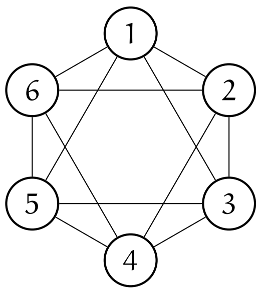
Definition 2.9 (Connected k-plex)**.** Given a graph $G = (V, E)$, a subset C of its vertices is called a connected k-plek if it is a k-plex and it is connected.



2.2a: Clique with six nodes



2.2b: 4-plex with six nodes



2.2c: Connected 2-plex with six nodes

An example of a clique, a 4-plex and a connected 2-plex can be found in Figure 2.2a, Figure 2.2b and Figure 2.2c respectively. In particular, Figure 2.2b shows a degenerate example of a k-plex that is not even connected, which should not be considered interesting as the objective of finding k-plexes is to find dense structures.

It is easy to see that any k-plex C of size at least $2k - 1$ is also a connected k-plex: indeed, consider any two nodes a and b in C. If they are connected by an edge, then there is a path between them and we are done. Otherwise, suppose that their neighbourhoods are disjoint in C. Then:

$$|C| - 2 = |C \setminus \{a, b\}| \geqslant |C \cap (N(a) \cup N(b))| = |C \cap N(a)| + |C \cap N(b)| \geqslant 2|C| - 2k$$

[1]This reduces to the definition of clique when $k = 1$.

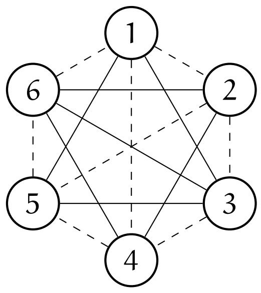


Figure 2.3: Black-connected clique with six nodes. White edges are dashed, black edges are represented with a continuous line.

which implies that $2k - 2 \geqslant |C|$, which is a contraddiction. So their neighbourhoods must intersect in some node $v \in C$, providing a valid path a, v, b between the two nodes.

This implies that considering just connected k-plexes does not alter the results for bigger elements of the family, while removing a good amount of small sets.

Unfortunately, connected k-plexes do not form an independent system (as can easily be seen by considering the subset $\{1, 4\}$ of the connected 2-plex of Figure 2.2c: indeed, this subset is not a connected 2-plex because it is not connected), but they still are commutable set system. This can be easily seen by using Definition 2.6, as any subset of a k-plex is still a k-plex and the union of any two connected sets that share at least a node is still connected.

Another interesting family that is not an independent system is the family of black-connected cliques. In this context, we have a colored graph that is composed of white and black edges:

Definition 2.10 (Black-connected clique)**.** Given a graph $G = (V, E)$ in which every edge has an associated color (either black or white), a subset C of its vertices is a black-connected clique if it is a clique and if it is connected in the graph where all white edges are removed.

Figure 2.3 gives an example. Moreover, from the example it is clear that black-connected cliques do not form an independence system, as the subset $\{1, 4\}$ is not a black-connected clique.

This problem arises naturally while studying molecule similarity in bioinformatics and computational chemistry[22, 12]. As with connected k-plexes, this family is a commutable set system.

Another family that may be interesting to study for network analysis is the family of connected bipartite (induced) subgraphs of a given graph G:

Definition 2.11 (Connected bipartite (induced) subgraph)**.** Given a graph $G = (V, E)$, a subset C of its edges (vertices) is a connected bipartite (induced) subgraph if the subgraph it defines is both bipartite and connected.

The previous examples of set systems that are not independence systems all share one common trait: they are obtained from independence systems by adding a connectivity constraint. These kinds of independence systems have been called *connected hereditary*, and an algorithm for enumerating them (that uses exponential memory) was given in [10].

It is natural, then, to wonder if any commutable set system is, in fact, a connected hereditary property. This conjecture can be proven false by considering the family $\mathcal{F}$ of cliques that all share a given node v in common: indeed, they are a commutable set system, but they are not an independent set system nor they can be obtained as the family of all cliques that satisfy a connectivity constraint (because in that case any singleton would be a member of the family, while in this case $\{v\}$ is the only singleton that belongs to $\mathcal{F}$).

The techniques used for the enumeration of set systems may be used to solve enumeration problems that one would not ordinarily associate with set systems. For example, one might enumerate all the s-t paths by enumerating the maximal elements of the set system made of all paths that begin in a certain node s and may be extended to reach a certain destination t.

In a similar way, the set of all sub-forests in a graph form an independence system, whose maximal elements are spanning trees: so it is possible to enumerate spanning trees with techniques that enumerate independence systems.

Another family that may be considered an independence system is the (complement of) s-t (vertex-)cuts in a graph:

Definition 2.12 (s-t (vertex-)cuts)**.** Given a connected graph $G = (V, E)$ and two vertices s, t, a subset C of E ($V \setminus \{s, t\}$) is an s-t (vertex-)cut if and only if the graph obtained by removing C from G has s and t in two different connected components.

This family is clearly not even accessible, but the family formed by its complement is. It may be defined directly as the family of all subsets of E (or $V \setminus \{s, t\}$) that induce connected components such that s and t belong to different components.

A similar family to the previous one is given by the feedback vertex (arc) set problem:

Definition 2.13 (Feedback vertex (arc) set)**.** Given a graph (possibly directed) $G = (V, E)$, a feedback vertex (arc) set is a subset F of V (E) such that the graph obtained from G by removing the elements in F contains no cycles.

As before, this is not an accessible family, but considering its complement we obtain an independence system. If we furthermore reduce ourselves to consider subset that leave the remaining graph connected, we obtain a commutable set system. This problem has been studied before, but known algorithms use an exponential amount of memory [47, 36].

2.3 Lower bounds

It was proven in [26] that it is not reasonable to try to find an algorithm that is able to find the maximal sets of **any** independence system in polynomial total time, as shown by the following reduction:

Theorem 2.1 (Lawler et al [26]). *There is no algorithm that enumerates all the maximal sets in any independence system in polynomial total time, unless* $P = NP$.

Proof. Let $F(X_1, \dots, X_n)$ be a boolean expression in conjunctive normal form. We will define an independence system on the ground set $E = \{T_1, F_1, \dots, T_n, F_n\}$ as follows. Let J be any subset of U and $x_i(J)$ be defined as:

$$x_i(J) = \begin{cases} \mathtt{true} & \text{if } T_i \in J,\ F_i \notin J \\ \mathtt{false} & \text{if } F_i \in J,\ T_i \notin J \\ \mathtt{undefined} & \text{if } T_i \notin J,\ F_i \notin J \\ \mathtt{overdefined} & \text{otherwise} \end{cases}$$

We say that $J \in \mathcal{F}$ if and only if one of the following holds:

- The assignment $X_i = x_i(J)$ satisfies the boolean formula F, or
- We have $T_i \notin J$ and $F_i \notin J$.

This is clearly an independence system, as any subset of a J that satisfies the second condition still satisfies it and any subset of a J that satisfies the first condition has at least one undefined literal.

Moreover, any solution to F provides a maximal set for $\mathcal{F}$; maximal sets not obtained this way must have at least an undefined literal, so they are of the form $E \setminus \{T_i, F_i\}$ for some i. We call these maximal sets trivial. Note that there are n trivial maximal sets.

Let us now suppose that there is an algorithm which is able to enumerate all the maximal elements of $\mathcal{F}$ in $P(n, \alpha)$ time. Then, if we let this algorithm run for $P(n, n)$ time, one of the following situations may happen:

- The algorithm terminates and finds no non-trivial maximal set: we know that F has no solution.
- The algorithm terminates and finds a non-trivial maximal set: we know that F has a solution.
- The algorithm does not terminate: this implies that $\mathcal{F}$ has more than n maximal solutions, so at least one of them must be non-trivial and thus F must have a solution.

This reasoning gives an algorithm that takes $P(n, n)$ (i.e. polynomial) time to check if F has a solution or not, thus proving the reduction.

□

Moreover, from the same proof another condional lower bound follows, based on the well-known Strong Exponential Time Hypothesis of Impagliazzo and Paturi[19].

Corollary 2.1. *There is no algorithm that enumerates all the maximal sets in any independence system in* $O(2^{\frac{q}{2}-\epsilon}P(\alpha, n))$, *unless SETH is false.*

To circumvent this issue, any general technique that promises to enumerate maximal elements in strongly accessible set systems relies on some form of problem-specific insight. In particular, *certificates* and *restricted problems* have been considered.

Definition 2.14 (Certificate)**.** A certificate for a given enumeration problem is some kind of information that can be computed in polynomial time and guarantees the existence of at least one solution for the given subproblem that must be solved during the recursive enumeration.

Definition 2.15 (Restricted problem)**.** Given a strongly accessible set system $\mathcal{F}$ on U, a maximal feasible set F and an element $v \in E \setminus F$, the restricted problem $\mathcal{P}(F, v)$ is the problem of enumerating all the maximal elements of the family

$$\mathcal{G}_F^v = \{A \in \mathcal{F} : A \subseteq F \cup \{v\}\}$$

Figure 2.4 represents the solutions of the restricted problem that is obtained for the connected k-plex problem from the k-plex of Figure 2.2c by adding node 7. The solution on top is the trivial solution (i.e. the solution that generated the restricted problem), while the other four figures are obtained by choosing 7 and all the possible subsets of non-neighbours of 7, and choosing the other nodes accordingly.

2.4 Previous algorithms

Previous works on the enumeration of maximal elements of set systems are mainly based on two different techniques: binary partition and reverse search. See for example [34] for an overview of enumeration algorithms, where its Chapter 9 deals with enumeration of subgraphs satisfying certain properties.

2.4.1 Binary partition

Binary partition is a general, recursive scheme that works by splitting the problem into the following subproblems: "enumerate all maximal sets that contain all the elements that belong to set S, no elements that belong to set X and possibly some of the elements that do not belong to either of them". The recursion starts with $S = X = \varnothing$, and

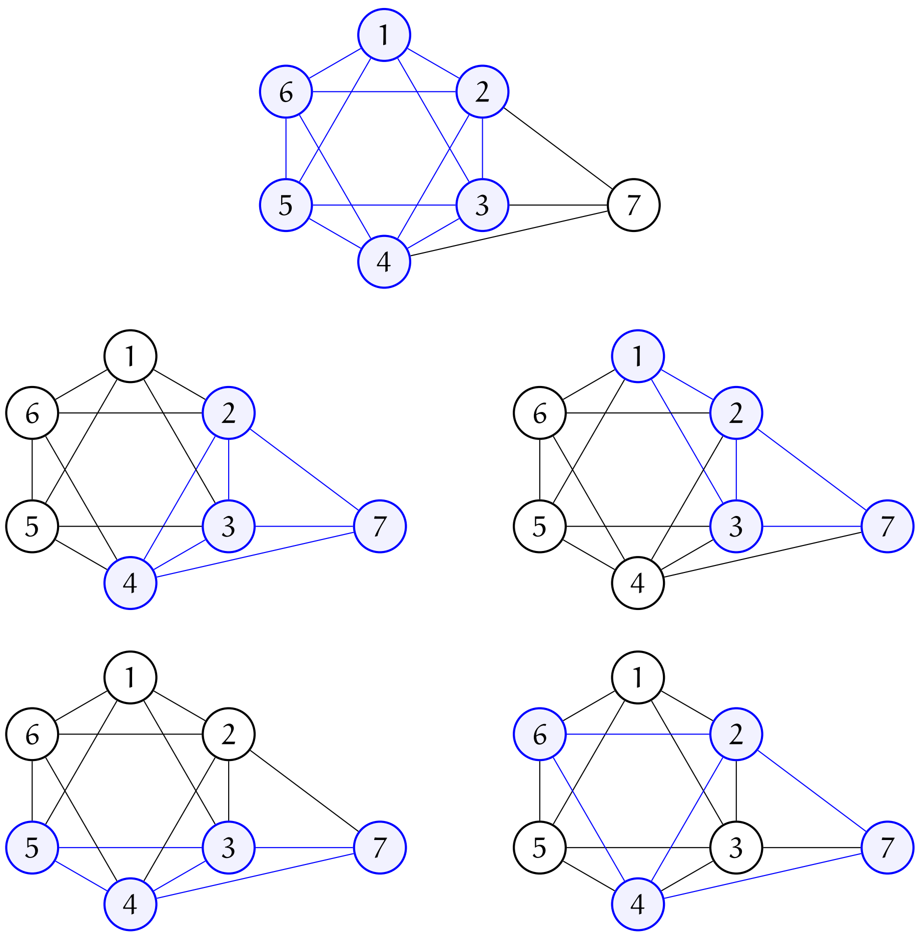


Figure 2.4: Example of a restricted problem using the connected 2-plex of Figure 2.2c and the extra node 7. The figures below highlight the non-trivial solutions, the one on the top the original k-plex that defines the restricted problem.

proceeds by choosing, at every recursive step, some node v in $V \setminus (S \cup X)$ and then by producing one or two recursive children, with v added to X or to S. This second option is only possible if we have that $S \cup \{v\} \in \mathcal{F}$, as otherwise the algorithm would produce incorrect solutions. When $S \cup X = V$, the recursion ends and the current solution S is generated as output.

