**SAPIENZA**
UNIVERSITÀ DI ROMA

# Efficient Parity Decision Trees and Their Connections to Logical Proofs and Total Search Problems in NP

Faculty of Information Engineering, Computer Science and Statistics
Bachelor's Degree in Computer Science

**Simone Bianco**
ID number 1986936

Advisor                                        Co-Advisor
Prof. Nicola Galesi                            Prof. Massimo Lauria

Academic Year 2023/2024

**Efficient Parity Decision Trees and Their Connections to Logical Proofs and Total Search Problems in NP**

Bachelor's Thesis. Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: bianco.simone@outlook.it

*«Either mathematics is too big for the human mind,*
*or the human mind is more than a machine»*
*— Kurt Gödel*

# Abstract

In computability theory, a *search problem* is a type of computational problem based on finding a specific property, object or structure in a given instance of a particular entity. Search problems describe any input-output-based problem, even everyday problems, ranging from number factorization to complex graph theory questions. For a given instance, some problems may even take the age of the universe to be solved by a machine. Complexity theory studies computational resources to identify what can and cannot be computed in a reasonable amount of time.

In recent years, the interest in *total search problem*, i.e. search problems that have at least one solution for all possible instances of the problem, has grown. Extensive study of these problems has shown that it is sufficient to restrict our interest to a small set of problems, each corresponding to a basic combinatorial principle, defining what is now referred to as the TFNP hierarchy. This hierarchy has been shown to be deeply linked with other complexity theory branches, such as logical proofs, circuits and protocols. These connections inspired researchers to extend the known results in hope of achieving an *universal theory*.

The thesis has two main goals: to summarize the main results in the TFNP world and to show new ones obtained through the introduction of *Parity Decision Trees*, an extension of the decision tree computational model based on linear forms in $\mathbb{F}_2$.

In the first chapter, we discuss the origins of computability theory, the basic concepts of computational complexity and the reasons behind the study of such concepts.

In the second chapter, we discuss the differences between decision problems and search problems, while also defining the decision classes P, NP and the functional classes FP, FNP and TFNP. Then, we explain reductions between problems, how they lead to the concept of completeness and how they apply to total search problems, giving birth to the TFNP hierarchy.

In the third chapter, we focus on the black-box version of the TFNP hierarchy. We discuss how this model characterizes relativization through oracles and how this translates to the use of decision trees. Then, we explain how the connections between decision trees and proof complexity give an alternative lens under which this model can be studied.

In the final chapter, we introduce the concept of parity in the black-box model through the use of parity decision trees. We show how parity defines a computational model stronger than the traditional one, introducing a new class $\mathsf{FP}^{pdt}$, i.e. the class of TFNP problems efficiently solvable by a PDT. Then, we show that this class is characterized by the Tree-like Linear Resolution over $\mathbb{F}_2$ proof system, proving that the complexity of PDTs is equivalent to the complexity of proofs in this system. Finally, we show that short proofs in $\mathsf{TreeRes}_\oplus$ can be converted into short proofs in the Nullstellensatz proof system, which characterizes all problems reducible to the Polynomial Parity Argument (PPA) principle. These results define new relations between our new class $\mathsf{FP}^{pdt}$ and the already known classes, in particular the inclusions $\mathsf{FP}^{dt} \subsetneq \mathsf{FP}^{pdt} \subseteq \mathsf{PPA}^{dt}$ and the non-inclusion $\mathsf{PLS}^{dt} \not\subseteq \mathsf{FP}^{dt}$.

# Contents

# Chapter 1

# Introduction

## 1.1 Computation and Turing machines

Throughout history, humans have been solving problems through a wide variety of models capable of computing valid results, ranging from their intellect to mechanical devices capable of solving problems. In particular, a computation made by a model can be described as a list of sequential operations and initial conditions that will always yield the same result each time the computation is executed.

In modern mathematics, this idea is formalized through the concept of **algorithm**, a finite list of unambiguous instructions that, given some set of initial conditions, can be performed to compute the answer to a given problem. Even though this is a straightforward definition, it isn't as "mathematically stable" as it seems: each computational model could have access to a different set of possible operations, meaning that the same problem could be solved by different computational models in various ways. This innate nature of computational models makes life difficult for mathematicians, who want to prove results that are as general as possible.

In 1933, Kurt Gödel - one of the greatest logicians of all time - tried to capture the concept of computation through logic, formalizing the definition of the class of **general recursive functions**, i.e. the class functions $f : \mathbb{N} \to \mathbb{N}$ that are "computable" in an intuitive sense. Formally, this corresponds to the smallest class of functions $f : \mathbb{N} \to \mathbb{N}$ that is closed under composition, recursion and minimization while also including the value zero, the successor operator and all the projection operators. *Gödel's thesis* states that every mechanically calculable function can be in some way defined using general recursive functions [Dav82]. However, Gödel's definition of computation wasn't good enough due to general recursive functions being a hard tool to work with.

In 1936, inspired by Gödel's ideas, Alonzo Church defined **lambda calculus**, a formal system in mathematical logic that expresses computation through abstraction, function application, variable binding and substitution. Church [Chu36] and his student Stephen Kleene [Kle36] were able to prove that a function is lambda-computable if and only if it is general recursive, showing that the two models were equivalently able to capture the concept of computation. Despite lambda calculus being simpler to work with compared to the axioms of general recursive functions,

this model was still considered too much abstract compared to "real" computations that were being executed by the very distant ancestors of modern computers. In 1936, Alan Turing - another of Church's distinguished students - defined the now-called **Turing machine**, an abstract machine capable of capturing the concept of computation itself through simple - but sufficient - operations.

Informally, a Turing machine is made of:

- A *tape* divided into cells, where each cell contains a symbol from a finite set called *alphabet*, usually assumed to contain only 0 and 1, or a special symbol ␣, namely the *blank character*. The tape is finite on the left side but infinite on the right side.

- A *read-write head* capable of reading and writing symbols on the tape. The head is always positioned on a single cell of the tape and can shift left and right only one cell per shift.

- A finite set of *states* that can be assumed by the machine. At all times the machine only knows its current state. The set contains at least one state that is capable of immediately halting the machine when reached (such states could be unreachable, making the machine go in an infinite loop).

- A finite set of *instructions* which, given the current state and the current cell read by the read-write head, dictate how the machine behaves. Each instruction tells the machine to do three things: replace the symbol of the current cell (which can be replaced with itself), move the head one cell to the left or one cell to the right and move from the current state to a new one (which can be the current state itself).

Initially, the machine's tape contains only an *input string*, while all the other infinite cells contain the blank character. At the end of the computation, the tape contains only the *output string*, which is the result of the computation.
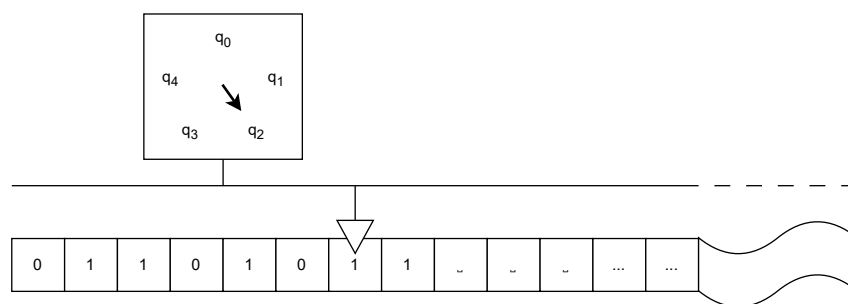


**Figure 1.** A Turing machine.

**Definition 1.** A Turing machine is a 7-uple $M = (Q, F, \Gamma, \Sigma, q_0, \delta)$ where:

- $Q$ is a finite set of states, $F \subseteq Q$ is a finite set of halting states and $q_0 \in Q$ is the initial state taken by the machine.

- $\Gamma$ is a finite set of symbols, usually called the tape alphabet. The tape alphabet always contains the symbol $\sqcup$, i.e. the black tape cell symbol.

- $\Sigma$ is a finite set of symbols, usually called the input alphabet, where $\Sigma \subseteq \Gamma - \{\sqcup\}$. The input string can be formed only of these characters.

- $\delta : (Q - F) \times \Gamma \to Q \times \Gamma \times \{\mathsf{L}, \mathsf{R}\}$ is a partial function, usually called the transition function, where $\mathsf{L}$ and $\mathsf{R}$ represents a left or right shift of the read-write head. Intuitively, if $\delta(q, a) = (p, b, L)$ then, when the machine is in state $q$ and reads the symbol $a$ on the current cell of the tape, it transitions to the state $p$, replaces the symbol $a$ with $b$ and moves the head to the left.

Turing [Tur37] showed that a function can be computed by his theoretical machine if and only if it is lambda-computable, proving that the two models are equivalent. Turing's characterization of computation was acclaimed by Gödel himself, who deemed it «*an absolute definition of an interesting epistemological notion*». The computational characterizations devised by Church and Turing would later be referred to as the **Church-Turing thesis**, constituting a formal definition of algorithm and computation.

Moreover, Turing proved the existence of an *Universal Turing machine*, a $\mathsf{TM}$ that is capable of simulating any other Turing machine. In other words, an $\mathsf{UTM}$ is capable of computing any function computable by another Turing machine. This result shouldn't be a surprise: modern computers are nothing more than a $\mathsf{UTMs}$ that can execute any given algorithm, producing an output for a given input.

The concept of Universal Turing machine also allows us to easily prove that many other computational models are capable of characterizing computation: if a model is capable of simulating an $\mathsf{UTM}$ then it is capable of making any possible computation. This idea is known as *Turing completeness.*

After achieving a mathematically stable definition of computation through Turing machines, Church and Turing's focus shifted to understanding which problems are computable. In particular, they showed that some functions are **uncomputable** by proving that there cannot exist a Turing machine capable of carrying out their computation without going in infinite loops, i.e. never halting and thus never finding a solution. The main example of such a problem is Turing's famous *Halting problem* which asks «*does the given machine halt or not for a given input?*». Turing's proof is based on Cantor's *diagonal argument*, a technique first used to show the non-numerability of the real numbers.

The existence of uncomputable functions - and thus uncomputable problems - gives a negative answer to the *Entscheidungsproblem* (german for *decision problem*), a question posed by David Hilbert in 1928 which asks if there is an algorithm that for each input statement answers "yes" or "no" according to whether or not the statement is universally true.

## 1.2   Complexity measures

Once the existence of the absolute limits of computation was discovered, researchers shifter their focus on the "practical" limits of computation. A "good" algorithm (or TM) should be able to solve a problem, in an efficient way. But what does it mean for a computation to be *efficient*? To formally describe this idea, computer scientists defined **complexity measures** to quantify the amount of computational resources needed by a Turing machine. An efficient TM should be able to solve a task with low computational resources. Above all, we are interested in studying the amount of steps needed and the amount of cells needed to achieve such computations. These two concepts are referred to as the *time complexity* and the *space complexity* of a Turing machine.

**Definition 2.** Given a Turing machine $M$ that halts on every input, we define the time complexity and space complexity of $M$ as the two functions $t, s : \mathbb{N} \to \mathbb{R}^+$ such that $t(n)$ and $s(n)$ are respectively the maximum number of steps and cells used by $M$ during the computation for any input of length $n$.

