PhD Dissertation

International Doctorate School in Information and Communication Technologies

DIT - University of Trento

On the computational complexity of enumerating certificates of NP problems

Marco Rospocher

Advisor:

Prof. Romeo Rizzi

Dipartimento di Matematica e Informatica (DIMI)

Università degli Studi di Udine

Member ICT Graduate School

Università degli Studi di Trento

March 2006

to Giuditta

Abstract

In this thesis we investigate the complexity of listing problems, where it is required to generate all possible solutions for any given input instance. The new contributions proposed are twofold. On one side, we propose a new structural computational complexity theory for listing problems associated to **NP** *relations. Under a reasonable notion of efficiency for listing algorithms, we provide several completeness results for the considered class of listing problems. On the other side, we provide a general efficient method for listing all (optimal) solutions of a combinatorial problem, whenever we have a good knowledge of the polyhedral description of the underlying combinatorial problem. To conclude, exploiting the underlying structure of the problem, we provide very efficient algorithms for listing all satisfying truth assignments of some peculiar classes of boolean formulas, like for example 2SAT formulas, outperforming the state-of-the-art time bounds.*

Keywords
listing, computational complexity, completeness, combinatorial optimization, polyhedral description.

Acknowledgements

My first and foremost thanks goes to my advisor, Romeo Rizzi, for introducing me to very challenging research topics, for his time, enthusiasm, encouragement and particularly his inexhaustible supply of interesting problems. Without his valuable guidance in research, this thesis would never have been written.

My sincere thanks goes to Pablo Moscato and all the members of the Newcastle Bioinformatics Initiative group at the University of Newcastle, NSW, Australia, for their support and hospitality during my three and a half months stay "down under".

A special thanks goes to my whole family, for their invaluable support over all these academic years.

Last but not least, my most grateful thanks goes to my beloved half, Giuditta, for having been so patient and supportive over the last three years, specially during my stay abroad. This thesis is dedicated to her.

Contents

List of Algorithms

List of Figures

Chapter 1

Introduction

Computational complexity is a lively and major branch of theoretical computer science that analyzes and classifies problems in terms of the computational resources which are necessary and sufficient to solve them. Since the introduction of the **NP**-completeness theory [7, 38, 43], computational complexity has rapidly expanded and a remarkable list of celebrated achievements has been obtained.

Historically, computational problems have been considered from a decisional perspective, in which problems have been identified with sets or languages. The goal in a decision problem is indeed to establish if a given input instance (a string) belongs or not to a set (a language). Hence, the output of the computation of a decision problem is just a yes or no answer. For example, SAT is the problem of deciding whether a given boolean formula φ admits a satisfying truth assignment.

Computational decision problems are classified into complexity classes according to the amount of resources needed to solve them. Class **P** contains decision problems solvable in polynomial time in the length of the input. Intuitively, **P** represents the class of efficiently solvable problems since algorithms with running time which grows faster than polynomial (e.g. exponential) are not considered to be practically usable. Unfortunately, for many well-known problems like SAT, no polynomial time algorithm is known.

NP contains decision problems solvable in polynomial time by nondeterministic algorithms. Equivalently, **NP** is characterized by certificates: a yes instance of the problem admits a short certificate with which one can prove the correctness of the answer in polynomial time. Note that SAT belongs to **NP**: a certificate for yes answer of a given boolean formula φ is precisely a satisfying truth assignment of φ, and its correctness can be trivially checked in polynomial time. Clearly, $\mathbf{P} \subseteq \mathbf{NP}$, but it is still an open question whether $\mathbf{P} \subset \mathbf{NP}$. **NP** contains **NP**-complete problems: if we solve in polynomial time any of them then all problems in **NP** are solvable in polynomial time. In particular, SAT is **NP**-complete [7].

Although, historically, much effort has been spent in studying the computational complexity of decision problems, it is clear that in many computational tasks, just a yes or no answer is not enough. For example, given a boolean formula, we may ask for a satisfying truth assignment (if one exist). Hence, more generally, we assume to work with a binary relation which associates instances to solutions, and we consider some computational problem associated to this relation. For example, the decision problem associated to a binary relation asks to decide if, for a given instance, the set of solutions corresponding to that instance is empty or not. Of particular interest are the so called **NP** relations, where the size of a solution is polynomially bounded by the size of the instance, and the correctness of the solution is verifiable in polynomial time. As suggested by the name, there is a strong connection between languages in **NP** and **NP** relations: a language L is in **NP** if and only if there exists an **NP** relation R, such that the set of instances of R

which admit at least a solution, is the language L itself. As noted in [5], to any language in **NP** corresponds a single decision problem and many **NP** relations.

In 1979, Valiant [54] proposed a computational complexity theory for counting problems associated to **NP** relations. In a counting problem, the goal is to calculate the cardinality of the set of solutions for any given instance. Hence, in view of the above considerations, Valiant investigated the complexity of counting, with respect to a specific **NP** relation, the certificates of any instance of a language in **NP**. For example, in #SAT the goal is to find the number of satisfying truth assignments of a given boolean formula. Valiant introduced class **#P**, the class of counting problems associated to **NP** relations, and the class of **#P**-complete problems (**#PC**), which is the class of those counting problems in **#P** such that solving in polynomial time any of them implies that all problems in **#P** are solvable in polynomial time. Recalling our example, #SAT is **#P**-complete [55]. Indeed, Valiant [54, 55] has also shown that there exist some **NP** relations such that the associated counting problem is **#P**-complete, while the associated decision problem is in **P**: for example, the problem of counting the number of perfect matchings in a bipartite graph G is **#P**-complete (as a consequence of the **#P**-completeness of PERMANENT, the problem of computing the permanent of a matrix), while deciding if a bipartite graph G has a perfect matching is in **P**.

In this thesis, we investigate the complexity of listing problems, that is those problems where the goal is to produce in output the set of all solutions for any given instance. Inspired by the approach proposed by Valiant [54], we propose a complexity theory for listing problems associated to **NP** relations. Hence, we investigate the complexity of listing, with respect to a specific **NP** relation, the certificates of any instance of a language in **NP**.

Although many positive results have been offered (where most of them concerns developing a fast ad-hoc algorithm for listing the solutions of a specific problem), only few and isolated negative results have been proposed, and, up to now, no structural computational complexity theory has been developed for the gender of problems here considered. Nonetheless, there are strong motivations in our opinion to study the computational complexity of listing problems:

1. due to the enduring growth of computing power and storing capacity, the task of listing all the solutions of a problem is nowadays feasible for reasonable size instances;

2. there are many fields in which it is required to efficiently list the solutions of a given problem: among them, computational biology (e.g. listing perfect phylogenies of species [36]), combinatorial optimization (e.g. listing perfect matchings of a graph [26, 27, 52]) and computational geometry (e.g. listing vertices of a polytope [6]);

3. listing all solutions of a problem can help when searching for a counter-example to some conjecture;

4. studying and classifying the difficulty of a listing problem may help to better understand the structure underlying it;

5. correlations and intersections of a complexity theory for listing problems with those already proposed for decision problems and other genders of problems may lead to new insights toward the main open questions in computational complexity, like the mighty $\mathbf{P} \subset \mathbf{NP}$ conjecture.

6. understanding the space of solutions of a problem can help to design

fast ad-hoc heuristics for it.

1.1 Organization and contributions

The chapters of this thesis are organized as follows. Chapter 2, contains some definitions and results used through out of the thesis. Chapters 3 through 5 contain a detailed description of the new contributions proposed, while in Chapters 6 we draw some conclusions. Summarizing, the main contributions proposed in this thesis are the following.

Chapter 3

In this chapter of the thesis, we introduce a new complexity class, called $\mathcal{L}\mathbf{P}$. This class is the listing analogue of class $\#\mathbf{P}$ for counting problems: it contains the listing problems associated to $\mathbf{NP}$ relations. We define some subclasses of $\mathcal{L}\mathbf{P}$ accordingly to the various notions of efficient listing proposed in literature. In this chapter, we mainly consider the weakest of them (polynomial total time), in order to obtain stronger results. We show that $\mathcal{L}\mathbf{P}$ contains $\mathcal{L}\mathbf{P}$-complete problems, that is listing problems such that if we efficiently solve one of them, then all listing problems in $\mathcal{L}\mathbf{P}$ can be efficiently solved. One of them, is $\mathcal{L}$Sat. We show that many listing problems turn out to be $\mathcal{L}\mathbf{P}$-complete thanks to the notion of one-to-one certificates reduction for $\mathbf{NP}$ relations: actually, many of the listing problems associated to $\mathbf{NP}$ relations such that the associated decision problem is $\mathbf{NP}$-complete, turn out to be $\mathcal{L}\mathbf{P}$-complete due to one-to-one certificates reductions. We conclude the chapter proving that the $\mathcal{L}\mathbf{P}$-completeness of the listing problem associated to an $\mathbf{NP}$ relation does not imply the $\mathbf{NP}$-completeness of the decision problem associated to the relation: we show that,

- the problem of listing all truth assignments of a 1Valid boolean formula;
- the problem of listing all prime implicants of a monotone boolean formula;

are all $\mathcal{L}\mathbf{P}$-complete.

Chapter 4

Differently from the negative results presented in Chapter 3, in this chapter and the next one we mainly provide positive results.

In this chapter, we consider the listing problem associated to combinatorial ensembles. A combinatorial ensemble is a family of couples $\langle S, \mathcal{F} \rangle$, where S is a set of element (called the ground set) and $\mathcal{F}$, given implicity, is a family of subsets of S (called feasible solutions). Strengthening a result proposed in [6], we show that every time we have a good knowledge of the polyhedral description of the combinatorial ensemble, then we can efficiently list all feasible solutions for any given instance. Actually, this result follows as a consequence of a more general one, where feasible solutions are additionally evaluated by an objective function. In particular, to each element of the ground set is assigned a real value weight, the value of a feasible solution is the sum of the weights of the elements in it, and we consider the problem of listing all optimal feasible solutions of the combinatorial ensemble. In particular, we show that every time we can compute in polynomial time a minimum value feasible solution for arbitrary real value weights, condition which is equivalent to have a good knowledge of the polyhedral description of the combinatorial ensemble, then we can list in polynomial space and polynomial delay all:

- the feasible solutions;
- the minimum/maximum cardinality feasible solutions;
- the minimum/maximum value feasible solutions;
- the minimum/maximum value minimum/maximum cardinality feasible solutions.

This result implies that we can efficiently list all (optimal) solutions of a broad class of combinatorial optimization problems: examples are the perfect matching problem, the spanning tree problem, and the T-join problem.

The same results cannot be achieved if we made the assumption that we can compute in polynomial time a minimum value feasible solution just for arbitrary nonnegative value weights. In particular, we provide an example of a combinatorial optimization problem for which it is $\mathcal{L}\mathbf{P}$-hard to list all minimum value feasible solutions for nonnegative value weights. However, if we restrict to strictly positive weights, then we show that it is possible to list in polynomial space and polynomial delay all:

- the minimum cardinality feasible solutions;
- the minimum value feasible solutions;
- the minimum value minimum cardinality feasible solutions.

The value of the above results is that from a single framework we can derive many positive results, whereas up to now many of these results had been worked out one by one.

Chapter 5

In this chapter, we consider the problem of listing all satisfying truth assignments for two particular classes of boolean formulas: CNF formulas with XOR clauses and 2SAT formulas. In [10], Creignou and Hébrard proposed a polynomial space polynomial delay algorithm for solving the listing problem in the two cases above. Their algorithm, based on a general fact due to Valiant [55], runs with a delay which is not linear in the size of the instance. Exploiting the underlying structure of the problem, we present in both cases a polynomial space linear delay listing algorithm.

Chapter 2

Preliminaries

In this chapter we introduce some basic notions and we recall some results that we will use later. This chapter has by no means the ambition of being exhaustive or detailed: references to comprehensive sources will be provided in each section.

2.1 Sets, relations and functions

See also [41, 49, 9].

A set of element is *finite* if it contains a finite number of elements, otherwise it is *infinite*. For example, the set $A = \{a, b, c\}$ is finite, while the set of odd natural numbers is infinite.
The *cardinality* or *size* of a finite set A, denoted by $|A|$, is the number of elements of A. The size of the empty set is 0.

The power set of a set A, denoted by 2^A, is the set of all possible subsets of A, including the empty set and A itself. If set A is finite, than 2^A contains $2^{|A|}$ elements.

Two special sets of great interest to us are $\mathbb{N}$ and $\mathbb{R}$. We denote by $\mathbb{N}$ the set of natural numbers and $\mathbb{R}$ the set of real numbers. Moreover, $\mathbb{R}_{\geq 0} := \{x : x \in \mathbb{R}, x \geq 0\}$ and $\mathbb{R}_{>0} := \{x : x \in \mathbb{R}, x > 0\}$.

Given two elements a and b, let us denote by (a, b) the *ordered pair* of a and b. Note that (a, b) is different from (b, a) and it is distinct from $\{a, b\}$. The *Cartesian product* of two sets A and B, denoted by $A \times B$, is the set of all ordered pairs (a, b) with $a \in A$ and $b \in B$.
A *binary relation* R between two sets A and B is a subset of the Cartesian product $A \times B$. To every binary relation R we can associate a predicate $R(a, b)$ such that $R(a, b)$ is true (resp. $R(a, b) = 1$) if and only if $(a, b) \in R$, otherwise $R(a, b)$ is false (resp. $R(a, b) = 0$).
We define the *domain* (resp. *codomain*) of a binary relation R between A and B as the set of all $a \in A$ (resp. $b \in B$) such that $R(a, b) = 1$ for some $b \in B$ (resp. $a \in A$).
The concept of ordered pair can be extended to a sequence of n elements, called *n-tuple*, denoted by $(a_1, a_2, \ldots, a_n)$. The Cartesian product of n sets $A_1, A_2, \ldots, A_n$ is the set of n-tuples

$$A_1 \times A_2 \times \cdots \times A_n = \{(a_1, a_2, \ldots, a_n) : a_i \in A_i, i = 1, \ldots, n\}.$$

Given two sets A and B, with B finite, we denote by $A^{|B|}$, or simply A^B, the cartesian product of $|B|$ sets each one equal to $|A|$. We call the elements of A^B *vectors* and, given a vector $a \in A^B$, we denote by a_b the b-th entry in the $|B|$-tuple a.
An *n-ary relation* on sets $A_1, A_2, \ldots, A_n$ is a subset of $A_1 \times A_2 \times \cdots \times A_n$.

A *function* from a set A to a set B (denoted by $f : A \to B$) is a binary relation between A and B such that there exists at most one ordered pair (a, b) for any member a of A. If $(a, b) \in f$, we usually write $f(a) = b$, where

a is called the *argument* of f, while b is called the *value* of f.
A function $f : A \to B$ may be:

1. *injective* (or *one-to-one*), if for every $a, a' \in A$ with $a \neq a'$, then $f(a) \neq f(a')$;

2. *surjective* (or *onto*), if for each $b \in B$ exists $a \in A$ such that $f(a) = b$;

3. *bijective* (or *one-to-one and onto*), if f is both injective and surjective.

2.2 Graphs and Trees

See also [49, 9, 8].

A *directed graph* (or *digraph*) G is a pair (V, A), where $V = V(G)$ is a finite set and $A = A(G)$ is a binary relation on V, that is, A is a subset of $V \times V$. The set V is called the *node set* of G, and its elements are called *nodes* (or *vertices*). The set A is called the *arc set* of G, and its elements are called *arcs*. Given an arc (u, v), or simply uv, we have that u is the *tail node* of uv and v is the *head node* of uv. The *in-degree* (resp. *out-degree*) of a node u is the number of arcs with u as head node (resp. tail node).

In an *undirected graph* $G = (V, E)$, the edge set E consists of unordered pairs of nodes, rather than ordered pairs as in directed graphs. That is, an edge is a set $\{u, v\}$, where $u, v \in V$ and $u \neq v$. By convention, we use the notation uv for an edge, rather than the set notation $\{u, v\}$, and uv and vu are considered to be the same edge. Furthermore, u and v are called the *end-points* of edge uv. Many of the following definitions hold both for directed and undirected graphs, so we state them just for undirected graphs.

A *weighted graph* is an undirected graph for which each edge $e \in E$ has an associated *weight* $w(e)$ (or, w_e), typically given by a *weight function* $w : E \to \mathbb{R}$.

We say that an undirected graph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. Given a set $V' \subseteq V$, *the subgraph of* G *induced by* V' is the undirected graph $G' = (V', E')$, where $E' = \{uv \in E : u, v \in V'\}$. A subgraph G' of G is *spanning* if $V' = V$.

A *bipartite graph* is an undirected graph $G = (V, E)$ in which V can be partitioned into two sets V_1 and V_2 such that $uv \in E$ implies either $u \in V_1$ and $v \in V_2$ or $u \in V_2$ and $v \in V_1$. That is, all edges go between the two sets V_1 and V_2.

Let $G = (V, E)$ be an undirected graph. For any $U \subseteq V$, we define $\delta(U) \subseteq E$ as the set of the edges with one endpoint in U and the other in $V \setminus U$. A *cut* is a subset $F \subseteq E$, such that $F = \delta(U)$ for some $U \subseteq V$.
A *matching* in an undirected graph $G = (V, E)$ is a subset M of edges such that no node of G is the endpoint of more than one edge in M.

A *walk* from u to v in an undirected graph $G = (V, E)$ is a sequence $u = v_0, e_1, v_1, \ldots, e_k, v_k = v$ with $k \geq 1$, where v_i is a node and e_i is the edge $v_{i-1}v_i$. A walk with $v_0, v_1, \ldots, v_k$ pairwise distinct is called a *path*. A walk with $v_0 = v_k$ and $v_1, v_2, \ldots, v_{k-1}$ pairwise distinct is called a *cycle*. An undirected graph $G = (V, E)$ is *connected* if there exists a path between each couple of nodes of G. A graph $G = (V, E)$ is *acyclic* if there are no cycles in G.

A *tree* $T = (V, E)$ is a connected acyclic undirected graph. Given a graph $G = (V, E)$, a *spanning tree* of G is a subgraph of G which is a tree and it is spanning.

2.3 Boolean formulas and Boolean Circuits

See also [49, 46].

A *boolean variable* x is a variable which can take only two possible values, called *boolean values*: 1 (*true*) or 0 (*false*).
The *negation* (or, NOT) of a boolean value is the other boolean value and it is usually indicated by $\bar{\cdot}$. Hence, $\bar{1} = 0$ and $\bar{0} = 1$.
The *conjunction* (or, AND) of two boolean values is equal to 1 only if both boolean values are equal to 1 and it is usually indicated by $\wedge$. Hence, $x \wedge y = 1$ only if $x = y = 1$.
The *disjunction* (or, OR) of two boolean values is equal to 1 if either or both boolean values are equal to 1 and it is usually indicated by $\vee$. Hence, $x \vee y = 1$ if $x = 1$ or $y = 1$.

A *boolean formula* can be any one of the following:

1. a boolean value;
2. a boolean variable;
3. the negation of a boolean formula;
4. the conjunction of two boolean formulas;

5. the disjunction of two boolean formulas.

Given a boolean variable x, a *literal* is a boolean formula x (*positive literal*) or $\bar{x}$ (*negative literal*).

We define a *truth assignment* (or, 0/1-*assignment*) T as a mapping which assigns to each variable a value in $\{0, 1\}$. If no ambiguity arises, sometimes we denote T by a 0/1-vector. The *evaluation* of a boolean formula φ under a truth assignment T is a process in which each variable instance in the expression is replaced by the boolean value assigned by T, after which the formula is simplified to 1 or 0 applying the above operators' definitions. A truth assignment T *satisfies* (resp. *does not satisfy*) a boolean formula φ, if φ evaluates to 1 (resp. 0) under T. If T satisfies φ, we write $T \models \varphi$, otherwise $T \not\models \varphi$. A boolean formula φ is *satisfiable* if there exists at least one truth assignment T such that $T \models \varphi$.

A boolean formula φ is in *conjunctive normal form* (or, is a *CNF-formula*) if $\varphi = \bigwedge_{j=1,\dots,m} c_j := c_1 \wedge c_2 \wedge \ldots \wedge c_m$, where $m \geq 1$, and each of the c_j, called *clause*, is either a literal or a disjunction of two or more literals.