This scheme guarantees a polynomial time per solution whenever we have a certificate for the problem we are considering. Indeed, the presence of a certificate allows us to immediately stop the recursion in nodes where we know there will not be any solution, producing a recursion tree that has at most αn nodes. As the cost of the computations done in every node of the recursion tree is polynomial in the input size, the total cost is polynomial in the input and the number of solutions. Further optimizations (dependent on the problem) usually allow to give algorithms that are extremely efficient in these cases.

When no certificate is available, no polynomial upper bound in the input and output size can usually be given. Nonetheless, algorithms that are very efficient in practice may be obtained by using some sort of heuristic that allows to eliminate a good percentage of the "dead ends" of the recursion tree.

As an example, binary partition can easily be used to enumerate spanning trees or s-t paths in polynomial time per solution. For the first case, a simple certificate is given by the check that the graph obtained by removing all the excluded edges is still connected, as that guarantees the existence of a spanning tree. As for s-t path, a certificate can be obtained by checking that the endpoint of path defined by the current set of "taken" edges is still able to reach t, without using any excluded edge.

The Bron-Kerbosch algorithm [7], used for enumerating maximal cliques and the basis of a lot of algorithms for the enumeration of cliques, k-plexes and black-connected cliques [22], is a straightforward application of binary partition, with a smart pivoting rule that allows to remove most "unwanted" recursive nodes.

2.4.2 Reverse search

Reverse search is a general technique in enumeration which was initially introduced by Avis and Fukuda [3]. It works by implicitely constructing a directed graph that has as nodes all the solutions to the enumeration problem, with an arc from a certain solution P to another solution Q if the second can be obtained in a certain way from the first. As long as the graph is properly connected, a simple graph visit starting from any solution will then enumerate all the solutions to the problem.

As this graph may contain multiple paths to a single solution, we either need to keep track of all the solutions already found so far or we need some way to remove edges from the graph, making it a directed forest whose roots are known.

In the first case, there is no performance penalty, but the resulting algorithm takes exponential memory to run. In the second case, some more computations may be required to discard unneeded edges.

Regarding set system enumeration, this technique is usually applied by building a graph over all the maximal elements of $\mathcal{F}$. Given a solution S, outgoing edges are obtained by using a node from $V \setminus S$ as a "guide" to find new maximal elements. In particular, this is usually done by solving the restricted problem on (S, v) and applying some form of post-processing to the found solutions. Examples and further details are provided in Section 3.2.

This idea is the one at the basis of most algorithms for maximal clique enumeration that guarantee polynomial time per solution, as in [13]. Moreover, it was also employed in [10] to give an algorithm for the enumeration of maximal elements in an independence system. In the same paper, the authors give an algorithm for the enumeration of maximal sets that satisfy "connected hereditary" properties, using exponential space. As the sets that satisfy connected hereditary properties are a special case of commutable set systems, this thesis improves their work by removing the need for exponential space.

Chapter 3

Framework description

This chapter will explain how the new framework for the enumeration of maximal elements in commutable set systems works. We will start by giving some necessary definitions, then we will proceed to explain the algorithm itself. Throughout this section, we assume the elements of U to be ordered with an arbitrary total order.

3.1 Core concepts

We will now introduce some concepts that will be fundamental while explaining how the algorithm works. As they are not very intuitive, most definitions will have an example on the black-connected clique C of Figure 2.3.

Definition 3.1. Given a commutable set system $\mathcal{F}$ and one of its feasible sets S, we say that s is a **seed** of S if $s \in S$ and $\{s\} \in \mathcal{F}$. The **canonical seed** of S, denoted by $\mathtt{seed}(S)$, is the smallest possible seed according to the ordering of the elements of U.

Any singleton in C is a seed, since it is both a clique and connected with black edges. So, its canonical seed is 1.

The concept of level, that is introduced with the next definition, will be fundamental to define a $\mathtt{complete}$ function (one of the main ingredients in the reverse search algorithm) that satisfies the property stated in Lemma 3.2 but is not NP-hard to compute (see Lemma 3.1). Moreover, the properties of the $\mathtt{complete}$ function obtained using this definition will allow us to prove Theorem 3.1.

Definition 3.2. Given a commutable set system $\mathcal{F}$, one of its feasible sets S and a seed s of S, we define the **level** of an element v with respect to s as follows:

- if $v = s$, then the level of v is 0.
- let k be the smallest integer such that there is a set $S' \subset S$ that is composed of elements of level $\leqslant k$ and such that $S' \cup \{v\} \in \mathcal{F}$ and $s \in S'$. Then the level of v is $k + 1$.

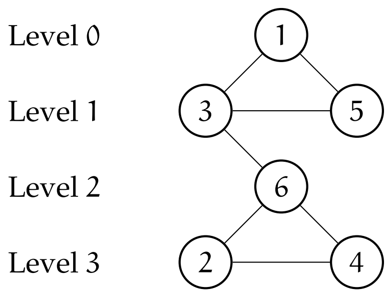


Figure 3.1: Black-connected clique of Figure 2.3 with nodes partitioned into groups according to their level. Only black edges are represented, and the canonical seed 1 is used.

- if there is no such subset, we say that the level of v is ∞.

We will use $\mathtt{level}_S^s(v)$ to denote the level of v in S with respect to seed s. When the seed is not specified, it is assumed to be the canonical seed.

Figure 3.1 represents C according to its levels. It follows immediately from the definition of level that the level of a node v with respect to a given seed s is given by the distance between v and s (i.e. the length of the shortest path) according to black edges. This actually holds true for any connected hereditary family.

These two definitions are crucial to allow us to define the following order between two feasible sets.

Definition 3.3. The **level order** $\prec$ between any two solutions P, Q of a commutable set system $\mathcal{F}$ is defined as follows:

- Let $lP = \{(\mathtt{level}_P(v), v)\ \forall v \in P\}$, the set pairs made by the level of an element and the element itself.
- Let lQ be defined in the same way.
- We say that $P \prec Q$ if and only if lP is smaller than lQ using the lexicographical order between sets defined in Section 1.1.

According to this definition, the set corresponding to C, i.e. the set of pairs of the form $(\mathtt{dist}(s, v), v)$, is $\{(0,1), (1,3), (1,5), (2,6), (3,2), (3,4)\}$.

We define $\mathtt{complete}(S, s)$ as the maximal solution that is obtained from S by iteratively adding the element $v \in E \setminus S$ such that adding v keeps the current set in $\mathcal{F}$ and that, among all those possible choices, minimizes $(\mathtt{level}_S^s(v), v)$. The definition of $\mathtt{complete}(S)$ takes into account that, while we are adding elements to S, we might add an element s' that is smaller than $\mathtt{seed}(S)$. In that case, subsequent iterations consider the levels to be relative to the new canonical seed.

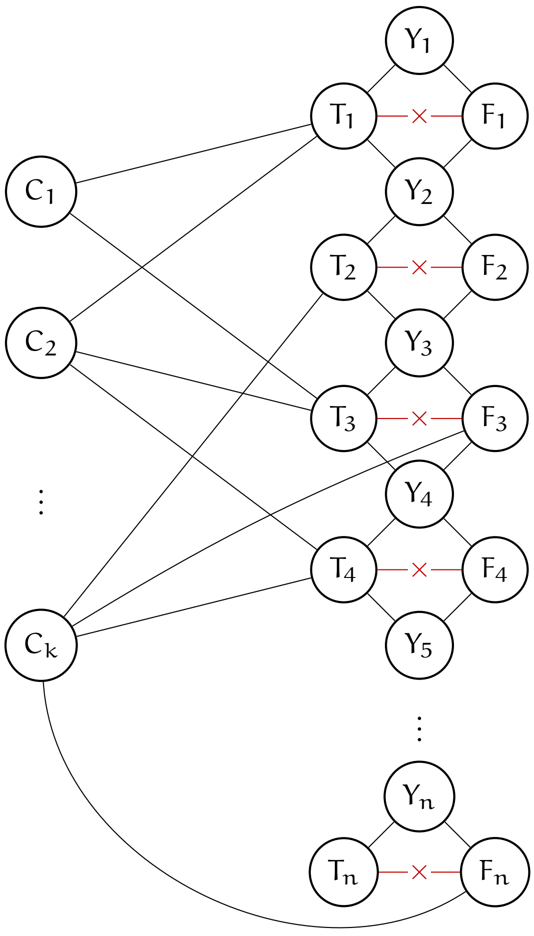


Figure 3.2: A graph with black and white edges that is used in the proof of Lemma 3.1. Only black edges are represented, white edges are implicitely assumed to be present between any two nodes, except for the ones that already have a black edge or that have a crossed-out red edge. Assuming $k = 3$ and $n = 5$, the figure represents the formula $(x_1 \lor x_3) \land (x_1 \lor x_3 \lor x_4) \land (x_2 \lor \neg x_3 \lor x_4 \lor \neg x_5)$

In previous works, $\mathtt{complete}(S)$ was simply defined as the lexicographically minimum maximal solution that contains S. Unfortunately, for a generic commutable set system this function is NP-hard to compute, as stated by the following lemma.

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

Proof. We prove that a $\mathtt{complete}(X)$ function that returns the lexicographically minimum black-connected clique containing X can be used to solve a SAT problem in polynomial time, by building a graph with a number of nodes linear in the amount of the clauses and variables in the formula.

Given a SAT formula in conjunctive normal form with n variables $x_1 \dots x_n$ and k clauses $d_1 \dots d_k$, we build the the graph in Figure 3.2, 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., nodes $C_1 \dots C_k$ have smaller label than all other nodes). Each Y_i is connected with a *black* edge to T_i and F_i, and, except for Y_1, also with T_{i-1} and F_{i-1}. Each C_i, which corresponds to d_i, is connected with *black* edge to T_j (resp. F_j) whether d_i contains a positive (resp. negative) occurrence of 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 Yi node. All other pairs of nodes are connected with a *white* edge, except for the pairs T_i,F_i (symbolized by the crossed-out red edge in Figure 3.2).

It is straightforward to see that any maximal black-connected clique in this graph will contain exactly one of T_i and F_i, for any i, and that any maximal black-connected clique containing all nodes in $C_1 \dots C_k$ will be lexicographically smaller than any that does not contain all of them (as they have the smallest labels).

Consider $\mathtt{complete}(\{Y_1\})$, the lexicographically smallest maximal black-connected clique containing Y_1. Any black-connected clique containing Y_1 and all C_i nodes represents a satisfying assignment for the formula at hand. Indeed, 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 black-connected clique; the set of T_i and F_i nodes in the black-connected clique will thus give us the value (true or false) of the corresponding variable x_i (recall that we cannot have the pair T_j-F_j in the same black connected clique). Hence, in order to verify that the formula is satisfiable, we only need to compute $\mathtt{complete}(\{Y_1\})$ and check whether this contains all C_i nodes.

□

Analogously, we say that P is a *prefix* of S if it is obtained from S by iteratively removing the element with the the highest level and, in case of ties, the highest element among those with maximal level. Note that the definition of levels, combined with the properties of commutative set systems, ensures that removing a node with maximal level will not cause S to become unfeasible. We also say that P is a *prefix of* S *with respect to* s to explicitely specify that s is the seed to be used.

Given the set form of C, it follows that its prefixes are given by $\{1\}$, $\{1,3\}$, $\{1,3,5\}$, $\{1,3,5,6\}$, $\{1,3,5,6,2\}$ and $\{1,3,5,6,2,4\}$, that are all black-connected cliques.

Note that, if P is a subset of S, $\mathtt{level}_P^s(v) \geqslant \mathtt{level}_S^s(v)$. If P is a prefix of S, then the equality holds for any v such that both levels are finite.

By the definition of $\mathtt{complete}$, any prefix of S may be computed by executing a limited number of steps of $\mathtt{complete}(P)$, where P is any smaller prefix of S, while limiting the selection of elements to add to the current solution to only elements that are part of S.

Thanks to the definition of $\mathtt{complete}(S)$ and the definition of level ordering, we obtain the following important lemma.

Lemma 3.2. *Given a feasible set* S*, if* Q *is any maximal feasible set of* $\mathcal{F}$ *such that* $S \subseteq Q$ *and* $\mathtt{seed}(S) = \mathtt{seed}(Q)$*, we have* $\mathtt{complete}(S) \preceq Q$.

Proof. If $\mathtt{seed}(\mathtt{complete}(S)) \neq \mathtt{seed}(S)$, then the thesis clearly follows, as the seed is the only level 0 element. Moreover, if $Q = \mathtt{complete}(S)$, there is nothing to prove.