Even between small powers there is a huge difference in resources. For instance, an algorithm that requires $n$ steps is clearly a lot more efficient than one that requires $n^2$ steps. In complexity theory, we consider as **efficient** any solution that requires a "reasonable" amount of resources, even when they are huge: an algorithm that requires $n^{1000}$ steps is still considered "reasonable", while the same doesn't hold for an algorithm that requires $2^n$ steps. In other words, efficiency dictates whether a problem is feasible or not in the real world: if a problem is computable by a TM but it requires an immense amount of time or space to get to the result, then the computation is practically unachievable. These problems are often referred to as **intractable problems** [AB09; Sip96].

Time and space complexity are intrinsically related one to the other. For instance, if a machine has time complexity $t(n)$ then the number of cells that can be used is limited by $t(n)$, hence $s(n) \leq t(n)$. Usually, these two measures are proportionally inverse: if we allow our algorithm to use more space then the computation can be sped up, while if we want to lower the amount space needed then the computation will require more steps. Larger inputs require a larger amount of computational resources, making the values $t(n)$ and $s(n)$ proportional to the size $n$ of the input. For this reason, as the input size grows, we are interested in the asymptotic behavior of these measures. This concept is captured by the so-called *big-Oh notation*.

**Definition 3.** Given two functions $f, g : \mathbb{N} \to \mathbb{R}^+$, we say that:

1. $f$ is in big-Oh of $g$, written as $f(n) = O(g(n))$, if there are two constants $c \in \mathbb{R}$ and $N \in \mathbb{N}_{>0}$ such that $\forall n \geq N$ it holds that $f(n) \leq cg(n)$.

2. $f$ is in Omega of $g$, written as $f(n) = \Omega(g(n))$, if there are two constants $c \in \mathbb{R}$ and $N \in \mathbb{N}_{>0}$ such that $\forall n \geq N$ it holds that $f(n) \geq cg(n)$.

3. $f$ is in Theta of $g$, written as $f(n) = \Theta(g(n))$, iif there are two constants $c \in \mathbb{R}$ and $N \in \mathbb{N}_{>0}$ such that $\forall n \geq N$ it holds that $cg(n) \leq f(n) \leq dg(n)$.

In other words, as the input size $n$ grows the function $f$ can dominate, be dominated or both by a function $g$, defining the *lower* and *upper* bounds of the value $f(n)$. In particular, when $f(n) = \Theta(g(n))$ the two functions can be considered to be *almost* the same due to them following the same growth pattern. Additionally, its easy to see that $f(n) = \Omega(g(n))$ if and only if $g(n) = O(f(n))$ and likewise that $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

In general, we consider an algorithm as *time efficient* if it can compute the answer to any input in at most a polynomial amount of time, i.e. in $O(n^k)$ time for some $k \in \mathbb{N}$. Likewise, it is considered *space efficient* if it can compute the answer to any input in at most a polynomial amount of space.

For example, consider the following informally defined Turing machine $M$ which takes the binary encoding $\langle m \rangle$ of a natural number $m \in \mathbb{N}$ as the input string and returns $\langle m^2 \rangle$ as the output string. The computation made by $M$ is achieved by repeatedly adding the value $m$.

$M =$ "Given the input string $\langle m \rangle$:

1. Copy the string $\langle m \rangle$ on a blank set of contiguous cells. This copied string will be referred to as the value $k$.

2. Copy the string $\langle m \rangle$ on a blank set of contiguous cells. This copied string will be referred to as the value $y$.

3. Repeat while the value $k$ is bigger than 1:

    3. Copy the string $\langle y \rangle$ on a blank set of contiguous cells. This copied string will be referred to as the value $x$.

    4. Compute $x + n$ and store the result on the space occupied by the string $\langle y \rangle$. In other words, compute $y \leftarrow x + n$.

    5. Compute $k - 1$ and store the result on the space occupied by the string $\langle k \rangle$. In other words, compute $k \leftarrow k - 1$.

6. Write $\sqcup$ on all the cells on the tape, except for the cells of the string $\langle o \rangle$.

7. Halt the machine and return the output string $\langle o \rangle$."

We know that any natural number $m \in \mathbb{N}$ can be encoded in binary with $\log_2 m$ bits. This means that the length $n$ of the input string $\langle m \rangle$ is $\log_2 m$.

Consider now the values $k$ and $o$ obtained in the computation. These values are natural numbers and they are bounded by $cm$ for some $c \in \mathbb{R}$. This means that $k, x, y = O(m)$ and thus that they can be encoded with $O(\log m)$ bits (asymptotic notation allows us to drop the subscript of the logarithm), therefore requiring $3 \cdot O(\log m)$ cells, which is asymptotically equivalent to $O(\log m)$ cells. We conclude that the space complexity of our TM is $O(\log m) = O(n)$.

To copy a string of length $\ell$, the Turing machine needs to copy $\ell$ cells but also requires to make additional shifts in order to repeatedly move from the original string to the copied one, making the total amount of steps required $O(\ell)$. In a similar fashion, binary addition (or subtraction) between two numbers $a$ and $b$ can

be computed in $O(\log a + \log b)$ steps. Since we initially set $k = m$ and the machine decrements the value of $k$ by 1 on each iteration of the loop, the computations inside the loop get executed $m - 1$ times. This means that the total number of loop steps is $O((m - 1) \log m)$. By adding the initial two copy procedures, the total number of steps done by the machine is $O(2 \log m + (m - 1) \log m)$, which is asymptotically equivalent to $O(m \log m)$. Thus, we conclude that the time complexity of such TM is $O(m \log m) = O(2^n n)$.

These complexity measures clearly imply that this TM is highly time inefficient since it requires an exponential amount of time. But does this mean that the problem is intractable? Modern computers can square a number in milliseconds, so the answer to this question should clearly be no. In fact, even by implementing the common column method for multiplying numbers usually taught to kids, this problem can be solved efficiently.

Efficiency is one of the lingering question in modern computer scientists. We know that some problems are computationally unattainable, but where is the line that separates tractable and intractable problems? What property defines problems that cannot be solved efficiently? Finding an answer to these questions may seem easy, but after more than 70 years of research it still persists in the minds of complexity theorists.

# Chapter 2

# Search problems

## 2.1 Decision vs. Search

For many years, the study of **decision problems** has been the main focus of computability theory. These problems can be described as simple questions with a «*yes*» or «*no*» answer, such as asking if some input object has some property or not. Given a language $\Sigma^*$, where $\Sigma$ is an alphabet of symbols and $\Sigma^*$ is the set of all strings on $\Sigma$, each decision problem can be described as a subset of $\Sigma^*$, where a string $\langle o \rangle$ that encodes an object $o$ is in the subset if and only if the answer to the problem for that object is positive. A «*yes*» answer is represented by a 1, while a «*no*» answer is represented by a 0. For example, given the language $\mathbb{N}$, the question «*is n a prime number?*» is modeled by the decision problem PRIMES $= \{n \in \mathbb{N} \mid n \text{ is prime}\}$. Since any symbol of an alphabet $\Sigma$ can be encoded as an unique sequence of bits, we can assume that $\{0, 1\}$ is our unique alphabet of interest [AB09; Sip96].

**Definition 4.** A decision problem for a property $P$ is a subset $L$ of a language $\{0, 1\}^*$ such that $L = \{x \in \{0, 1\}^* \mid P(x) = 1\}$.

A decision problem is said to be *decidable* if there is a Turing machine answers the question posed by the problem with 0 or 1 for any input $x \in \{0, 1\}^*$. This also implies that the machine has to halt for every input. Decidability theory plays a large role in mathematics and computer science since most problems can be modeled through it. However, by their nature, decision problems are limited. Some problems require a more complex result than a simple yes-or-no answer. Instead of asking the question «*does this object have the required property?*», we may be more interested in the question «*what gives this object the following property?*».

These kinds of questions are modeled by **functional problems**, i.e. any problem where an output that is more complex than a yes-or-no answer is expected for a given input. Functional problems are by nature "harder" than decision problems, describing any possible type of computation achievable through the concept of computable function, even decidability itself (any decision problem is just a functional problem with only two possible outputs).

Functional problems are described through the concept of relation: given a set of inputs $X$ and a set of possible outputs $Y$, a functional problem is as a relation $R \subseteq X \times Y$ such that the pair $(x, y)$ is in $R$ if and only if $y$ is the output to the input $x$ for the given question. For instance, the question «*what is the prime factorization of n?*» is modeled by the functional problem $\text{FACTOR} = \{(n, (p_1, \ldots, p_k)) \in \mathbb{N} \times \mathbb{N}^k \mid n = p_1 \cdot \ldots \cdot p_k\}$.

We observe that questions like «*is y a valid output for the input x?*» are still modeled by decision problems due to them requiring a simple yes-or-no answer, while a function problem would ask the question «*what is the output for the input x?*». For example, the question «*is $p_1, \ldots, p_k$ the prime factorization of n?*» corresponds to the decision problem $\text{FACTORIZATION}_n = \{(p_1, \ldots, p_k) \in \mathbb{N}^k \mid n = p_1 \cdot \ldots \cdot p_k\}$.

Even though decision problems can indeed be modeled as functional problems whose outputs are only «*yes*» and «*no*», they aren't effectively a subset of functional problems due to them being defined differently: the decision problem PRIMES can be converted into the functional problem $\{(n, b) \in \mathbb{N} \times \{0, 1\} \mid b = 1$ if $n$ is prime, $b = 0$ otherwise$\}$, but they aren't effectively the same problem despite answering the same question.

Another important thing to notice is that, even though the name implies a correlation to standard mathematical functions due to the concept of input-output being involved, the given definition also includes *partial* and *multivalued* functions, i.e. functions for which not all inputs have a corresponding output and functions for which one input can have more outputs. For these reasons, the term *functional problem* is considered to be slightly abused. In recent years, this issue was solved by the introduction of the more general term **search problems**, describing the idea of finding a valid output for the given input, better suiting the previous formal definition.

To give a more detailed definition of search problems, we assume that these problems all share the language $\{0, 1\}^*$, describing all inputs as a sequence of bits. Since each problem could have inputs of different lengths, researchers have defined search problems through the use of a sequence of relations rather than a single relation [BCE+98; RGR22; BFI23]. This also allows separation between different types of outputs based on the length of the inputs.

**Definition 5.** A search problem is a sequence $R = (R_n)_{n \in \mathbb{N}}$ of relations $R_n \subseteq \{0, 1\}^n \times O_n$, one for each $n \in \mathbb{N}$, where each $O_n$ is a finite set called outcome set.

Since it includes partial functions, this definition allows search problems to be "undefined" for some inputs, meaning that there is no answer for some inputs. A search problem is said to be **total** if for each $R_n$ in the sequence it holds that $\forall x \in \{0, 1\}^n$ there is an answer $y \in O_n$ such that $(x, y) \in R_n$. In other words, a total search problem has at least an output for all possible inputs, removing partial functions from the context, while multivalued functions are still allowed. For example, FACTORING is a total non-multivalued search problem due to each natural number having a guaranteed unique prime factorization by the Fundamental Theorem of Arithmetic.