In Chapter 5 we will use a further boolean operator, called *exclusive disjunction* (or, XOR), such that the exclusive disjunction of two boolean values is equal to 1 if exactly one of the two boolean values is equal to 1, and it is usually indicated by $\oplus$. Hence, $x \oplus y = 1$ if $x = 1$ and $y = 0$, or $x = 0$ and $y = 1$.

A *boolean circuit* is a directed acyclic graph $C = (V, A)$ such that to each node of C, also called *gate*, is assigned one of the following labels: $\wedge, \vee, \bar{\cdot}, x_1, x_2, \ldots, x_m, 0, 1,$ *output*. For each label $x_1, x_2, \ldots, x_m,$ *output* there is exactly one node in C. All nodes have out-degree at least 1, except the node labeled *output*, which has out-degree 0 and in-degree 1. The nodes labeled $x_1, x_2, \ldots, x_m$ have in-degree 0 and are called *inputs* of the circuit. The nodes labeled $\bar{\cdot}$ have in-degree 1 while the nodes labeled 0 or 1 have in-degree 0. To conclude, the nodes labeled $\wedge, \vee$ have in-degree 2.

Let $T \in \{0,1\}^m$ be an assignment to the circuit's inputs. Given a gate v, we define recursively the *truth value* $T(v)$ of gate v as follows:

1. if gate v is labeled 1 (resp. 0), then $T(v) = 1$ (resp. $T(v) = 0$);
2. if gate v is labeled x_i, then $T(v) = T(x_i)$;
3. if gate v is labeled $\bar{\cdot}$, then $T(v) = \overline{T(w)}$, where wv is the only arc in C with v as head node;
4. if gate v is labeled $\wedge$, then $T(v) = T(u) \wedge T(w)$, where wv, uv are the only two arcs in C with v as head node;
5. if gate v is labeled $\vee$, then $T(v) = T(u) \vee T(w)$, where wv, uv are the only two arcs in C with v as head node;
6. if gate v is labeled *output*, then $T(v) = T(w)$, where wv is the only arc in C with v as head node.

Given an assignment $T \in \{0,1\}^m$, we define the *circuit's output* $C(T)$ as the truth value of the gate labelled *output*. Hence, $C(T) \in \{0,1\}$.

2.4 Complexity Theory

In this section we assume the reader to be familiar with the notions of algorithm, Turing Machine, polynomial-time and the big $\mathcal{O}$ notation. See also [46, 5, 29] for a detailed explanation of these notions and the others reported in this section.

An *alphabet* Σ is a finite set of elements, called *symbols*. A *word*, or *string of symbols*, is an ordered finite sequence of symbols in Σ. We indicate with Σ^* the set of all finite words over Σ. Given a word x, we indicate with $|x|$ the *length*, or *size*, of x, i.e. the number of symbols occurring in x. A *language* over Σ is a subset of Σ^*. In this thesis we assume that Σ is the *binary* alphabet, that is, $\Sigma = \{0, 1\}$.

An *abstract problem* P is a binary relation between a set I of problem *instances* and a set S of problem *solutions*. An example of abstract problem is the one of finding an Hamiltonian Cycle C in a graph G. An instance for this problem is the graph G, while a solution is a cycle in G which traverses each node exactly once, or an empty set if no such cycle exists.

An *encoding* of a set O of objects is a mapping e from O to the set of binary strings. Any "finite" mathematical object can be encoded by a binary string. For example, natural numbers can be easily encoded by binary strings, or a graph can be represented by its adjacency matrix, which in turn can be encoded by a binary string.

Encodings are used to map abstract problems to problems which can be solved by a computer. A computer algorithm that solves some abstract problem takes an encoding of a problem instance in input and produces an encoding of a problem solution in output. We call a problem whose instances set and solutions set are sets of binary strings a *computational problem*, and we say that a computational problem is *polynomial-time solvable* if there exists an algorithm to solve it in time $\mathcal{O}(n^k)$, where n is the length of the input string and k is a constant.

The extension of the notion of polynomial-time solvability from computational problems to abstract problems depends on the particular encoding chosen. In fact, different encodings of the same object can have different size (think for example to the different encodings of an integer number in unary or binary). However, if we rule out some exceptions and we assume to work with concise and reasonable encodings, all natural encodings have polynomial related size: this implies that if an abstract problem P is polynomial-time solvable with respect to an encoding e, then P is polynomial-time solvable with respect to any other concise and reasonable encoding.

In a *decision problem* it is given an input $x \in \Sigma^*$ and it is required to verify whether the input satisfies a certain property. Hence, the output required in a decision problem is just a *yes* or *no* answer. If, given x, the answer returned is yes (resp. no), then we say that x is a *yes-instance* (resp. *no-instance*). Clearly, every decision problem can be thought as a language L over Σ, the one containing all and only the yes-instances of the decision problem. Hence, the terms decision problem and language is used interchangeably in this thesis as in the literature.

We indicate with **P** the class of decision problems which can be solved in polynomial-time by a deterministic Turing Machine. We indicate with **NP** the class of decision problems which can be solved in polynomial-time by a non-deterministic Turing Machine. By definition, $\mathbf{P} \subseteq \mathbf{NP}$.

There is an equivalent way to define class **NP** which turns out to be very useful in the context of this thesis. Given a binary relation R, we define the *language* $L(R)$ *associated to* R as $L(R) := \{x \ : \ (x, y) \in R \text{ for some } y\}$. A binary relation $R \subseteq \Sigma^* \times \Sigma^*$ is said to be *polynomially balanced* if, for any $(x, y) \in R$, the length of y is bounded by a polynomial in the length of x, that is, $|y| \leq |x|^k$. A binary relation $R \subseteq \Sigma^* \times \Sigma^*$ is said to be *polynomially decidable* if there exists a polynomial time algorithm that, given $x, y \in \Sigma^*$, decides whether $(x, y) \in R$. The following proposition holds.

Proposition 2.1. *A language* $L \in \mathbf{NP}$ *if and only if there is a polynomially balanced polynomially decidable relation* R*, such that* $L = L(R)$*.*

Given a string x, the strings y such that $(x, y) \in R$ are called *succinct certificates* (or *polynomial witnesses*). They attest that, with respect to relation R, string x is a word of language $L(R)$.

Let A and B be two decision problems. A polynomial time *Karp reduction*, or simply reduction, from A to B is a polynomial time computable function f such that $x \in A$ if and only if $f(x) \in B$. If there exists a polynomial time reduction from A to B, we say that A *reduces* to B, and we write $A \leq B$. Clearly, if $A \leq B$ and B is in $\mathbf{P}$, it follows that $A \in \mathbf{P}$. Conversely, if $A \leq B$ and A is not in $\mathbf{P}$, it follows that $B \notin \mathbf{P}$. Furthermore, if $A \leq B$ and $B \leq C$, then $A \leq C$.

A decision problem A is **NP**-*hard* if for every problem $B \in \mathbf{NP}$ we have $B \leq A$. A decision problem A is **NP**-*complete* if it is **NP**-hard and it belongs to **NP**. Hence, if an **NP**-complete problem A is polynomially solvable, then all decision problems in **NP** are polynomially solvable.

Let $R \subseteq \Sigma^* \times \Sigma^*$ be a binary relation. Given a string $x \in \Sigma^*$, we define $Y(x) := \{y \ : \ (x, y) \in R\}$. We define the following problems associated to a relation R:

- **Decision Problem:** given a string x, decide whether $Y(x) \neq \emptyset$.
- **Search Problem:** given a string x, output a string $y \in Y(x)$ (if any).
- **Counting Problem:** given a string x, output $|Y(x)|$.
- **Listing Problem:** given a string x, output all strings in $Y(x)$ (if any).

Note that, if R is a polynomially balanced polynomially decidable relation, for any string x, $Y(x)$ is the set of succinct certificates attesting that, with respect to relation R, x belongs to the **NP** language $L(R)$. Hence, the class of decision problems associated to polynomially balanced polynomially decidable relations is exactly **NP**.

The class of search problems (also know as *function problems*) associated to polynomially balanced polynomially decidable relations is **FNP**. The class of counting problems associated to polynomially balanced polynomially decidable relations is **#P** [55, 54].

Given two polynomially balanced polynomially decidable relations R_1 and R_2, a *Levin reduction* from R_1 to R_2 is a triplet of polynomial time computable functions $\langle f, g, h \rangle$ such that:

- $\forall x \in \Sigma^*, x \in L(R_1) \Longleftrightarrow f(x) \in L(R_2)$;

- $\forall x, y \in \Sigma^*, (x, y) \in R_1 \Longrightarrow (f(x), g(x, y)) \in R_2$;
- $\forall x, z \in \Sigma^*, (f(x), z) \in R_2 \Longrightarrow (x, h(x, z)) \in R_2$.

Note that a Levin reduction from R_1 to R_2 implies a Karp reduction from $L(R_1)$ to $L(R_2)$. Actually, as observed in [1], all known Karp reductions for proving **NP**-completeness of decision problems are, or can easily modified to be, Levin reductions between some natural **NP** relations defining those decision problems.

2.5 Polytopes, Polyhedra and Linear Programming

See also [49, 48, 8, 33].

Let $x_1, \ldots, x_m \in \mathbb{R}^n$, and let $\lambda_1, \ldots, \lambda_m \in \mathbb{R}_{\geq 0}$. We say that $y := \lambda_1 x_1 + \cdots + \lambda_m x_m$ is a *nonnegative combination* of vectors $x_1, \ldots, x_m$. Furthermore, if $\lambda_1 + \cdots + \lambda_m = 1$, we say that y is a *convex combination* of $x_1, \ldots, x_m$.

A subset C of $\mathbb{R}^n$ is called a *cone* if $C \neq \emptyset$ and for each $x, y \in C$, every nonnegative combination of x, y belongs to C. The cone generated by a set X of vectors is the smallest cone containing X, and it is denoted by *cone*(X).

Given a set $S \subseteq \mathbb{R}^n$, the *convex hull* of S, denoted by conv.hull(S), is the set of all convex combinations of elements of S. An important result concerning the convex hull of a finite set of elements is that, for any finite $S \subseteq \mathbb{R}^n$ and any $c \in \mathbb{R}^n$, we have:

$$\min\{cx : x \in S\} = \min\{cx : x \in \text{conv.hull}(S)\}.$$

A *polyhedron* P is the set of solutions of a finite system of linear inequalities, that is, given a matrix $A \in \mathbb{R}^{n \times m}$ and a vector $b \in \mathbb{R}^n$, the polyhedron determined by A and b is the set $P = \{x \in \mathbb{R}^m : Ax \leq b\}$. If $A \in \mathbb{Q}^{n \times m}$ and $b \in \mathbb{Q}^n$, then $P = \{x \in \mathbb{R}^m : Ax \leq b\}$ is said to be a *rational* polyhedron. In this thesis, we consider only rational polyhedra.

Given $w \in \mathbb{R}^m$ and $t \in \mathbb{R}$, the inequality $wx \leq t$ is *valid* for a polyhedron P if $P \subseteq \{x \in \mathbb{R}^m : wx \leq t\}$. A polyhedron P is *bounded* if there exists $w \in \mathbb{R}_{\geq 0}$ such that $P \subseteq \{x \in \mathbb{R}^m : -w \leq x_i \leq w \text{ for } i = 1, \ldots, m\}$. A bounded polyhedron is called a *polytope*. Motzkin [44] showed that a set P is a polyhedron if and only if $P = Q + C$ for some polytope Q and some cone C. In $P \neq \emptyset$, then C is unique and is called the *characteristic cone* char.cone(P) of P.

Given $c \in \mathbb{R}^m$ and $d \in \mathbb{R}$, the *hyperplane* determined by c and d is the set $H = \{x \in \mathbb{R}^m : cx = d\}$. Given a polyhedron $P = \{x \in \mathbb{R}^m : Ax \leq b\}$ and an hyperplane $H = \{x \in \mathbb{R}^m : cx = d\}$, we say that H is a *supporting hyperplane* of P if $cx \leq d$ is a valid inequalities for P and $P \cap H \neq \emptyset$. A subset F of P is called a *face* of P if either $F = P$ or $F = P \cap H$ for some supporting hyperplane H of P.

Given a polyhedron $P = \{x \in \mathbb{R}^m : Ax \leq b\}$, a vector $x \in P$ is called a *vertex* of P if $\{x\}$ is a face of P, or equivalently if x cannot be written as a convex combination of vectors in P. We call P *pointed* if it has at least one vertex. In a pointed polyhedron every (inclusionwise) minimal face of P is a vertex. Nonempty polytopes are pointed. Indeed, a polytope is equal to the convex hull of its vertices. Actually, a set P is a polytope if and only if there exists a set S such that P is equal to conv.hull(S).

A set S in $\mathbb{R}^n$ is called *up-monotone* if for each $y \in S$ all vectors $x \in \mathbb{R}^n$ with $x \geq y$ are in S. The *dominant* $S^\uparrow$ of S is the smallest up-monotone convex set containing S. Note that,

$$S^\uparrow = \text{conv.hull}(S) + \mathbb{R}^n_{\geq 0} := \{x + y \ : \ x \in \text{conv.hull}(S), y \in \mathbb{R}^n_{\geq 0}\}.$$

Furthermore, if P is a nonempty polytope, then $P^\uparrow$ is an unbounded polyhedron.

Let $P \subseteq \mathbb{R}^n$ be a rational polyhedron. The *facet complexity* of P is the smallest number f such that $f \geq n$ and there exists a system $Ax \leq b$ of linear inequalities defining P where each inequality has size[1] at most f. The *vertex complexity* of P is the smallest number v such that $v \geq n$ and there exist rational vectors $x_1, \dots, x_k$, $y_1, \dots, y_h$, with

$$P = conv.hull(x_1, \dots, x_k) + cone(y_1, \dots, y_h)$$

where each of $x_1, \dots, x_k$, $y_1, \dots, y_h$ has size at most v. The following result shows that vertex and facet complexity of a polyhedron are polynomially related.

Theorem 2.2 ([48], page 121 - Theorem 10.2). *Let $P \subseteq \mathbb{R}^n$ be a rational polyhedron with facet complexity f and vertex complexity v. Then $v \leq 4n^2 f$ and $f \leq 4n^2 v$.*

A *Linear Programming* problem, or *LP* problem, is the problem of *optimizing* (i.e. maximizing or minimizing) a linear function over a polyhedron $P = \{x : Ax \leq b\}$, like, for example, $\max\{cx : Ax \leq b\}$ or $\min\{cx : Ax \leq$

[1]Here, the size of a rational linear inequality or of a vector is its encoding length.

$b\}$. The polyhedron P is called the *feasible region* and a vector $x \in P$ is called a *feasible solution.* If P is not empty, the LP problem is called *feasible*, and *infeasible* otherwise. If x is a feasible solution such that cx is optimal, we say that x is an *optimal* solution. A feasible LP problem is called *bounded* if it has optimal solutions, and *unbounded* otherwise.

In [12], Dantzig presented a method for solving LP problems called the *simplex method.* Although the simplex method turns out to work very well in practice, no worst case polynomial running time bound has ever been proved. The first polynomial time method for solving LP problems was given by Khachiyan [39] in 1979. However, this method, also known as the *ellipsoid method*, turns out to be practically infeasible. Karmarkar [37], in 1984, introduced the *interior point* method. This method solves LP problems in polynomial time and have very efficient implementations.

We conclude this section recalling an important result in polyhedral combinatorics which relates the polynomial time solvability of two problems on polyhedra. We consider the following two problems on polyhedra.

Problem. OPTIMIZATION
Input: a rational polyhedron $P \subseteq \mathbb{R}^n$ and a rational vector $w \in \mathbb{R}^n$.
Goal: either (i) find $y \in P$ such that $wy \leq wx$ for each $x \in P$, or (ii) find a vector y in char.cone(P) with $cy \leq 0$, or (iii) conclude that $P = \emptyset$.

Problem. SEPARATION
Input: a rational polyhedron $P \subseteq \mathbb{R}^n$ and a rational vector $v \in \mathbb{R}^n$.
Question: either conclude that $v \in P$, or, if not, find a rational vector $w \in \mathbb{R}^n$ such that $wx < wv$ for all $x \in P$.

A class of polyhedra $\mathcal{P} = \{P_t : t \in \mathcal{O}\}$, where $\mathcal{O}$ is a collection of objects and P_t is a rational polyhedra for each $t \in \mathcal{O}$, is called *proper* if for each object $t \in \mathcal{O}$ we can compute in polynomial time in the size of t two natural numbers n_t and s_t such that $P_t \subseteq \mathbb{R}^{n_t}$ and P_t has facet complexity at most s_t.

We say that problem SEPARATION is *polynomially solvable* over class $\mathcal{P}$ if, for any $P_t \in \mathcal{P}$ and any rational vector in $v \in \mathbb{R}^{n_t}$, there exists a polynomial time (in the size of t and v) algorithm for solving problem SEPARATION. Analogously, we say that problem OPTIMIZATION is *polynomially solvable* over class $\mathcal{P}$ if there exists a polynomial time algorithm for solving problem OPTIMIZATION for any instance $\langle P_t, w\rangle$, where $t \in \mathcal{O}$ and w is a rational vector in $\mathbb{R}^{n_t}$. Grötschel, Lovasz and Schrijver [33] proved the following equivalence.

Theorem 2.3 (Grötschel, Lovasz and Schrijver [33]). *For any proper class of polyhedra, problem* OPTIMIZATION *is solvable in polynomial time if and only if problem* SEPARATION *is solvable in polynomial time.*

Chapter 3

Structural results regarding a listing complexity theory

In this chapter of the thesis, we put the basis for developing a computational complexity theory for listing problems. Informally, given an input instance x and a property P, a listing problem asks to output all solutions y associated to input x that satisfy property P. We introduce complexity class $\mathcal{L}\mathbf{P}$ containing the listing problems associated to polynomially balanced polynomially decidable binary relations, together with some relevant subclasses. Note that $\mathcal{L}\mathbf{P}$ is the listing analogous of class $\#\mathbf{P}$ for counting problems. We show that $\mathcal{L}\mathbf{P}$ contains *complete* problems, that is, problems that if we can efficiently list their solutions for any given instance, then we can efficiently solve any listing problem in $\mathcal{L}\mathbf{P}$. In particular, we conclude the chapter showing that some members of $\mathcal{L}\mathbf{P}$ are complete, although their decision and search versions are polynomially solvable.

3.1 Listing Problems and class $\mathcal{L}\mathbf{P}$

Without loss of generality, we assume that Σ is the binary alphabet. We next recall the definition of listing problem associated to a binary relation.

Definition 3.1 (Listing Problem)**.** *Let $R \subseteq \Sigma^* \times \Sigma^*$ be a binary relation on strings. Given any string $x \in \Sigma^*$, the* listing problem *associated with relation R asks to return the set $\{y \ : \ (x, y) \in R\}$.*

Usually, x is called *input instance* of the problem, while every element of $\{y \ : \ (x, y) \in R\}$ is called a *solution*. Among all listing problems associated with binary relations, we focus our attention on studying listing problems associated to **NP** *relations*, that is, relations R that are polynomially balanced (i.e. $(x, y) \in R$ implies $|y| \leq |x|^k$ for some constant k), and polynomially decidable (i.e. there exists a polynomial time algorithm deciding whether $(x, y) \in R$ or not). An example of **NP** relation is the *Hamiltonian Cycle* relation R_{HC} defined as follows (we recall that an Hamiltonian cycle in a graph is a cycle which traverses all nodes in the graph exactly once):

$$R_{HC} := \{(G, C) : G \text{ is a graph, } C \text{ is an Hamiltonian cycle in } G\} .$$

We recall that **NP** relations are strictly related to **NP** languages: a language $L \in \mathbf{NP}$ if and only if there is a polynomially balanced polynomially decidable relation R, such that

$$L = L(R) := \{x \ : \ (x, y) \in R \text{ for some } y\} .$$

For example, the relation *Hamiltonian Cycle* R_{HC} is associated to language Hamiltonian Cycle, a well-know member of **NP**. However, it is important to note that for any particular **NP** language L, there are many **NP** relations R such that $L = L(R)$ (see [3]).

In this chapter of the thesis we are interested in studying the computational complexity of listing, with respect to a specified **NP** relation R, the succinct certificates y attesting that string x belongs to a language $L(R) \in \mathbf{NP}$.

For example, when we list all Hamiltonian cycles of a given graph G, we are listing, with respect to relation R_{HC}, all certificates attesting that G is a member of the family of Hamiltonian graphs.