Let now $i > 0$ be the smallest level such that the set of elements of level i in Q is different from the set of elements of level i in $\mathtt{complete}(S)$. Moreover, let v be the smallest element that belongs to exactly one of these two sets.

We will now prove that v does, in fact, belong to level i of $\mathtt{complete}(S)$. Let X be the union of all levels smaller than i in Q, as well as of all elements on level i that are smaller than Q. Thanks to the definitions of levels and to the alternative definition of commutable set systems (Definition 2.6), we have that $X \cup \{v\}$ is feasible, as it may be obtained as the union of feasible sets all containing $\mathtt{seed}(Q)$ and all contained in Q. Moreover, since S is feasible, is contained in Q and contains $\mathtt{seed}(Q)$, we have that $X \cup \{v\} \cup S$ is feasible. Since it can be easily seen that $X \cup S$ is one of the sets obtained during the iterations that build $\mathtt{complete}(S)$, we have that v was considered for adding during that iteration. As v was chosen as the smallest node in the difference between level i of the two sets, we know that at that iteration no node smaller than v was viable, and so v was added to $X \cup S$ by $\mathtt{complete}$.

Now, if v is bigger than all the elements in level i of Q, we have that level i of $\mathtt{complete}(S)$ is a superset of level i of Q. Since Q is maximal, it cannot be a proper subset of $\mathtt{complete}(S)$, so it must have at least one element on level $i+1$. Otherwise, level i of $\mathtt{complete}(S)$ compares smaller than level i of Q, so by the definition of level order it follows that $\mathtt{complete}(S) \prec Q$.

□

All this machinery allows us to definte the concepts used for executing the reverse search.

Definition 3.4. If a maximal solution S is such that $\mathtt{complete}(\mathtt{seed}(S)) = S$, we call it a $\mathtt{root}$. If S is not a $\mathtt{root}$, we call one of its nonempty prefixes $\mathtt{core}(S)$ if it is the longest prefix such that $\mathtt{complete}(\mathtt{core}(S)) \neq S$. Moreover, in that case the smallest element of $S \setminus \mathtt{core}(S)$ (according to the level order) is called parent index of S (denoted by $\mathtt{parind}(S)$) and $\mathtt{complete}(\mathtt{core}(S))$ is called $\mathtt{parent}(S)$.

Note that Lemma 3.2 shows that, for any non-$\mathtt{root}$ maximal solution, $\mathtt{parent}(S) \prec S$. This property will be crucial when proving that the solution graph defined by $\mathtt{parent}$ does not contain any directed cycles.

3.2 Algorithm

This section will first outline the reverse search algorithm employed by the framework and will then provide further details on how the graph of the reverse search is generated.

3.2.1 Reverse search explained

We will now describe the graph that is explored by the reverse search algorithm used by the framework. While doing this, we use the definition of $\mathtt{parent}$ from the Section 3.1, but the results are valid for any definition of $\mathtt{parent}$ that satisfies $\mathtt{parent}(S) \prec S$ for some order $\prec$.

Consider the graph $\mathcal{R}$ that has all the maximal sets of $\mathcal{F}$ as nodes. The arcs of $\mathcal{R}$ are defined as follows: for every maximal non-$\mathtt{root}$ solution S, we have an arc from $\mathtt{parent}(S)$ to S.

Thanks to the fact that $\mathtt{parent}(S) \prec S$ and to the properties of orders, we know that the arcs of $\mathcal{R}$ cannot contain any directed cycle. Moreover, as every node has at most one ingoing edge, there can be no undirected cycle either, as otherwise one node would have at least two ingoing edges.

Since $\mathcal{R}$ contains no cycles, it must be a forest. Let us now consider any maximal subtree in this forest. Its root has no ingoing edge, so it is, in fact, a maximal solution with no parent (i.e. a $\mathtt{root}$). Moreover, any other solution in that subtree is reachable from its $\mathtt{root}$: since a $\mathtt{root}$ is the only node in that tree that has no parent, walking back from S to $\mathtt{parent}(S)$ as long as it is possible must eventually reach the $\mathtt{root}$.

Let now S be any maximal solution. We define $\mathtt{children}(S)$ to be the family of maximal sets such that, for any maximal solution Q, we have that the parent of Q is in S if and only if Q belongs to $\mathtt{children}(S)$.

It is now easy to define an algorithm that enumerates all the maximal solutions in a commutable set system $\mathcal{F}$: simply start the explorations from the root of every forest, exploring every tree in a depth-first manner. Pseudocode for this algorithm is shown in Algorithm 1, which is a simple recursive visit on a directed forest. A more complicated algorithm, that avoids using a recursion stack (thus potentially improving memory usage), can be found in Algorithm 2. More in detail, this algorithm performs a depth-first search that jumps from a node to the next without storing anywhere what the last node was. So, to find the next node we must first check if the current node has some child that is yet unvisited (this can be obtained by visiting the children in some deterministic order, thus allowing to skip all the children that have already been visited just by knowing the last explored node). If that is not the case, then the algorithm has to climb up in the tree, by executing the $\mathtt{parent}$ function.

Note that in some cases the parent of a given maximal solution, or the exact set of children, may be hard to compute. We suppose that, instead of $\mathtt{children}$, we have a similar $\mathtt{candidates}$ function available, that satisfies $\mathtt{candidates}(S) \supseteq \mathtt{children}(S)$. In

Algorithm 1: Reverse search algorithm

```
for every solution $S$ that is a $\mathtt{root}$ do
    $\mathtt{spawn}(S)$
end
Function $\mathtt{spawn}(P)$
    for $S \in \mathtt{children}(P)$ do
        $\mathtt{spawn}(S)$
    end
```

this case, we may define a graph $\mathcal{S}$ that has the same nodes of $\mathcal{R}$ but different arcs. In particular, the arcs of $\mathcal{S}$ are defined as follows: for any pair of maximal solutions S, Q, we have an arc from S to Q if and only if $Q \in \mathtt{candidates}(S)$. Clearly $\mathcal{R}$ is a subgraph of $\mathcal{S}$, so an exploration of this graph starting from all the $\mathtt{roots}$ will report all the maximal solutions. The issue of avoiding duplication may be solved by keeping a global set of all solutions discovered so far. The pseudocode for this approach (that corresponds to a "traditional" implementation of a depth-first search on a generic graph) can be found in Algorithm 3 and is the one used to enumerate maximal sets that satisfy connected hereditary properties in [10].

Finally, note that by using the technique of *alternative output*, as described in [39], that is by outputting a solution when going down in the computational tree if the current height is even and when going up otherwise, we obtain algorithms whose maximum **delay** is given by the cost of running the algorithm for $\mathtt{children}$.

3.2.2 Computing $\mathtt{children}(S)$

We will now explain how to compute the children of a given maximal solution S of a commutable set system $\mathcal{F}$. As this cannot be done in the generic case in sub-exponential time (since doing so would give a polynomial total time algorithm for the enumeration of any commutable set system, in violation of Theorem 2.1), we will use the concept of restricted problem from Definition 2.15 as a blackbox to aid us in the search of the children of S. Moreover, when doing so some care will be necessary to avoid generating the same child multiple times.

The following key fact will give us an the algorithm that generates $\mathtt{children}(S)$, as it gives us a reasonably-sized set of candidates that is assured to contain all the children of S.

Theorem 3.1. *Given a maximal solution C such that $\mathtt{parent}(C) = S$, we have that $\mathtt{core}(C)$ is the prefix ending just before $\mathtt{parind}(C)$ of a solution of $\mathcal{P}(S, \mathtt{parind}(C))$.*

Proof. Let v be $\mathtt{parind}(C)$ and let Cv be $\mathtt{core}(C) \cup \{v\}$. We will first prove that $v \notin S$.

Suppose by contraddiction $v \in S$. Since we know $\mathtt{parent}(C) = \mathtt{complete}(\mathtt{core}(C))$, let y be the first element that is added to $\mathtt{core}(C)$ by $\mathtt{complete}$. Clearly, it cannot

Algorithm 2: Reverse search algorithm without recursion

```
for every solution $S$ that is a root do
    $\mathtt{Last} \leftarrow \varnothing$;
    loop
        $\mathtt{Child} \leftarrow \varnothing$;
        for $C \in \mathtt{children}(S)$ that comes after $\mathtt{Last}$ do
            $\mathtt{Child} \leftarrow C$;
            break;
        end
        if $\mathtt{Child} = \varnothing$ then
            if $S$ is a root then
                break;
            end
            $\mathtt{Last} \leftarrow S$;
            $S \leftarrow \mathtt{parent}(S)$;
        else
            $\mathtt{Last} \leftarrow \varnothing$;
            $S \leftarrow \mathtt{Child}$;
        end
    end
end
```

be v, as otherwise we would have $\mathtt{parent}(C) = \mathtt{complete}(Cv) = C$, that contraddicts the definition of parent. By the definition of $\mathtt{complete}$, we immediately know that $(\mathtt{level}_{\mathtt{core}(C)}(y), y) < (\mathtt{level}_{\mathtt{core}(C)}(v), v)$ and that, as $\mathtt{core}(C)$ is a prefix of C, $(\mathtt{level}_C(y), y) < (\mathtt{level}_C(v), v)$. Thus, $y \notin C$, as that would mean that v is not the element immediately after $\mathtt{core}(C)$ in C.

By Definition 2.6, we know that, since both Cv and $\mathtt{core}(C) \cup \{y\}$ belong to $\mathcal{F}$, $\mathtt{core}(C) \cup \{v, y\}$ belongs to $\mathcal{F}$ too. Thus, y is a viable choice for $\mathtt{complete}$ when it is expanding the set Cv. As any element that is a viable candidate for Cv but not for $\mathtt{core}(C)$ must have a level of $\mathtt{level}_{Cv}(v) = \mathtt{level}_{\mathtt{core}(C)}(v) + 1$, then $(\mathtt{level}_{Cv}(y), y)$ is still the lowest level-value pair among the viable nodes for $\mathtt{complete}(Cv)$, and so it will be chosen as the next element. This implies that $\mathtt{complete}(Cv)$ contains y, which contraddicts $y \notin C$ since, by the definition of core, we know $\mathtt{complete}(Cv) = C$. This proves that $v \notin S$.

As $Cv \in \mathcal{F}$, we have that $Cv \in \mathcal{G}_S^v$. So Cv must be contained in a maximal solution of $\mathcal{G}_S^v$, i.e. in a solution of $\mathcal{P}(S, v)$. We will call this solution R.

We now need to prove that Cv is indeed a prefix of R with respect to s, the canonical seed of C. If $C = Cv$, then there is nothing to prove, as then $R \supseteq C$ and so $R = C$ (since

Algorithm 3: Reverse search algorithm with exponential memory

```
$\mathcal{S} = \varnothing$;
for every solution $S$ that is a $\mathtt{root}$ do
    $\mathtt{spawn}(S)$
end
Function $\mathtt{spawn}(P)$
    if $S \in \mathcal{S}$ then
        return;
    end
    add $S$ to $\mathcal{S}$;
    for $S \in \mathtt{children}(P)$ do
        $\mathtt{spawn}(S)$
    end
```

C is maximal).

Otherwise, R strictly contains Cv. Let r be the smallest element in $R \setminus Cv$ and suppose by contraddiction that $(\mathtt{level}^s_R(r), r) < (\mathtt{level}^s_R(v), v)$. Then we also have that $(\mathtt{level}^s_R(r), r) < (\mathtt{level}^s_{Cv}(v), v)$, thanks to the fact that the level may not decrease when taking subsets.

Let us denote by Rp the prefix of R that ends just before r: note that Rp is a prefix of C. Moreover, by Definition 2.6, we know that $Cv \cup \{r\}$ is in $\mathcal{F}$ too, as both Cv and $Rp \cup \{r\}$ are and they both contain Rp.

Let y be the element of C that immediately follows Cv in C. Then, by the definition of prefix, we know that $(\mathtt{level}^s_C(v), v) < (\mathtt{level}^s_C(y), y)$. Since the level of a node does not change by taking prefixes (as long as it does not become infinite), it follows that $(\mathtt{level}^s_{Cv}(v), v) < (\mathtt{level}^s_{Cv}(y), y)$. Moreover, we also have the following equality:

$$\mathtt{level}^s_R(r) = \mathtt{level}^s_{Rp}(r) = \mathtt{level}^s_{Cv}(r)$$

Putting all this together allows us to conclude that

$$(\mathtt{level}^s_{Cv}(r), r) < (\mathtt{level}^s_{Cv}(y), y)$$

that imples that the first step of $\mathtt{complete}(Cv)$ chooses r, so that $\mathtt{complete}(Cv) \neq C$, contraddicting the definition of core. This proves that Cv is indeed a prefix of R. □

We will denote with $R(C)$ the solution of $\mathcal{P}(\mathtt{parent}(C), \mathtt{parind}(C))$ (according to level order) that is obtained by running $\mathtt{complete}(\mathtt{core}(C) \cup \{\mathtt{parind}(C)\})$ using only candidates from $\mathtt{parent}(C) \cup \{\mathtt{parind}(C)\}$.