## 2.2 The complexity classes **FP**, **FNP** and **TFNP**

In complexity theory, decision problems are grouped into numerous categories, each defining a subclass. The most important subclass corresponds to the set of problems that can be **efficiently solved**. This class is referred to as P, i.e. the class of problems solvable by a Turing machine in polynomial time. Not all decision problems are efficiently solvable, i.e. the so-called *intractable problems* (see Chapter 1). Several problems for which there is currently no answer regardless of whether or not they are efficiently solvable have been shown to be **efficiently verifiable**, meaning that there is a Turing machine called *verifier* that given an additional input $c$, namely the *certificate*, is capable of telling in polynomial time if the value $y$ is the output of an input $x$.

**Definition 6.** A verifier for a decision problem $L$ is a Turing machine $V$ such that for each input $x \in \Sigma^*$ there is a certificate $c \in \Sigma^*$ for which $V(x, c) = 1$ if and only if $x \in L$.

The class of problems that are verifiable by a polynomial time verifier with certificates of polynomial length is referred to as NP. This class is equivalent to the class of problems efficiently solvable by a *non-deterministic Turing machine*, a TM that on each step of the computation can choose between a set of possible actions, branching the computation. Originally, the class NP was defined through this type of TM - hence the name of the class being an abbreviation for *non-deterministic polynomial time* - but it quickly got replaced with the verifier definition due to NTMs being only a theoretical computational model that is physically unrealizable [AB09]. For our purposes, we will consider the modern definition of NP.

**Definition 7.** We define P as the set of decision problems that can be solved by a polynomial time TM. We define NP as the set of decision problems that can be verified by a polynomial time verifier.

It's easy to see that $P \subseteq NP$ since every efficiently solvable problem can also be efficiently verified by simply ignoring the certificate and solving the problem. However, it is currently not known whether $P = NP$ or not. The answer to this question is considered to be one of the most important questions in mathematics. In fact, if $P = NP$ were to be true, a lot of key problems in mathematics that are currently only efficiently verifiable could be solved in a reasonable amount of time by a modern computer. On the other hand, a large number of current technologies are based on the **assumption** that $P \neq NP$. Examples include the field of cryptography, which assumes that it's easy to check that each encrypted string is the result of the encryption scheme being applied to the original message, which works as the certificate, and very hard to find this message only through the encrypted string. If $P \neq NP$ were proven false, we would have to reconsider a large portion of the modern world, even digital currencies themselves.

In the context of search problems, we define the class FP - *functional* P - as the class of search problems efficiently solvable by an algorithm and FNP - *functional* NP - as the class of search problems whose solutions are efficiently verifiable by a verifier.

**Definition 8.** We define FP as the set of search problems $R = (R_n)_{n \in \mathbb{N}}$ whereby $\forall n \in \mathbb{N}$ there is a polynomial time TM $T_n$ such that $T_n(x) = y$ if and only if $(x, y) \in R_n$. We define FNP as the set of search problems $R = (R_n)_{n \in \mathbb{N}}$ whereby $\forall n \in \mathbb{N}$ there is a polynomial time verifier $V_n$ such that $\exists w \in \{0, 1\}^{n^k}$ for which $V_n(x, y, w) = 1$ if and only if $(x, y) \in R_n$.

An important remark to be made is that, even though any decision problem can be transformed into a search problem with only two possible outputs, since they are defined on two different types of problems it doesn't make sense to say that P $\subseteq$ FP or that NP $\subseteq$ FNP. However, an important result shows that it can hold that P $=$ NP if and only if FP $=$ FNP [BG94; DK14]. This implies that, despite search problems being by definition more complex than decision problems, the functional version of the conjecture is as hard as the decisional one.

**Theorem 1.** *P $=$ NP if and only if FP $=$ FNP*

*Proof.* Since each decision problem can be translated into a search problem with only two possible outcomes, we trivially get that if FP $=$ FNP then P $=$ NP.

Suppose now that P $=$ NP. We already know that FP $\subseteq$ FNP, so we have to show that FP $\subseteq$ FNP. Let $R = (R_n)_{n \in \mathbb{N}} \in$ FNP be a search problem verifiable in polynomial time.

For each $n \in \mathbb{N}$, let $L_n$ be the set of pairs $(x, z)$ such that $z$ is the prefix of an outcome $zw$ for the problem $R_n$ with input $x$, formally $L_n = \{(x, y) \mid \exists z \in \{0, 1\}^k, k \leq n$ s.t. $(x, zw) \in R_n\}$. It's easy to see that $L_n \in$ NP since each pair $(x, z)$ is certified by the string $zw$ itself and the correctness of this certificate can be polynomially verified given that $R \in$ FNP.

Since $L_n \in$ NP $=$ P, we know that there is a polynomial time algorithm $\text{Partial}_n$ that decides $L_n$. Thus, for each $n \in \mathbb{N}$, we can construct the following polynomial time algorithm $\text{Solve}_n$ which directly concludes that $R \in$ FP and thus that FNP $\subseteq$ FP.

```
function Solve_n(x)
    y = ε                                          ▷ ε is the empty string
    while True do
        if Partial_n(x, y0) = True then
            y = y0
        else if Partial_n(x, y1) = True then
            y = y1
        else
            return y
        end if
    end while
end function
```

$\square$

As discussed in the previous section, not all search problems are **total**, meaning that a solution could not exist for some inputs. A lot of real-world problems have a guaranteed solution for each input, ranging from simple number functions to harder problems, making total search problems more interesting than non-total ones.

**Definition 9.** We define the class TFNP as the subset of FNP problems that are also total.

For simplicity, we assume that each search problem in FP is also total: since problems in FP are solvable in polynomial time, when a solution doesn't exist we can output a pre-chosen «*doesn't exist*» solution, making the problem total. This assumption easily implies that FP $\subseteq$ TFNP $\subseteq$ FNP, giving us a proper hierarchy. For natural reasons, this assumption wouldn't work for FNP problems: the only way to polynomially verify that a solution doesn't exist would be to solve the problem itself and find that there is no solution, implying that FP = FNP would be trivially true.

Another way to view total search problems is through the lens of *polynomial disqualification*. In decisional problems, the class coNP contains all the problems whose complement is in NP. If the complementary problem is polynomially verifiable, this means that there is a polynomial verifier that can decide if an input doesn't have the required property, effectively disqualifying it. Proving that a decision problem is in coNP is also equivalent to proving that for each input of that problem there is no string capable of certifying that the solution is correct. Researchers currently believe that NP $\neq$ coNP, even though this is still an open question. If the answer to this question is proven negative, we would also have a direct answer to the P $\stackrel{?}{=}$ NP question: we know that if NP $\neq$ coNP then P $\neq$ NP [AB09; Sip96]

For search problems, we define the class FcoNP in the same way. In particular, the class TFNP corresponds to the class F(NP $\cap$ coNP), which contains search problems whose inputs can be certified and disqualified in polynomial time [MP91].

**Proposition 1.** TFNP = F(NP $\cap$ coNP)

*Proof.* If $R = (R_n)_{n \in \mathbb{N}} \in$ TFNP then we know that every input $x$ has an output $y$. However, this means that the complementary problem $\overline{R}$ is empty, meaning that each input is trivially verifiable in polynomial time and thus that $\overline{R} \in$ FNP. Hence, we conclude that $R \in$ F(NP $\cap$ coNP).

Vice versa, if $S \in$ F(NP $\cap$ coNP) then trivially we have that $S \in$ FNP. Moreover, since $S \in$ F(NP $\cap$ coNP) we know that each input $x$ can be easily certified or disqualified in polynomial time, meaning that each input must have a solution polynomially verifiable and thus that $S \in$ TFNP.

$\square$

## 2.3   Reductions between problems

One of the most interesting aspects of computable (and uncomputable) problems is the ability to be transformed into another problem in order to achieve a solution. Suppose that we have an instance $a$ of problem $A$ and that we know an algorithm that transforms $a$ into an instance $b$ of a problem $B$ such that $a$ is a «*yes*» answer if and only if $b$ is a «*yes*» answer. Then, by solving $b$ we would get an answer to $a$. In computer science, this concept is known as **reduction**: a problem $A$ is said to be reducible into a problem $B$, written as $A \leq B$, if any instance $a$ of $A$ can be mapped into an instance $b$ of $B$ whose solution gives a solution to the former.

In decision problems, this concept is described through **many-to-one mappings**, computable functions that map instances of the original problem to instances of the reduced problem. This particular type of reduction from $A$ to $B$ is denoted by $A \leq_m B$.

**Definition 10.** A decision problem $A$ is many-to-one reducible to a decision problem $B$ if there is a computable function $f$ such that $x \in A$ if and only if $f(x) \in B$.

To better understand how reductions work, consider the 3COL problem, which asks the question «*is this graph 3-colorable?*». A graph is said to be 3-colorable if we can color each node of the graph with a color different from all of its neighbors by using at most 3 colors.
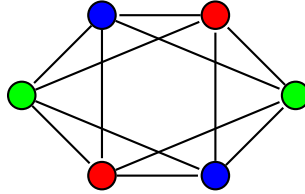


**Figure 2.** Example of a 3-colorable graph.

This problem can easily be reduced to the SAT problem, which asks the question «*does this propositional formula have an assignment that satisfies it?*». This problem is clearly in NP: if $\phi$ is a satisfiable formula, we can use an assignment $\alpha$ that satisfies $\phi$ as the certificate, which can easily be done in linear time. To reduce 3COL to SAT, we construct the following sub-formulas:

- For each node $v \in V(G)$, we define the sub-formula $\phi_v = (r_v \lor g_v \lor b_v)$ which imposes that each node must have at least one color assigned

- For each edge $(u,v) \in E(G)$, we define the sub-formula $\phi_{(u,v)} = (\overline{r_u} \land \overline{r_v}) \land (\overline{g_u} \land \overline{g_v}) \land (\overline{b_u} \land \overline{b_v})$ which imposes that the nodes of each edge must have different colors.

Then, we construct the final formula that encodes the 3-colorization of $G$:

$$\phi_G = \left( \bigwedge_{v \in V(G)} \phi_v \right) \land \left( \bigwedge_{(u,v) \in E(G)} \phi_{(u,v)} \right)$$

By definition of $\phi_G$, it holds that $G$ is 3-colorable if and only if $\phi_G$ is satisfiable. In particular, there is a bijection between $G$'s color assignments and $\phi_G$'s variable assignments. This concludes that $3\text{COL} \leq_m \text{SAT}$. Moreover, this reduction can be computed by a Turing machine in polynomial time. When this happens, we write $3\text{COL} \leq_p \text{SAT}$

Many-to-one reductions between problems are transitive: starting from a problem $A$, we can reduce it to a problem $B$ through a function $f$ and then reduce it to a problem $C$ through a function $g$. This implies that the composition $g \circ f$ is a reduction from $A$ to $C$. For instance, the 4-COL problem, a variant of the 3-COL with four colors instead of three colors, can be reduced in polynomial time to 3-COL, giving us the reduction chain $4\text{-COL} \leq_p 3\text{-COL} \leq_p \text{SAT}$.