Definition 3.2 (Class $\mathcal{L}$**P**). *We define $\mathcal{L}$**P** as the class of listing problems associated with polynomially balanced polynomially decidable relations.*

Notice that $\mathcal{L}$**P** is the listing analogue of class #**P** for counting problems.

3.2 Listing algorithms and their notions of efficiency

A *listing algorithm* for solving the listing problem associated to a relation R is an algorithm that, for any x, returns all y such that $(x, y) \in R$ without duplicates. In order to define a notion of efficiency (polynomial time) for listing algorithms some caution has to be taken. Let us consider the following relation:

$$R_{SAT} := \left\{ (\varphi, T) : \begin{array}{l} \varphi \text{ is a boolean CNF-formula,} \\ T \text{ is a truth assignment satisfying } \varphi \end{array} \right\}.$$

Clearly, R_{SAT} is an **NP** relation: the length of T is indeed polynomial in the number of variables (i.e., R_{SAT} is polynomially balanced) and we can easily check if $T \models \varphi$ in polynomial time in the input size (i.e., R_{SAT} is polynomially decidable). Consider the listing problem associated to relation R_{SAT}.

Problem. $\mathcal{L}$SAT
Input: a boolean CNF-formula φ of n variables.
Output: all truth assignments T satisfying φ.

Assume that we have a boolean CNF-formula φ of n variables such that, setting $T(x_1) = 1$, φ is satisfied whatever value is assigned to the remaining variables. Clearly, there are at least $2^{(n-1)}$ truth assignments satisfying φ. That is, the number of solutions is exponential in the size of the input. Hence, a listing algorithm of time complexity polynomial in the input size only is not possible, and the output size, i.e. the number of solutions, has to be taken into account. Even so, several notions of efficiency for listing algorithms can be considered:

(1) **Polynomial Total Time.** A listing algorithm runs in *polynomial total time* if its time complexity is polynomial in the input size and the output size. This notion was introduced for the first time by Tarjan in [53];

(2) **P-enumerability.** We say that a listing algorithm **P**-*enumerates* the solutions of a relation R if the time required to output them is bounded by $p(n)C$, where $p(n)$ is a polynomial in the input size n and C is the number of solutions to output. If such an algorithm exists for a relation R, then we say that R is **P**-*enumerable* (with some slight abuse of notation, we will also say that the listing problem associated to R is **P**-enumerable). The notion of **P**-enumerability was defined by Valiant in [54]. Clearly, **P**-enumerability is a stronger notion than polynomial total time. Furthermore, if the space complexity of an algorithm that **P**-enumerates the solutions of a relation R is polynomial in the input size only, we say that R is *strongly* **P**-*enumerable*. To the best of our knowledge, strong **P**-enumerability was introduced in [25];

(3) **Polynomial Delay.** A listing algorithm is said to have *D delay* if it lists all solutions one after the other in some order, in such a way that:

1. it takes polynomial time in the input size before producing the first

solution or halting;

2. after returning any solution, it takes D time before producing another solution or halting.

If D is polynomial (resp. linear, constant) in the input size, then we say that the algorithm runs with *polynomial delay* (resp. *linear delay*, *constant delay*). The notion of polynomial delay was introduced by Johnson, Yannakakis and Papadimitriou in [34]. Clearly, polynomial delay is a stronger notion than **P**-enumerability.

These various notions of efficiency for listing algorithms allow us to define some subclasses of $\mathbf{LP}$. In particular, we take into consideration the following subclasses:

Class EP. A listing problem $E \in \mathbf{LP}$ belongs to class $\mathbf{EP}$ if it admits a polynomial total time listing algorithm. This class has been introduced in [24];

Class $\mathbf{P_{enu}}$. A listing problem $E \in \mathbf{LP}$ belongs to class $\mathbf{P_{enu}}$ if E is **P**-enumerable;

Class $\mathbf{P_{del}}$. A listing problem $E \in \mathbf{LP}$ belongs to class $\mathbf{P_{del}}$ if it admits a polynomial delay listing algorithm;

Class $\mathbf{L_{del}}$. A listing problem $E \in \mathbf{LP}$ belongs to class $\mathbf{L_{del}}$ if it admits a linear delay listing algorithm.

The inclusion relationships between these subclasses of $\mathbf{LP}$ are established in the following observation.

Observation 3.3. $\mathbf{L_{del}} \subseteq \mathbf{P_{del}} \subseteq \mathbf{P_{enu}} \subseteq \mathbf{EP} \subseteq \mathbf{LP}$.

Proof. Follows from the definitions. □

Note that these classes are nonempty. In fact, as we will show in Chapter 5, there exists a linear delay algorithm for listing all truth assignments satisfying a 2SAT boolean formula.

3.3 $\mathcal{L}$**P**-complete listing problems

In this section we define a new subclass of $\mathcal{L}$**P**, called $\mathcal{L}$**PC**, which contains $\mathcal{L}$**P**-*complete* listing problems.

Definition 3.4 ($\mathcal{L}$**P**-complete problems)**.** *A listing problem E is* $\mathcal{L}$**P**-*complete if*

1. *$E \in \mathcal{L}\mathbf{P}$;*
2. *the existence of a polynomial total time algorithm for listing the solutions of E implies $\mathcal{L}\mathbf{P} = \mathbf{EP}$.*

More generally, we say that a listing problem E is $\mathcal{L}$**P**-*hard* if Condition 2 in Definition 3.4 is satisfied, but we don't know if $E \in \mathcal{L}\mathbf{P}$.

Definition 3.5 (Class $\mathcal{L}$**PC**)**.** *$\mathcal{L}\mathbf{PC} \subseteq \mathcal{L}\mathbf{P}$ is the class of* $\mathcal{L}$**P**-*complete listing problems.*

By definition, if $\mathcal{L}\mathbf{PC} \cap \mathbf{EP} \neq \emptyset$, then class $\mathcal{L}$**P** and class **EP** collapse. In the next subsection we show that class $\mathcal{L}$**PC** is not empty.

3.3.1 An $\mathcal{L}$P-complete problem: $\mathcal{L}$Bounded Halting

Let's consider the binary relation R_{BH} defined as follows[1]:

$$R_{BH} := \left\{ (\langle M, x, 1^t \rangle, y) : \begin{array}{l} M \text{ is the description of a deterministic Turing} \\ \text{machine that accepts } (x, y) \text{ within } t \text{ steps} \end{array} \right\}.$$

Note that R_{BH} is an **NP** relation. In fact, $|y|$ is bounded by t, and so $|y|$ is polynomially bounded by the length of the input $\langle M, x, 1^t \rangle$ (hence R_{BH} is polynomially bounded). Furthermore, whether M accepts (x, y) within t steps or not is decidable in time polynomial in length of the input $\langle M, x, 1^t \rangle$: we just need to simulate M on (x, y) for t steps (hence R_{BH} is polynomially decidable).

Next, we show that the listing problem $\mathcal{L}$Bounded Halting associated to relation R_{BH} is $\mathcal{L}$**P**-complete. By definition, $\mathcal{L}$Bounded Halting is the following problem.

Problem. $\mathcal{L}$Bounded Halting
Input: a triplet $\langle M, x, 1^t \rangle$, where M is the description of a deterministic Turing machine, x is a string and t is a natural number.
Output: all strings y such that M accepts (x, y) within t steps.

Our $\mathcal{L}$**P**-completeness proof of $\mathcal{L}$Bounded Halting is based on the following result. Levin reductions have been defined in Section 2.4.

[1] Here, 1^t indicates a string of 1s of length t.

Lemma 3.6. *Let $R \subseteq \Sigma^* \times \Sigma^*$ be an arbitrary* **NP** *relation. Then, there exists a Levin reduction $\langle f, g, h\rangle$ from R to R_{BH} such that,*

$$(x, y) \in R \Leftrightarrow (f(x), y) \in R_{BH},$$

for each $x, y \in \Sigma^*$.

Proof. The proof that we propose is an adaptation of the proof presented in Claim 2.3.1 of Lecture 2 in [32]. Let R be an arbitrary **NP** relation. Let p_R be a polynomial such that $|y| \leq p_R(|x|)$ for any $(x, y) \in R$. Let M_R be a polynomial time deterministic Turing machine that, given $x, y \in \Sigma^*$, decides whether $(x, y) \in R$, and let t_R be a polynomial bounding the running time of M_R. We define f, g, h as follows:

1. f maps an instance $x \in L(R)$ to an instance $f(x) := \langle M_R, x, 1^{t_R(|x|+p_R(|x|))}\rangle$ of $L(R_{BH})$;

2. $\forall x, y \in \Sigma^*, g(x, y) := y$;

3. $\forall x, z \in \Sigma^*, h(x, z) := z$.

We claim that $\langle f, g, h\rangle$ is a Levin reduction from R to R_{BH} such that, for each $x, y \in \Sigma^*$, $(x, y) \in R$ if and only if $(f(x), y) \in R_{BH}$. First of all, note that f, g, h are all polynomial time computable (note that the description of M_R is a constant string for the reduction). We next prove that $(x, y) \in R$ if and only if $(f(x), y) \in R_{BH}$. Suppose that $(x, y) \in R$. Clearly, $|y| \leq p_R(|x|)$. Hence, M_R accepts (x, y) within $t_R(|x| + p_R(|x|))$ steps. That is, $(\langle M_R, x, 1^{t_R(|x|+p_R(|x|))}\rangle, y) \in R_{BH}$. The other implication follows analogously. $\square$

Actually, Lemma 3.6 says that there exists a Levin reduction from the generic **NP** relation R to R_{BH} that *preserves the certificates*: $y \in \Sigma^*$ is a certificate attesting that $x \in \Sigma^*$ is a yes-instance for the decision problem associated to R, if and only if the *same* y is a certificate attesting that instance $\langle M, x, 1^t \rangle$, to which x is mapped by function f in the Levin reduction, is a yes-instance for the decision problem associated to R_{BH}. We can now prove the following result.

Theorem 3.7. $\mathcal{L}$BOUNDED HALTING *is* $\mathcal{LP}$*-complete.*

Proof. By the argumentation spent at the beginning of this subsection, we have that $\mathcal{L}$BOUNDED HALTING belongs to $\mathcal{LP}$. Hence, Condition 1 in Definition 3.4 is satisfied. We now prove that also Condition 2 in Definition 3.4 holds.
Let $R \subseteq \Sigma^* \times \Sigma^*$ be an arbitrary **NP** relation and let E be the listing problem in $\mathcal{LP}$ associated to R. By Lemma 3.6, we have that there exists a Levin reduction from relation R to R_{BH} that preserves the certificates. Hence, if there exists a polynomial total time algorithm for solving $\mathcal{L}$BOUNDED HALTING, then there exists a polynomial total time algorithm for solving E. Since R has been chosen arbitrary, we have that $\mathcal{L}$BOUNDED HALTING is $\mathcal{LP}$-complete. □

Furthermore, by Lemma 3.6, we can say that $\mathcal{L}$BOUNDED HALTING is somehow a *strong* member of $\mathcal{LPC}$.

Observation 3.8. *If* $\mathcal{L}$BOUNDED HALTING *is (strongly)* **P***-enumerable, then any listing problem in* $\mathcal{LP}$ *is (strongly)* **P***-enumerable. If there exists a polynomial delay (resp. linear delay, constant delay) algorithm for solving* $\mathcal{L}$BOUNDED HALTING*, then there exists a polynomial delay (resp. linear delay, constant delay) algorithm for solving any listing problem in* $\mathcal{LP}$.

Proof. The statement follows from the same argumentations spent in the proof of Theorem 3.7. □

Actually, $\mathcal{L}$**PC** does not contain only $\mathcal{L}$BOUNDED HALTING. Indeed, many other problems are in $\mathcal{L}$**PC**, thanks to the notion of one-to-one certificates reduction.

3.3.2 One-to-one certificates reductions: a powerful tool for our $\mathcal{L}$P-completeness theory

A *parsimonious reduction* $\langle f, g, h\rangle$ from an **NP** relation R_A to an **NP** relation R_B is a Levin reduction from R_A to R_B such that $|\{y : (x, y) \in R_A\}| = |\{z : (f(x), z) \in R_B\}|$. We also say that parsimonious reductions *preserve the number of certificates* between two **NP** relations. The notion of parsimonious reduction was introduced in [50], and since then it has played a key role in the counting complexity theory. In this subsection we consider a special type of parsimonious reduction for **NP** relations.

A *one-to-one certificates reduction* $\langle f, g, h\rangle$ from an **NP** relation R_A to an **NP** relation R_B is a parsimonious reduction from R_A to R_B such that, for each $x \in L(R_A)$, $h(x, z)$ is injective with respect to z (i.e. for each z_1, z_2, $h(x, z_1) = h(x, z_2)$ if and only if $z_1 = z_2$). Hence, a one-to-one certificates reduction from R_A to R_B provides a one-to-one and onto mapping between certificates for yes-instance x of R_A and certificates for yes-instance $f(x)$ of R_B. The reader can easily verify that the Levin reduction proposed in Lemma 3.6 is a special case of one-to-one certificates reduction.

One-to-one certificates reductions between **NP** relations allow us to include more members in class $\mathcal{L}$**PC**, thanks to the following observation.

Observation 3.9. *Let R_A and R_B be two* **NP** *relations. Let E_A and E_B be the listing problems corresponding to R_A and R_B respectively. Assume that E_A belongs to* $\mathcal{L}$**PC**. *If there exists a one-to-one certificates reduction $\langle f, g, h\rangle$ from R_A to R_B, then $E_B \in$* $\mathcal{L}$**PC**.

Proof. Clearly, $E_B \in$ $\mathcal{L}$**P**. Suppose that there exists a polynomial total time algorithm that solves problem E_B. We show that there exists a polynomial total time algorithm that solves problem E_A. In fact, thanks to the one-to-one certificates reduction $\langle f, g, h\rangle$ from R_A to R_B, given a string $x \in \sum^*$ we can produce in polynomial time a string $f(x)$ such that $x \in L(R_A)$ if and only if $f(x) \in L(R_B)$. Now, we can apply the polynomial total time algorithm that solves problem E_B in order to list all certificates of yes-instance $f(x)$. Note that to any certificate z of yes-instance $f(x)$ in E_B there corresponds exactly one certificate $y = h(x, z)$ of yes-instance x in E_A; furthermore, h is polynomial time computable. Hence, combining the polynomial total time algorithm that solves problem E_B with the polynomial time computable function h, we can list in polynomial total time all certificates of yes-instance x in E_A. That is, $E_A \in$ **EP** $\cap$ $\mathcal{L}$**PC** and then $\mathcal{L}$**P** $=$ **EP**. Hence, $E_B \in$ $\mathcal{L}$**PC**. $\square$

Let's consider the binary relation R_{CS} defined as follows[2]:

$$R_{CS} := \left\{ (C,T) : \begin{array}{l} C \text{ is a boolean circuit, } T \text{ is a 0/1-assignment to the} \\ \text{circuit's inputs such that the circuit's output } C(T) = 1 \end{array} \right\}.$$

Note that R_{CS} is an **NP** relation: obviously $|T|$ is polynomially bounded by the size of $|C|$, and it is easy to verify in polynomial time if $C(T) = 1$ (know-

[2]See Section 2.3 for some details on boolean circuits

ing the value of a gate's inputs, which are at most 2, the evaluation of a gate takes $\mathcal{O}(1)$). Indeed, the **NP** language associated to relation R_{CS} is the well-known **NP**-complete language CIRCUIT SAT. In Claim 2.4.1 of Lecture 2 in [32], it is presented a Levin reduction from R_{BH} to R_{CS}. Actually, the Levin reduction proposed [32] is a one-to-one certificates reduction from R_{BH} to R_{CS} (a one-to-one certificates reduction from the generic **NP** relation to R_{CS} is implicitly given in several treatments of the theory of computational complexity: [46] (see Theorem 8.2 at page 171 and Theorem 18.1 at page 442) and [9] (see Lemma 34.5 and Lemma 34.6, pages 987–994)). Hence, it follows that $\mathcal{L}$CIRCUIT SAT, the listing problem associated to relation R_{CS}, is $\mathcal{L}$**P**-complete.

Even if, to the best of our knowledge, this thesis is the first place where the notion of one-to-one certificates reduction is explicitly given, several of the parsimonious reductions for **NP** relations which have been proposed in the literature are actually one-to-one certificates reductions. For example, consider relation R_{SAT} defined early in Section 3.2. There exists a one-to-one certificates reduction from R_{CS} to R_{SAT} (for a full description of it, we address the reader to [46], Example 8.3, page 163): given a boolean circuit C, we can easily construct in linear time a boolean CNF-formula φ (to each gate of C there corresponds a variable and some clauses in φ) such that, to each truth assignment satisfying φ, there corresponds exactly one 0/1-assignment to C's inputs such that the C's output is equal to 1. Hence, $\mathcal{L}$SAT is $\mathcal{L}$**P**-complete. Some other listing problems which turn out to be $\mathcal{L}$**P**-complete due to one-to-one certificates reductions are those associated with the following **NP** relations:

1. $R_{3SAT} := \{(\varphi, T) : \varphi$ is a boolean CNF-formula with at most 3 literals per clause, T is a truth assignment satisfying $\varphi\}$. One-to-one certificates

reduction from: R_{SAT} ([55] or [46]);

2. $R_{HP} := \{(G, P) : G$ is an undirected graph, P is an Hamiltonian path in $G\}$. One-to-one certificates reduction from: R_{SAT} ([46]);

3. $R_{ILP} := \{(\langle A, b\rangle, x) : A$ is an integer matrix, b is an integer vector and x is an integer vector such that $Ax \leq b\}$. One-to-one certificates reduction from: R_{3SAT} ([51]);

4. $R_{HS} := \{(\langle S, C, K\rangle, S') : S$ is a finite set, C is a collection of subsets of S, $K \leq |S|$, and S' is an hitting set for C of size at most K. One-to-one certificates reduction from: R_{3SAT} ([56]);

Note that, by Observation 3.8 and the proof of Observation 3.9, listing problems proved to be $\mathcal{L}\mathbf{P}$-complete by one-to-one certificates reductions are somehow *quite strong* members of $\mathcal{L}\mathbf{PC}$.

Observation 3.10. *Let R be an* $\mathbf{NP}$ *relation. Let E be the listing problem associated to relation R. Assume that E has been proved to be $\mathcal{L}\mathbf{P}$-complete showing a one-to-one certificates reduction (or a composition of one-to-one certificates reductions) from the generic* $\mathbf{NP}$ *relation to R. Then, if E is (strongly)* $\mathbf{P}$*-enumerable, then any listing problem in* $\mathcal{L}\mathbf{P}$ *is (strongly)* $\mathbf{P}$*-enumerable. If there exists a polynomial delay algorithm for solving E, then there exists a polynomial delay algorithm for solving any listing problem in* $\mathcal{L}\mathbf{P}$.

Proof. The statement follows from Observation 3.8 and from analogous argumentations as those spent in the proof of Observation 3.9. □

At this point of the dissertation, the reader might have gotten the impression that the $\mathcal{L}$**P**-completeness of the listing problem associated to an **NP** relation R implies the **NP**-completeness of language $L(R)$ associated to R. In the following Section we show that this is not the case.

3.4 $\mathcal{L}$P-completeness of some listing problems whose decision version is in P

A *1Valid CNF-formula* (resp. *0Valid CNF-formula*) of n variables is a boolean CNF-formula φ such that the truth assignment 1^n (resp. 0^n), which maps all n variables to 1 (resp. 0), satisfies φ. Consider the following binary relation:

$$R_{1VSAT} := \left\{ (\varphi, T) : \begin{array}{c} \varphi \text{ is a 1Valid boolean CNF-formula,} \\ T \text{ is a truth assignment satisfying } \varphi \end{array} \right\}.$$

It can be easily verified that R_{1VSAT} is an **NP** relation. Furthermore, little effort is needed to show that the language $L(R_{1VSAT})$ associated to relation R_{1VSAT} is in **P**: by definition, 1^n is a truth assignment that satisfies every 1Valid CNF-formula. Let's now consider the listing problem associated to relation R_{1VSAT}.

Problem. $\mathcal{L}$1Valid-Sat
Input: a 1Valid CNF-formula φ of n variables.
Output: all truth assignments that satisfy φ.

In [10] it is proved that there is no polynomial delay algorithm that lists all truth assignments satisfying a 1Valid CNF-formula unless **P** = **NP**. Next, we show that $\mathcal{L}$1Valid-Sat is $\mathcal{L}$**P**-complete.