We now know that any child C of S may be found by first chosing the node v that should be $\mathtt{parind}(C)$, then trying all the possible solutions R of $\mathcal{P}(S, v)$ as candidates for $R(C)$, then checking if the prefix with respect to any seed of C ending just before

v is, in fact, the core of a child. To avoid duplication, we check that the tuple (v, R, s) choosen to generate the child was indeed the one that can be found from C itself as $(\mathtt{parind}(C), R(C), \mathtt{seed}(C))$. Since we have proven that when that specific tuple is chosen then C is obtained by the previous procedure, this argument shows the correctness of the algorithm, whose pseudocode can be found in Algorithm 4.

Algorithm 4: Children generation algorithm

```
Function children(S)
    for $v \in E \setminus S$ do
        for R solution of $\mathcal{P}(S, v)$ different from S do
            for s possible seed of R do
                $Cv \leftarrow$ the prefix of R with respect to s ending with $v$;
                $C \leftarrow \mathtt{complete}(Cv)$;
                if $\mathtt{parind}(C) = v$ and $\mathtt{parent}(C) = S$ and $R = R(C)$ and
                 $\mathtt{seed}(C) = s$ then
                    yield C;
                end
            end
        end
    end
```

Note that some speed-ups may be obtained by choosing v from a smaller set, as long as any v such that $\{v\}$ is not the only non-trivial solution to $\mathcal{P}(S, v)$ belongs to that set. This is because $\mathtt{complete}(\{v\})$ is either a root or does not satisfy $\mathtt{seed}(\mathtt{complete}(\{v\})) = v$.

3.2.3 An algorithm that does not require the restricted problem

In Subsection 3.2.2, the restricted problem is only used as a tool to find the cores of the children of the current node. Another way to reach the same objective is trying all the possible subsets of $S \cup \{v\}$. This algorithm would be slower in the majority of cases, but requires no knowledge of the problem to be applied. Moreover, with slight modifications to the definition of level it can be used on set systems that are not commutable but only strongly accessible.

More precisely, we may define the level of a node as follows:

- If $v \notin S$, then $\mathtt{level}(v) = |S|$.
- If $v \in S$, then run $\mathtt{complete}(\{s\})$ while considering only candidates from S. Let S' be the solution such that the next iteration of $\mathtt{complete}$ adds v. Then $\mathtt{level}(v) = |S'|$.

When replacing Definition 3.2 with this one, both $\mathtt{complete}$ and $\mathtt{parent}$ are still well-defined, and the same proof for Lemma 3.2 still holds. We thus obtain that the algorithms of Subsection 3.2.1 may be used to enumerate any strongly accessible set system.

3.2.4 Analysis

The running time of Algorithm 4 critically depends on the cost of solving $\mathcal{P}$. The following lemma will tie together α and the number of solutions of $\mathcal{P}$.

Lemma 3.3. *The number of solutions of $\mathcal{P}(S, v)$ is at most $(|S| + 1)\alpha$.*

Proof. Let us fix a node s of $S \cup \{v\}$ and consider all the solutions of $\mathcal{P}(S, v)$ that have s as a seed. We will prove that there are at most α such solutions: as s may be chosen in $|S| + 1$ ways, this proves the thesis.

Indeed, let us suppose by contraddiction that there are more that α solutions of $\mathcal{P}(S, v)$ that contain s. As any solution of $\mathcal{P}(S, v)$ is contained in a maximal solution of $\mathcal{F}$, by the pidgeonhole principle, at least two such solutions (say Q and R) must be contained in the same maximal solution B of $\mathcal{F}$. From Definition 2.6 then follows that, since Q and R belong to $\mathcal{F}$, are contained in an element of $\mathcal{F}$ and both contain $\{s\}$, another viable solution, $Q \cup R \in \mathcal{F}$ too. But this contraddicts the maximality of Q and R in $\mathcal{G}_S^v$, so the thesis is proven. □

This fact, combined with the algorithm explained above, gives us the following theorem:

Theorem 3.2. *Let $\mathcal{F}$ be a commutable set system over a set U. If the restricted problem can be solved in polynomial time (resp. polynomial total time), then the maximal solutions of $\mathcal{F}$ may be enumerated with polynomial delay (resp. in polynomial total time).*

Note that algorithms obtained with this framework have a slightly higher computational cost with respect to the corresponding ones given by the framework in [10]. However, algorithms obtained with our framework have an huge advantage in a parallelized or distributed settings, as they do not require a centralized set that stores all the maximal solutions generated so far. Such a set would create a huge bottleneck, as it would be the source of a lot of inherently sequential work, thus limiting the maximum speed-up that can be obtained. In contrast, our framework requires almost no coordination, with the exception of the work required to split the forest in roughly equal-sized chunks to be assigned to each worker.

Let us now consider the variant of the algorithm that does an exaustive search of the possible $\mathtt{cores}$ of the children. That algorithm gives us a delay bounded by $O(P(n)2^q)$, which is the square of the lower bound given in Corollary 2.1.

This bound gives us the following corollary:

Corollary 3.1. *Let $\mathcal{F}$ be a commutable set system over a set U such that any element $S \in \mathcal{F}$ has size at most $q = O(\log(n))$. Then all the maximal solutions of $\mathcal{F}$ may be enumerated with polynomial delay.*

Note that the trivial algorithm (try all the sets with size up to q) does not guarantee this, as its running time is $O(n^q) = O(n^{c \log n})$, that is super-polynomial.

Finally, we will spend a couple words on (possible) implementations of $\mathtt{complete}$ and $\mathtt{parent}$ and their runtime costs. Suppose that determining the level of a node takes $\mathcal{L}(q)$ time, and that we know that the viable elements to be added belong to a set A of size at most $\mathcal{A}$ that can be listed in $O(\mathcal{A})$ time. Then, $\mathtt{complete}$ may be easily implemented by scanning A, finding the level of each element, adding to the current solution the one with the lowest level-value pair and repeating. The total cost of each of these steps is $O(\mathcal{A}\mathcal{L}(q))$, and there is a total of $O(q)$ steps, so the total running time of $\mathtt{complete}$ is $O(q\mathcal{A}\mathcal{L}(q))$.

As for $\mathtt{parent}$, we can proceed by removing an element v from the current solution and executing one step of $\mathtt{complete}$. If this step adds v back, then v was not the $\mathtt{parind}$ and we need to remove another element, otherwise what we have left is the $\mathtt{core}$ of the solution and we just need to end the execution of $\mathtt{complete}$. This algorithm has the same running time as the one above, i.e. $O(q\mathcal{A}\mathcal{L}(q))$.

We may summarize these results with the following lemma.

Lemma 3.4. *If we may find the level of a node in $\mathcal{L}(q)$ time, and we know that the elements that may be added to a partial solution can be iterated on using at most $O(\mathcal{A})$ total time, then both* $\mathtt{complete}$ *and* $\mathtt{parent}$ *may be implemented to run in $O(q\mathcal{A}\mathcal{L}(q))$ time, while using only $O(q)$ memory.*

Chapter 4

Applications of the framework

This chapter will show some applications of the algorithmic framework of Section 3.2 to problems that were presented in Section 2.2. In particular, the main focus will be on solving the restricted problem, as in many cases this is not trivial.

4.1 Cliques and (connected) k-plexes

Applying the framework to cliques is quite straightforward, since the restricted problem is very simple: if C is a clique, then $C \cup \{v\}$ has C and $C \cap N(v) \cup \{v\}$ as the only maximal cliques. As the restricted problem may be solved in polynomial time, the maximal clique problem may be solved with polynomial delay.

This is a well-known result, as seen for example in [13].

Solving the restricted problem for maximal k-plexes is not so simple: in fact, it may have an exponential number of solution. In [5], the authors find a polynomial-delay algorithm (assuming k to be a constant) to generate all the solutions of the restricted problem.

Applying this solution of the restricted problem to our framework gives a polynomial total time algorithm to find all the maximal k-plexes in a graph, a weaker result than the one in [5]. Because the delay between two solutions of the restricted problem is not very useful in our framework, we will give an upper bound on the number of solutions of the restricted problem and an algorithm to compute them.

Lemma 4.1. *Assuming k to be a constant, if S is a k-plex and v is a node in $V \setminus S$, then there are at most $1 + f(k)|S|^{k-1}$ maximal k-plexes in $S \cup \{v\}$, with $f(k) = (k-1)^{2k}$ for $k > 1$ and $f(1) = 1$. Moreover, they can be computed in $O(kf(k)|S|^k)$ time using only $O(kq)$ memory.*

Proof. During the proof, we will use the following upper bound on the sum of certain binomial coefficients, that can easily be derived from the fact that $\binom{n}{i} \leqslant n^i$:

$$\sum_{i=0}^{k} \binom{n}{i} \leqslant kn^k \qquad \forall n > 0$$

S is clearly a maximal k-plex in $S \cup \{v\}$. From now on, we will only consider maximal k-plexes that include v. Any of those maximal k-plexes may contain up to $k-1$ elements of S that are not neighbours of v. So, we can partition the maximal k-plexes that contain v according to the set $\tilde{N}$ of non-neighbours of v that they contain.

Consider now $K(\tilde{N}) = \{v\} \cup \tilde{N} \cup (N(v) \cap S)$. If it is not a k-plex, then it cannot be because of any element in $N(v)$: indeed, the number of non-neighbours that these elements have in $K(\tilde{N})$ is at most the number of non-neighbours that they have in S, and so it is at most k. Moreover, it cannot be due to v either, as we know $|\tilde{N}| \leqslant k-1$.

So any element x that breaks the k-plex constraint must be in $\tilde{N}$. Call $\tilde{N}_b$ the set of such xs. The constraint may only be broken by having $k+1$ non-neighbours, as v is the only element in $K(\tilde{N})$ that was not in S. Since we are only interested in k-plexes that contain v, the only way to fix this issue for any $x \in \tilde{N}_b$ is by removing from $K(\tilde{N})$ some element that belongs to $B(x) = \tilde{N}(v) \cap S \setminus N(x)$. As we need to do this for all the x in $\tilde{N}_b$, we need to find some set X such that:

- $X \subseteq \bigcup_{x \in \tilde{N}_b} B(x)$,
- $X \cup B(x) \neq \varnothing$ for any x in $\tilde{N}_b$,
- X is minimal among all the sets that satisfy the first two properties, as otherwise the resulting k-plex would not be maximal.

We will now prove that any minimal X satisfies $|X| \leqslant |\tilde{N}_b| \leqslant k-1$. Let $f(y)$ be the function that maps any element of X into the set of $B(x)$ such that $B(x) \cap X = \{y\}$. We will now prove that if $|X| > |N_b|$ then there is an y for which $f(y) = \varnothing$. Indeed, if it were not the case, we would have that, since the various $f(y)$ are disjoint, $|\bigcup_{y \in X} f(y)| > |\tilde{N}_b|$. But this is a contraddiction, as the number of $B(x)$ is smaller than $|\tilde{N}_b|$ because of their definition. So there is an y such that $f(y) = \varnothing$. Removing such an y would not alter the number of sets covered by X, and so we have proven that X is not minimal.

So, we have that X must be a set of size at most $|\tilde{N}_b|$ contained in a set of size at most $|N_b| \times (k-1) \leqslant (k-1)^2$.

Taking these results together, we have that the number of maximal k-plexes in $S \cup \{v\}$ containing v is at most

$$\sum_{i=0}^{k-1} \binom{|S|}{i} \sum_{i=0}^{k-1} \binom{(k-1)^2}{i} \leqslant |S|^{k-1}(k-1)^{2k}$$

Finally, since these sets can be easily enumerated in $O(|S|)$ time per set, it is enough to show that we may check if they are a k-plex and their maximality in $O(k|S|)$ time.

Indeed, given a preprocessing time of $O(|S|^2)$ (that reduces to $O(|S|)$ if $k = 1$), due to the computation of the set of non-neighbours for every node in S, we may check if a set is a k-plex in $O(k|S|)$ time; moreover we can check if extending a given k-plex with one extra node still gives a k-plex in $O(k)$ time per node (since we only need to check that node and its neighbours). This completes the proof.

□

Unfortunately, there is no easy bound on the size of the candidate set for k-plexes (either for $\mathtt{complete}$ or for the choice of v in Algorithm 4), so the resulting enumeration algorithm still has a time complexity of $O(n^2 q^{k+2} f(k)\alpha)$, which is obtained as the n (for the choice of v) times the number of solutions of the restricted problem $O(f(k)q^{k-1})$ times q (the number of choices for a seed in every solution) times the cost of running $\mathtt{complete}$ or $\mathtt{parent}$, which according to Lemma 3.4 is given by the cost of checking the level of a node ($O(q)$ by updating it from one iteration to the next) times q times $\mathcal{A} = O(n)$. Thus running $\mathtt{complete}$ takes $O(nq^2)$. Multiplying all these values together gives the cost of computing $\mathtt{children}$ as $O(n^2 q^{k+2} f(k))$. As this is basically the only cost in the reverse search, we obtain a delay of $O(n^2 q^{k+2} f(k))$ and thus the given total time.