Reductions between decision problems map any «*yes*» answers of problem $A$ to some «*yes*» answers of problem $B$ and the same goes for «*no*» answers. In search problems, however, there is no concept of a negative answer: even if a problem has only two possible outputs, both of them are still a solution. Some people could argue that an input for which there is no solution is a negative answer to the search problem. But how could we map inputs without solutions to other inputs without solutions? What if one of the two problems involved is total and the other isn't? This clearly doesn't make sense and, even if it did, we are only interested in finding solutions. We give the following definition of search problem reduction:

**Definition 11.** A search problem $R = (R_m)_{m \in \mathbb{N}}$, where $R_m \subseteq \{0,1\}^m \times O_m$ is said to be many-to-one reducible to a search problem $S = (S_n)_{n \in \mathbb{N}}$, where $S_n \subseteq \{0,1\}^n \times O'_n$, if for all $m \in \mathbb{N}$ there is an $n \in \mathbb{N}$ for which there is a computable function $f : \{0,1\}^m \to \{0,1\}^n$ and a computable function $g : \{0,1\}^m \times O'_n \to O_m$ such that:
$$\forall x \in \{0,1\}^m \ (f(x), y) \in S \implies (x, g(x,y)) \in R$$

In other words, the function $f$ maps inputs of $R$ into inputs of $S$, while the function $g$ maps solutions of $S$ into solutions of $R$.

When a reduction can be efficiently computed by a TM with a time (or space) complexity that is in the order of the complexity of $B$, the problem $A$ can be solved by a machine that first computes the reduction and then solves the problem $B$. For instance, if $A \leq_p B$ through a function $f$ and $B$ can be solved in polynomial time then we can build a machine that first computes $f(x)$ for a given input $x$ and then compute whether $f(x) \in B$ or not. Since both the reduction and the final computation in this case require polynomial time, we conclude that $A$ can also be solved in polynomial time.

Reductions play a critical role in computer science. In particular, they allow us to define the concept of **completeness**. A problem is considered complete for a class $\mathcal{C}$ when every problem from its class can be reduced to it under a specific constraint. This constraint usually depends on the complexity constraints dictated by a subclass $\mathcal{D} \subseteq \mathcal{C}$: if every problem in $\mathcal{C}$ is reducible to $B$ under the same constraints that define the subclass $\mathcal{D}$ and we can show that $B \in \mathcal{D}$, then the whole class $\mathcal{C}$ collapses onto its subclass, that is $\mathcal{C} = \mathcal{D}$.

For instance, since $P \subseteq NP$, the completeness constraint behind NP-Completeness would be to maintain polynomial time reductions. In P-Completeness, instead, the completeness constraint would be to maintain logarithmic space reductions due to how $L \subseteq P$, where $L$ is the class of problems decidable in logarithmic space.

**Definition 12.** A problem $B$ is said to be NP-Complete if $B \in \mathcal{C}$ and $\forall A \in NP$ it holds that $A \leq_p B$.

From this definition, we derive that an NP-Complete problem can be solved in polynomial time if and only if $P = NP$. The SAT problem previously described is actually the first ever known NP-Complete problem, a seminal result proven by Cook in 1971 [Coo71] and later by Levin in 1973 [Lev73]. In particular, Levin proved this result through the functional version of this complete problem FSAT, which asks the question «*which assignment satisfies the following formula?*», modeling what he called *universal sequential search problem*. In fact, the functional versions can be used to prove that the decisional versions are complete and vice versa [BCE+98].

**Theorem 2.** *The decisional problem $A$ is NP-Complete if and only if the functional problem $FA$ is FNP-Complete.*

Since Cook's result, many important problems have been shown to be NP-Complete, such as Karp's 21 problems [Kar72]. Nowadays, thousands of problems fall into this class. Since most researchers believe that $P \neq NP$, once a problem is proven to be NP-Complete they start looking to approximations of the problem, due to a general optimal solution being out of reach. Proving that $P \neq NP$ would imply that all such problems are actually *intractable*, making approximation algorithms the best we can hope for.

## 2.4 The TFNP hierarchy

Currently, it is not known if there is a FNP-Complete problem that is also *total*. For example, the problem FSAT isn't total due to some formulas being unsatisfiable, thus no satisfying assignment can be returned as a solution. Researchers believe that the existence of such a problem is very unlikely.

For these reasons, in the TFNP world the concept of completeness is studied under a *different approach*: instead of considering problems that are complete for the whole class, we consider important problems that have a lot of TFNP problems reducible to them. These important problems form additional subclasses of TFNP. With a slight abuse of notation, we denote with $S$ the class of problems efficiently reducible to the search problem $S$.

**Definition 13.** Given TFNP problem $S$, we define the class $S$ as the subset of TFNP problems efficiently reducible to the problem $S$ in polynomial time, formally $S = \{R \in TFNP \mid R \leq_m S \text{ in polynomial time}\}$

The extensive study of TFNP classes has been successful in capturing the complexity of many branches of mathematics, such as problems from cryptography, game theory and economics that are reducible to TFNP complete problems. Unexpectedly, a vast majority of total search problems can be characterized with very few subclasses, which form the **TFNP hierarchy**.

Each of these subclasses is characterized by a complete total search problem that describes an elementary question, such as determining if a mapping doesn't have collision or not - or equivalently, if a function is injective or not [RGR22; BFI23]. These complete problems are guaranteed to be total by the very *combinatorial principles* that dictate them:

- PLS (Polynomial Local Search): the class of search problems designed to model the process of finding the local optimum of a function or the class of problems whose solution is guaranteed by the «*Every directed acyclic graph has a sink*» principle. It is formally defined as the class of search problems that are polynomial-time reducible to the SINK-OF-DAG problem.

- PPP (Polynomial Pigeonhole Principle): the class of problems whose solution is guaranteed by the «*Every mapping from a set of $n + 1$ elements to a set of $n$ elements has a collision*» principle. It is defined as the class of problems that are polynomial-time reducible to the PIGEON problem.

- PPA (Polynomial Parity Argument): the class of problems whose solution is guaranteed by the «*Every undirected graph with an odd-degree node must have another odd-degree node*» principle. It is defined as the class of problems that are polynomial-time reducible to the LEAF problem.

- PPADS (Polynomial Parity Argument - Directed with Sink): the class of problems whose solution is guaranteed the «*Every directed graph with a positively unbalanced node (out-degree > in-degree) must have a negatively unbalanced node*» principle. It is defined as the class of problems that are polynomial-time reducible to the SINK-OF-LINE problem.

- SOPL (Sink of Potential Line): the class of problems that are polynomial-time reducible to the SINK-OF-POTENTIAL-LINE problem. It has been proven that SOPL = PLS ∩ PPADS [GHJ+22a]

- PPAD (Polynomial Parity Argument - Directed): the class of problems whose solution is guaranteed the «*Every directed graph with an unbalanced node must have another unbalanced node*» principle. It is defined as the class of problems that are polynomial-time reducible to the END-OF-LINE problem.

- CLS (Continuous Local Search): the class of search problems designed to model the process of finding a local optimum of a continuous function over a continuous domain. It is defined as the class of problems that are polynomial-time reducible to the CONT-LOCALPOINT problem. It has been proven that CLS = EOPL = PLS ∩ PPAD [FGH+22; GHJ+22a], where EOPL is the class of search problems that are polynomial-time reducible to the END-OF-POTENTIAL-LINE problem.
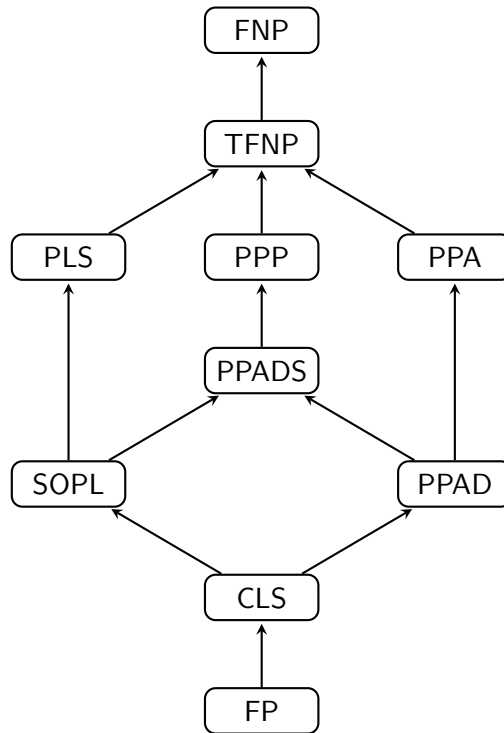
**Figure 3.** Hierarchy of the main total search problem subclasses.
An arrow from class $A$ to class $B$ means that $A \subseteq B$.

Interestingly, lots of complex problems have been proven to be reducible to these basic problems. For example, the NASH problem relative to finding a Nash equilibrium of a given game has been shown to not only lie inside PPAD but to also be PPAD-Complete [DGP06; CDT09]. One should ponder what it really means for a problem to be complex.

Proving any unconditional separation between these subclasses, which can be achieved by showing that one of them is not efficiently reducible to the other, would directly imply that FP $\neq$ TFNP, answering the P $\overset{?}{=}$ NP question. By the hardness of the question itself, finding such unconditional separation seems to be completely out of reach. However, it turns out that the TFNP model indeed has conditional separations, in particular relative to *oracles* (see Chapter 3).

## 2.5 White-box **TFNP**

In computer science and engineering, systems and models are distinguished between white-box and black-box systems. A system is **white-box** if its internal workings are known, meaning that given any input it is possible to know how the system achieves a result. Contrary, the computational process is unknown in a **black-box** system. Black-box models allow us to consider only the result for a given input, ignoring how that result is achieved. For example, a programmer uses both white-box and black-box systems: personal functions are white-boxes, while ready-to-go library functions are black-boxes.

Each TFNP problem can be analyzed through the lens of both white-box and black-box systems. Originally, these two models were characterized by solvability and verifiability through Turing machines [BG94; BCE+98]. In recent years, researchers have shifted to another characterization: the white-box TFNP model is studied through *Boolean circuits*, while black-box TFNP is studied through *decision trees* [GHJ+22b; GKR+19; BFI23]. Any reader who has come this far will have asked himself the following question: why shift to other computational models? The answer is pretty straightforward: they are easier to work with. This shift of perspective allowed researchers to perform complex reasoning more easily, reaching otherwise unintuitive results. In this section, we will briefly discuss the white-box model, while the black-box model will be extensively discussed in the following chapter.

In this context, Boolean circuits are defined as sets of logical AND and logical OR gates connected by cables. Boolean circuits have been proven to be Turing complete due to Turing machines and families of circuits being capable of simulating each other up to a logarithmic factor [AB09]. Again, none should be dumbfounded by this result: any modern computer is just a large amount of Boolean gates wired together. We give the following definition of a Boolean circuit. [RYM+22]

**Definition 14.** A Boolean circuit is a directed acyclic graph whose nodes, called gates, each associated with either an input variable, its negation or a Boolean operator. Each input gate has in-degree 0 and unlimited out-degree. Each Boolean gate has an out-degree gate equal to 1 (except for the output gate which has out-degree 0) and in-degree equal to either 1 or 2. All the 2 in-degree gates compute the logical AND or the logical OR of their given input variables or Boolean function.