Theorem 3.11. *$\mathcal{L}$1VALID-SAT is $\mathcal{L}\mathbf{P}$-complete.*

Proof. We consider a generic instance of problem SAT. Let φ be a CNF-formula of n variables and m clauses. Hence, $\varphi = \bigwedge_{j=1,\ldots,m} c_j$. Without loss of generality, we can assume that φ is not a 1Valid CNF-formula. We define an instance of problem 1VALID-SAT as follows:

$$\varphi' := \bigwedge_{i=1,\ldots,n} \bigwedge_{j=1,\ldots,m} (c_j \vee x_i).$$

Clearly, φ' is a 1Valid CNF-formula. We denote by S_f the set of all truth assignments satisfying a CNF-formula f. Then, the following holds.

> *Claim.* $S_{\varphi'} = S_\varphi \cup \{1^n\}$.
> By construction, $S_{\varphi'} \supseteq S_\varphi \cup \{1^n\}$. We need to prove that $S_{\varphi'} \setminus \{1^n\} \subseteq S_\varphi$.
> Let T be a truth assignment in $S_{\varphi'} \setminus \{1^n\}$. Hence, there exists $k \in \{1, 2, \ldots, n\}$ such that $T(x_k) = 0$. Note that,
>
> $$\varphi' = \left(\bigwedge_{j=1,\ldots,m} (c_j \vee x_k) \right) \bigwedge \left(\bigwedge_{i \neq k} \bigwedge_{j=1,\ldots,m} (c_j \vee x_i) \right).$$
>
> Hence, in order to satisfy φ', T must satisfy $\bigwedge_{j=1,\ldots,m} c_j$, that is φ.
> Hence, $T \in S_\varphi$.

Now, suppose that we have a polynomial total time algorithm A for listing all truth assignments of φ'. Then, we can derive a polynomial total time algorithm B for listing all satisfying truth assignments of φ: the algorithm is basically algorithm A except that it rejects solution 1^n. Clearly, B is a polynomial total time algorithm since its time complexity is polynomial in the input size and polynomial in $|S_\varphi| + 1$. Hence, $\mathcal{L}$SAT $\in \mathbf{EP}$, and this implies that $\mathcal{L}\mathbf{P} = \mathbf{EP}$. □

More generally, let's consider the binary relation

$$R_{VSAT} := \left\{ (\langle \varphi, T^* \rangle, T) : \begin{array}{l} \varphi \text{ is a boolean CNF-formula,} \\ T, T^* \text{ are truth assignments satisfying } \varphi \end{array} \right\}.$$

Obviously, R_{VSAT} is an **NP** relation. Furthermore, analogously to R_{1VSAT}, the language $L(R_{VSAT})$ associated to relation R_{VSAT} is in **P**: by definition, $T^* \models \varphi$.

The listing problem $\mathcal{L}$VALID-SAT associated to R_{VSAT} is the following.

Problem. $\mathcal{L}$VALID-SAT
Input: a CNF-formula φ of n variables and a truth assignment such that $T \models \varphi$.
Output: all truth assignments that satisfy φ.

Corollary 3.12. $\mathcal{L}$VALID-SAT *is* $\mathcal{L}$**P**-*complete.*

Proof. Follows from Theorem 3.11. □

The reader may argue that relations R_{1VSAT} and R_{VSAT} are somehow artificial. We conclude this chapter by describing a more natural **NP** relation such that, the decision problem associated to it is in **P**, while the listing problem associated to it is $\mathcal{L}$**P**-complete.

3.4.1 $\mathcal{L}$Prime Implicants is $\mathcal{L}$P-complete

A *monotone boolean formula* φ is a boolean formula in which negation symbols do not appear, that is, all literals are positive and only disjunction and conjunction operations are allowed. An *implicant* of such formula is a subset I of the variables such that, setting all variables in I equal to 1, φ is satisfied whatever value is assigned to the variables not in I. A *prime implicant* is an implicant which is *minimal*, that is, it does not contain any other implicant as a proper subset. Consider the following binary relation:

$$R_{PI} := \left\{ (\varphi, I) : \begin{array}{l} \varphi \text{ is a monotone boolean formula,} \\ I \text{ is a prime implicant of } \varphi \end{array} \right\}$$

Note that relation R_{PI} is an **NP** relation. In fact, a prime implicant is a subset of the variables (hence R_{PI} is polynomially balanced). Furthermore, give a monotone boolean formula φ and a subset I of its variables, I is an implicant of φ if and only if the truth assignment T^I, defined as

$$T^I(x_i) = \begin{cases} 1, & \text{if } x_i \in I; \\ 0, & \text{otherwise,} \end{cases}$$

satisfies φ. Moreover, an implicant I is a prime implicant if, for every $x \in I$, $I \setminus \{x\}$ is not an implicant of φ. Hence, R_{PI} is polynomially decidable.

Let us call Prime Implicants the decision problem associated to relation R_{PI}. The following result holds.

Observation 3.13. Prime Implicants *is in* **P**.

Proof. Every monotone boolean formula has a prime implicant since every monotone boolean formula has at least one implicant: the set of all variables. □

Furthermore, given a monotone boolean formula, it is easy to compute in polynomial time one of its prime implicants applying the greedy strategy described in Algorithm 3.1. The correctness of Algorithm 3.1 can be easily verified.

Algorithm 3.1 FindPI(φ)

```
Parameters: $\varphi$ is a monotone boolean formula of $n$ variables $x_1, x_2, \ldots, x_n$;
Output: a prime implicant $I$ of $\varphi$;

  Set $I := \{x_1, x_2, \ldots, x_n\}$;
  for $j := 1$ to $n$ do
    if $I \setminus \{x_j\}$ is an implicant of $\varphi$ then
      remove $x_j$ from $I$;
  return $I$.
```

We now consider the listing problem $\mathcal{L}$Prime Implicants associated to relation R_{PI}.

Problem. $\mathcal{L}$Prime Implicants
Input: a monotone boolean formula φ of n variables.
Output: all prime implicants of φ.

In [31] it is proved that there is no polynomial total time algorithm listing all prime implicants of a monotone boolean formula unless $\mathbf{P} = \mathbf{NP}$. Next, we show that the problem of listing all prime implicants of a monotone boolean formula is $\mathcal{L}\mathbf{P}$-complete.

Theorem 3.14. $\mathcal{L}$Prime Implicants *is* $\mathcal{L}\mathbf{P}$*-complete.*

Proof. We prove that $\mathcal{L}$Prime Implicants is $\mathcal{L}$**P**-complete showing that the existence of a polynomial total time algorithm for $\mathcal{L}$Prime Implicants would imply a polynomial total time algorithm for $\mathcal{L}$Sat.
Let φ be a boolean CNF formula with variables $x_1, x_2, \ldots, x_n$, with $n > 1$. Let φ' be the boolean CNF formula with variables $t_1, f_1, t_2, f_2, \ldots, t_n, f_n$ obtained from φ applying the following replacement rules:

- replace each positive literal x_i with variable t_i;
- replace each negative literal $\bar{x}_i$ with variable f_i.

Let φ'' be the monotone boolean formula defined as

$$\varphi'' := A \vee (B \wedge \varphi'),$$

where

- $A := \bigvee_{j=1,\ldots,n} (t_j \wedge f_j)$;
- $B := \bigwedge_{j=1,\ldots,n} (t_j \vee f_j)$.

The following claims regarding the prime implicants of φ'' hold.

Claim 1. If I is a prime implicant of φ'', then I is a prime implicant either of A or of $B \wedge \varphi'$.
The claim trivially follows from the definition of φ''.

Claim 2. If I is a prime implicant of $B \wedge \varphi'$, then $t_j \in I$ or $f_j \in I$ for every $j \in \{1, 2, \ldots, n\}$. Hence, $|I| \geq n$.
The claim trivially follows from the definition of B.

Claim 3. If I is a prime implicant of A, then $I = \{t_j, f_j\}$ for some

$j \in \{1, 2, \dots, n\}$. Furthermore, $I = \{t_j, f_j\}$ is a prime implicant of φ'' for every $j \in \{1, 2, \dots, n\}$.

The claim follows from the definition of A and Claim 2.

Claim 4. I is a prime implicant of φ'' if and only if either I is a prime implicant of A, or I is a prime implicant of $B \wedge \varphi'$ such that $|I| = n$.

The claim follows from Claim 2 and Claim 3.

Now, let T be a truth assignment satisfying φ. We define $I_T := \{t_i : T(x_i) = 1\} \cup \{f_i : T(x_i) = 0\}$. Clearly, I_T is an implicant of φ'. Furthermore, by definition, exactly one between t_j and f_j belongs to I_T for every $j \in \{1, 2, \dots, n\}$. Hence, I_T is a prime implicant of $B \wedge \varphi'$. By Claim 4, we conclude that I_T is a prime implicant of φ''.

Conversely, let I be a prime implicant of both φ'' and $B \wedge \varphi'$. Hence, $|I| = n$ and I contains exactly one between t_j and f_j for every $j \in \{1, 2, \dots, n\}$. We define a truth assignment T^I for φ as follows:

$$T^I(x_i) = \begin{cases} 1, & \text{if } t_i \in I; \\ 0, & \text{if } f_i \in I. \end{cases}$$

By our assumptions, since I is a prime implicant of $B \wedge \varphi'$, we have that I is an implicant of φ'. Hence, by definition of φ', we have that T^I satisfies φ. Furthermore, given any two different prime implicants I_1 and I_2, the corresponding truth assignments T^{I_1} and T^{I_2} are different.

Hence, we have proved that the number of prime implicants of φ'' is equal to the number of truth assignments satisfying φ plus n. Furthermore, given a prime implicant of φ'' different than $\{t_j, f_j\}$ for every $j \in \{1, 2, \dots, n\}$, we can easily compute in linear time a truth assignment satisfying φ.

Assume that there exists a polynomial total time algorithm for listing all prime implicants of a monotone boolean formula. Then, we can apply it to φ''. Every time it outputs a prime implicant I of φ'', if $I = \{t_j, f_j\}$ for some $j \in \{1, 2, \ldots, n\}$ then we discard it, otherwise we compute an assignment T^I for φ as described above. Hence, we obtained an algorithm for listing all truth assignments satisfying φ of time complexity polynomial in the input size and polynomial in $|X| + n$, where $|X|$ is the number of truth assignments satisfying φ. That is, we obtained a polynomial total time algorithm for $\mathcal{L}$SAT. □

Theorem 3.14 shows that there exist natural **NP** relations such that the listing problem associated to them is $\mathcal{L}$**P**-complete, although the decision problem associated to them is in **P**. We conclude the chapter providing a result which relates the difficulty to efficiently list $\mathcal{L}$**P**-complete problems with the mighty question "Is **P** = **NP**?". In [54], Valiant claim that the listing problem associated to an **NP** relation whose corresponding decision problem is **NP**-complete is not **P**-enumerable in general unless **P** = **UP**, where **UP** is the class of languages accepted in polynomial time by *unambiguous non-deterministic Turing machines*, those with the property that for any input of the machine there is at most one accepting computation. Furthermore, it is also obvious that there is no polynomial delay listing algorithm for listing problem associated to **NP** relations whose corresponding decision problem is **NP**-complete, unless **P** = **NP** (e.g., see [10]). Thanks to Theorem 3.14 and since there is no polynomial total time algorithm listing all prime implicants of a monotone boolean formula unless **P** = **NP** [31], we can prove the following result.

Theorem 3.15. *There is no polynomial total time algorithm for any $\mathcal{L}$**P**-complete problem unless* **P** = **NP**.

Proof. Let E be an arbitrary $\mathcal{L}\mathbf{P}$-complete problem. Suppose that there exists a polynomial total time algorithm for solving E. By definition of $\mathcal{L}\mathbf{P}$-completeness, this implies that there exists a polynomial total time algorithm for solving any problem in $\mathcal{L}\mathbf{P}$. In particular, this implies that there exists a polynomial total time algorithm for solving $\mathcal{L}$PRIME IMPLICANTS. But, there is no polynomial total time algorithm listing all prime implicants of a monotone boolean formula unless $\mathbf{P} = \mathbf{NP}$ [31]. $\square$

Chapter 4

On listing solutions of a broad class of combinatorial optimization problems

In this chapter, we investigate the complexity of listing solutions of a broad class of combinatorial optimization problems. We show that every time we have a good polyhedral description of a combinatorial problem, then we can efficiently list all (optimal) solutions of the combinatorial problem.

4.1 Introduction

A *combinatorial ensemble* $\mathcal{C}$ is a family of couples $\langle S, \mathcal{F} \rangle$, also called *instances* of the combinatorial ensemble, where S, also called the *ground set*, is a finite set of elements and $\mathcal{F}$, also called the *feasible family*, is a family of subsets of S. The members of $\mathcal{F}$ are called *feasible solutions*. We assume $\mathcal{F}$ to be given implicitly by a *compact representation*, as shown by the following examples of combinatorial ensemble:

- the matching ensemble, where S is the edge set of a graph G, and $\mathcal{F}$ is the family of matchings of G. Here, G is a compact representation of $\mathcal{F}$;

- the spanning tree ensemble, where S is the edge set of a graph G, and $\mathcal{F}$ is the family of spanning trees of G. Here, G is a compact representation of $\mathcal{F}$;
- the truth assignment ensemble, where S is the set of variables of a boolean formula φ and $\mathcal{F}$ is the family of truth assignments satisfying φ. Here, φ is a compact representation of $\mathcal{F}$.

The size of an instance of a combinatorial ensemble is actually the size of the compact representation of $\mathcal{F}$.

In the *combinatorial decision problem* associated to a combinatorial ensemble, given an instance $\langle S, \mathcal{F} \rangle$, the goal is to decide whether $\mathcal{F}$ is an empty family or not. An example of combinatorial decision problem is the following one.

Example 1. Satisfiability
Given a boolean formula φ of n variables, is φ satisfiable?

In the *combinatorial search problem* associated to a combinatorial ensemble, given an instance $\langle S, \mathcal{F} \rangle$, the goal is to return a member of $\mathcal{F}$ (if $\mathcal{F}$ is not empty). An example of combinatorial search problem is the following one.

Example 2. Matching Problem
Given a graph $G = (V, E)$, find a matching of G.

Among the problems that we consider in this chapter of the thesis, there is the *combinatorial listing problem* associated to a combinatorial ensemble:

given an instance $\langle S, \mathcal{F} \rangle$, list all members of $\mathcal{F}$ (if $\mathcal{F}$ is not empty).

Example 3. Spanning Tree Listing Problem
Given a graph $G = (V, E)$, list all spanning trees of G.

Given a ground set S, to each subset F of S we can associate univocally a vector $\chi^F \in \{0, 1\}^S$, called the *incidence vector* of F, defined as follows: $\chi^F_e = 1$ if element $e \in F$, while $\chi^F_e = 0$ if element $e \notin F$. Given an instance $\langle S, \mathcal{F} \rangle$ of a combinatorial ensemble, we denote by $I^{\mathcal{F}}$ the set of incidence vectors corresponding to the feasible solutions in $\mathcal{F}$. Note that if $\mathcal{F} = 2^S$, $I^{\mathcal{F}}$ is the set of the vertices of the 0/1-hypercube in $\mathbb{R}^S$. Given any instance $\langle S, \mathcal{F} \rangle$, the convex hull of $I^{\mathcal{F}}$ is a 0/1-polytope $P_{\langle S,\mathcal{F} \rangle}$ whose vertices are exactly the vectors in $I^{\mathcal{F}}$. It follows that, given an instance $\langle S, \mathcal{F} \rangle$ of a combinatorial ensemble, the combinatorial decision problem corresponds to deciding whether $P_{\langle S,\mathcal{F} \rangle}$ is empty or not, the combinatorial search problem corresponds to returning a vertex of $P_{\langle S,\mathcal{F} \rangle}$ (if $P_{\langle S,\mathcal{F} \rangle}$ is not empty), and the combinatorial listing problem corresponds to returning all vertices of $P_{\langle S,\mathcal{F} \rangle}$ (if $P_{\langle S,\mathcal{F} \rangle}$ is not empty).

We say that a combinatorial ensemble has a *compact description* (resp. *dominant compact description*), if the description of $P_{\langle S,\mathcal{F} \rangle}$ (resp. $P^{\uparrow}_{\langle S,\mathcal{F} \rangle}$, the dominant of $P_{\langle S,\mathcal{F} \rangle}$) in terms of inequalities can be obtained in polynomial time in the size of instance $\langle S, \mathcal{F} \rangle$.

We say that a combinatorial ensemble is *separable* (resp. *dominant separable*), if we have a *separation algorithm* for $P_{\langle S,\mathcal{F} \rangle}$ (resp. $P^{\uparrow}_{\langle S,\mathcal{F} \rangle}$), that is, a polynomial time algorithm that, given a rational $z \in \mathbb{R}^S_{\geq 0}$, tests if z belongs to $P_{\langle S,\mathcal{F} \rangle}$ (resp. $P^{\uparrow}_{\langle S,\mathcal{F} \rangle}$), or, if not, returns a rational vector $c \in \mathbb{R}^S$ such that

$cx < cz$ for each $x \in P_{\langle S,\mathcal{F}\rangle}$ (resp. $x \in P^{\uparrow}_{\langle S,\mathcal{F}\rangle}$). It follows by definition that if a combinatorial ensemble has a compact description, then it is separable: in fact, given a rational $z \in \mathbb{R}^S_{\geq 0}$, if z satisfies all the inequalities in the description of $P_{\langle S,\mathcal{F}\rangle}$, then $z \in P_{\langle S,\mathcal{F}\rangle}$; otherwise, if $ax \leq b$ is an inequality in the description of $P_{\langle S,\mathcal{F}\rangle}$ such that $az > b$, then we return vector a. However, note that the contrary is not true. That is, there exist separable combinatorial ensembles which do not have a compact description: an example, is the matching ensemble. Analogously, it follows that if a combinatorial ensemble has a dominant compact description, then it is dominant separable.

In [6], it is shown that the combinatorial listing problem is strongly **P**-enumerable for combinatorial ensembles which have a compact description: actually, the algorithm proposed runs in polynomial space and with polynomial delay. One of the new contributions presented in this chapter of the thesis, is that the combinatorial listing problem is strongly **P**-enumerable for separable combinatorial ensembles.

Theorem 4.1. *For any separable combinatorial ensemble, the combinatorial listing problem is polynomial space polynomial delay solvable.*

Theorem 4.1 follows as corollary of Theorem 4.4 introduced later in this chapter.

In the *combinatorial optimization problem* associated to a combinatorial ensemble, feasible solutions are additionally evaluated by an *objective function* and the goal is to find a feasible solution with minimum objective function value. In details, to each element e of the ground set S corresponds a *weight* $w_e \in \mathbb{R}$ and we define the objective function as $w(F) := \sum_{e \in F} w_e$, for any $F \subseteq S$. Hence, the input of a combinatorial optimization problem is a triplet $\langle S, \mathcal{F}, w\rangle$. An example of combinatorial optimization problem is the following one.

Example 4. Minimum Weight Spanning Tree Problem
Given a graph $G = (V, E)$ and a weight $w_e \in \mathbb{R}$ for each edge $e \in E$, find a minimum weight spanning tree of G.

From a polyhedral combinatorics point of view, the combinatorial optimization problem corresponds to the problem of returning a vertex χ^F of $P_{\langle S,\mathcal{F}\rangle}$ such that $w \cdot \chi^F$ is minimum (if $P_{\langle S,\mathcal{F}\rangle}$ is not empty).

In this chapter of the thesis we investigate the complexity of listing all minimum value solutions of combinatorial ensembles. With a slight abuse of notation, we also say that we are interested in listing all optimal solutions of combinatorial optimization problems. In particular, we study this listing problem when,

Case 1 the combinatorial optimization problem is polynomial time solvable for any weight vector $w \in \mathbb{R}^S$;

Case 2 the combinatorial optimization problem is polynomial time solvable for any weight vector $w \in \mathbb{R}^S_{\geq 0}$.

In Section 4.2, we consider those combinatorial ensembles satisfying the hypothesis of Case 1 above. Examples are the perfect matching ensemble (Edmonds' strongly polynomial time algorithm [17] solves the minimum weight perfect matchings problem) or the spanning tree ensemble (Kruskal's method

[40] or Prim's method [47] return a minimum weight spanning tree in a graph). More generally, as we show next, the hypothesis of Case 1 above is equivalent to assume that the combinatorial ensemble is separable.