A more interesting problem is the one of enumerating connected k-plexes. This problem was studied in [5] too, where a polynomial-delay algorithm for the restricted problem was given. The resulting algorithm using their framework takes incremental polynomial time but uses additional exponential memory. With our framework, we obtain a polynomial-delay algorithm (for fixed k) that only takes $O(kq)$ memory:

Theorem 4.1. *All the connected k-plexes in a graph G can be enumerated with $O(q^{k+4}\Delta^2 f(k))$ delay, using only $O(kq)$ extra memory (other than the memory used to store G).*

The theorem holds because we can restrict our choice of candidates to nodes that only have at least one neighbour in the current solution, thus replacing all the instances of n in the running time of the enumeration of k-plexes with $q\Delta$. Moreover, we may re-use Lemma 4.1 with the following simple observation: any connected k-plex must be contained in a "normal" k-plex, and may be obtained from it by extracting the connected component containing v. Moreover, duplicates may be avoided by choosing an easily-computable canonical k-plex that should generate a given maximal connected k-plex (for example, the one obtained by adding iteratively adding the smallest element in the input to the restricted problem that would keep the current set a k-plex). So, when we generate a connected k-plex C from a given k-plex K, we check if K is the canonical k-plex associated to C and, if it is not, we discard C, as we know that it will be (or has already been) generated when K is the canonical k-plex.

4.2 Black-connected cliques

In the case of black-connected cliques, the restricted problem $\mathcal{P}(S, v)$ is easy to solve: it is enough to find the only maximal clique in $S \cup \{v\}$ that is not S and to remove any vertex from it that is not in the same black-connected component of v. This takes $O(q)$ time and gives exactly one solution.

Let us now consider $\mathtt{complete}$ and $\mathtt{parent}$. We will denote by Δ_b the maximum "black degree" of a node in G. To execute a step of $\mathtt{complete}$, it is enough to consider the elements of G that are black neighbours of a node in the current solution. There are at most $q\Delta_b$ such elements, and it is easy to check if one such element may be added to the solution in $O(q)$ time. Thanks to Lemma 3.4, the execution of both $\mathtt{complete}$ and $\mathtt{parent}$ takes at most $O(q^3\Delta_b)$ time.

Moreover, note that in Algorithm 4 we only need to consider nodes that are black neighbours of a node in the current solution as a possible v, as otherwise the only non-trivial solution of the restricted problem would be $\{v\}$ itself. With a calculation similar to the one that gave us the bound for connected k-plexes, this gives the following result:

Theorem 4.2. *All the maximal black-connected cliques in a graph G may be enumerated with $O(q^5\Delta_b^2)$ delay, using only $O(q)$ extra memory (other than the memory used to store G).*

4.3 Connected bipartite (induced) subgraphs

A bipartite subgraph can be equivalently seen as a graph that is two-colorable, or as a graph that contains no cycle with odd length.

We will first consider non-induced subgraphs. In this case, when solving the restricted problem we have a bipartite graph with one extra edge e that connects two nodes of the same color, as otherwise the solution that generates the restricted problem would not be maximal. Any non-trivial subset of the edges in the restricted problem that gives a solution must then be missing enough edges to disconnect these two nodes in the original bipartite graph, i.e. the set of removed edge must be a cut in the bipartite graph. Moreover, since we are interested in maximal solutions, the cut must be minimal. Note that, since minimal cuts divide the graph into two connected components, and that adding e to the graph connects these two components, the connectivity constraint is not an issue. Since enumeration of minimal cuts can be done in polynomial total time, as in [31], the restricted problem may be solved in polynomial total time.

For induced subgraphs, the extra node v will have some neighbours of one color and some of the other. We will call these two sets of neighbours 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 in such a way that either

- B becomes empty, or

- W becomes empty, or
- there is no path from any node in B to any node in W

This can be done by enumerating the minimal vertex-cuts between two fake nodes b and w, where b (w) have all B (W) as neighbours (respectively). As before, removing minimal vertex-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 vertex-cuts can be enumerated in polynomial total time, as in [37].

Putting it all together, we obtain the following result.

Theorem 4.3. *All the maximal connected bipartite (induced) subgraphs in* G *may be enumerated in polynomial total time, using only polynomial space.*

4.4 Feedback vertex (arc) sets

As with bipartite subgraphs, the restricted problem requires one to break all the cycles that are formed in a graph with no cycles by adding a single node or a single arc. Thus, it may be solved by enumerating maximal cuts or vertex-cuts in the original graph. As vertex cuts in undirected graphs may be enumerated in polynomial total time (see [37]), and edge cuts may be enumerated in polynomial total time both in undirected and directed graphs (see [31]), we have the following result.

Theorem 4.4. *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.*

Chapter 5

Conclusions

This thesis analyzed known results regarding the enumeration of set systems, discussing various classes of set systems (independence systems, commutable set systems, (strongly) accessible set systems).

Despite the hardness result given in Theorem 2.1, many techniques have succeded in producing efficient algorithms, in practice if not in theory (for example [16]). In particular, we focused on binary partition, which was presented in Section 2.4, and on reverse search, initially presented in Section 2.4 and presented more in-depth in Section 3.2.1.

We then explained our new algorithmic framework, based on the technique of reverse search, that achieves under suitable conditions polynomial total time enumeration of commutable set systems while using a limited amount of memory. More precisely, our framework allows us to transform, in a space-efficient way, an algorithm for the enumeration of maximal solutions to a restricted problem that consists in enumerating the maximal solutions in a set obtained by adding an extra element to a given maximal solution.

The runtime performance of this algorithm influences heavily the running time of the algorithm obtained by applying our framework: a polynomial algorithm for the restricted problem gives a polynomial delay for the general enumeration, while a polynomial total time algorithm for the restricted problem translates to a polynomial total time algorithm for the general problem (this last implication clearly also holds in the other direction).

Thus, we studied in Chapter 4 how to solve the restricted problem in various commutable set systems. This analysis is by no means exhaustive, and a possible direction for future work would be to solve the restricted problem for more set systems, or even to find some general techniques related to restricted problems.

Other interesting extensions to this work could be related to the performance of the algorithm obtained. This could be done both on a theoretical aspect, by studying further properties of these families in order to lower the time complexity of the algorithm, and on a practical aspect, as the algorithm should be easy to implement in a distributed manner. Indeed, we expect that the ability to easily implement a distributed version of the algorithm will make the biggest difference in practice between this framework and previous work.

Bibliography

[1] N. K. Ahmed, J. Neville, R. A. Rossi, and N. Duffield. Efficient graphlet counting for large networks. In *ICDM*, pages 1–10. IEEE, 2015.

[2] R. Angles and C. Gutierrez. Survey of graph database models. *CSUR*, 40(1):1, 2008.

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

[4] H. G. Barrow and R. M. Burstall. Subgraph isomorphism, matching relational structures and maximal cliques. *Information Processing Letters*, 4(4):83–84, 1976.

[5] D. Berlowitz, S. Cohen, and B. Kimelfeld. Efficient enumeration of maximal k-plexes. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 431–444. ACM, 2015.

[6] M. Boley, T. Horváth, A. Poigné, and S. Wrobel. Listing closed sets of strongly accessible set systems with applications to data mining. *Theoretical Computer Science*, 411(3):691–700, 2010.

[7] C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

[8] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J Comput*, 14(1):210–223, 1985.

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

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

[11] S. Cohen and Y. Sagiv. An abstract framework for generating maximal answers to queries. In *ICDT*, pages 129–143. Springer, 2005.

[12] A. Conte, R. Grossi, A. Marino, L. Tattini, and L. Versari. A fast algorithm for large common connected induced subgraphs. *Algorithms for Computational Biology*, page 62.

[13] A. Conte, R. Grossi, A. Marino, and L. Versari. Sublinear-space bounded-delay enumeration for massive network analytics: Maximal cliques. In *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, volume 148, pages 1–148, 2016.

[14] W. H. Day and D. Sankoff. Computational complexity of inferring phylogenies by compatibility. *Systematic Biology*, 35(2):224–229, 1986.

[15] N. Du, B. Wu, X. Pei, B. Wang, and L. 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.

[16] D. Eppstein and D. Strash. Listing all maximal cliques in large sparse real-world graphs. *Experimental Algorithms*, pages 364–375, 2011.

[17] L. A. Goldberg. *Efficient algorithms for listing combinatorial structures*, volume 5. Cambridge University Press, 1992.

[18] F. Harary and I. C. Ross. A procedure for clique detection using the group matrix. *Sociometry*, 20(3):205–215, 1957.

[19] R. Impagliazzo and R. Paturi. Complexity of k-SAT. In *Computational Complexity, 1999. Proceedings. Fourteenth Annual IEEE Conference on*, pages 237–240. IEEE, 1999.

[20] D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou. On generating all maximal independent sets. *Inform Process Lett*, 27(3):119 – 123, 1988.

[21] C. Klein, A. Marino, M.-F. Sagot, P. V. Milreu, and M. Brilli. Structural and dynamical analysis of biological networks. *Brief Funct Genomics*, 11(6):420–433, 2012.

[22] I. Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theoretical Computer Science*, 250(1-2):1–30, 2001.

[23] F. S. Kuhl, G. M. Crippen, and D. K. Friesen. A combinatorial algorithm for calculating ligand binding. *Journal of Computational Chemistry*, 5(1):24–34, 1984.

[24] W. Kunhiro. Enumeration of enumeration algorithms and its complexity.

[25] V. Lacroix, L. Cottret, P. Thébault, and M.-F. Sagot. An introduction to metabolic networks and their structural analysis. *IEEE ACM T Comput Bi*, 5(4):594–617, 2008.

[26] E. L. Lawler, J. K. Lenstra, and A. Rinnooy Kan. Generating all maximal independent sets: NP-hardness and polynomial-time algorithms. *SIAM Journal on Computing*, 9(3):558–565, 1980.

[27] V. E. Lee, N. Ruan, R. Jin, and C. Aggarwal. A survey of algorithms for dense subgraph discovery. In *Managing and Mining Graph Data*, pages 303–336. Springer, 2010.

[28] P. V. Milreu, C. C. Klein, L. Cottret, V. Acuña, E. Birmelé, M. Borassi, C. Junot, A. Marchetti-Spaccamela, A. Marino, L. 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.

[29] N. Modani and K. Dey. Large maximal cliques enumeration in sparse graphs. In *CIKM*, pages 1377–1378. ACM, 2008.

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

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

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

[33] N. Rhodes, P. Willett, A. Calvet, J. B. Dunbar, and C. Humblet. Clip: similarity searching of 3d databases using clique detection. *Journal of chemical information and computer sciences*, 43(2):443–448, 2003.

[34] F. Ruskey. Combinatorial generation. *Preliminary working draft. University of Victoria, Victoria, BC, Canada*, 11:20, 2003.

[35] R. Samudrala and J. Moult. A graph-theoretic algorithm for comparative modeling of protein structure. *Journal of molecular biology*, 279(1):287–302, 1998.

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

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

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

[39] U. Takeaki. Two general methods to reduce delay and change of enumeration algorithms. 2003.

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

[41] E. Tomita, A. Tanaka, and H. Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical Computer Science*, 363(1):28 – 42, 2006. Computing and Combinatorics.

[42] C. Tsourakakis, F. Bonchi, A. Gionis, F. Gullo, and M. Tsiarli. Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees. In *SIGKDD*, pages 104–112. ACM, 2013.

[43] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM J Comput*, 6(3):505–517, 1977.

[44] L. G. Valiant. The complexity of computing the permanent. *Theor Comput Sci*, 8(2):189–201, 1979.

[45] S. Wasserman and K. Faust. *Social network analysis: Methods and applications*, volume 8. Cambridge university press, 1994.

[46] D. W. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In *ICDE*, pages 976–985. IEEE, 2007.

[47] 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.

Appendix A

Code for the framework

This appendix contains a simple implementation of the enumeration framework described in this thesis. It contains the generic implementations of the functions described in Chapter 3, plus some specific implementations of the functions needed to apply the framework to black-connected cliques in graphs obtained by computing the product of two connected graphs.

The implementation provided here is not as memory-efficient as the one described in the rest of the work, but improves a bit the running time of the algorithm. More in detail, for black-connected cliques, we obtain an algorithm that has a delay of $O(q^4\Delta_b^2)$ while using $O(q\Delta_b)$ memory per recursive node. For simpicity, only the recursive version was implemented.