Each gate $v$ is associated with the Boolean function $f_v$ computed by it. A function $f$ is said to be computed by a circuit with output gate $u$ if for all inputs $x \in \{0, 1\}^n$ it holds that $f(x) = f_u(x)$.

The complexity of Boolean circuits is measured in terms of their *size* and *depth*, i.e. the number of gates of the circuit and the length of the longest directed path from an input gate to the output gate. However, differently from protocols, the **circuit complexity** of a function $f$ is defined as the size of the smallest Boolean circuit that computes it.
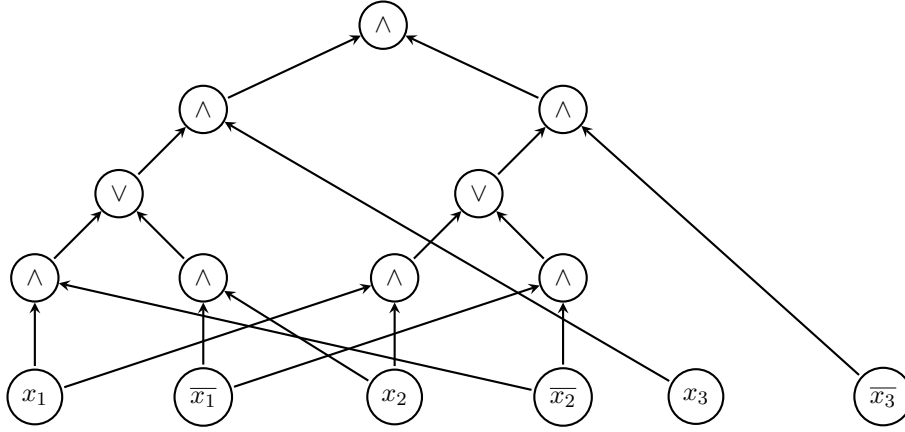
**Figure 4.** A Boolean circuit of size 15 and depth 4 computing $x_1 \oplus x_2 \oplus x_3$.

One of the most interesting properties of circuits is their strict relation to *protocols*. Suppose that we have two parties, namely Alice and Bob, who want to cooperate in order to achieve a common objective, like computing a function. To reach their goal, Alice and Bob must carry out separate computations, communicating the result to the other party in a pre-defined sequence of steps. This idea serves as groundwork for a definition of protocols, algorithms that dictate such alternations between computation and communications. We give the following formal definition of protocol. [RYM+22]

**Definition 15.** Let $X$ be Alice's input set and let $Y$ be Bob's input set. A protocol $\pi$ is a rooted directed binary tree whose leaves are associated with outputs and internal nodes are owned by either Alice or Bob, where the owner of $v$ is noted by owner($v$). Each leaf is labeled with an output $o \in O$, where $O$ is the outcome set. Each internal node $v$ is also associated to a function $g_v : Z \to \{0, 1\}$, where $Z = X$ if owner($v$) = A and $Z = Y$ if owner($v$) = B.

When given the input $(x, y) \in X \times Y$, the protocol computes the associated function of the current node (starting from the root), proceeding on the left child if the output is 0 and on the right child if the output is 1. When a leaf is reached, the protocol returns the associated output. The output of the protocol for a given input $(x, y)$ is denoted with $\pi(x, y)$. A function $f$ is said to be computed by the protocol $\pi$ if for all inputs $(x, y)$ it holds that $f(x, y) = \pi(x, y)$.

The complexity of protocols is measured in terms of their *size* and *depth*, that being the number of nodes of the protocol and the length of the longest directed path from the root node to a leaf. The **communication complexity** of a function $f$ is defined as the depth of the smallest protocol that computes $f$, corresponding to the minimal number of bits that must be communicated by Alice and Bob to compute $f$ for all possible inputs.
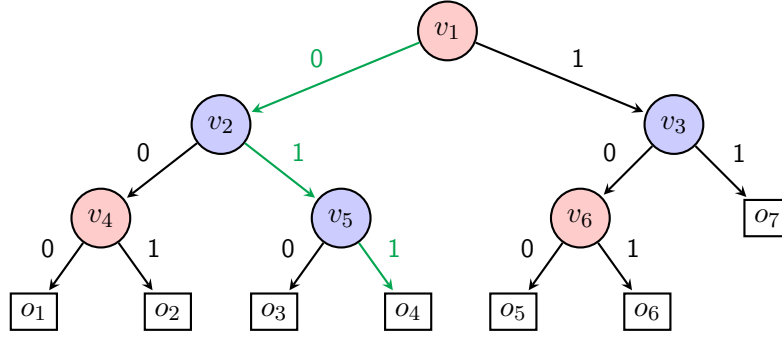
**Figure 5.** An example of a protocol of size 13 and depth 3 where the red nodes are owned by Alice and the blue nodes are owned by Bob. The green path shows the computation given by $f_{v_1}(x) = 0$, $f_{v_2}(y) = 1$ and $f_{v_5}(y) = 1$ for the input $(x, y)$.

A protocol encodes all possible messages that may be sent by the parties during any conceivable conversation, producing the expected output. This means that a protocol always returns an answer for all possible inputs, making any function computed by a protocol *total*. This makes protocols are also a valid way to solve or verify total search problems.

In particular, for each TFNP problem $R$, we denote with $R^{cc}$ the equivalent TFNP$^{cc}$ problem, where *cc* stands for *communication complexity*. Due to them being defined on two inputs instead of one, communication search problems are defined on two sets of input values instead of one.

**Definition 16.** A communication search problem is a sequence $R = (R_n)_{n \in \mathbb{N}}$ of relations $R_n \subseteq \{0, 1\}^n \times \{0, 1\}^n \times O_n$, one for each $n \in \mathbb{N}$, where each $O_n$ is a finite set called "outcome set".

A protocol is considered to be efficient when its communication complexity is polylogarithmic with respect to the bit-size of the inputs, i.e. equal to $O(\log^k n)$. This ensures that there is a Turing machine capable of simulating the protocol in polynomial time. We give the following definitions of FP$^{cc}$ and FNP$^{cc}$

**Definition 17.** We define FP$^{cc}$ as the set of communication search problems $R = (R_n)_{n \in \mathbb{N}}$ for which there exists a polylogarithmic depth protocol $\pi_n$ such that $\pi_n(x, y) = z$ if and only if $((x, y), z) \in R_n$. We define FNP$^{cc}$ as the set of communication search problems $R = (R_n)_{n \in \mathbb{N}}$ for which there exists a polylogarithmic depth protocol $V_n$ such that $V_n((x, y), z) = 1$ if and only if $((x, y), z) \in R_n$.

In this case, the certificate is the protocol itself: it defines a schema through which a Turing machine can verify the solution. The concept of reduction also applies to communication search problems, but only under a pre-fixed value $t$ of the maximum amount of bits usable in the reduction, i.e. the maximum depth of the reduction protocol, which is necessary for computational reasons that we won't discuss. This allows us to define a $t$-bit TFNP$^{cc}$ hierarchy that follows the same structure as the standard one.

**Definition 18.** A communication search problem $R = (R_m)_{m \in \mathbb{N}}$, where $R_m \subseteq \{0,1\}^m \times \{0,1\}^m \times O_m$, is said to be many-to-one reducible into a search problem $S = (S_n)_{n \in \mathbb{N}}$, where $S_n \subseteq \{0,1\}^n \times \{0,1\}^n \times O'_n$, if for all $m \in \mathbb{N}$ there is an $n \in \mathbb{N}$ for which there are two functions $f_X, f_Y : \{0,1\}^m \to \{0,1\}^n$ and a $t$-bit protocol $g : (\{0,1\}^m \times \{0,1\}^n) \times O'_n \to O_m$ such that:

$$\forall (x,y) \in \{0,1\}^m \times \{0,1\}^m \ (f_X(x), f_Y(y), z) \in S \implies (x, y, \pi((x,y), z)) \in R$$

In other words, the functions $f_X, f_Y$ map inputs of $R$ into inputs of $S$, while the protocol $g$ maps solutions of $S$ into solutions of $R$.

Circuits and protocols are related one another through the **Karchmer-Widgerson game**. The game has a simple objective: given two inputs with different outputs, Alice and Bob have to cooperate to find a bit that differs between the two inputs. When the function is monotone, that being a function for which given any pair of inputs $x, y$ such that $x \leq y$ it also holds that $f(x) \leq f(y)$, the game is also called monotone.

**Definition 19.** Given a Boolean function $f : \{0,1\}^n \to \{0,1\}$, we define the Karchmer-Wigderson game of $f$, denoted with $\mathrm{KW}(f)$, as the following communication problem: given the two inputs $x$ and $y$, where $f(x) = 0$ and $f(y) = 1$, find an index $i \in [n]$ such that $x_i \neq y_i$.

If $f$ is a monotone Boolean function, meaning that given two inputs $x, y$ if $x \leq y$ then $f(x) \leq f(y)$, the monotone Karchmer-Widgerson game of $f$, denoted with $\mathrm{mKW}(f)$, finds an index $i \in [n]$ such that $x_i < y_i$.

These games were originally introduced in 1990 by Karchmer and Widgerson [KW88] to show how the communication complexity of a game for a function $f$ is equal to the circuit complexity of a *Boolean circuit* that solves the game on $f$.

**Theorem 3.** *Given a function $f : \{0,1\}^n \to \{0,1\}$, there is a circuit of depth $d$ that computes $f$ if and only if there is a protocol of depth $d$ that solves $\mathrm{KW}(f)$. Moreover, if $f$ is monotone, the circuit is monotone and the protocol solves $\mathrm{mKW}(f)$*

A surprising result [Gál02; GKR+19] proved that any communication search problem is equivalent to the monotone KW game of some Boolean function. This result implies that $\mathsf{TFNP}^{cc}$ exactly is the study of the monotone Karchmer-Widgerson game.

**Lemma 1.** *For any communication search problem $R = (R_n)_{n \in \mathbb{N}}$, where $R_n \subseteq \{0,1\}^n \times \{0,1\}^n \times O_n$, in $t$-bit $\mathsf{TFNP}^{cc}$, there is a function $f$ on $2^t |O_n|$ variables such that $R$ is communication equivalent to $\mathrm{mKW}(f)$ under $t$-bit mapping reductions.*

These results further extend the already known connections between search problems, communication complexity and circuit complexity, establishing that any result obtained in one of these models can be in some way established to the others.

# Chapter 3

# Black-box TFNP

## 3.1  Oracles and decision trees

In the previous chapter, we have briefly shown how TFNP subclasses are defined in terms of basic existence principles that capture white-box total search problems solvable by protocols reducible to Karchmer-Widgerson games. From now on, we will shift our focus to the black-box model.