Given a combinatorial ensemble $\mathcal{C}$, we define the class of polytopes $\mathcal{P}_{\mathcal{C}} := \{P_{\langle S,\mathcal{F}\rangle} \ : \ \langle S,\mathcal{F}\rangle$ is an instance of $\mathcal{C}\}$. We refer to $\mathcal{P}_{\mathcal{C}}$ as the class of polytopes associated to the combinatorial ensemble $\mathcal{C}$.

Observation 4.2. *Given any combinatorial ensemble $\mathcal{C}$, $\mathcal{P}_{\mathcal{C}}$ is a proper class of polytopes.*

Proof. See Appendix A. □

Hence, since $\mathcal{P}_{\mathcal{C}}$ is a proper class of polytopes, the *Optimization≡Separation* Theorem (see Theorem 2.3) implies the following.

Theorem 4.3. *A combinatorial ensemble is separable if and only if the combinatorial optimization problem associated to the combinatorial ensemble is polynomial time solvable for any weight vector $w \in \mathbb{R}^S$.*

One of the major result proposed in this chapter of the thesis is the following one.

Theorem 4.4. *For any separable combinatorial ensemble, the following listing problems are polynomial space polynomial delay solvable:*

- *list all feasible solutions;*

- *list all feasible solutions of maximum or minimum cardinality;*
- *given a weight vector $w \in \mathbb{R}^S$, list all feasible solutions of maximum or minimum value;*
- *given a weight vector $w \in \mathbb{R}^S$, list all maximum or minimum cardinality feasible solutions of maximum or minimum value.*

In Section 4.2, we provide an algorithmic proof of Theorem 4.4. Theorem 4.4 implies for example that, the (minimum weight) perfect matchings of a graph and the (minimum weight) spanning trees of a graph are polynomial space polynomial delay listable.

In Section 4.3, we consider those combinatorial ensembles satisfying the hypothesis of Case 2 above. Clearly, the hypothesis of Case 1 implies the hypothesis of Case 2. However, the contrary is not true unless **P** = **NP**. Examples are the $s-t$ cut ensemble ([22, 35, 19]) or the $s-t$ path ensemble ([16]) for which we can compute in polynomial time a minimum value solution only when the weights are nonnegative. Note that, when the weights are nonnegative, the problem of minimizing a linear function over a 0/1-polytope P is equivalent to the problem of minimizing the same linear function over the dominant of P. Actually, the problem of minimizing $w \cdot x$ with $w \in \mathbb{R}^S_{\geq 0}$ over the dominant $P^{\uparrow}$ of a 0/1-polytope P is polynomial time solvable if and only if the optimization problem over $P^{\uparrow}$ is polynomial time solvable. More generally, as we show next, the hypothesis of Case 2 above is equivalent to assume that the combinatorial ensemble is dominant separable.

Given a combinatorial ensemble $\mathcal{C}$, we define the class of polyhedra $\mathcal{P}^{\uparrow}_{\mathcal{C}} := \{P^{\uparrow}_{\langle S,\mathcal{F}\rangle} : \langle S, \mathcal{F}\rangle \text{ is an instance of } \mathcal{C}\}$. We refer to $\mathcal{P}^{\uparrow}_{\mathcal{C}}$ as the class of dominant polyhedra associated to the combinatorial ensemble $\mathcal{C}$.

Observation 4.5. *Given any combinatorial ensemble $\mathcal{C}$, $\mathcal{P}_{\mathcal{C}}^{\uparrow}$ is a proper class of polyhedra.*

Proof. See Appendix A. $\square$

Hence, since $\mathcal{P}_{\mathcal{C}}^{\uparrow}$ is a proper class of polyhedra, the *Optimization≡Separation* Theorem (see Theorem 2.3) implies the following.

Theorem 4.6. *A combinatorial ensemble is dominant separable if and only if the combinatorial optimization problem associated to the combinatorial ensemble is polynomial time solvable for any weight vector $w \in \mathbb{R}^S_{\geq 0}$.*

In Section 4.3, we provide an example of dominant separable combinatorial ensemble for which there exists no polynomial total time algorithm to list all optimal solutions unless a polynomial total time algorithm exists to list all truth assignments of a CNF boolean formula. That is, we show that for this dominant separable combinatorial ensemble, the problem of listing all optimal solutions is $\mathcal{L}$**P**-hard.

However, we show that there exists a polynomial space polynomial delay algorithm that lists all optimal solutions when $w \in \mathbb{R}^S_{>0}$.

Theorem 4.7. *For any dominant separable combinatorial ensemble, the following listing problems are polynomial space polynomial delay solvable:*

- *list all feasible solutions of minimum cardinality;*
- *given a weight vector $w \in \mathbb{R}^S_{>0}$, list all feasible solutions of minimum value;*

- *given a weight vector $w \in \mathbb{R}^S_{>0}$, list all minimum cardinality feasible solutions of minimum value.*

In Section 4.3, we provide an algorithmic proof of Theorem 4.7. Theorem 4.7 implies that, when the weights are strictly positive, we can list in polynomial space and polynomial delay all minimum weight $s-t$ cuts of a graph and all minimum weight $s-t$ paths of a graph.

Many ad-hoc algorithms as well as a general method have been proposed for the listing problems considered in this chapter of the thesis. We propose a brief review of some of these results in Section 4.4 at the end of the chapter, where we compare them with our results.

4.2 Case 1. The combinatorial optimization problem is polynomial time solvable for any weight vector $w \in \mathbb{R}^S$

Let $\langle S, \mathcal{F} \rangle$ denote the generic instance of a specific combinatorial ensemble, where S is the ground set and $\mathcal{F}$ is a family of subsets of S. Let $w \in \mathbb{R}^S$ be a weight vector. Our main hypothesis in this section of the thesis is that there exists a polynomial time algorithm $\textsc{Min}(S, \mathcal{F}, w)$ that, for any instance $\langle S, \mathcal{F} \rangle$ of the combinatorial ensemble and any weight vector $w \in \mathbb{R}^S$, returns *nil* if $\mathcal{F}$ is empty, or $\min_{F \in \mathcal{F}} w \cdot \chi^F$ otherwise.

We show that for separable combinatorial ensembles, the problems:

A. list all members of $\mathcal{F}$;

B. list all members of maximum or minimum cardinality of $\mathcal{F}$;

C. list all members F of $\mathcal{F}$ such that $w \cdot \chi^F$ is maximum or minimum;

D. list all members F of maximum or minimum cardinality of $\mathcal{F}$ such that $w \cdot \chi^F$ is maximum or minimum;

are all strongly **P**-enumerable. Indeed, for these listing problems we provide polynomial space polynomial delay algorithms. Actually, among the four listing problems proposed, we just need to solve one of the two variants of Problem C, since the other problems are special cases of Problem C. In fact, assuming that we have an algorithm for listing all members F of $\mathcal{F}$ such that $w \cdot \chi^F$ is minimum, we have that:

- taking $w \equiv 0$ we list all feasible solutions (i.e., we get Problem A);
- taking $w \equiv 1$ we list all minimum cardinality feasible solutions, while taking $w \equiv -1$ we list all maximum cardinality feasible solutions (i.e., we get Problem B);
- considering $-w$ instead of w, we list all members F of $\mathcal{F}$ such that $w \cdot \chi^F$ is maximum;
- let $W := (1 + \sum_{e \in S} w_e)\chi^S$. Considering weight vector $+W + w$ (resp. $+W - w$) instead of w, we list all minimum cardinality members F of $\mathcal{F}$ such that $w \cdot \chi^F$ is minimum (resp. maximum). Similarly, considering weight vector $-W + w$ (resp. $-W - w$) instead of w, we list all maximum cardinality members F of $\mathcal{F}$ such that $w \cdot \chi^F$ is minimum (resp. maximum).

Before defining formally our algorithm for solving Problem C, we would like to give a flavour of the idea inspiring it. We use $\textsc{Min}(S, \mathcal{F}, w)$ as an oracle, in order to check if there are optimal solutions of the combinatorial optimization problem considered which include or not include a particular element s of the ground set. To be more precise, to check if there exists a minimum value solution that does not contain element s, we can simply augment its weight by 1, and apply algorithm $\textsc{Min}(S, \mathcal{F}, w)$ on this modified instance. If the minimum value is the same as the starting instance, then there exists a minimum value solution that does not contain element s, otherwise there are no minimum value solutions of the problem which do not contain s. Analogously, to check if there exists a minimum value solution that contains element s, we can simply decrease its weight by 1, and apply algorithm $\textsc{Min}(S, \mathcal{F}, w)$ on this modified instance. If the minimum value of this modified instance is equal to the minimum value of the starting instance minus 1, then there exists a minimum value solution that contains element s, otherwise there are no minimum value solutions of the problem which contain s. We can apply this idea recursively on each element of the ground set, obtaining a depth first search like algorithm, where the leaves of the recursion tree correspond to solutions of minimum value for the given instance of the combinatorial optimization problem considered.

Algorithm 4.1 solves Problem C, where ENUMERATE is the recursive procedure defined in Procedure 4.2.

Algorithm 4.1 $\textsc{List}(S, \mathcal{F}, w)$

Parameters: S is a finite set of elements, $\mathcal{F}$ is the family of feasible solutions, $w \in \mathbb{R}^S$;

```
1: if $\textsc{Min}(S, \mathcal{F}, w) \neq nil$ then
2:     $opt \leftarrow \textsc{Min}(S, \mathcal{F}, w)$;
3:     ENUMERATE $(S, \mathcal{F}, \emptyset, S, w, opt)$.
```

Procedure 4.2 ENUMERATE($S, \mathcal{F}, \check{F}, \tilde{S}, w, opt$)

Parameters: S is a finite set of elements, $\mathcal{F}$ is the family of feasible solutions, $\check{F}$ is a partial optimal solution, $\tilde{S}$ is the subset of S still to be considered, $w \in \mathbb{R}^S$, *opt* is the value of any optimal solution of instance $\langle S, \mathcal{F}, w\rangle$;

```
1: if $\tilde{S} = \emptyset$ then
2:     output $\check{F}$; return;
3: let $s$ be any element of $\tilde{S}$;
4: $w^+ \leftarrow w + 1 \cdot \chi^{\{s\}}$; $w^- \leftarrow w - 1 \cdot \chi^{\{s\}}$;
5: $opt^+ \leftarrow$ MIN($S, \mathcal{F}, w^+$); $opt^- \leftarrow$ MIN($S, \mathcal{F}, w^-$);
6: if $opt^+ = opt$ then
7:     ENUMERATE ($S, \mathcal{F}, \check{F}, \tilde{S} \setminus \{s\}, w^+, opt^+$);
8: if $opt^- = opt - 1$ then
9:     ENUMERATE ($S, \mathcal{F}, \check{F} \cup \{s\}, \tilde{S} \setminus \{s\}, w^-, opt^-$).
```

4.2.1 Correctness of Algorithm 4.1

In this section, we prove the correctness of Algorithm 4.1. Let $S := \{s_1, s_2, \ldots, s_{|S|}\}$. To simplify our exposition, we can assume without loss of generality that we choose the elements from $\tilde{S}$ following a fixed order. For example, we can assume that element s chosen at line 3 of Procedure 4.2 is the one with the smallest index in $\tilde{S}$, i.e. $s := min_i\{s_i : s_i \in \tilde{S}\}$.

Lemma 4.8. *Let* ENUMERATE($S, \mathcal{F}, \check{F}', \tilde{S}', w', opt'$) *be an arbitrary recursive call of Procedure 4.2. For any optimal solution F^* of instance $\langle S, \mathcal{F}, w'\rangle$ we have that $w'(F^*) = opt'$ and $\check{F}' \subseteq F^* \subseteq \check{F}' \cup \tilde{S}'$.*

Proof. We prove the statement of the lemma by induction on the cardinality of $S \setminus \tilde{S}'$.

Base Case: $|S \setminus \tilde{S}'| = 0$. This situation arises when Algorithm 4.1 invokes

Procedure 4.2. Hence, we have that $\check{F}' = \emptyset$, $\tilde{S}' = S$, $w' = w$ and $opt' = opt$. Therefore, the statement is clearly true since for any optimal solution F^* of instance $\langle S, \mathcal{F}, w\rangle$, we have that $w(F^*) = opt$ and $\emptyset \subseteq F^* \subseteq S$.

Inductive Case: $|S \setminus \tilde{S}'| = k + 1$. We assume that the statement of the lemma is true for every recursive call such $|S \setminus \tilde{S}''| = k$ and we prove it for any recursive call such that $|S \setminus \tilde{S}'| = k + 1$.
Let ENUMERATE$(S, \mathcal{F}, \check{F}_{k+1}, \tilde{S}_{k+1}, w_{k+1}, opt_{k+1})$ be an arbitrary recursive call such that $|S \setminus \tilde{S}_{k+1}| = k+1$. Let F^* be any optimal solution for instance $\langle S, \mathcal{F}, w_{k+1}\rangle$.
By construction, let ENUMERATE$(S, \mathcal{F}, \check{F}_k, \tilde{S}_k, w_k, opt_k)$ be the (unique) recursive call which invoke ENUMERATE$(S, \mathcal{F}, \check{F}_{k+1}, \tilde{S}_{k+1}, w_{k+1}, opt_{k+1})$. Two situations need to be considered.

- $\check{F}_{k+1} = \check{F}_k$, $\tilde{S}_{k+1} = \tilde{S}_k \setminus \{s_{k+1}\}$, $w_{k+1} = w_k + 1\chi^{\{s_{k+1}\}}$ and $opt_{k+1} = opt_k$.
  Firstly, note that $w_{k+1}(F^*) = opt_{k+1}$ follows from line 5 of Procedure 4.2.
  Clearly, $s_{k+1} \notin F^*$, otherwise F^* would be a solution of $\langle S, \mathcal{F}, w_k\rangle$ of cost $opt_k - 1$. Absurd, since the cost of an optimal solution in $\langle S, \mathcal{F}, w_k\rangle$ is opt_k. Furthermore, since $opt_{k+1} = opt_k$ and $s_{k+1} \notin F^*$, F^* is also an optimal solution for $\langle S, \mathcal{F}, w_k\rangle$. By inductive hypothesis we have that $\check{F}_k \subseteq F^* \subseteq \tilde{S}_k \cup \check{F}_k$. Hence, $\check{F}_{k+1} \subseteq F^* \subseteq \tilde{S}_{k+1} \cup \check{F}_{k+1}$.

- $\check{F}_{k+1} = \check{F}_k \cup \{s_{k+1}\}$, $\tilde{S}_{k+1} = \tilde{S}_k \setminus \{s_{k+1}\}$, $w_{k+1} = w_k - 1\chi^{\{s_{k+1}\}}$ and $opt_{k+1} = opt_k - 1$.
  Firstly, note that $w_{k+1}(F^*) = opt_{k+1}$ follows from line 5 of Procedure 4.2.
  Clearly, $s_{k+1} \in F^*$, otherwise F^* would be a solution of $\langle S, \mathcal{F}, w_k\rangle$ of cost $opt_k - 1$. Absurd, since the cost of an optimal solution in $\langle S, \mathcal{F}, w_k\rangle$ is opt_k. Furthermore, since $opt_{k+1} = opt_k - 1$ and $s_{k+1} \in F^*$, F^* is also

an optimal solution for $\langle S, \mathcal{F}, w_k \rangle$. By inductive hypothesis we have that $\check{F}_k \subseteq F^* \subseteq \tilde{S}_k \cup \check{F}_k$. Hence, $\check{F}_{k+1} \subseteq F^* \subseteq \tilde{S}_{k+1} \cup \check{F}_{k+1}$.

$\square$

Lemma 4.9. *Consider any recursive call* ENUMERATE$(S, \mathcal{F}, F', \tilde{S}', w', \mathit{opt}')$ *of Procedure 4.2 such that* $\tilde{S}' \neq \emptyset$. *Let* $OPT(S, \mathcal{F}, w')$ *be the set of optimal solution of instance* $\langle S, \mathcal{F}, w' \rangle$. *We have that,*

$$OPT(S, \mathcal{F}, w') = OPT(S, \mathcal{F}, w^+) \,\dot{\cup}\, OPT(S, \mathcal{F}, w^-).^1$$

Proof. Let s be the element chosen at line 3 of Procedure 4.2 during the recursive call ENUMERATE$(S, \mathcal{F}, F', \tilde{S}', w', \mathit{opt}')$. In order to prove the statement of the lemma, we need to consider separately the two inclusions.

$OPT(S, \mathcal{F}, w') \supseteq OPT(S, \mathcal{F}, w^+) \cup OPT(S, \mathcal{F}, w^-)$**.** Let F^+ be any optimal solution of instance $\langle S, \mathcal{F}, w^+ \rangle$. By Lemma 4.8 we have that $s \notin F^+$. Furthermore, $w^+(F^+) = \mathit{opt}'$. Hence, F^+ is an optimal solution of instance $\langle S, \mathcal{F}, w' \rangle$. Analogously, let F^- be any optimal solution of instance $\langle S, \mathcal{F}, w^- \rangle$. By Lemma 4.8 we have that $s \in F^-$. Furthermore, $w^-(F^-) = \mathit{opt}' - 1$. Hence, F^- is an optimal solution of instance $\langle S, \mathcal{F}, w' \rangle$.

$OPT(S, \mathcal{F}, w') \subseteq OPT(S, \mathcal{F}, w^+) \cup OPT(S, \mathcal{F}, w^-)$**.** Let F^* be any optimal solution of instance $\langle S, \mathcal{F}, w' \rangle$. If $s \in F^*$, then $w^-(F^*) = \mathit{opt}_k - 1$ and F^* is an optimal solution for $\langle S, \mathcal{F}, w^- \rangle$. Otherwise, if $s \notin F^*$, then $w^+(F^*) = \mathit{opt}_k$ and F^* is an optimal solution for $\langle S, \mathcal{F}, w^+ \rangle$.

$^1\dot{\cup}$ stands for the disjoint union.

Furthermore, by Lemma 4.8 we have that $OPT(S, \mathcal{F}, w^+) \cap OPT(S, \mathcal{F}, w^-) = \emptyset$. ☐

Theorem 4.10. *F is a solution returned by algorithm* LIST$(S, \mathcal{F}, w)$ *if and only if F is an optimal solution of Problem C associated to instance $\langle S, \mathcal{F}, w \rangle$.*

Proof. Follows from Lemma 4.8 and Lemma 4.9. ☐

Before proving the time and space complexity of our algorithm, we would like to observe that if we label with $s_1, s_2, \dots s_n$ the elements of the ground set, and we choose element s at line 3 of Procedure 4.2 to be $s := min_i\{s_i : s_i \in \tilde{S}\}$, then our listing algorithm outputs the (optimal) feasible solutions in lexicographic order according to the labeling: that is, given two solutions F_1 and F_2, if s_k is the element of the ground set with smallest index in the symmetric difference between F_1 and F_2, then the solution containing s_k is outputted after the solution not containing s_k.

4.2.2 Complexity of Algorithm 4.1

In this subsection we discuss both time and space complexity of Algorithm 4.1.

We recall that we assumed MIN$(S, \mathcal{F}, w)$ to be polynomial time computable in the size of the instance. Note that $\mathcal{O}(|S|)$ recursive calls to Procedure 4.2 are needed to return the first solution. Furthermore, once a solution is returned, at most $\mathcal{O}(|S|)$ recursive calls to Procedure 4.2 are needed to return the next solution or to halt. Hence, Algorithm 4.1 is a polynomial

delay algorithm, and this implies that Problem C is **P**-enumerable.
It can be easily seen that the space complexity of Algorithm 4.1 is polynomial in the input size only. Hence, Algorithm 4.1 is a polynomial space polynomial delay algorithm, and this implies that Problem C is strongly **P**-enumerable.

4.2.3 Example: Algorithm 4.1 lists all minimum spanning trees

In this section we present a simple application of Algorithm 4.1 to the Minimum Spanning Tree Problem.

Finding a minimum spanning tree is a classical combinatorial optimization problem. There exist many algorithms that solve this problem: among them, for example, Kruskal's algorithm [40] and Prim's algorithm [47]. We can apply one of these algorithms as black box $\textsc{Min}(S, \mathcal{F}, w)$ in our algorithm $\textsc{List}(S, \mathcal{F}, w)$, in order to list all minimum spanning trees.

The input of the Minimum Spanning Tree problem is a graph $G = (V, E)$ and a weight vector $w \in \mathbb{R}^E$. According to our definition of combinatorial ensemble, we have that $S = E$ and the graph G is a compact representation of $\mathcal{F}$.