A.1 Generic algorithm and graph data structures

A.1.1 `framework.hpp`

This file contains the implementation of the functions defined in Chapter 3.

```
#ifndef FRAMEWORK_HPP
#define FRAMEWORK_HPP
#include <vector>
#include <cstdint>
#include <functional>
#include <algorithm>
#include <memory>
#include <set>
#include <unordered_map>
#include <unordered_set>

template <typename univ_t>
class CommutableSystem {
public:
    const univ_t& e;

    CommutableSystem(const univ_t& e): e(e) {}

    typedef typename univ_t::elem_t elem_t;
    /**
     * Checks if a given subset is a solution.
     */
    virtual bool is_good(const std::vector<elem_t>& s) = 0;

    /**
     * Solves the restricted problem
     */
    virtual void restricted_problem(
        const std::vector<elem_t>& s,
        elem_t v,
        const std::function<bool(std::vector<elem_t>)>& cb
    ) = 0;

    /**
     * Reports a solution
     */
    virtual void report_solution(const std::vector<elem_t>& s) = 0;


    /**
     * Checks if we can add a given element to a solution
     */
    virtual bool can_add(const std::vector<elem_t>& s, elem_t v) {
        auto cnd = s;
        cnd.push_back(v);
        return is_good(cnd);
    }

    /**
     * Returns true if the resticted problem may have at least two solutions.
     */
    virtual bool restr_multiple() {
        return true;
    }

    /**
     * Checks if the given element can be a valid seed of a solution,
     * or a root if NULL is specified.
     */
    virtual bool is_seed(elem_t v, const std::unordered_set<elem_t>* s) {
        return is_good({v});
    }

    /**
     * Iterates over all the possible new elements that could be added
     * because of a single new element in a solution.
     */
    virtual void complete_cands(
        const std::vector<elem_t>* ground_set,
        elem_t new_elem,
        const std::function<bool(elem_t)>& cb
    ) {
        if (!ground_set) {
            for (elem_t i=0; i<e.size(); i++) {
                if (cb(i))
                    break;
            }
        } else {
            for (auto i: *ground_set) {
                if (cb(i))
                    break;
            }
        }
    }

    /**
     * Iterates over all the possible new elements that could be used
     * for the restricted problem
     */
    virtual void restricted_cands(
        const std::vector<elem_t>& s,
        const std::vector<int32_t>& level,
        const std::function<bool(elem_t)>& cb
    ) {
        auto ss = s;
        std::sort(ss.begin(), ss.end());
        for (elem_t i=0; i<e.size(); i++) {
            if (std::binary_search(ss.begin(), ss.end(), i)) continue;
            if (cb(i))
                break;
        }
    }

    /**
     * Checks if complete of a given element is a root.
     */
    virtual bool get_root(
        elem_t v,
        std::vector<elem_t>& s,
        std::vector<int32_t>& level
    ) {
        if (!is_seed(v, nullptr)) return false;
        s.clear();
        level.clear();
        s.push_back(v);
        level.push_back(0);
        auto ret = !complete(s, level, true);
        return ret;
    }

    /**
     * Update candidate list when a new element is added to the solution.
     */
    virtual void update_step(
        std::vector<elem_t>& s,
        elem_t v,
        int32_t level,
        std::set<std::pair<int32_t, elem_t>>& candidates,
        std::unordered_map<elem_t, int32_t>& cand_level,
        const std::vector<elem_t>* ground_set
    ) {
        complete_cands(ground_set, v, [&](elem_t cnd) {
            if (!can_add(s, cnd)) return false;
            if (cand_level.count(cnd)) return false;
            cand_level[cnd] = level+1;
            candidates.emplace(level+1, cnd);
            return false;
        });
    }

    /**
     * Extracts the next valid cand from candidates
     */
    virtual std::pair<elem_t, int32_t> next_cand(
        const std::vector<elem_t>& s,
        std::set<std::pair<int32_t, elem_t>>& candidates
    ) {
        while (!candidates.empty()) {
            auto p = *candidates.begin();
            candidates.erase(candidates.begin());
            if (!can_add(s, p.second)) continue;
            return {p.second, p.first};
        }
        return {e.size(), -1};
    }

    /**
     * Recomputes the order and the level of the elements in s with another
↪   seed.
     */
    virtual void resort(
        std::vector<elem_t>& s,
        std::vector<int32_t>& level,
        elem_t seed
    ) {
        std::vector<elem_t> sn{seed};
        std::vector<int32_t> ln{0};
        complete_inside(sn, ln, s, false);
        s = sn;
        level = ln;
    }

    /**
     * Complete function. Returns true if there was a seed change, false
↪   otherwise
     */
    virtual bool complete(
        std::vector<elem_t>& s,
        std::vector<int32_t>& level,
        bool stop_on_seed_change = false
    ) {
        if (s.empty()) throw std::runtime_error("??");
        std::set<std::pair<int32_t, elem_t>> candidates;
        std::unordered_map<elem_t, int32_t> cand_level;
        for (uint32_t i=0; i<s.size(); i++) {
            update_step(s, s[i], level[i], candidates, cand_level, nullptr);
        }
        bool seed_change = false;
        while (true) {
            elem_t n;
            int32_t l;
            std::tie(n, l) = next_cand(s, candidates);
            if (n == e.size()) break;
            unsigned pos = s.size();
            while (pos > 0 && (l < level[pos-1] || (l==level[pos-1] && n <
            ↪  s[pos-1]))) pos--;
            s.insert(s.begin()+pos, n);
            level.insert(level.begin()+pos, l);
            if (n < s[0]) { // Seed change
                if (stop_on_seed_change) return true;
                seed_change = true;
                resort(s, level, n);
                cand_level.clear();
                candidates.clear();
                for (uint32_t i=0; i<s.size(); i++) {
                    update_step(s, s[i], level[i], candidates, cand_level,
                    ↪  nullptr);
                }
            } else {
                update_step(s, n, l, candidates, cand_level, nullptr);
            }
        }
        return seed_change;
    }

    /**
     * Runs complete inside a given set.
     */
    virtual void complete_inside(
        std::vector<elem_t>& s,
        std::vector<int32_t>& level,
        const std::vector<elem_t>& inside,
        bool change_seed = true
    ) {
        if (s.empty()) throw std::runtime_error("??");
        std::set<std::pair<int32_t, elem_t>> candidates;
        std::unordered_map<elem_t, int32_t> cand_level;
        for (uint32_t i=0; i<s.size(); i++) {
            update_step(s, s[i], level[i], candidates, cand_level, &inside);
        }
        while (true) {
            elem_t n;
            int32_t l;
            std::tie(n, l) = next_cand(s, candidates);
            if (n == e.size()) break;
            unsigned pos = s.size();
            while (pos > 0 && (l < level[pos-1] || (l==level[pos-1] && n < s[pos-1]))) pos--;
            s.insert(s.begin()+pos, n);
            level.insert(level.begin()+pos, l);
            if (n < s[0] && change_seed) { // Seed change
                resort(s, level, n);
                cand_level.clear();
                candidates.clear();
                for (uint32_t i=0; i<s.size(); i++) {
                    update_step(s, s[i], level[i], candidates, cand_level, &inside);
                }
            } else {
                update_step(s, n, l, candidates, cand_level, &inside);
            }
        }
    }

    /**
     * Computes the prefix of the solution with a given seed and ending with v
     */
    virtual void get_prefix(
        std::vector<elem_t>& s,
        std::vector<int32_t>& level,
        elem_t seed,
        elem_t v
    ) {
        resort(s, level, seed);
        std::size_t i;
        for (i=0; i<s.size(); i++)
            if (s[i] == v)
                break;
        s.resize(i+1);
```

```cpp
            level.resize(i+1);
    }

    /**
     * Parent function, returns the parent index.
     */
    virtual elem_t parent(
        const std::vector<elem_t>& s,
        const std::vector<int32_t>& level,
        std::vector<elem_t>& parent,
        std::vector<int32_t>& parent_level
    ) {
        for (unsigned parind_pos = s.size()-1; parind_pos > 0; parind_pos--) {
            parent = s;
            parent_level = level;
            parent.resize(parind_pos);
            parent_level.resize(parind_pos);
            complete(parent, parent_level);
            if (parent != s) {
                return s[parind_pos];
            }
        }
        parent.clear();
        return e.size();
    }

    /**
     * Computes the children of a given solution. Returns true if we stopped generating
     * them because the callback returned true.
     */
    virtual bool children(
        const std::vector<elem_t>& s,
        const std::vector<int32_t>& level,
        const std::function<bool(const std::vector<elem_t>&, const std::vector<int32_t>&)>& cb
    ) {
        bool done = false;
        restricted_cands(s, level, [&] (elem_t cand) {
            restricted_problem(s, cand, [&](const std::vector<elem_t>& sol) {
                std::unordered_set<elem_t> sol_set(sol.begin(), sol.end());
                for (auto seed: sol) {
                    if (!is_seed(seed, &sol_set)) continue;
                    if (cand <= seed) continue;
                    std::vector<elem_t> core = sol;
                    std::vector<int32_t> clvl = level;
                    get_prefix(core, clvl, seed, cand);
                    std::vector<elem_t> child = core;
                    std::vector<int32_t> lvl = clvl;
                    // There was a seed change
                    if (complete(child, lvl, true)) continue;
                    std::vector<elem_t> p;
                    std::vector<int32_t> plvl;
                    elem_t parind = parent(child, lvl, p, plvl);
                    // Not the parent of this child
                    if (p != s) continue;
                    // Wrong parent index
                    if (parind != cand) continue;
                    if (restr_multiple()) {
                        p.push_back(parind);
                        complete_inside(core, clvl, p);
                        // Wrong restricted problem solution
                        if (core != sol) continue;
                    }
                    if (cb(child, lvl)) {
                        done = true;
                        break;
                    }
                }
                return done;
            });
            return done;
        });
        return done;
    }
};

template<typename CS>
class ReverseSearch {
protected:
    std::unique_ptr<CS> cs;
    typedef typename CS::elem_t elem_t;
    virtual void handle_solution(const std::vector<elem_t>& s, const
     ↪ std::vector<int32_t>& level) {
        cs->report_solution(s);
        cs->children(s, level, [this](const std::vector<elem_t>& sn, const
         ↪ std::vector<int32_t>& leveln) {
            handle_solution(sn, leveln);
            return false;
        });
    }
public:
    template <typename... Args>
    ReverseSearch(const Args&... args): cs(std::make_unique<CS>(args...)) {}

    void run() {
        std::vector<elem_t> s, p;
        std::vector<int32_t> level, pl;
        for (elem_t i=0; i<cs->e.size(); i++) {
            if (cs->get_root(i, s, level)) {
                handle_solution(s, level);
                }
            }
        }
};

#endif
```