Black boxes have been used by complexity theorists since the early days, mostly through the concept of **oracle**, a device capable of instantly solving an instance of a designated problem. In particular, these problems may even be uncomputable, an assumption that allows us to view oracles as magical devices. Turing machines can be allowed to query such oracles to an additional *oracle tape*. The machine writes a string on such tape, asking the oracle to solve the problem for that input. The output of the oracle is then written on the same tape, which can then be read by the Turing machine. Any query made to the oracle requires $\Theta(1)$ time, meaning that they don't influence the cost of the computation.

**Definition 20.** An oracle for a problem $A$ is an external device that is capable of instantaneously solving an instance of $A$. An oracle Turing machine is a Turing machine provided with the ability to query an oracle. We write $M^A$ to describe a Turing machine provided with an oracle for the problem $A$.

Given a class $\mathcal{C}$ and an oracle for a problem $A$, the *relativized version* of the class $\mathcal{C}$, written $\mathcal{C}^A$ is the set of all problems of $\mathcal{C}$ solvable (or verifiable) with access to the oracle of $A$. For example, $\mathsf{P}^{\mathrm{SAT}}$ is the class of problems solvable in polynomial time by a Turing machine with an oracle for the SAT problem. More generally, given two classes $\mathcal{C}, \mathcal{B}$, we write $\mathcal{C}^{\mathcal{B}}$ to denote the set of all problems of $\mathcal{C}$ solvable (or verifiable) with access to an oracle for any problem that lies in $\mathcal{B}$. In other words, we have that $\mathcal{C}^{\mathcal{B}} = \bigcup_{A \in \mathcal{B}} \mathcal{C}^A$.

Oracles proved to be surprisingly useful for studying the relationship between $\mathsf{P}$ and $\mathsf{NP}$ by considering the relationship between $\mathsf{P}^A$ and $\mathsf{NP}^A$ for an oracle $A$. In a celebrated result [BGS75], Baker et al. showed that there are two problems $A$ and $B$ such that $\mathsf{P}^A = \mathsf{NP}^A$ and $\mathsf{P}^B \neq \mathsf{NP}^B$. This fact makes many commonly used proof techniques useless against the conjecture, meaning that any answer to the $\mathsf{P} \overset{?}{=} \mathsf{NP}$

question will require techniques that are invariant with respect to the presence of an oracle.

Oracles provide a simple yet effective way to generalize the concept of reduction through the so-called *Turing reductions*: if a Turing machine provided with an oracle for the problem $B$ is capable of resolving a problem $A$ then the problem $A$ can be reduced to solving multiple instances of the problem $B$. When $A$ is Turing reducible to $B$, we write $A \leq_T B$. Clearly, if the oracle machine $M^B$ can solve $A$ then any query to the oracle can be replaced with a call to a subroutine that solves $B$. This conversion is often referred to as *de-relativization*. Many-to-one reductions can be seen as a specific case of Turing reductions, where the machine makes exactly one query to the oracle and then returns the output of such query.

In the particular case of total search problems, it was proven that the reducibility between search problems is strictly connected to the reducibility of their relativized versions up to all oracles [BCE+98].

**Theorem 4.** *Given two black-box search problems $R, S \in$ TFNP and their relative classes it holds that $R \leq_m S$ if and only if $R^A \leq_m S^A$ for all oracles $A$.*

This result implies that proving any relativized separation is equivalent to proving a non-relativized separation, allowing us to use the intuitive nature of oracles to rule out possible collapses in TFNP subclasses. Many TFNP subclasses have been proven to be different through separations between the respective query search problems. [RGR22; BFI23]

**Definition 21.** A query search problem is a sequence $R = (R_n)_{n \in \mathbb{N}}$ of relations $R_n \subseteq \{0,1\}^n \times O_n$, one for each $n \in \mathbb{N}$, where each $O_n$ is a finite set called "outcome set".

A good eye will surely notice that the previous definition does not vary from the normal definition of search problems, unlike communication search problems. The only true difference resides in their computational models: query search problems are solved (or verified) through **decision trees**.

**Definition 22** ([LNN+95])**.** A decision tree is a rooted directed binary tree whose nodes are associated with either an output value or an input Boolean variable. Each leaf is labeled with an output $o \in O$, where $O$ is the outcome set. Each internal node is labeled by a variable and the two outgoing edges are labeled by the two possible values of that variable.

Decision trees can be viewed as nothing more than the black-box version of protocols: we don't care about who computes the next step or how they do it, we only care about the result being either a 0 or a 1 to proceed with the computation. In fact, like protocols, decision trees encode all possible ways to obtain a result, making them *total*. Likewise, the complexity of a decision tree computing a function follows the same complexity measures as a protocol, i.e. its *size* and its *depth*. A function $f$ is said to be computed by the decision tree $T$ if for all inputs $x$ it holds that $f(x) = T(x)$.
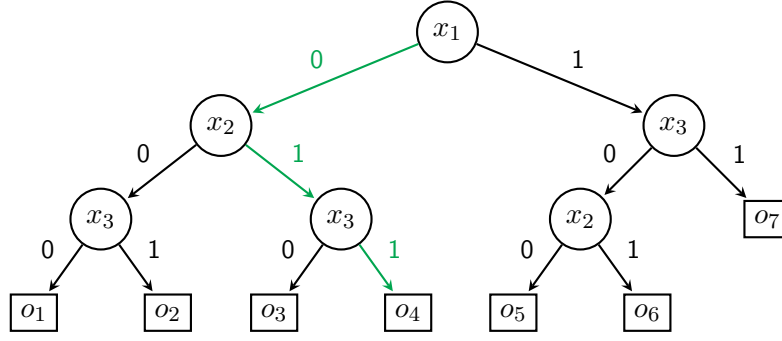
**Figure 6.** An example of a decision tree of size 13 and depth 3. The green path shows the computation made for the input $x = 011$.

Decision trees give an easier way to describe the computation of an oracle Turing machine: if $M^B$ solves (or verifies) a problem $A$ then the $i$-th query made by the procedure corresponds to a variable $x_i$ for the decision tree where $x_i = 1$ if the query returns a positive result and 0 otherwise. In other words, the computation tree of an oracle Turing machine can be viewed as a decision tree.

**Proposition 2.** *Given a search problem $A \in$ TFNP, if there is an oracle Turing machine $M^B$ that solves (or verifies) $A$ then there is a decision tree that solves (or verifies) $A$.*

The above proposition gives a strong result that allows us to characterize black-box TFNP through decision trees instead of oracles: *any decision tree separation implies a relativized separation for some oracle* [RGR22; BFI23]. As in the communication complexity formulation, given a TFNP problem $R$, we denote with $R^{dt}$ the equivalent TFNP$^{dt}$ problem, where *dt* stands for *decision tree*. We will omit this notation when the context makes it clear.

**Definition 23.** We define FP$^{dt}$ as the set of query search problems $R = (R_n)_{n \in \mathbb{N}}$ for which there exists a polylogarithmic depth decision tree $T_n$ such that $T_n(x) = y$ if and only if $(x, y) \in R_n$. Likewise, we define FNP$^{dt}$ as the set of query search problems $R = (R_n)_{n \in \mathbb{N}}$ for which there exists a polylogarithmic depth decision tree family $\{T_y\}_{y \in \{0,1\}^m}$ such that $T_y(x) = 1$ if and only if $(x, y) \in R_n$.

Like protocols, in query search problems the certificate is the verifying decision tree itself. Decision tree reductions are based on a more fine-grained definition, where the function that maps inputs of the first problem to inputs of the second problem is computed by many decision trees with output $\{0, 1\}$.

**Definition 24.** A query search problem $R = (R_m)_{m \in \mathbb{N}}$, where $R_m \subseteq \{0,1\}^m \times O_m$ is said to be many-to-one reducible to a query search problem $S = (S_n)_{n \in \mathbb{N}}$, where $S_n \subseteq \{0,1\}^n \times O'_n$, if for all $m \in \mathbb{N}$ there is an $n \in \mathbb{N}$ for which there is sequence $T = (T_i)_{i \in [n]}$ of decision trees $T_i : \{0,1\}^m \to \{0,1\}$ and a decision tree $T^o_y : \{0,1\}^m \to O_m$ for each $y \in O'_n$ such that:

$$\forall x \in \{0,1\}^m \ \ (T(x), y) \in S \implies (x, T^o_y(x)) \in R$$

where $T(x) := (T_1(x), \ldots, T_n(x))$.

The difference in notation between $T_1, \ldots, T_n$ and $T_y^o$ underlines the fact that the former return a $\{0, 1\}$ output, while the latter returns an output in $O_n$. The *size* of the reduction is the number of input bits to $S$, that being $n$. The *depth $d$* of the reduction is the maximum depth of any tree involved in the reduction, meaning that

$$d = \max(\{\text{depth}(T_i) : i \in [n]\} \cup \{\text{depth}(T_y^o) : o \in O_m\})$$

The complexity of a reduction from $R$ to $S$, written as $S(R)$, is equal to the sum of the log of the size and the minimal depth of a decision tree reduction from $R$ to $S$, that is $S(R) = \log(m) + d$.

**Definition 25.** Given $S \in \mathsf{TFNP}^{dt}$, we define the class $S^{dt}$ as the subset of $\mathsf{TFNP}^{dt}$ problems efficiently reducible through decision trees to the problem $S$, formally $S^{dt} = \{R \in \mathsf{TFNP} \mid S(R) = O(\log^k n)\}$

## 3.2 Proof Complexity

Like the white-box model, black-box total search problems can be studied under multiple lenses, such as **proof complexity**. This branch of complexity theory studies the complexity measures needed for a propositional formula to be proved by propositional proof systems, that being any system of rules that can prove the truthfulness of a propositional formula, i.e. a string made of logical operators applied on a set of $n$ variables, such as $F = x_1 \wedge (x_1 \rightarrow \neg x_2 \vee x_3)$.

Any statement can be encoded by propositional formulas, which is either a *tautology* (a statement that is always true), a *satisfiable* formula (a statement that can be true or false based on the assignment) or an *unsatisfiable* formula (a statement that is always false). Proving that a formula $F$ is a tautology is equivalent to proving that $\neg F$ is unsatisfiable.

Proof systems can be viewed as an algorithm that manipulates propositional formulas, producing a new formula. When a formula $G$ is derived by the formula $F$ in the proof system $S$, we write $F \overset{S}{\vdash} G$. Proof systems must be *sound*: if $F \overset{S}{\vdash} G$ then $G$ is a *logical consequence* of $F$, which means that $F \rightarrow G$ is a tautology. In 1979, Cook and Reckhow gave the following formal definition of propositional proof system - often called Cook-Reckhow proof systems.

**Definition 26.** A propositional proof system (or pps) is a polynomial time computable surjective function $f : \Sigma^* \rightarrow \text{TAUT}$, where TAUT is the set of logical tautologies.

Given a formula $F \in \text{TAUT}$ a string $s \in \Sigma^*$ and a proof system $f$, we say that $s$ encodes $F$ for the pps $f$ if it holds that $f(s) = F$. This idea justifies why we want proof systems to be surjective: any true statement must have a valid encoding in the proof system. This property is called *completeness* of the proof system.