Consider for example the graph G drawn on the left side of Figure 4.1. Note that, the trees T_1, T_2, T_3, T_4 drawn on the right side of Figure 4.1 are all the spanning trees of G. We define the weight vector w as follow: $w(s_1) := 1, w(s_2) := 2, w(s_3) := 3, w(s_4) := 3, w(s_5) := 2$, that is $w := (1, 2, 3, 3, 2)$. According to this weight vector, we have that the minimum spanning tree value is 8, and there are two minimum spanning trees, T_1 and T_2. Indeed, $w(T_1) = 8, w(T_2) = 8, w(T_3) = 9$ and $w(T_4) = 9$.

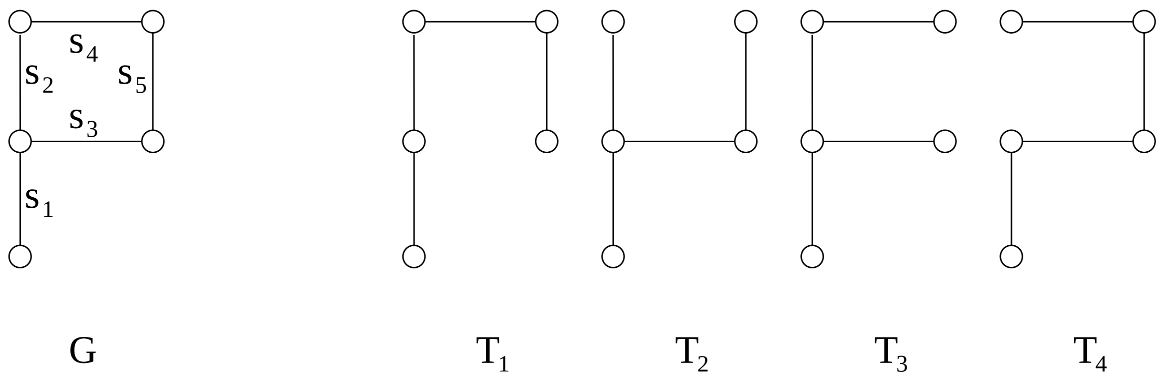

Figure 4.1: Minimum Weight Spanning Tree Example: a graph G and all its spanning trees T_1, T_2, T_3, T_4.

Now, we apply Algorithm 4.1 to the given input instance, in order to obtain all minimum spanning trees. Without loss of generality, we can assume that element s chosen at line 3 of Procedure 4.2 is the one with the smallest index in $\tilde{S}$, i.e. $s := min_i\{s_i : s_i \in \tilde{S}\}$.

Figure 4.2 describes the behavior of our listing algorithm when applied on the given input instance. Each arrow labeled `yes` in Figure 4.2 means that in the recursive call considered, condition $opt^+ = opt$ at line 6 of Procedure 4.2 is true, while each arrow labeled `no` in Figure 4.2 means that in the recursive call considered, condition $opt^- = opt - 1$ at line 8 of Procedure 4.2 is true. The leaves of the recursive tree drawn in Figure 4.2 correspond to the two minimum spanning tree value solutions T_1 and T_2.

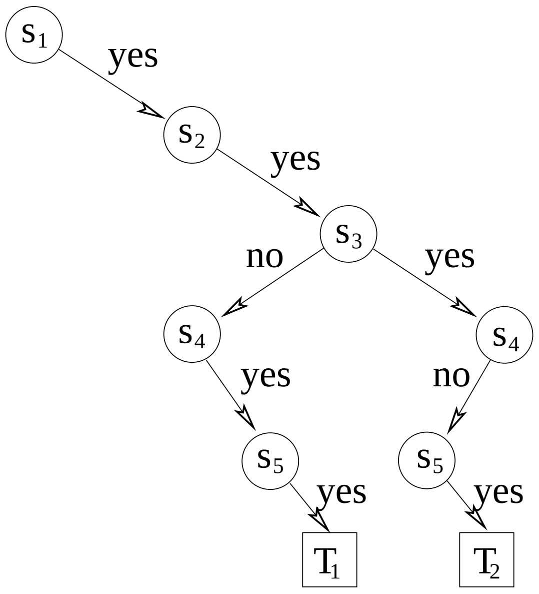

Figure 4.2: Minimum Weight Spanning Tree Example: Recursion Tree.

4.3 Case 2. The combinatorial optimization problem is polynomial time solvable for any weight vector $w \in \mathbb{R}^S_{\geq 0}$

In Section 4.2 we have considered those combinatorial optimization problems for which we are able to compute $\text{MIN}(S, \mathcal{F}, w)$ for any weight vector $w \in \mathbb{R}^S$. Unfortunately, there are many combinatorial optimization problems for which this is not always the case. Think for example to the minimum weight cut problem: the problem of computing a minimum weight cut is polynomially solvable only if the weights are non-negative, while computing a maximum weight cut is an **NP**-hard problem.

In this section we consider those combinatorial optimization problems such that there exists a polynomial time algorithm $\text{MIN}(S, \mathcal{F}, w)$ that, for any instance $\langle S, \mathcal{F}, w\rangle$ with $w \in \mathbb{R}^S_{\geq 0}$, returns *nil* if $\mathcal{F}$ is empty, or $\min_{F\in\mathcal{F}} w \cdot \chi^F$

otherwise. We recall that this hypothesis is equivalent to consider dominant separable combinatorial ensembles.

It should be clear that Algorithm 4.1 defined in Section 4.2 cannot do the job: in fact, we are not allowed to decrease the weight of an element of the ground set to a negative value, otherwise $\textsc{Min}(S, \mathcal{F}, w)$ cannot be applied. Indeed, we show that there exists a combinatorial optimization problem such that we can compute a minimum objective value solution for any $w \in \mathbb{R}^S_{\geq 0}$, but such that, listing all its optimal solutions is an $\mathcal{L}\mathbf{P}$-hard problem, that is, if we can list all its optimal solutions, then we can list all truth assignments that satisfies a CNF-formula.

However, as a partial compensation of this negative result, we first of all show that for any instance with $w \in \mathbb{R}^S_{>0}$, there exists a polynomial space polynomial delay algorithm that solves the problem:

A. list all members F of $\mathcal{F}$ such that $w \cdot \chi^F$ is minimum.

As a by-product of this positive result, we get that for any dominant separable combinatorial ensemble, there exists a polynomial space polynomial delay algorithm that solves the problems:

B. list all members of minimum cardinality of $\mathcal{F}$;

C. list all members F of minimum cardinality of $\mathcal{F}$ such that $w \cdot \chi^F$ is minimum.

Clearly, Problem B and Problem C are special cases of Problem A. In fact,

- taking $w \equiv 1$ we list all minimum cardinality feasible solutions, (i.e., we get Problem B);

- let $W := (1 + \sum_{e \in S} w_e)\chi^S$. Considering weight vector $+W + w$ instead of w, we list all minimum cardinality members F of $\mathcal{F}$ such that $w \cdot \chi^F$ is minimum (i.e., we get Problem C).

The algorithm that we present for solving Problem A is an adaptation of Algorithm 4.1 in order to work with those combinatorial optimization problems considered in this section. The adaptation is very simple: we change the value used to increase/decrease the weight of the elements of the ground set in line 4 of procedure Procedure 4.2. Instead of value 1, we add/remove a value $\epsilon > 0$ such that $\epsilon \in \left(0, \min_{i=1,\ldots,|S|}\{w_i\}\right)$, where w_i is the i-th component of vector w. With this modification we ensure that no zero or negative weights arise during the execution of the algorithm.

Algorithm 4.3 solves Problem A, where ENUMERATE$_{>0}$ is the recursive procedure defined in Procedure 4.4.

Algorithm 4.3 LIST$_{>0}(S, \mathcal{F}, w)$

Parameters: S is a finite set of elements, $\mathcal{F}$ is the family of feasible solutions, $w \in \mathbb{R}^S_{>0}$;

```
1: if MIN($S, \mathcal{F}, w$) $\neq$ nil then
2:     opt ←MIN($S, \mathcal{F}, w$); $\epsilon \leftarrow \frac{\min_{i=1,\ldots,|S|}\{w_i\}}{2}$;
3:     ENUMERATE$_{>0}$($S, \mathcal{F}, \emptyset, S, w, opt, \epsilon$).
```

The proof of correctness of Algorithm 4.3 and the complexity analysis are the same as those presented for Algorithm 4.1 in Case 1. Hence Algorithm 4.3 can list efficiently all optimal solutions of combinatorial optimization problems like the minimum weight $s - t$ cut problem when all weights involved are strictly positive.

Procedure 4.4 ENUMERATE$_{>0}(S, \mathcal{F}, F, \tilde{S}, w, opt, \epsilon)$

Parameters: S is a finite set of elements, $\mathcal{F}$ is the family of feasible solutions, $\check{F}$ is a partial optimal solution, $\tilde{S}$ is the subset of S still to be considered, $w \in \mathbb{R}^S_{>0}$, *opt* is the value of any optimal solution of instance $\langle S, \mathcal{F}, w\rangle$, $\epsilon \in \mathbb{R}_{>0}$;

```
1: if $\tilde{S} = \emptyset$ then
2:    output $F$; return;
3: let $s$ be any element of $\tilde{S}$;
4: $w^+ \leftarrow w + \epsilon\chi^{\{s\}}$; $w^- \leftarrow w - \epsilon\chi^{\{s\}}$;
5: $opt^+ \leftarrow$ MIN$(S, \mathcal{F}, w^+)$; $opt^- \leftarrow$ MIN$(S, \mathcal{F}, w^-)$;
6: if $opt^+ = opt$ then
7:    ENUMERATE$_{>0}(S, \mathcal{F}, F, \tilde{S} \setminus \{s\}, w^+, opt^+, \epsilon)$;
8: if $opt^- = opt - \epsilon$ then
9:    ENUMERATE$_{>0}(S, \mathcal{F}, F \cup \{s\}, \tilde{S} \setminus \{s\}, w^-, opt^-, \epsilon)$.
```

Now, as anticipated, we show that there exists a combinatorial optimization problem such that the minimum objective value feasible solution can be computed in polynomial time for any $w \in \mathbb{R}^S_{\geq 0}$, but such that it is $\mathcal{L}$**P**-hard to list all its optimal solutions. The problem that we consider is the following one.

Problem. WEIGHTED-0VALID-SAT
Input: a 0Valid CNF-formula φ of n variables, a non-negative weight w_i associated to each variable x_i, with $i = 1, \ldots, n$.
Goal: a minimum value truth assignment of φ, where the value of a truth assignment T is defined as $w(T) := \Sigma^n_{i=1} w_i T(x_i)$.

The following observations hold for problem WEIGHTED-0VALID-SAT.

Observation 4.11. WEIGHTED-0VALID-SAT *is polynomial time solvable.*

Proof. Since the boolean formula is 0Valid and the weights are nonnegative, 0^n is a minimum value solution for every instance of problem WEIGHTED-0VALID-SAT. Hence, the minimum value is always 0. □

Observation 4.12. *If all variable weights are strictly positive, then 0^n is the only minimum value truth assignment for problem* WEIGHTED-0VALID-SAT.

Proof. Let T be a minimum value satisfying truth assignment of a 0Valid formula φ. Hence, $\Sigma_{i=1}^{n} w_i T(x_i) = 0$. Since all weights are strictly positive and $T(x_i) \geq 0$, we have that $T(x_i) = 0$ for every $i = 1, \ldots, n$. That is, $T = 0^n$. □

Note that if all variable weights are strictly positive we can apply Algorithm 4.3 in order to list all minimum value solutions, that is, just solution $T = 0^n$. Therefore, $\mathcal{L}$WEIGHTED-0VALID-SAT, the problem of listing all minimum value truth assignments of a 0Valid formula, is strongly **P**-enumerable when all variable weights are strictly positive. But, if the variable weights are nonnegative, then $\mathcal{L}$WEIGHTED-0VALID-SAT is $\mathcal{L}$**P**-hard.

Theorem 4.13. $\mathcal{L}$WEIGHTED-0VALID-SAT *is* $\mathcal{L}$**P***-hard.*

Proof. We consider an instance of problem $\mathcal{L}$0VALID-SAT, that is a 0Valid boolean formula φ of variables $x_1, x_2, \ldots, x_n$. We build an instance of problem $\mathcal{L}$WEIGHTED-0VALID-SAT as follows. We define $\varphi' := \varphi$ and $w(x_i) := 0$ for each variable x_i. It is easy to check that $\langle \varphi', w \rangle$ is an instance of

$\mathcal{L}$WEIGHTED-0VALID-SAT. Furthermore, every truth assignment satisfying φ is an optimal truth assignment that satisfies φ', and viceversa. This implies that if there exists a polynomial total time algorithm solving $\mathcal{L}$WEIGHTED-0VALID-SAT, then there exists a polynomial total time algorithm solving $\mathcal{L}$0VALID-SAT. But, by Corollary 3.12 in Chapter 3, we have that $\mathcal{L}$0VALID-SAT is $\mathcal{LP}$-complete. Hence, $\mathcal{L}$WEIGHTED-0VALID-SAT is $\mathcal{LP}$-hard. □

Hence, we have proved that when variable weights are non-negative a polynomial total time algorithm for listing all minimum value satisfying truth assignments of $\mathcal{L}$WEIGHTED-1VALID-SAT does not exist unless a polynomial total time algorithm exists for all listing problems in $\mathcal{LP}$.

4.4 Comparison with other state-of-the-art listing algorithms

The listing method that we propose can be applied to a broad class of combinatorial ensembles. As such, our listing method implies that for many well-known combinatorial ensembles, the problem of listing all (optimal) feasible solutions is polynomial space polynomial delay solvable. Up to now, for many separable combinatorial ensembles the same result has be achieved working on the single combinatorial ensemble and developing "ad hoc" algorithms for it (see [28] for a catalog of listing algorithms and results), while with our method we obtain the same result "a priori" (i.e. without developing an "ad hoc" algorithm) and from a very general framework (even if, to be fair we have to say that in some cases, the "ad hoc" algorithms proposed in the literature achieve a linear or constant delay rather than the polynomial delay we can achieve with our general algorithm: see [28] for some examples).

Furthermore, our method implies that the listing problem associated to several dozens of combinatorial ensembles is polynomial space polynomial delay solvable, whereas this result was unknown before this thesis: think for example to the problem of listing all T-joins of a graph and the problem of listing all simple b-matchings or b-factors of a graph (we address the reader to [33, 49] to find many more separable combinatorial ensembles on which our listing algorithm can be applied).

In [6], Bussieck and Lübbecke proved that, given a linear description of a 0/1-polytope P, the vertex set of P is strongly **P**-enumerable; actually, the algorithm proposed runs in polynomial space polynomial delay. As a consequence, they obtain that if a combinatorial ensemble admits a compact description, then the problem of listing all feasible solutions of the combinatorial ensemble is strongly **P**-enumerable.
It should be clear to the reader that, given a linear description of a 0/1-polytope P, our algorithm can list all vertices of P in polynomial space and polynomial delay. Furthermore, as observed in Section 4.1 at the beginning of this chapter, every combinatorial ensemble which admits a compact description is separable. The contrary is not true. Consider for example the perfect matching ensemble for non-bipartite graphs: a complete description of the perfect matching polytope in the non-bipartite case includes the blossom inequalities, which are an exponential number in the size of the instance of the combinatorial ensemble. Instead, the perfect matching polytope admits a polynomial time separation algorithm (see [45]).

Chapter 5

Listing satisfying truth assignments of XOR and 2SAT formulas

In this chapter of the thesis we investigate the complexity of listing all the satisfying truth assignments of a:

- CNF-formula with XOR-clauses;
- 2SAT formula.

In both cases, we present a polynomial space linear delay listing algorithm, whereas only polynomial space polynomial delay listing algorithms were previously known [10]. We recall to the reader that the size of the input is the number of literal occurrences in the formula.

5.1 Listing satisfying truth assignments of CNF-formulas with XOR-clauses

A *CNF-formula with XOR-clauses* is a CNF boolean formula such that each clause consists of literals connected by XOR operator (*XOR-clause*).

Example 1. An example of a CNF-formula φ with XOR-clauses.

$$\varphi = (x_1 \oplus \bar{x}_2 \oplus x_3) \wedge (x_2 \oplus x_3 \oplus \bar{x}_4) \wedge (\bar{x}_1 \oplus x_4) \wedge (x_1 \oplus \bar{x}_3 \oplus \bar{x}_4)$$

$x_1, \ldots, x_4$ are boolean variables.

The problem of deciding whether a CNF-formula φ with XOR-clauses is satisfiable or not is in **P**: actually, finding a truth assignment for a CNF-formula with XOR-clauses is equivalent to solving a linear equality system over $\mathbf{GF(2)}$, a problem which is polynomially solvable applying Gaussian elimination ([18]). In [11], a polynomial time algorithm for counting the number of truth assignments satisfying a CNF-formula with XOR-clauses is described.

In this section, we consider the problem of listing all the truth assignments satisfying a CNF-formula with XOR-clauses. A polynomial space polynomial delay algorithm for solving this problem is proposed in [10]. In this section, we describe a polynomial space linear delay listing algorithm. Actually, the algorithm that we propose has a delay which is linear in the number of variables only. We also show that we can efficiently compute ranking and unranking functions for the satisfying truth assignments of a given CNF-formula with XOR-clauses.

5.1.1 A polynomial space linear delay listing algorithm for CNF-formulas with XOR-clauses

Our algorithm is based on a well-known characterization that relates CNF-formulas with XOR-clauses to systems of linear equations over field $\mathbf{GF(2)}$.

In particular, given a CNF-formula φ with XOR-clauses having m clauses and n variables, we can define a system of m linear equations and n variables as follows. To each clause c_j of φ corresponds a linear equation over $\mathbf{GF(2)}$, where the right-end side is 1, while the left-end side is obtained by c_j according to the following replacements:

1. XOR operator is replaced by addition operator over $\mathbf{GF(2)}$;
2. each negated literal $\bar{x}_i$ is replaced by $1 + x_i$.

It follows immediately from this construction that each truth assignment satisfying φ corresponds exactly to a solution of the linear system.

Example 2. The linear system corresponding to the CNF-formula φ with XOR-clauses proposed in Example 1.

$$\begin{cases} x_1 + x_2 + x_3 & = & 0 \\ x_2 + x_3 + x_4 & = & 0 \\ x_1 + x_4 & = & 0 \\ x_1 + x_3 + x_4 & = & 1 \end{cases}$$

In order to solve the linear system obtained, we can apply the Gauss-Jordan elimination scheme. We briefly describes Gauss-Jordan method since our listing algorithm relies on the compact representation of the space of solutions produced by this elimination scheme. In order to simplify our description, in what follow we assume that $m \geq n$ (however, the case $m < n$ follows analogously).

Each linear system can be formulated as a matrix equation in the form

$$A \cdot X = B,$$

where $A = (a_{i,j})$ is the $m \times n$ matrix of coefficients of the linear system, that is, $a_{i,j}$ is the coefficient of variable x_j in the i-th equation (since we are considering field $\mathbf{GF(2)}$, we have that $a_{i,j} := 1$ if x_j appear in the i-th equation, while $a_{i,j} := 0$ otherwise), $X = (x_i)$ is the vector of the variables of the linear system and $B = (b_j)$ is the vector of the right-end side elements of the linear system. Note that, in our hypothesis, A is a 0/1-matrix and B is a 0/1-vector.

Example 3. The matrix equation corresponding to the linear system in Example 2.

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix} \cdot X = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

The Gauss-Jordan elimination scheme operates on the so called *augmented matrix*, a $m\times(n+1)$ matrix $\hat{A}$ obtained considering vector B as the additional $(n+1)$-th column of A. The method consists in two phases: in the first phase we manipulate the matrix $\hat{A}$ using the following operations:

- add a multiple of one row to another row;
- interchange rows or columns (except for the last column);

in order to put the matrix into almost tridiagonal form. A matrix A is in *almost tridiagonal form* if:

1. If there are any *zero rows*, that is rows entirely made up of zeros except at most the element in the last column, then they are grouped at the bottom of the matrix;

2. In any two successive non zero rows $i, i{+}1$, we have that $a_{i,i} = a_{i+1,i+1} = 1$ and $a_{i+1,k} = 0$ for $k = 1, \ldots, i$.