A.1.2 `graph.hpp`

This file contains the implementation of the graph data structures, including the product graph as defined in [12]. This product graph is the graph with colored edges in which black connected cliques are enumerated.

```
#ifndef GRAPH_HPP
#define GRAPH_HPP
#include <stdint.h>
#include <assert.h>
#include "dynarray.hpp"
#include "cuckoo.hpp"
#include "binary_search.hpp"
#include "common.hpp"

#include <vector>
#include <algorithm>
#include <unordered_map>
#include <functional>

template <typename node_t_ = uint32_t, bool lowmem = false>
class graph_t {
public:
    typedef node_t_ node_t;
    typedef node_t elem_t;
private:
    node_t N_;
    dynarray<binary_search_t<node_t>> edges;
    // Structures for the "fast" version
    dynarray<cuckoo_hash_set<node_t>> edges_fast;
    dynarray<typename binary_search_t<node_t>::iterator> fwd_iter;

protected:
    static int64_t nextInt(FILE* in) {
        int64_t n = 0;
        int64_t ch = getc_unlocked(in);
        while (ch != EOF && (ch < '0' || ch > '9')) ch = getc_unlocked(in);
            if (ch == EOF) return EOF;
            while (ch >= '0' && ch <= '9') {
            n = 10*n + ch - '0';
            ch = getc_unlocked(in);
        }
        return n;
    }
    static void read_edges(FILE* in, bool directed,
     ↪ std::vector<std::vector<node_t>>& graph) {
        while(true) {
            int a = nextInt(in);
            int b = nextInt(in);
            if (a == EOF || b == EOF) return;
            if(a == b) continue;
            graph[a].push_back(b);
            if (!directed) graph[b].push_back(a);
        }
    }
public:
    graph_t(node_t N, const std::vector<std::vector<node_t>>& edg, bool
     ↪ sorted = false): N_(N) {
        edges.resize(N);
        for (node_t i=0; i<N; i++) {
            edges[i].init(edg[i]);
            if (!sorted) std::sort(edges[i].data().begin(),
             ↪ edges[i].data().end());
        }
        if (!lowmem) {
            edges_fast.resize(N);
            fwd_iter.resize(N);
            for (node_t i=0; i<N; i++) {
                for (auto x: edg[i])
                    edges_fast[i].insert(x);
                 fwd_iter[i] = edges[i].upper_bound(i);
            }
        }
    }

    static graph_t read_oly(FILE* in = stdin, bool directed = false) {
        node_t N = nextInt(in);
        nextInt(in);
        std::vector<std::vector<node_t>> graph(N);
        read_edges(in, directed, graph);
        for (node_t i=0; i<N; i++) {
            sort(graph[i].begin(), graph[i].end());
            graph[i].erase(unique(graph[i].begin(), graph[i].end()),
             ↪ graph[i].end());
        }
        return {N, graph, true};
    }

    static graph_t read_nde(FILE* in = stdin, bool directed = false) {
        node_t N = nextInt(in);
        std::vector<std::vector<node_t>> graph(N);
        for(node_t i=0; i<N; i++) {
            int a = nextInt(in);
            int b = nextInt(in);
            graph[a].reserve(b);
        }
        read_edges(in, directed, graph);
        for (node_t i=0; i<N; i++) {
            sort(graph[i].begin(), graph[i].end());
            graph[i].erase(unique(graph[i].begin(), graph[i].end()),
             ↪  graph[i].end());
        }
        return {N, graph, true};
    }

    /**
     *  Node new_order[i] will go in position i.
     */
    template <typename G>
    static graph_t permute(G g, const std::vector<node_t>& new_order) {
        assert(new_order.size() == (size_t) g.size());
        std::vector<node_t> new_pos(g.size(), -1);
        for (node_t i=0; i<g.size(); i++) new_pos[new_order[i]] = i;
        std::vector<std::vector<node_t>> new_edges(g.size());
        for (node_t i=0; i<g.size(); i++) {
            for (auto x: g.neighs(i)) {
                new_edges[new_pos[i]].push_back(new_pos[x]);
            }
        }
        return {g.size(), new_edges};
    }

    node_t size() const {
        return N_;
    }

    node_t degree(node_t i) const {
        return edges[i].size();
    }

    const binary_search_t<node_t>& neighs(node_t i) const {
        return edges[i];
    }

    class fwd_neighs_t {
        node_t n;
        const graph_t* g;
    public:
        fwd_neighs_t(const graph_t* g, node_t n): g(g), n(n) {}
        typename binary_search_t<node_t>::iterator begin() const {
            if (lowmem) return g->edges[n].upper_bound(n);
            else return g->fwd_iter[n];
        }
        typename binary_search_t<node_t>::iterator end() const {
            return g->edges[n].end();
        }
        node_t size() const {
            return end() - begin();
        }
    };

    friend class fwd_neighs_t;

    const fwd_neighs_t fwd_neighs(node_t n) const {
        return {this, n};
    }

    node_t fwd_degree(node_t n) const {
        return fwd_neighs(n).size();
    }

    bool are_neighs(node_t a, node_t b) const {
        if (lowmem) return edges[a].count(b);
        else return edges_fast[a].count(b);
    }
};

template <typename label_t_ = uint32_t, typename node_t_ = uint32_t, bool
↪  lowmem = false>
class labeled_graph_t: public graph_t<node_t_, lowmem> {
    std::vector<label_t_> labels;
    using ugraph_t = graph_t<node_t_, lowmem>;
public:
    typedef label_t_ label_t;
    typedef typename ugraph_t::node_t node_t;
    typedef typename ugraph_t::elem_t elem_t;

    labeled_graph_t(
        node_t N, std::vector<label_t> labels,
        const std::vector<std::vector<node_t>>& edg,
        bool sorted = false
    ): ugraph_t(N, edg, sorted), labels(labels) {}

    static labeled_graph_t read_oly(FILE* in = stdin, bool directed = false)
    ↪  {
        node_t N = ugraph_t::nextInt(in);
        ugraph_t::nextInt(in);
        std::vector<std::vector<node_t>> graph(N);
        std::vector<label_t> labels(N);
        for (node_t i=0; i<N; i++) {
            labels[i] = ugraph_t::nextInt(in);
        }
        ugraph_t::read_edges(in, directed, graph);
        for (node_t i=0; i<N; i++) {
            sort(graph[i].begin(), graph[i].end());
            graph[i].erase(unique(graph[i].begin(), graph[i].end()),
             ↪  graph[i].end());
        }
        return {N, labels, graph, true};
    }

    label_t get_label(node_t node) const {
        return labels[node];
    }
};

template <typename label_t_ = uint32_t, typename node_t_ = uint32_t, bool
 ↪  lowmem = false>
class product_graph_t {
    typedef label_t_ label_t;
    using ugraph_t = graph_t<node_t_, lowmem>;
    using lgraph_t = labeled_graph_t<label_t, node_t_, lowmem>;
    lgraph_t g1;
    lgraph_t g2;
    std::vector<std::pair<node_t_, node_t_>> nds;
    std::unordered_map<std::pair<node_t_, node_t_>, node_t_, pair_hash> rmp;
    void gen_node_list() {
        std::unordered_map<label_t, std::vector<node_t_>> g2_nodes;
        for (node_t_ i=0; i<g2.size(); i++)
            g2_nodes[g2.get_label(i)].push_back(i);
        for (node_t_ i=0; i<g1.size(); i++) {
            for (auto second: g2_nodes[g1.get_label(i)]) {
                nds.emplace_back(i, second);
                rmp[nds.back()] = nds.size()-1;
            }
        }
    }
public:
    typedef typename ugraph_t::node_t node_t;
    typedef typename ugraph_t::elem_t elem_t;
    product_graph_t(lgraph_t&& g1, lgraph_t&& g2): g1(g1), g2(g2) {
        gen_node_list();
    }
    product_graph_t(const lgraph_t& g1, const lgraph_t& g2): g1(g1), g2(g2) {
        gen_node_list();
    }

    static product_graph_t read_oly(FILE* in1, FILE* in2, bool directed =
     ↪  false) {
        return {lgraph_t::read_oly(in1, directed), lgraph_t::read_oly(in2,
         ↪  directed)};
    }

    node_t size() const {
        return nds.size();
    }

    std::pair<node_t, node_t> to_pair(node_t node) const {
        return nds[node];
    }

    void black_neighs(node_t node, const std::function<bool(node_t)>& cb)
    ↪   const {
        for (auto a: g1.neighs(nds[node].first)) {
            for (auto b: g2.neighs(nds[node].second)) {
                auto p = std::make_pair(a, b);
                if (rmp.count(p) == 0) continue;
                if (cb(rmp.at(p))) break;
            }
        }
    }

    bool are_neighs(node_t a, node_t b) const {
        return nds[a].first != nds[b].first && nds[a].second != nds[b].second
            && g1.are_neighs(nds[a].first, nds[b].first) ==
               g2.are_neighs(nds[a].second, nds[b].second);
    }

    bool are_black_neighs(node_t a, node_t b) const {
        bool ans = g1.are_neighs(nds[a].first, nds[b].first) &&
               g2.are_neighs(nds[a].second, nds[b].second);
        return ans;
    }
};
#endif
```

A.2 Black connected cliques

A.2.1 bccliques.cpp

This file contains the implementation of the main function.

```
#include "graph.hpp"
#include "permute.hpp"
#include "bccliques.hpp"

int main(int argc, char** argv) {
    if (argc < 3) {
        fprintf(stderr, "Usage: %s g1 g2\n", argv[0]);
        return 1;
    }
    FILE* f1 = fopen(argv[1], "r");
    FILE* f2 = fopen(argv[2], "r");
    auto tmp = product_graph_t<>::read_oly(f1, f2);
    auto rs = ReverseSearch<BlackConnectedCliques<>>(tmp);
    rs.run();
}
```

A.2.2 bccliques.hpp

This file contains the functions that are needed by the framework to enumerate black-connected cliques.

```
#ifndef BCCLIQUES_HPP
#define BCCLIQUES_HPP
#include "framework.hpp"
#include "graph.hpp"
#include "cuckoo.hpp"
#include <queue>

template <typename node_t = uint32_t>
class BlackConnectedCliques: public CommutableSystem<product_graph_t<node_t>>
↪  {
public:
    using CommutableSystem<product_graph_t<node_t>>::CommutableSystem;
    typedef typename CommutableSystem<product_graph_t<node_t>>::elem_t
    ↪  elem_t;
    /**
     * Checks if a given subset is a solution.
     */
    virtual bool is_good(const std::vector<elem_t>& s) override {
        throw std::runtime_error("This function should never be called!");
    }

    /**
     * Solves the restricted problem
     */
    virtual void restricted_problem(
        const std::vector<elem_t>& s,
        elem_t v,
        const std::function<bool(std::vector<elem_t>)>& cb
    ) override {
        cuckoo_hash_set<elem_t> ok;
        ok.insert(v);
        for (auto n: s)
            if (this->e.are_neighs(v, n))
                ok.insert(n);
        std::vector<elem_t> sol;
        cuckoo_hash_set<elem_t> visited;
        std::queue<elem_t> q;
        q.push(v);
        while (!q.empty()) {
            auto t = q.front(); q.pop();
            if (!ok.count(t)) continue;
            if (visited.count(t)) continue;
```

```
                visited.insert(t);
                sol.push_back(t);
                this->e.black_neighs(t, [&](node_t n) -> bool {
                    q.push(n);
                    return false;
                });
            }
            cb(sol);
        }

        /**
         * Reports a solution
         */
        virtual void report_solution(const std::vector<elem_t>& s) override {
            printf("{");
            for (auto n: s) printf("%u, ", this->e.to_pair(n).first);
            printf("\b\b} -> {");
            for (auto n: s) printf("%u, ", this->e.to_pair(n).second);;
            printf("\b\b}\n");
            fflush(stdout);
        }

        /**
         * Checks if we can add a given element to a solution
         */
        virtual bool can_add(const std::vector<elem_t>& s, elem_t v) override {
            uint32_t black_cnt = 0;
            uint32_t neigh_cnt = 0;
            for (auto n: s) {
                if (this->e.are_neighs(v, n)) neigh_cnt++;
                if (this->e.are_black_neighs(v, n)) black_cnt++;
            }
            return black_cnt > 0 && neigh_cnt == s.size();
        }

        /**
         * Returns true if the resticted problem may have at least two solutions.
         */
        virtual bool restr_multiple() override {
            return false;
        }

        /**
         * Checks if the given element can be a valid seed of a solution,
         * or a root if NULL is specified.
         */
        virtual bool is_seed(elem_t v, const std::unordered_set<elem_t>* s)
         ↪ override {
            bool can_be = true;
            this->e.black_neighs(v, [&] (elem_t e) {
                if (e > v) return true;
                if (s == nullptr || s->count(e)) {
                    can_be = false;
                    return true;
                }
                return false;
            });
            return can_be;
        }

        /**
         * Iterates over all the possible new elements that could be added
         * because of a single new element in a solution.
         */
        virtual void complete_cands(
            const std::vector<elem_t>* ground_set,
            elem_t new_elem,
            const std::function<bool(elem_t)>& cb
        ) override {
            if (!ground_set) {
                this->e.black_neighs(new_elem, cb);
            } else {
                for (auto i: *ground_set) {
                    if (cb(i))
                        break;
                }
            }
        }

        /**
         * Iterates over all the possible new elements that could be used
         * for the restricted problem
         */
        virtual void restricted_cands(
            const std::vector<elem_t>& s,
            const std::vector<int32_t>& level,
            const std::function<bool(elem_t)>& cb
        ) override {
            std::set<elem_t> els;
            for (auto i: s) {
                this->e.black_neighs(i, [&](elem_t v) {
                    els.insert(v);
                    return false;
                });
            }
            for (auto i: s) els.erase(i);
            for (auto i: els) {
                if (cb(i))
                    break;
            }
        }
};
```

```

#endif
```

A.3 Data structures and other common things

A.3.1 `binary_search.hpp`

This file contains a fast implementation of binary search.

```
#ifndef _BINARY_SEARCH_T
#define _BINARY_SEARCH_T
#include "dynarray.hpp"
#include <vector>

template<typename T = uint32_t>
class binary_search_t {
private:
    dynarray<T> support;
public:
    typedef dynarray<T>& data_type;
    typedef const T* iterator;

    void init(const std::vector<T>& v) {
        support.resize(v.size());
        unsigned cnt = 0;
        while (cnt != v.size()) {
            support[cnt] = v[cnt];
            cnt++;
        }
    }

    iterator begin() const {
        return support.begin();
    }

    iterator it_at(size_t p) const {
        return begin() + p;
    }

    iterator end() const {
        return support.end();
    }

    size_t size() const {return support.size();}

    T get_at(size_t idx) const {
        return support[idx];
    }

    bool count(T v) const {
        int64_t n = support.size();
        const T* arr = &support[0];
        while (n > 1) {
            const int64_t half = n/2;
            arr = (arr[half] < v)?(arr+half):arr;
            n -= half;
        }
        const T* tmp = (*arr < v)+arr;
        return tmp < support.end() && *tmp == v;
    }

    iterator lower_bound(T v) const {
        return std::lower_bound(support.begin(), support.end(), v);
    }

    iterator upper_bound(T v) const {
        return std::upper_bound(support.begin(), support.end(), v);
    }

    data_type data() {
        return support;
    }
};
#endif
```