The most studied proof system is **Resolution** (or $\mathsf{Res}$). Given a formula $F \in \text{TAUT}$, this proof system can prove that it is a tautology by proving that $\neg F \in \text{UNSAT}$.

A *conjunctive normal form* (CNF) formula $F$ is a conjunction of $m$ *clauses* $C_1, \ldots, C_m$, where $C_i$ is a disjunction of $k_i$ *literals*, that being either a variable defined on $F$ or its negation. For example, the following formula is in conjunctive normal $F = (x_1 \lor x_2 \lor \neg x_3 \lor x_4) \land (x_1 \lor \neg x_2) \land x_3$

Any formula can be expressed as an equivalent CNF formula, making Resolution a *complete* and *sound* proof system. Resolution proofs are based on repeated applications of the following simple rule called the *resolution rule*:

$$\frac{C \lor x \qquad D \lor \neg x}{C \lor D}$$

Given a CNF formula $F = C_1 \land \ldots \land C_m$ and a clause $C$, we have that $F \overset{\text{Res}}{\vdash} C$ if there is a sequence of clauses $D_1, \ldots, D_k$ such that $D_k = C$ and each $D_i$ in the sequence is either an *axiom* of $F$ (meaning that $D_i = C_j$ for some $j$) or is obtained by applying the resolution rule on $D_p$ and $D_q$ for some $p, q < i$. Resolution is able to prove that a CNF formula $\neg F$ is unsatisfiable by deriving the empty clause $\bot$ starting from the axioms of the formula itself. A Resolution proof is often referred to as a *refutation*. Given the following unsatisfiable CNF formula $(y \lor z) \land (y \lor \neg z) \land (x \lor \neg y \lor z) \land (\neg x \lor z) \land \neg z$, a Resolution proof is given by:

$$
\begin{array}{lccl}
D_1: & \neg y \lor z & \text{Axiom} \\
D_2: & y \lor z & \text{Axiom} \\
D_3: & y & \text{Res. on } D_1, D_2 \\
D_4: & x \lor \neg y \lor z & \text{Axiom} \\
D_5: & \neg z & \text{Axiom} \\
D_6: & x \lor \neg y & \text{Res. on } D_4, D_5 \\
D_7: & \neg x \lor \neg y \lor z & \text{Axiom} \\
D_8: & \neg x \lor \neg y & \text{Res on } D_7, D_5 \\
D_9: & x & \text{Res on } D_3, D_6 \\
D_{10}: & \neg x & \text{Res on } D_3, D_8 \\
D_{11}: & \bot & \text{Res on } D_9, D_{10}
\end{array}
$$

Each refutation can also be graphically represented by connecting clauses with lines:
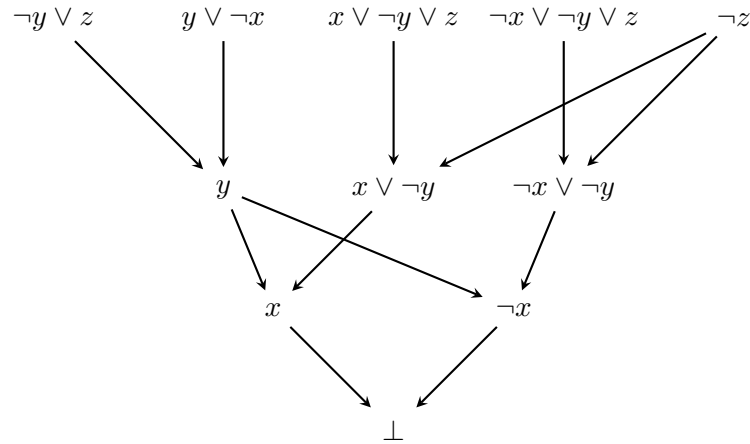


**Figure 7.** Dag-like refutation of the previous formula.

Through this representation, each refutation produces a directed acyclic graph (DAG), also known as dag-like refutation. When each clause - thus excluding the axioms - appears only once in a refutation, we say that it is a **tree-like** refutation due to how the clauses form a tree. This restriction implies that we cannot "store" intermediate results. Any dag-like refutation can be converted into a tree-like refutation by derive multiple times each clause that we have to reuse.
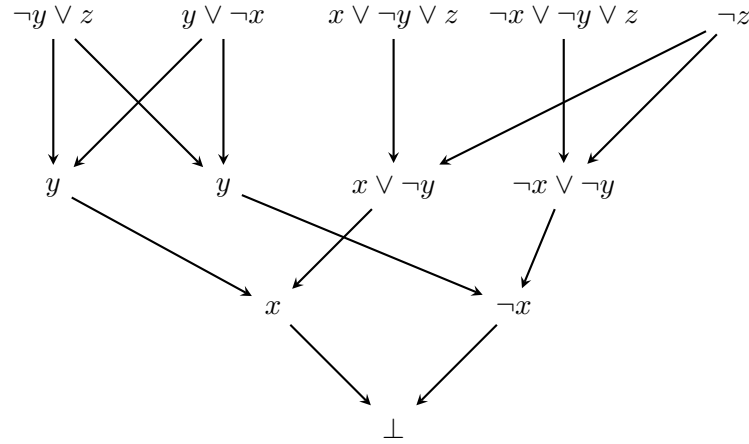


**Figure 8.** Tree-like refutation of the previous formula.

This subset of proofs defines a more specific proof system called *Tree-like Resolution* (or TreeRes). Generally, this type of refutation produces a proof of exponential length compared to the number of variables defined on the formula itself.Resolution and Tree-like Resolution are separated, meaning that some proofs are easy for the former and hard for the latter, making Resolution a stronger proof system.

Resolution has three main complexity measures: size, depth and width. The *size* of a Resolution proof is the total number of nodes appearing in the proof. The *depth* of a Resolution proof is the length of the longest path from an axiom to the empty clause. The *width* of a Resolution proof is the maximum number of literals appearing in a clause of the proof. For example, the proof shown in Figure 8 has size 9, depth 3 and width 3. These three complexity measures are highly related. For example, if a Tree-like Resolution proof has depth $d$ the size of such proof is $O(2^{d+1})$ since a $d$-depth tree can have at most $2^{d+1}$ nodes.

But why are we interested in proving or refusing propositional formulas? We discussed how the SAT problem is NP-Complete. This clearly implies that the problem $\overline{SAT}$ is coNP-Complete. This fact can be used to show that $\overline{SAT} \leq_m$ UNSAT $\leq_m$ TAUT, implying that both UNSAT and TAUT are also coNP-Complete. Showing that any of these problems is also in NP would answer the NP $\overset{?}{=}$ coNP question.

Proof systems are essential to work on this question: given the encoding $\Pi$ of a proof of $F$ in a proof system, a verifier can follow the rules defined by the proof system to prove that $F$ is indeed a tautology. In this case, $\Pi$ serves as a certificate for $F$ while the pps defines the verifier. We give the following equivalent definition of a propositional proof system.

**Definition 27.** A propositional proof system (or pps) is a polynomial verifier $V$ such that $F \in \mathrm{TAUT}$ if and only if there is a string $\Pi \in \Sigma^*$ such that $V(F, \Pi) = 1$.

At first glance, one could think that this definition implies that any complete and sound pps proves that $\mathrm{TAUT} \in \mathsf{NP}$. However, we must also consider the length of such proofs: to be an efficient verifier, the length of the certificates must be polynomially bounded by the length of $F$. In other words, it must hold that $|\Pi| = O(|F|^k)$ for some $k \in \mathbb{N}$. This means that in order to prove that $\mathsf{NP} \neq \mathsf{coNP}$, or equivalently that $\mathrm{TAUT} \in \mathsf{NP}$, we must find a **polynomially bounded proof system**, a pps that uses only polynomially bounded proofs for all tautologies.

**Proposition 3.** *There is a polynomially bounded proof system if and only if* $\mathsf{NP} \neq \mathsf{coNP}$.

We already discussed how researchers believe that $\mathsf{NP} \neq \mathsf{coNP}$ is the expected answer to the conjecture. Proving this statement is no easy task: we would have to prove that there is a particular formula $F$ that strictly requires an exponential length encoding for every discovered and undiscovered proof system.

## 3.3 The Black-box model and Proof complexity

Proof complexity is highly related to other branches of complexity theory, including the study of total search problems. In order to get to this relation between proof complexity and $\mathsf{TFNP}^{dt}$, we have to restrict our focus to CNF formulas. By construction, a CNF formula can be unsatisfiable if and only if for all assignments $\alpha(x_1, \ldots, x_n)$ there is a clause $C_i$ that is false. It's easy to see this fact implies that any CNF formula gives rise to an associated search problem: finding a falsified clause inside the formula (if there is any) for each possible assignment.

**Definition 28.** Given a CNF $F = C_1 \wedge \ldots \wedge C_m$ over $n$ variables, we define $\mathrm{Search}(F)$ as the following search problem: given an input assignment $\alpha(x_1, \ldots, x_n)$, return an index $i$ such that the assignment falsifies $C_i$.

This problem is usually referred to as the **false clause search problem**. When $F$ is an unsatisfiable CNF formula, $\mathrm{Search}(F)$ is a total search problem since for any input assignment there will always be an unsatisfied clause. Moreover, the search problem of any unsatisfiable formula can easily be solved (or verified) by a decision tree for any formula $F$: if the assignment $\alpha(x_1 = b_1, \ldots, x_n = b_2)$ falsifies the clause $C_i$, define a path $x_1 = b_1, \ldots, x_n = b_n$ on the decision tree and let $C_i$ be the output of such path. In other words, for all $\neg F \in \mathrm{TAUT}$ it holds that $\mathrm{Search}(F) \in \mathsf{TFNP}^{dt}$.
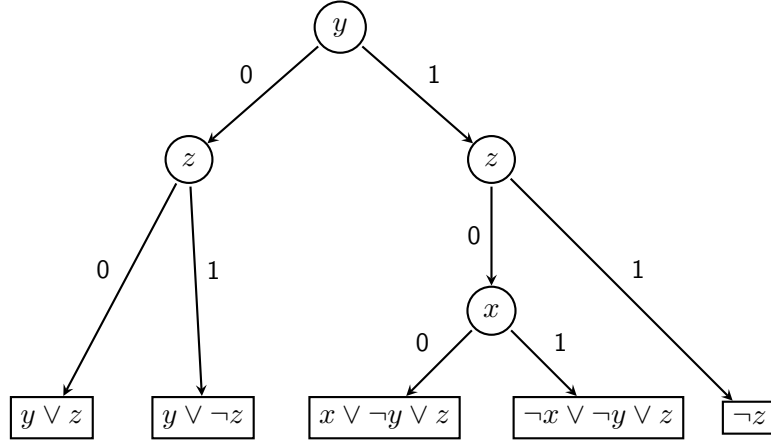
**Figure 9.** Decision tree for the previous unsatisfiable formula.