Note that we have to follow some precautions if some columns interchanges are involved. In fact, since each column corresponds to a variable of the linear system, if some columns interchanges are performed, then we have to update this correspondence. Note that this is equivalent to relabel the variables for which the correspondening columns are involved in an interchange. However, without loss of generality and in order to avoid confusion, in what follows we assume that at the end of first phase variable x_i corresponds to i-th column.

Example 4. Almost tridiagonal form of the augmented matrix $\hat{A}$ corresponding to the matrix equation in Example 3.

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Note that, a linear system has no solutions if and only if after the first phase there is a zero row of matrix $\hat{A}$ such that the element in the last column is 1. Otherwise, in the second phase, we manipulate the non zero rows of matrix $\hat{A}$ using only row operations (adding a multiple of one row to another row) in order to put the matrix into *almost diagonal form.* That is, if there are k non zero rows in $\hat{A}$, then:

1. the $m - k$ zero rows are grouped at the bottom of the matrix;
2. $\hat{A}_{[1..k][1..k]}$ is the identity matrix of size k.

Example 5. Almost diagonal form of the augmented matrix $\hat{A}$ corresponding to the matrix equation in Example 3.

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Once the augmented matrix $\hat{A}$ is in almost diagonal form, we can show how to list efficiently all the solutions of the linear system associated. If there are n non zero rows in matrix $\hat{A}$ (that is, $k = n$), then there is only one solution of the linear system, and the value assigned to each variable is exactly the value in each entry of the last column of matrix $\hat{A}$, that is, $x_i = \hat{A}_{i,n+1}$. Otherwise, the system as 2^{n-k} solutions, where the value of variables $x_{k+1}, \ldots, x_n$ can be arbitrary chosen (we call $x_{k+1}, \ldots, x_n$ *free variables*), while the value of variables $x_1, \ldots, x_k$ may depend both on the value assigned to $x_{k+1}, \ldots, x_n$ and on the corresponding entry of the last column of matrix $\hat{A}$, which is however fixed. That is, the space of solutions of the linear system admits a compact representation as follow:

$$x_{k+1}, \ldots, x_n \;=\; \text{arbitrary value choice;}$$

$$\begin{aligned} x_1 &= f(x_{k+1}, \ldots, x_n); \\ &\vdots \\ x_k &= f(x_{k+1}, \ldots, x_n). \end{aligned}$$

Example 6. Compact representation of the space of solutions of the linear system in Example 3.

$$\begin{aligned} x_4 &= \text{arbitrary value choice}; \\ x_1 &= x_4; \\ x_2 &= x_4 + 1; \\ x_3 &= 1. \end{aligned}$$

Note that this compact representation can be obtained in polynomial time (the Gauss-Jordan method runs in polynomial time in the size of the linear system [18]). The listing algorithm that we propose takes advantage of this compact representation. The idea is the following. Using a $(n-k)$-bits Gray code [30], we can generate with constant delay (for example, see [4]) the 2^{n-k} assignments of the free variables. Each time a new assignment for the free variables is produced, we compute the value of variables $x_1, \ldots, x_k$. We show that this latter computation can be performed in linear time in the number of variables using the adequate data structure.

Firstly, we permanently fix the value of those variables among $x_1, \ldots, x_k$ that do not depend on the value of variables $x_{k+1}, \ldots, x_n$. In Example 6, we set variable $x_3 = 1$.

Then, for each free variable $x_i = x_{k+1}, \ldots, x_n$ we make a list of all the variables $x_1, \ldots, x_k$ that depends on the value of x_i, that is, $x_i \longleftarrow \{x_j : j = 1, \ldots, k \ \wedge \ x_j = f(x_i)\}$. In Example 6, we have that $x_4 \longleftarrow \{x_1, x_2\}$.

To follow, we generate the first $(n-k)$-bits vector (*codeword*) of the Gray code, we assign the value to the free variables and we compute by substitution the value of the remaining variables. In Example 6, the first codeword of the 1-bit Gray code is 0, so $x_4 = 0$ and we have that $x_1 = 0$ and $x_2 = 1$. Note that, the time required to generate the first solution of the linear system, including the time to obtain the compact representation, is polynomial in the input size.

Then, iteratively, we produce the next codeword of the Gray code for the free variables. In Example 6, the next codeword is 1, so $x_4 = 1$. In general, this takes constant time and constant change, since a codeword differs in just the one bit respect to the previous codeword. Hence, only the value of one of the last $n-k$ variables changes, lets say x_i. Therefore, only the value of the variables in the list of x_i changes in this new solution (apart from the value of variable x_i itself). But, since we are in a $\mathbf{GF(2)}$ field, the new value of each variable in the list of x_i is equal to its value in the previously generate solution plus 1 (if the value in the previous solution was 0, now the value in the new solution is 1, and viceversa). In Example 6, we have $x_1 = 1$ and $x_2 = 0$. Since a list contains at most k variables, we can output the solutions of the linear system with linear delay.

The algorithm that we propose for listing all solutions of a CNF formula with XOR-clauses is schematized in Algorithm 5.1.

Procedure 5.1 XOR-Listing(φ)

Parameters: φ is a CNF boolean formula with XOR-clauses.

```
1: Build the augmented matrix $\hat{A}$ corresponding to the given boolean formula $\varphi$;
2: Apply the two phases Gauss-Jordan methods;
3: **if** there is a zero row with 1 in the last column **then**
4:     the linear system has no solutions;
5:     exit;
6: If some column interchanges have been performed, relabel variables accordingly;
7: Let $k$ be the number of non zero rows;
8: **if** $k = n$ **then**
9:     output the unique truth assignment of the boolean formula (first $n$ elements of
       column $n + 1$ of $\hat{A}$);
10:    exit;
11: **for** each variable $x_j$ with $j = k + 1, \dots, n$ **do**
12:     create a list of elements $x_i$ such that $x_i = f(x_j)$;
13: **for** each variable $x_i$ with $i = 1, \dots, k$ **do**
14:     **if** $x_i \neq f(x_j)$ for every variable $x_j$ with $j = k + 1, \dots, n$ **then**
15:         assign the value of $x_i$ permanently;
16: produce the first codeword of the $n-k$ bits Gray code, where the $i$-th bit corresponds
    to the value of $x_{k+i}$;
17: assign value to variables $x_i$ with $j = 1, \dots, k$;
18: output the first satisfying truth assignment;
19: **for** $2^{n-k} - 1$ times **do**
20:     generate the next codeword of the Gray code;
21:     let $j$ be the index of the bit such that its value has been changed from the previous
        codeword;
22:     flip the value of $x_{k+j}$ and of the variables in the list of $x_{k+j}$;
23:     output the new truth assignment.
```

5.1.2 Ranking and unranking functions for CNF-formula with XOR-clauses

The compact representation of the space of solutions of the linear system associated to CNF-formulas with XOR-clauses has another relevant consequence. Given a set F of n elements, a *ranking function* assigns a unique integer in the range $[0, n-1]$ to each element of F. The *unranking function* is the inverse of the ranking function: given an integer $k \in [0, n-1]$, the value of the function is the element of F of rank k.

We next show that there are very efficient ranking and unranking functions for the satisfying truth assignments of a CNF-formula φ with XOR-clauses.

Clearly, if the linear system has only one solution, then the functions are trivial. Hence, we can assume that the linear system has 2^z solutions, where z is the number of free variables. Without loss of generality, from now on we assume that these variables are the first z ones, that is, $x_1, x_2, \ldots, x_z$. Since each truth assignment of n variables can be seen as a binary number of n bits, we can identify each truth assignment with the integer number corresponding to its first z bits. Hence, the functions required can be easily defined. Given a satisfying truth assignment, the ranking function consists of computing the integer corresponding to the value of the first z variables. Given an integer k in $[0, 2^z - 1]$, the unranking function assigns to the first z variables the value according to the binary encoding of k, and computes the value of the remaining variables by back substitution using the compact representation of the space of solutions of the linear system associated. In Algorithm 5.2 and Algorithm 5.3 we formalize the algorithms for computing the ranking and unranking functions respectively.

Procedure 5.2 RANK($\mathcal{C}, X$)

Parameters: $\mathcal{C}$ is a compact representation of the truth assignment satisfying a CNF formula φ with XOR-clauses, X is a satisfying truth assignment of φ;

```
1: compute the integer $t$ corresponding to the value in $X$ of the free variables (according to $\mathcal{C}$);
2: output $t$.
```

Procedure 5.3 UNRANK($\mathcal{C}, t$)

Parameters: $\mathcal{C}$ is a compact representation of the truth assignment satisfying a CNF formula φ with XOR-clauses, t is the number of the satisfying truth assignment to output;

```
1: assign the value to the free variables according to the binary encoding of the given integer $t$;
2: compute the value of the remaining variables according to the compact representation $\mathcal{C}$;
3: output the variables assignment obtained.
```

Worthy of note is the fact that with our unranking procedure we can easily randomly generate a satisfying truth assignment of a given CNF-formula with XOR-clauses: we just need to randomly choose an integer number in $[0, 2^z - 1]$, where z is the number of free variables, and then apply our unranking procedure to obtain a randomly generated satisfying truth assignment of the formula.

5.2 Listing satisfying truth assignments of 2SAT formulas

A *2SAT formula* is a boolean formula in conjunctive normal form such that each clause contains at most two literals. The problem of deciding whether a 2SAT formula φ is satisfiable or not is in **P** ([7]). Actually, several linear time algorithms have been proposed to either find a truth assignment that satisfies a 2SAT formula or conclude that the 2SAT formula is not satisfiable ([21, 2, 15]).

In [55], Valiant proved that the problem of counting the number of satisfying truth assignments of a 2SAT formula is #**P**-complete even when restricted to the case of monotone 2SAT formulas (2SAT formulas with only positive literals). This implies that if we can count in polynomial time the number of satisfying truth assignments of a 2SAT formula, then we can solve in polynomial time the counting problem associated to any **NP** relation.

We consider the problem of listing all the satisfying truth assignments of a 2SAT formula. A polynomial space polynomial delay algorithm for solving this problem is proposed in [10]. In this section, we present a polynomial space linear delay algorithm that lists all satisfying truth assignments of a

2SAT formula.

5.2.1 A polynomial space linear delay listing algorithm for 2SAT formulas

To simplify exposition, we assume that a truth assignment of a formula is represented as a collection of literals, one for each variable of the formula. Furthermore, with a slight abuse of notation, we assume that the empty assignment is the only satisfying truth assignment of an empty formula.

First of all we define some notions we use in this section. An *empty-clause* in a CNF boolean formula is a clause which contains no literal. If a formula contains an empty-clause, then it is not satisfiable. A *unit-clause* in a CNF boolean formula is a clause which contains only one literal. Given a boolean formula φ, we say that φ is an *exactly* 2SAT boolean formula if each clause in φ contains exactly two literals. Given a CNF boolean formula φ, a CNF boolean formula φ' is a *sub-formula* of φ if each clause in φ' is a clause in φ (note that $\emptyset$ and φ are both sub-formulas of φ). Given a boolean formula φ, we indicate with $Var(\varphi)$ the set of variables which have at least one literal in φ. Given a set of literals T, we indicate with $Var(T)$ the set of variables which have at least one literal in T. Given a boolean formula φ, we denote by $Sol(\varphi)$ the set of all truth assignments which satisfies φ. Given a set V of boolean variables, we denote by $All(V)$ the set of all truth assignments for V: note that, if V is empty, then $All(V) = \{\emptyset\}$. Given two families A, B of sets, we define

$$A \uplus B := \begin{cases} \emptyset, & \text{if either } A \text{ or } B \text{ is empty;} \\ \{a \cup b \ : \ a \in A, b \in B\}, & \text{otherwise.} \end{cases}$$

The following easy to prove observation holds.

Observation 5.1. *Let φ be any 2SAT formula. Let x be one of the two literals of a variable in $Var(\varphi)$. Then,*

$$Sol(\varphi) = Sol(\varphi \wedge x) \,\dot{\cup}\, Sol(\varphi \wedge \overline{x}).$$

Most of the algorithms which return a satisfying truth assignment of 2SAT formulas are based on the *unit propagation* technique. Actually, unit propagation is used in many heuristics returning a satisfying truth assignment of a generic CNF boolean formula, like the Davis Putnam Logemann Loveland algorithm [14, 13]. Unit propagation relies on the fact that if x is the literal in a unit-clause of a 2SAT formula φ, then x must be in any satisfying truth assignment of φ (we say that literal x is *forced* in φ), as stated by the following observation.

Observation 5.2. *Let φ be any 2SAT formula. Let x be the literal in a unit-clause of φ. Let φ' be the 2SAT formula obtained removing from φ all clauses containing literal x and all occurrences of literal $\overline{x}$. Then,*

$$Sol(\varphi) = \{\{x\}\} \uplus Sol(\varphi') \uplus All(Var(\varphi) \setminus (Var(\varphi') \cup Var(\{x\}))).$$

Unit propagation is described Procedure 5.4. Note that if Procedure 5.4 gets in input a 2SAT formula φ, then it returns a 2SAT formula φ' and a set of literals T, such that:

1. either φ' contains an empty clause (hence, φ is not satisfiable), or φ' is an exactly 2SAT sub-formula of φ;
2. T is a set of forced literals for φ.

Procedure 5.4 UNITPROP(φ)

Parameters: φ is a 2SAT formula;

```
1: $T := \emptyset$;
2: while $\varphi$ contains some unit-clause do
3:     let $x$ be the literal in a unit-clause of $\varphi$;
4:     $T := T \cup \{x\}$;
5:     remove from $\varphi$ all clauses containing literal $x$;
6:     remove from $\varphi$ all occurrences of literal $\overline{x}$;
7: return $(\varphi, T)$.
```

Observation 5.2 generalizes as follows.

Observation 5.3. *Let φ be a 2SAT boolean formula. Let φ' and T be respectively the 2SAT formula and the set of literals returned by a call to* UNITPROP(φ)*. Then,*

$$Sol(\varphi) = \{T\} \uplus Sol(\varphi') \uplus All(Var(\varphi) \setminus (Var(\varphi') \cup Var(T))).$$

Observation 5.3 implies that if we can list all satisfying truth assignments of φ', then we can easily list all satisfying truth assignments of φ. We also say that the satisfying truth assignments of φ' are a *compact representation* of the satisfying truth assignments of φ.

Let φ be any 2SAT formula. Given a variable $x \in Var(\varphi)$, we say that φ is x*-free* if φ admits two satisfying truth assignments T, T' such that $x \in T$ and $\overline{x} \in T'$. We say that φ is *free* if φ is x-free for every variable $x \in Var(\varphi)$. Note that, every free 2SAT formula is an exactly 2SAT formula.

Let φ be any 2SAT formula. Then, either φ is not satisfiable, or it is possible to compute in polynomial time a free sub-formula φ' of φ such that the satisfying truth assignments of φ' are a compact representation of the satisfying truth assignments of φ. Consider subroutine PREPROCESSING(φ) described in Procedure 5.5.

Procedure 5.5 PREPROCESSING(φ)

Parameters: φ is a 2SAT boolean formula;

```
1: if $\varphi$ is not satisfiable then
2:     halt;
3: $(\varphi, T)$ :=UNITPROP($\varphi$);
4: $W := Var(\varphi)$;
5: while $W$ is not empty do
6:     extract a variable $x$ from $W$;
7:     if the formula $\varphi \wedge x$ is not satisfiable then
8:         $(\varphi, T')$ :=UNITPROP($\varphi \wedge \overline{x}$);
9:         $T := T \cup T'$;
10:    else if the formula $\varphi \wedge \overline{x}$ is not satisfiable then
11:        $(\varphi, T')$ :=UNITPROP($\varphi \wedge x$);
12:        $T := T \cup T'$;
13: return $(\varphi, T)$.
```

Lemma 5.4. *Let φ be a satisfiable 2SAT formula. Let φ' and T be respectively the 2SAT formula and the set of literals returned by a call to* PREPROCESSING(φ)*. Then, φ' is a free 2SAT sub-formula of φ. Furthermore,*

$$Sol(\varphi) = \{T\} \uplus Sol(\varphi') \uplus All(Var(\varphi) \setminus (Var(\varphi') \cup Var(T))).$$

Moreover, Procedure 5.5 runs in polynomial time in the size of φ.

Note that more efficient procedures than Procedure 5.5 may exist: however, our purpose here is just to show that we can obtain a free 2SAT sub-formula of a satisfiable 2SAT formula in polynomial time in the input size.

From now how we assume to work only on free 2SAT formulas. In fact, if we have a linear delay algorithm for listing all satisfying truth assignments of a free 2SAT formula, then, by Lemma 5.4, we can easily derive a linear delay algorithm for listing all satisfying truth assignments of an arbitrary 2SAT formula.

5.2.2 A polynomial space linear delay listing algorithm for free 2SAT formulas

The keystone of our algorithm is the following results regarding free 2SAT formulas.

Lemma 5.5. *Let φ be a free 2SAT formula. Let x be one of the two literals of a variable in $Var(\varphi)$. Let φ' be the formula returned by a call to* UnitProp$(\varphi \wedge x)$*. Then, φ' is a free 2SAT formula. Furthermore, the satisfying truth assignments of φ' are a compact representation of the satisfying truth assignments of φ that contains literal x.*

Proof. Clearly, φ' is an exactly 2SAT sub-formula of φ. Assume, for a contradiction, that φ' is not a free 2SAT formula. Hence, there exists $y \in Var(\varphi')$ such that φ' is not y-free. That is, either $\varphi' \wedge y$ or $\varphi' \wedge \overline{y}$ is not satisfiable. W.l.o.g., we can assume that $\varphi' \wedge y$ is not satisfiable. But, since φ' is a sub-formula of φ, this implies that $\varphi \wedge y$ is not satisfiable. Absurd, since φ is free.
The fact that the satisfying truth assignments of φ' are a compact representation of the satisfying truth assignments of φ that contains literal x follows from Observation 5.3. □

Note that Lemma 5.5 and Observation 5.1 implicitly define a recursive procedure to list all satisfying truth assignments of a free 2SAT formula: choose a variable of the formula, branch on its two possible truth assignments, unit propagate the variable's truth assignment in the formula, and call recursively on the reduce formula. The algorithm ListFree2SAT(φ) that we propose is described in Algorithm 5.6. The algorithm relies on some procedures defined below. All parameters in these procedures are passed by reference.

Algorithm 5.6 ListFree2SAT(φ)

Parameters: φ is a free 2SAT boolean formula;

```
1: Let $T$ be an empty set of literals (kept as a stack);
2: Let $C$ be an empty set of clauses (kept as a stack);
3: Branch($\varphi, T, C$).
```

Procedure 5.7 Branch(φ, T, C)

Parameters: φ is a free 2SAT boolean formula, T is a set of literals, C is a set of clauses;

```
1: if $\varphi$ is not empty then
2:     let $x$ be a variable in $Var(\varphi)$;
3:     LiterProp($\varphi, x, T, C$);
4:     Branch($\varphi, T, C$);
5:     UndoLiterProp($\varphi, x, T, C$);
6:     LiterProp($\varphi, \overline{x}, T, C$);
7:     Branch($\varphi, T, C$);
8:     UndoLiterProp($\varphi, \overline{x}, T, C$);
9: else
10:    Print all members of set $\{T\} \uplus All(V)$, where $V$ is the set of variables of the input
       formula which do not have a literal in $T$.
```

Procedure 5.8 LITERPROP(φ, x, T, C)

Parameters: φ is a free 2SAT boolean formula, x is a literal, T is a set of literals, C is a set of clauses;

```
1: create an empty stack $W$;
2: push literal $x$ into $W$;
3: push in $C$ a fake clause (a sentinel) containing $x$;
4: while $W$ is not empty do
5:    pop a literal $y$ from the top of $W$;
6:    push $y$ into $T$;
7:    for each clause $c$ containing $y$ do
8:       push $c$ into $C$;
9:       delete $c$ from $\varphi$;
10:   for each clause $c$ containing $\overline{y}$ do
11:      let $z$ be the other literal in $c$;
12:      push literal $z$ into $W$ (if $z$ not already in $W$).
```

Procedure 5.9 UNDOLITERPROP(φ, x, T, C)

Parameters: φ is a free 2SAT boolean formula, x is a literal, T is a set of literals, C is a set of clauses;

```
1: pop a clause $c$ from $C$;
2: while $c$ is not the fake clause containing $x$ do
3:    re-insert $c$ in formula $\varphi$;
4:    pop a clause $c$ from $C$;
5: repeat
6:    pop a literal $y$ from $T$;
7: until $y = x$.
```

The reader can easily check that Procedure LITERPROP acts on a free 2SAT boolean formula φ and a literal x (where x is the literal of a variable in $Var(\varphi)$) exactly as a call to Procedure UNITPROP($\varphi \wedge x$). Furthermore, since we work on a single copy of the formula (all parameters are passed by reference), in order to execute correctly the branch on the two possible truth assignments of a chosen variable we need to restore the formula as it was before the unit propagation on the first truth value: Procedure UNDOLITERPROP is somehow the "inverse" of LITERPROP, in the sense that it restores the formula as before a call to LITERPROP. Actually, the following observations hold.