A.3.2 `common.hpp`

This file contains an hash function for pairs.

```
#ifndef COMMON_HPP
#define COMMON_HPP
#include <unordered_map>

struct pair_hash {
    template <class T1, class T2>
    std::size_t operator () (const std::pair<T1,T2> &p) const {
        auto h1 = std::hash<T1>{}(p.first);
        auto h2 = std::hash<T2>{}(p.second);
        h1 ^= h2 + 0x9e3779b9 + (h1<<6) + (h1>>2);
        return h1;
    }
};
#endif
```

A.3.3 `cuckoo.hpp`

This file contains a fast implementation of a cuckoo hash set.

```
#ifndef _CUCKOO_H
#define _CUCKOO_H
```

```
#include <vector>
#include <assert.h>
#include <immintrin.h>
#include <stdlib.h>
#include <stdint.h>

template<typename T, T missing = -1,
#ifdef __KNC__
    int bucket_size = 64/sizeof(T)
#else
    int bucket_size = 16/sizeof(T)
#endif
>
class cuckoo_hash_set {
public:
    typedef T value_type;
    typedef value_type& reference;
    typedef const value_type& const_reference;
    typedef value_type* pointer;
    typedef const value_type* const_pointer;
    typedef std::ptrdiff_t difference_type;
    typedef size_t size_type;
private:
    pointer ht;
    size_t mask;
    size_t sz;

    size_t hash_1(const value_type& k) const {
        return  k & mask;
    }

    size_t hash_2(const value_type& k) const {
        return ~k & mask;
    }

    void insert(value_type& k, value_type*& table) {
        int h1 = hash_1(k);
        for (int pos=0; pos<bucket_size; pos++)
            if (table[h1*bucket_size+pos] == missing) {
                table[h1*bucket_size+pos] = std::move(k);
                return;
            }
        int h2 = hash_2(k);
        for (int pos=0; pos<bucket_size; pos++)
            if (table[h2*bucket_size+pos] == missing) {
                table[h2*bucket_size+pos] = std::move(k);
                return;
            }
        bool use_hash_1 = true;
        for (unsigned i=0; i<mask; i++) {
            value_type cuckooed;
            size_t hash;
            if (use_hash_1) hash = hash_1(k);
            else hash = hash_2(k);
            int pos = 0;
            for (; pos<bucket_size; pos++)
                if (table[hash*bucket_size+pos] == missing)
                    break;
            if (pos == bucket_size) {
                cuckooed = std::move(table[hash*bucket_size]);
                pos = 1;
                for (; pos<bucket_size; pos++)
                    table[hash*bucket_size+pos-1] =
                     ↪  std::move(table[hash*bucket_size+pos]);
                table[hash*bucket_size+pos-1] = std::move(k);
            } else {
                cuckooed = std::move(table[hash*bucket_size+pos]);
                table[hash*bucket_size+pos] = std::move(k);
            }
            use_hash_1 = hash == hash_2(cuckooed);
            k = std::move(cuckooed);
            if (k == missing) return;
        }
        rehash(table);
        insert(k, table);
    }

    void rehash(value_type*& table) {
        auto oldmask = mask;
        if (mask == 0) mask = 1;
        else mask = (mask<<1) | mask;
        pointer newt = 0;
        posix_memalign((void**)&newt, sizeof(T)*bucket_size,
         ↪  sizeof(T)*capacity());
        std::fill(newt, newt+capacity(), missing);
        for (size_t i=0; i<(oldmask+1)*bucket_size; i++)
            if (table[i] != missing)
                insert(table[i], newt);
        std::swap(table, newt);
        free(newt);
    }
public:
    class const_iterator {
    private:
        const cuckoo_hash_set& container;
        size_type offset;
    public:
        typedef cuckoo_hash_set::value_type value_type;
        typedef cuckoo_hash_set::const_reference const_reference;
        typedef cuckoo_hash_set::const_pointer const_pointer;
        typedef cuckoo_hash_set::difference_type difference_type;
        typedef std::forward_iterator_tag iterator_category;

        const_iterator(const cuckoo_hash_set& container, size_type offset_): container(container), offset(offset_) {
            while (offset != container.capacity() && container.ht[offset] == missing)
                ++offset;
        }
        bool operator==(const const_iterator& other) const {
            return &container == &other.container && offset == other.offset;
        }
        bool operator!=(const const_iterator& other) const {
            return !(*this == other);
        }

        const_iterator& operator++() {
            ++offset;
            while (offset != container.capacity() && container.ht[offset] == missing)
                ++offset;
            return *this;
        }
        const_iterator operator++(int) {
            const_iterator tmp = *this;
            ++*this;
            return tmp;
        }

        const_reference operator*() const {
            return container.ht[offset];
        };
    };

    typedef const_iterator iterator;
    friend class const_iterator;

    cuckoo_hash_set& operator=(const cuckoo_hash_set& other) = delete;

    cuckoo_hash_set(const cuckoo_hash_set& other): ht(nullptr) {
        posix_memalign((void**)&ht, sizeof(T)*bucket_size, sizeof(T)*other.capacity());
        for (uint64_t i=0; i<other.capacity(); i++)
            ht[i] = other.ht[i];
        mask = other.mask;
        sz = other.sz;
    };

    ~cuckoo_hash_set() {
        free(ht);
    }

    cuckoo_hash_set(): mask(0), sz(0) {
        posix_memalign((void**)&ht, sizeof(T)*bucket_size, sizeof(T)*bucket_size);
        std::fill(ht, ht+bucket_size, missing);
    }

    const_iterator begin() const {
        return const_iterator(*this, 0);
    }
    const_iterator end() const {
        return const_iterator(*this, capacity());
    }

    bool operator==(const cuckoo_hash_set<T>& oth) {
        if (oth.size() != size()) return false;
        for (const auto& x: oth)
            if (!count(x))
                return false;
        return true;
    }

    bool operator!=(const cuckoo_hash_set<T>& oth) {
        return !(*this == oth);
    }

    void insert(value_type k) {
        if (count(k)) {
            return;
        }
        insert(k, ht);
        sz++;
    }
    bool count(const value_type& k) const {
            int h1 = hash_1(k);
        int h2 = hash_2(k);
#ifndef __KNC__
        if (bucket_size == 4 && sizeof(T) == 4) {
            __m128i cmp = _mm_set1_epi32(k);
            __m128i b1 = _mm_load_si128((__m128i*)&ht[bucket_size*h1]);
            __m128i b2 = _mm_load_si128((__m128i*)&ht[bucket_size*h2]);
            __m128i flag = _mm_or_si128(_mm_cmpeq_epi32(cmp, b1), _mm_cmpeq_epi32(cmp, b2));
            return _mm_movemask_epi8(flag);
        }
#else
        if (bucket_size == 16 && sizeof(T) = 4) {
            __m512i cmp = _mm512_set1_epi32(k);
            __m512i b1 = _mm512_load_epi32(&ht[bucket_size*h1]);
            __m512i b2 = _mm512_load_epi32(&ht[bucket_size*h2]);
            return _mm512_cmpeq_epi32_mask(b1, cmp) || _mm512_cmpeq_epi32_mask(b2, cmp);
        }
```

```
#endif
        bool result = false;
        for (unsigned i=0; i<bucket_size; i++)
            result |= (ht[(bucket_size*h1)|i] == k || ht[(bucket_size*h2)|i]
             ↪  == k);
        return result;
    }
    void reserve(size_type sz) {
        if (sz <= capacity()) return;
        mask++;
        while (mask <= sz/bucket_size) mask <<= 1;
        free(ht);
        posix_memalign((void**)&ht, sizeof(T)*bucket_size,
         ↪  sizeof(T)*capacity());
        std::fill(ht, ht+capacity(), missing);
        mask--;
    }
    size_type size() const {
        return sz;
    }
    size_type capacity() const {
        return (mask+1)*bucket_size;
    }
    bool empty() const {
        return sz == 0;
    }
    void erase(const value_type& k) {
        int h1 = hash_1(k);
        for (int pos=0; pos<bucket_size; pos++)
            if (ht[h1*bucket_size+pos] == k) {
                ht[h1*bucket_size+pos] = missing;
                sz--;
                return;
            }
        int h2 = hash_2(k);
        for (int pos=0; pos<bucket_size; pos++)
            if (ht[h2*bucket_size+pos] == k) {
                ht[h2*bucket_size+pos] = missing;
                sz--;
                return;
            }
    }
    void clear() {
        std::fill(ht, ht+bucket_size*capacity(), missing);
    }

    int front() const {
        return *begin();
    }
};
#endif
```

A.3.4 `dynarray.hpp`

This file contains a simple, fixed size array whose size is decided at construction time.

```
#ifndef _DYNARRAY_H
#define _DYNARRAY_H
#include <cstddef>
#include <stdexcept>
#include <algorithm>
#include <memory>

template< class T >
struct dynarray {
    // types:
    typedef       T                                    value_type;
    typedef       T&                                   reference;
    typedef const T&                                   const_reference;
    typedef       T*                                   iterator;
    typedef const T*                                   const_iterator;
    typedef std::reverse_iterator<iterator>            reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
    typedef size_t                                     size_type;
    typedef ptrdiff_t                                  difference_type;

    // fields:
private:
    T*         store;
    size_type count;

    // helper functions:
    void check(size_type n) {
        if (store == 0) throw std::out_of_range("dynarray");
        if (n >= count) throw std::out_of_range("dynarray");
    }
    T* alloc(size_type n) {
        if (n>std::numeric_limits<size_type>::max()/sizeof(T))
            throw std::out_of_range("dynarray");
        return reinterpret_cast<T*>(malloc(n*sizeof(T)));
    }

    void init(size_t n) {
        size_type i;
        try {
            for (size_type i=0; i<count; ++i)
                new (store+i) T;
        } catch (...) {
            for (; i>0; --i)
                (store+(i-1))->~T();
            throw;
        }
    }

    void init(size_t n, const value_type& v) {
        size_type i;
        try {
            for (size_type i=0; i<count; ++i)
                new (store+i) T(v);
        } catch (...) {
            for (; i>0; --i)
                (store+(i-1))->~T();
            throw;
        }
    }
public:
    // construct and destruct:
    dynarray(): store(nullptr), count(0) {};
    const dynarray operator=(const dynarray&) = delete;

    explicit dynarray(size_type c): store(alloc(c)), count(c) {
        init(c);
    }

    explicit dynarray(size_type c, const value_type& v): store(alloc(c)),
    ↪   count(c) {
        init(c, v);
    }

    dynarray(const dynarray& d): store(alloc(d.count)), count(d.count) {
        try {
            std::uninitialized_copy(d.begin(), d.end(), begin());
        } catch (...) {
            free(store);
            throw;
        }
    }

    ~dynarray() {
        if (store == 0) return;
        for (size_type i = 0; i<count; ++i)
            (store+i)->~T();
        free(store);
    }

    void resize(size_type n) {
        this->~dynarray();
        store = alloc(n);
        count = n;
        init(n);
    }

    void resize(size_type n, const value_type& v) {
        this->~dynarray();
        store = alloc(n);
        count = n;
        init(n, v);
    }

    // iterators:
    iterator       begin()        { return store; }
    const_iterator begin()  const { return store; }
    const_iterator cbegin() const { return store; }
    iterator       end()          { return store + count; }
    const_iterator end()    const { return store + count; }
    const_iterator cend()   const { return store + count; }

    reverse_iterator       rbegin()        { return reverse_iterator(end());
    ↪   }
    const_reverse_iterator rbegin()  const { return reverse_iterator(end());
    ↪   }
    reverse_iterator       rend()          { return
    ↪   reverse_iterator(begin()); }
    const_reverse_iterator rend()    const { return
    ↪   reverse_iterator(begin()); }

    // capacity:
    size_type size()     const { return count; }
    size_type max_size() const { return count; }
    bool      empty()    const { return count == 0; }

    // element access:
    reference       operator[](size_type n)       { return store[n]; }
    const_reference operator[](size_type n) const { return store[n]; }

    reference       front()       { return store[0]; }
    const_reference front() const { return store[0]; }
    reference       back()        { return store[count-1]; }
    const_reference back()  const { return store[count-1]; }

    const_reference at(size_type n) const { check(n); return store[n]; }
    reference       at(size_type n)       { check(n); return store[n]; }

    // data access:
    T*       data()       { return store; }
    const T* data() const { return store; }

    friend void swap(dynarray<value_type>& a, dynarray<value_type>& b) {
        std::swap(a.store, b.store);
        std::swap(a.count, b.count);
    }
};

#endif
```