Observation 5.6. *Let φ be a free 2SAT formula, T be a set of literals, C be a set of clauses and x be the literal of a variable in $Var(\varphi)$. Let φ', T', C' be φ, T, C after a call to* LITERPROP(φ, x, T, C). *Let φ'', T'', C'' be φ', T', C' after a call to* UNDOLITERPROP(φ', x, T', C'). *Then, $\varphi = \varphi''$, $T = T''$ and $C = C''$.*

Observation 5.7. *Let φ be a free 2SAT formula, T be a set of literals and C be a set of clauses. Let φ', T', C' be φ, T, C after a call to* BRANCH(φ, T, C). *Then, $\varphi = \varphi'$, $T = T'$ and $C = C'$.*

Since Algorithm 5.6 is nothing more than a formal description of the procedure implied by Lemma 5.5 and Observation 5.1 for listing all satisfying truth assignments of a free 2SAT formula, the following theorem holds.

Theorem 5.8. *Algorithm 5.6 returns all satisfying truth assignments of any free 2SAT boolean formula.*

Next, we show that storing the formula with an adequate data structure, the delay between to consecutive satisfying truth assignments outputted by Algorithm 5.6 is linear in the size of the formula. First of all, note that every time the formula gets empty (lines 9-10 of Procedure 5.7), it is easy to list with linear delay all satisfying truth assignments T^* of the input formula with $T^* \supseteq T$, using for example a Gray code to generate all possible assignments for the variables of the input formula which don't have a literal in T.

The data structure that we need to store the formula, must obey the following requirements:

1. the cost of retrieving/deleting all clauses containing a literal x is $\mathcal{O}(K)$, where K is the number of clauses containing literal x;
2. the cost of re-inserting a clause c in φ is $\mathcal{O}(1)$.

To convince the reader that data structures satisfying the above requirements do exist, we describe one of them in Appendix B.

Note that, if the above requirements are satisfied, it is easy to check that if K is the number of clauses removed form φ in a call to Procedure LITERPROP, then Procedure LITERPROP runs in $\mathcal{O}(K)$. Analogously, if K is the number of clauses re-inserted in φ in a call to Procedure UNDOLITERPROP, then Procedure UNDOLITERPROP runs in $\mathcal{O}(K)$. Clearly, this implies that in at most linear time in the size of the input formula, Algorithm 5.6 outputs the first satisfying truth assignment. Furthermore, after Algorithm 5.6 outputs a satisfying truth assignment, in at most linear time in the size of the input formula the algorithm either outputs another truth assignment, or it terminates, because all satisfying truth assignments have been outputted.

Since the space used by Algorithm 5.6 is for sure polynomial in the input size, we can conclude that Algorithm 5.6 is a polynomial space linear delay algorithm.

Chapter 6

Conclusions

In this thesis, we have provided some new contributions regarding the general issue of listing solutions. These contributions can be summarized as follows.

Firstly, we introduced a computational complexity theory for the class of listing problems associated with **NP** relations. To the best of our knowledge, no similar contribution has been proposed before this thesis. We have shown that under the weak efficiency notion of polynomial total time, several listing problems turn out to be complete for the class considered. Although most of them are related to **NP**-complete decision problems, we have provided some examples of hard to solve listing problems whose decision version is easy to solve. These results strengthened our belief that listing problems deserve the development of their own computational complexity theory.

Secondly, we have described a very general and efficient listing method for combinatorial problems. Improving a result proposed in [6], we have shown that whenever we have a good knowledge of the polyhedral description of the polytope defined by the feasible solutions of a combinatorial problem, then all feasible solutions are listable in polynomial space and polynomial delay. Furthermore, we have proved that if the feasible solutions are additionally evaluated by a linear objective function, then we can list in polynomial space and polynomial delay all optimal value solutions. Indeed, we have proved as a general and fundamental fact, that the polynomial space polynomial delay listability of the (optimal) feasible solutions of a combinatorial problem follows from the good knowledge of the polyhedral description of the underlying combinatorial problem. One of the consequences is also that all minimum/maximum cardinality feasible solutions are efficiently listable. The value of these results is that from a single framework we can derive many positive results, whereas up to now many of these results had been worked out one by one. We have also shown that a good knowledge of the polyhedral description of the dominant of the polytope defined by the feasible solutions of a combinatorial problem, does not necessarily imply that we can efficiently list all (optimal) feasible solutions of the combinatorial problem: in fact, we have provide an example of dominant separable combinatorial problem where the existence of a polynomial total time algorithm for listing all (optimal) feasible solutions would imply a polynomial total time algorithm for the listing problem associated with every **NP** relations.

Thirdly, we have proposed new polynomial space linear delay algorithms for listing all satisfying truth assignments of two particular classes of boolean formulas: 2SAT formulas and CNF formulas with XOR-clauses. Previously, only polynomial space polynomial delay algorithms were known for these classes of boolean formulas [10]. Differently from the previously proposed approaches, our algorithms achieve this improved delay time bound by exploiting the structure of the space of solutions in the two considered situations.

We conclude recalling to the reader a relevant open problem related to the topics considered in this thesis and addressed in many works. In [34], Johnson, Yannakakis and Papadimitriou have proposed a polynomial delay algorithm for listing all maximal independent sets of a graph. Up to now, it is unknown whether a polynomial total time algorithm exists to list all maximal independent sets in *hypergraphs* (an hypergraph is a couple (V, H), where V is a set of vertices and H is a collection of subsets of V, called *hyperedges*; clearly, a graph is special case of hypergraph where every hyperedge contains precisely two elements). This problem, also known as the *hypergraph transversal problem* (the complement of a maximal independent set in an hypergraph is called a minimal transversal), has been investigated by several prominent researchers in the last three decades [42, 34, 20, 23]. In 1996, a breakthrough result by Fredman and Khachiyan [23] has established the existence of a *quasi-polynomial time* algorithm for the *hypergraph dualization* problem: given an hypergraph $\mathcal{H}$ and a set $\mathcal{X}$ of minimal transversal, either prove that $\mathcal{X}$ is the set of all minimal transversals of $\mathcal{H}$, or find a new minimal transversal $T \notin \mathcal{X}$ for $\mathcal{H}$. The algorithm proposed in [23] has time complexity $n^k + m^{o(\log m)}$, where k is a natural number, n is the number of vertices of the hypergraph, and $m = |\mathcal{H}| + |\mathcal{X}|$. As observed in [23], this result implies that the hypergraph dualization problem is not **NP**-hard, unless any **NP**-complete problem can be solved in quasi-polynomial time. Note that the existence of a polynomial time algorithm for the hypergraph dualization problem would imply the existence of a polynomial total time algorithm for the hypergraph transversal problem.

Bibliography

[1] S. Arora. Reductions, codes, PCPs, and inapproximability. In *IEEE Symposium on Foundations of Computer Science*, pages 404–413, 1995.

[2] B. Aspvall, M.F. Plass, and R.E. Tarjan. A linear time algorithm for testing the truth of certain quantified Boolean formulas. *Information Processing Letters*, 8:121–123, 1979.

[3] M. Bellare and S. Goldwasser. The complexity of decision versus search. *SIAM Journal on Computing*, 23(1):97–119, 1994.

[4] J.R. Bitner, G. Ehrlich, and E.M. Reingold. Efficient generation of the binary reflected gray code and its applications. *Communication of the ACM*, 19(9):517–521, 1976.

[5] D.P. Bovet and P. Crescenzi. *Introduction to the theory of complexity.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994.

[6] M.R. Bussieck and M.E. Lübbecke. The vertex set of a 0/1-polytope is strongly $\mathcal{P}$-enumerable. *Comput. Geom. Theory Appl.*, 11(2):103–109, 1998.

[7] S.A. Cook. The complexity of theorem proving procedures. *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[8] W.J. Cook, W.H. Cunningham, W.R. Pulleyblank, and A. Schrijver. *Combinatorial Optimization.* John Wiley & Sons, Inc., New York, NY, USA, 1998.

[9] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, Cambridge, Massachusetts, 2001.

[10] N. Creignou and J.J. Hébrard. On generating all solutions of generalized satisfiability problems. *Informatique Théorique et Applications*, 31(6):499–511, 1997.

[11] N. Creignou and M. Hermann. Complexity of generalized satisfiability counting problems. *Inf. Comput.*, 125(1):1–12, 1996.

[12] G.B. Dantzig. Maximization of a linear function of variables subject to linear inequalities. In T.C. Koopmans, editor, *Activity Analysis of Production and Allocation - Proceedings of a Conference*, volume 13 of *Cowles Commission Monograph*, pages 339–347. Wiley, New York, 1951.

[13] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communication of the ACM*, 5(7):394–397, 1962.

[14] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[15] A. del Val. On 2-sat and renamable horn. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 279–284. AAAI Press / The MIT Press, 2000.

[16] E.W. Dijkstra. A note on two problems in connexion with graphs. In *Numerische Mathematik*, volume 1, pages 269–271. Mathematisch Centrum, Amsterdam, The Netherlands, 1959.

[17] J. Edmonds. Maximum matching and a polyhedron with 0,1-vertices. *Journal of Research of the National Bureau of Standards (B)*, 69:125–130, 1965.

[18] J. Edmonds. Systems of distinct representatives and linear algebra. *Journal of Research of the National Bureau of Standards*, 71(B):241–245, 1967.

[19] J. Edmonds and R.M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.

[20] T. Eiter and G. Gottlob. Identifying the minimal transversals of a hypergraph and related problems. *SIAM Journal on Computing*, 24(6):1278–1304, 1995.

[21] S. Even, A. Itai, and A. Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM Journal on Computing*, 5(4):691–703, December 1976.

[22] L.R. Ford Jr and D.R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.

[23] M.L. Fredman and L. Khachiyan. On the complexity of dualization of monotone disjunctive normal forms. *Journal of Algorithms*, 21(3):618–628, 1996.

[24] K. Fukuda. Note on new complexity classes **ENP**, **EP** and **CEP**. Manuscript electronically available at URL http://www.ifor.math.ethz.ch/ifor/staff/fukuda/ENPhome/ENPnote.html, June 1996.

[25] K. Fukuda, T.M. Liebling, and F. Margot. Analysis of backtrack algorithms for listing all vertices and all faces of a convex polyhedron. *Computational Geometry*, 8:1–12, 1997.

[26] K. Fukuda and T. Matsui. Finding all minimum-cost perfect matchings in bipartite graphs. *Networks*, 22(5):461–468, 1992.

[27] K. Fukuda and T. Matsui. Finding all the perfect matchings in bipartite graphs. *AMLETS: Applied Mathematics Letters*, 7(1):15–18, 1994.

[28] K. Fukuda, T. Matsui, and Y. Matsui. A catalog of enumeration algorithms. Project (in progress), ROSO, Department of Mathematics, EPFL, 1996. Available on-line at http://dmawww.epfl.ch/roso.mosaic/kf/enum/enum.html.

[29] M.R. Garey and D.S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., San Francisco, CA, USA, 1979.

[30] E. Gilbert. Gray codes and paths on the n-cube. *Bell System Tech. J.*, 37:815–826, 1958.

[31] L.A. Goldberg. *Efficient Algorithms for Listing Combinatorial Structures.* Cambridge University Press, Cambridge, 1993.

[32] O. Goldreich. *Introduction to Complexity Theory.* Lecture Notes Series of the Electronic Colloquium on Computational Complexity, 1999. Downloadable at http://www.eccc.uni-trier.de/eccc-local/ECCC-LectureNotes/.

[33] M. Grötschel, L. Lovasz, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization (Algorithms and Combinatorics).* Springer, 1994.

[34] D.S. Johnson, M. Yannakakis, and C.H. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27(3):119–123, 1988.

[35] L.R. Ford Jr. and D.R. Fulkerson. A simple algorithm for finding maximal network flows and an application to the Hitchcock problem. *Canadian Journal of Mathematics*, 9:210–218, 1957.

[36] S. Kannan and T. Warnow. A fast algorithm for the computation and enumeration of perfect phylogenies. *SIAM Journal on Computing*, 26(6):1749–1763, 1997.

[37] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–396, 1984.

[38] R.M. Karp. Reducibility among combinatorial problems. In R.E. Miller and J.W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 1972. Plenum Press.

[39] L.G. Khachiyan. A polynomial algorithm in linear programming. *Doklady Akedamii Nauk SSSR*, 20:191–194, 1979.

[40] J.B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.

[41] S. Lang. *Linear algebra.* Addison-Wesley Series in Mathematics. Addison-Wesley, 1965.

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

[43] L.A. Levin. Universal sorting problems. *Problemy Peredachi Informatsii*, 9(3):265–266, 1973. In Russian.

[44] T.S. Motzkin. *Beiträge zur Theorie der linearen Ungleichungen.* PhD thesis, University Basel, 1936.

[45] M.W. Padberg and M.R. Rao. Odd minimum cut-sets and b-matchings. *Mathematics of Operations Research*, 7:67–80, 1982.

[46] C.M. Papadimitriou. *Computational Complexity.* Addison-Wesley Publishing Company, 1994.

[47] R.C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[48] A. Schrijver. *Theory of linear and integer programming.* John Wiley & Sons, Inc., New York, NY, USA, 1986.

[49] A. Schrijver. *Combinatorial Optimization - Polyhedra and Efficiency*, volume 24 of *Algorithms and Combinatorics.* Springer-Verlag, Berlin, 2003.

[50] J. Simon. *On some central problems in computational complexity.* PhD thesis, Cornell University, Ithaca, N.Y, 1975. Available as Cornell Department of Computer Science Technical Report TR75-224.

[51] M. Sipser. *Introduction to the Theory of Computation.* Second Edition. Course Technology, 2005.

[52] U. Takeaki. A fast algorithm for enumerating bipartite perfect matchings. In *International Symposium on Algorithm and Computation*, pages 349–359, 2001.

[53] R.E. Tarjan. Enumeration of the elementary circuits of a directed graph. *SIAM Journal on Computing*, 2(3):211–216, 1973.

[54] L.G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(3):189–201, 1979.

[55] L.G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[56] F. Zane. *Circuits, CNFs, and Satisfiability.* PhD thesis, Dept. of Computer Science and Engineering, University of California, San Diego, 1998.

Appendix A

$\mathcal{P}_{\mathcal{C}}$ and $\mathcal{P}^{\uparrow}_{\mathcal{C}}$ are proper classes of polyhedra

Proof of Observation 4.2. We have to show that for each instance $\langle S, \mathcal{F} \rangle$ of $\mathcal{C}$, we can compute in polynomial time in the instance size two natural numbers n_t and s_t such that, $P_{\langle S,\mathcal{F}\rangle} \subseteq \mathbb{R}^{n_t}$ and $P_{\langle S,\mathcal{F}\rangle}$ has facet complexity at most s_t. Clearly, $n_t := |S|$. Furthermore, since $P_{\langle S,\mathcal{F}\rangle}$ is the convex hull of some 0/1-vectors, the vertex complexity of $P_{\langle S,\mathcal{F}\rangle}$ is at most the size of vector 1^S, that is $\left|1^S\right|$. By Theorem 2.2, $s_t := 4n^2 \left|1^S\right|$ is an upper bound of the facet complexity of $P_{\langle S,\mathcal{F}\rangle}$. Both values can be computed in polynomial-time in the instance size. □

Proof of Observation 4.5. By definition, $P^{\uparrow}_{\langle S,\mathcal{F}\rangle} = P_{\langle S,\mathcal{F}\rangle} + \mathbb{R}^S_{\geq 0}$. Since $P_{\langle S,\mathcal{F}\rangle}$ is the convex hull of some 0/1-vectors, and $\mathbb{R}^S_{\geq 0} = cone(\{e_1, \ldots, e_S\})$, where e_i is the vector with all entries equal to 0 except the i-th entry which is equal to 1, the vertex complexity of $P^{\uparrow}_{\langle S,\mathcal{F}\rangle}$ is at most $\left|1^S\right|$. By Theorem 2.2, $s_t := 4n^2 \left|1^S\right|$ is an upper bound of the facet complexity of $P^{\uparrow}_{\langle S,\mathcal{F}\rangle}$. Both values can be computed in polynomial-time in the input size $|\langle S, \mathcal{F} \rangle|$. □

Appendix B

A data structure for listing all satisfying truth assignments of 2SAT formulas

In this appendix, we present a data structure to store free 2SAT formulas which satisfies the requirements, in terms of efficient operations' computation, required by Algorithm 5.6.

Let φ be a free 2SAT formula made of m clauses and containing n variables. In the data structure that we propose, we keep a list of the clauses in the formula, and, for each of the two literals of the variables in the formula, we keep a list of the clauses containing that literal. In detail, the data structure that we propose is made of the following components (see Figure B.1 for an example):

1. a doubly linked list $formula$, initially made of m elements; $formula$ represents φ, where each element in the list represents a clause c. Each element of $formula$ contains two couples $\langle l, p \rangle$, one for each literal l in c, where p is a pointer to the element corresponding to clause c in the list associated to literal l;

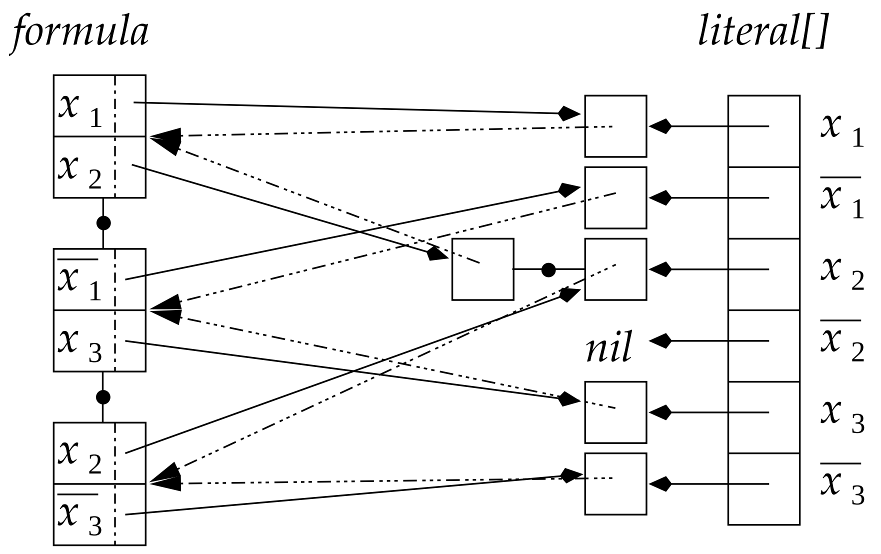

Figure B.1: Given the free 2SAT formula $\varphi = (x_1 \vee x_2) \wedge (\overline{x_1} \vee x_3) \wedge (x_2 \vee \overline{x_3})$, we can represent it with the above data structure.

2. a $2n$ entries array $literal[]$, with one entry for each of the two literals of every variable in φ; each entry $literal[y]$ is a pointer to a doubly linked list, where each element corresponds to a clause c of the formula. Actually, each element of this list is a pointer to the corresponding clause in $formula$ (note that this list may be empty if literal y is not contained in any clause of φ).

It is clear that we can retrieve all clauses containing literal x in $\mathcal{O}(K)$, where K is the number of clauses containing literal x, since we just need to scan the list pointed by $literal[x]$. Furthermore, the deletion from the formula of all the clauses containing literal x takes $\mathcal{O}(K)$ time, since for any clause c containing x, we just need to remove the element corresponding to c both from $formula$ and from the list of both literals which appear in clause c.

Suppose that we want to re-insert into φ a clause c, containing literals x and y. Then, we just need to insert a new element in $formula$ and in the list of both literal x and literal y (clearly, since the elements in the lists used in this data structure are kept in no particular order, an element is always add to the top of a list), and to fix coherently the pointers. Clearly, the re-insertion of a clause can be performed in $\mathcal{O}(1)$.

Index