Similarly, we can show that any total query search problem $R$ can be associated with the search problem of the formula $F$ that describes the set of decision trees that verify $R$. Consider a decision tree $T$ made of the paths $p_1, \ldots, p_k$, each leading to the leaves $\ell_1, \ldots, \ell_k$. The DNF encoding of $T$, written as $D(T)$, is the disjunction over the conjunction of the literals $\alpha_1, \ldots, \alpha_h$ along each of the accepting paths in $T$. In other words, we have that $D_T = p_1 \vee \ldots \vee p_k$ where each $p_i = \alpha_1 \wedge \ldots \wedge \alpha_h \wedge \ell_i$ is an accepting path of $T$. By De Morgan's theorem, $\neg D(T)$ is a CNF.

**Proposition 4.** *Given a total query search problem $R \subseteq \{0,1\}^n \times O$, for each $n \in \mathbb{N}$ there exists an unsatisfiable CNF formula $F_n$ defined over $|O|$-many variables such that $R_n = \mathrm{Search}(F_n)$. This formula is called canonical CNF encoding of $R_n$.*

*Proof.* Since $R = (R_n)_{n \in \mathbb{N}} \in \mathsf{TFNP}^{dt}$, for each $y \in O_n$ there is a polylog($n$)-depth decision tree $T_y$ that verifies $R_n$. Consider the CNF $F_n := \bigwedge_{y \in O_n} \neg D(T_y)$. Since $R$ is a total search problem, for each input $x$ there is a valid output, implying that at least one tree $T_y$ will have an accepting path, meaning that $D(T_y)$ with input $x$ accepts and therefore $\neg D(T_y)$ with input $x$ rejects, concluding that $F_n$ is unsatisfiable. Moreover, this formulation also concludes that:

$$(x, y) \in R_n \iff (x, y) \in \mathrm{Search}(F_n)$$

and thus that $R_n = \mathrm{Search}(F_n)$.

$\square$

This result clearly implies that $(R)_{n \in \mathbb{N}} = (\mathrm{Search}(F_n))_{n \in \mathbb{N}}$, where $F_1, F_2, \ldots$ is a family of CNF formulas, and by extension that black-box $\mathsf{TFNP}$ is exactly the study of the false clause search problem. Like in the communication $\mathsf{TFNP}$ case, the upshot is that it is sufficient to restrict our interests to the study of search problems associated with unsatisfiable CNF formulas.

Furthermore, we also notice that the width of each CNF encoding of a given search problem in $\mathsf{TFNP}^{dt}$ corresponds to the maximum depth of the decision trees that solve or verify it.

Through this connection, Göös et al. [GKR+19] showed that many important proof systems are characterized by an associated $\mathsf{TFNP}^{dt}$ search problem and vice versa. Given a proof system $P$ and an unsatisfiable CNF formula $F$, the **complexity** required by $P$ to prove $F$ is given by:

$$P(F) := \min\{\log \operatorname{size}(\Pi) + \deg(\Pi) : \Pi \text{ is a } P\text{-proof of } F\}$$

where $\operatorname{size}(\Pi)$ is the the total number of symbols in $\Pi$ and $\deg(\Pi)$ is the *degree* of $\Pi$ associated to $P$, which varies from proof system to proof system. For example, in Tree-like Resolution the degree is the *depth* of the proof, while in Resolution the degree is the *width* of the proof. This degree measure will be specified for the proof systems analyzed in the following sections.

To make things more readable, we will refer to Search($F$) as $S_F$. Since each $\mathsf{TFNP}^{dt}$ problem is equivalent to $S_F$ for some formula $F$, the complexity parameter defined above can be used to give another **characterization** of $\mathsf{TFNP}^{dt}$ problems.

**Definition 29.** We say that a proof system $P$ characterizes a $\mathsf{TFNP}^{dt}$ problem $R$ (and reflexively that $R$ characterizes $P$) if it holds that

$$R^{dt} = \{S_F : P(F) = \operatorname{polylog}(n)\}$$

where $F = (F_i)_{i \in \mathbb{N}}$ is a family of formulas. In a stronger sense, it must hold that $R^{dt}(S_F) = \Theta(P(F))$.

Again, this implies that the encoding CNF formulas must have a polylogarithmic width. Most of the $\mathsf{TFNP}$ subclasses discussed in previous sections have been shown to have a characterizing proof system:

- $\mathsf{FP}^{dt}(S_F) = \Theta(\mathsf{TreeRes}(F))$ [LNN+95]

- $\mathsf{PLS}^{dt}(S_F) = \Theta(\mathsf{Res}(F))$ [BKT14]

- $\mathsf{PPA}^{dt}(S_F) = \Theta(\mathbb{F}_2\text{-}\mathsf{NS}(F))$ [GKR+19]

- $\mathsf{PPADS}^{dt}(S_F) = \Theta(\mathsf{unary\text{-}NS}(F))$ [GHJ+22b]

- $\mathsf{PPAD}^{dt}(S_F) = \Theta(\mathsf{unary\text{-}SA}(F))$ [GHJ+22b]

- $\mathsf{SOPL}^{dt}(S_F) = \Theta(\mathsf{RevRes}(F))$ [GHJ+22b]

- $\mathsf{CLS}^{dt}(S_F) = \Theta(\mathsf{RevResT}(F))$ [GHJ+22b]

$$
\begin{array}{c}
\boxed{\mathsf{TFNP}^{dt}}
\end{array}
$$

$$
\mathsf{Res} = \boxed{\mathsf{PLS}^{dt}} \qquad \boxed{\mathsf{PPP}^{dt}} \qquad \boxed{\mathsf{PPA}^{dt}} = \mathbb{F}_2\text{-NS}
$$

$$
\text{unary-SA} = \boxed{\mathsf{PPADS}^{dt}}
$$

$$
\mathsf{RevRes} = \boxed{\mathsf{SOPL}^{dt}} \qquad \boxed{\mathsf{PPAD}^{dt}} = \text{unary-NS}
$$

$$
\mathsf{RevResT} = \boxed{\mathsf{CLS}^{dt}}
$$

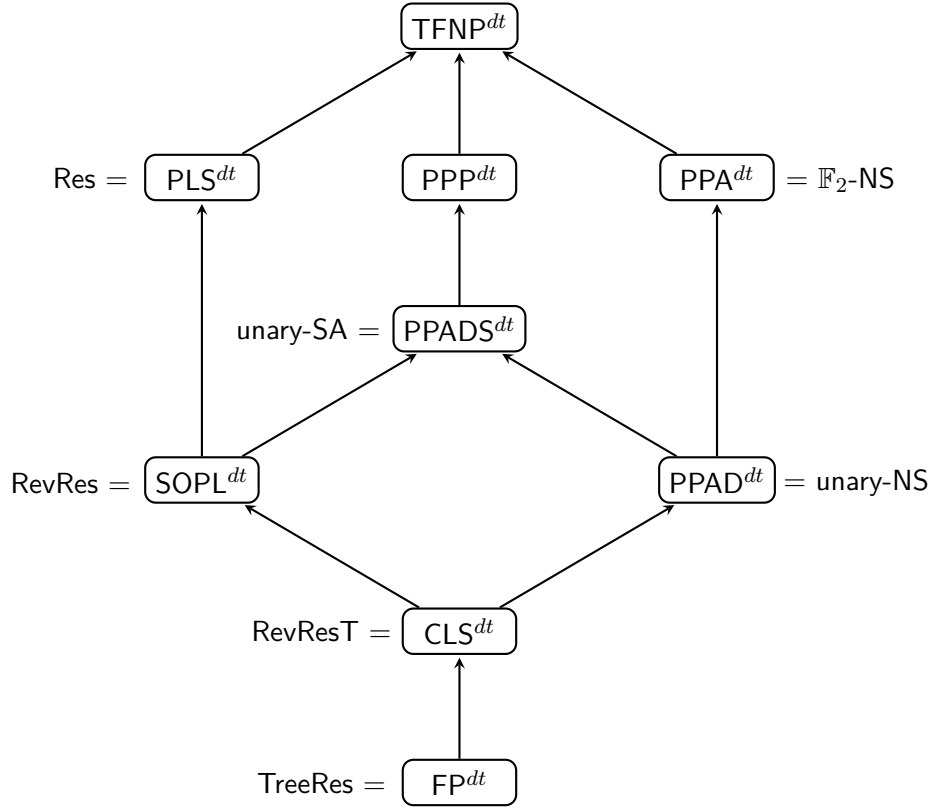$$
\mathsf{TreeRes} = \boxed{\mathsf{FP}^{dt}}
$$

**Figure 10.** Black-box TFNP classes and their characterizing proof systems.

## 3.4   Reductions through CNF formulas

Intuitively, the characterization given in the previous section shows that any $\mathsf{TFNP}^{dt}$ problem can be transformed into a proof system for refuting unsatisfiable CNF formulas of polylogarithmic width: since any $\mathsf{TFNP}^{dt}$ is equivalent to the search problem for some unsatisfiable CNF formula, any efficient decision tree reduction between problems is nothing more than an efficient proof in the characterizing proof system and vice versa. To formalize this idea, we introduce the concept of **reductions between CNF formulas** [BFI23].

Suppose that $C$ is a clause over $n$ variables and that $T = (T_i)_{i \in [n]}$ is a sequence of depth-$d$ decision trees, where $T_i : \{0,1\}^m \to \{0,1\}$. We refer to $C(T)$ as the CNF formula obtained by substituting each variable $x_i$ in $C$ with $D(T_i)$ and rewriting the result as a CNF, or more conveniently:

$$
C(T) := \bigwedge_{i \in [n]} \bigwedge_{\substack{r : \text{rejecting} \\ \text{path of } T_i}} \neg r
$$

**Definition 30.** Let $F = C_1 \wedge \ldots \wedge C_{m_F}$ be an unsatisfiable CNF over $n_F$ variables. We say that a CNF formula $H$ made of $m_H$ clauses over $n_H$ variables reduces to $F$ via depth-$d$ decision trees if there exist two sequences of depth-$d$ decision trees $T = (T_i)_{i \in [n_F]}$ and $T^o = (T_j^o)_{j \in [m_F]}$, where $T_i : \{0,1\}^{n_H} \to \{0,1\}$ and $T_j^o : \{0,1\}^{n_H} \to [m_H]$, such that given the following formula:

$$F_H := \bigwedge_{j \in [m_F]} \bigwedge_{\substack{p \,:\, \text{path} \\ \text{in } T_j^o}} C_i(T) \vee \neg p$$

it holds that if $F$ is unsatisfiable then $F_H$ is unsatisfiable and by consequence that $H$ is unsatisfiable.

In particular, we notice that $F_H$ can also be written as a CNF by simply distributing each $\neg p$ inside $C_i(T)$. Each clause $C_i(T) \vee \neg p$ must be either tautological (since it could contain a variable and its negation) or a weakening of the corresponding clause of $H$ - meaning that it is a formula $Q$ such that $H \to Q$ - indexed by the label at the end of the path $p$. Moreover, we notice that through this formulation any depth-$d$ decision tree reduction from $S_H$ to $S_F$ induces the search problem $S_{F_H}$. By construction, reductions between CNF formulas are just a formal way to say that reductions between search problems reduction are actually proof systems.

**Definition 31.** Given a problem $S_F \in \mathsf{TFNP}^{dt}$ the **canonical proof system** of such problem, written as $P_F$, is a proof system that refutes an unsatisfiable formula $H$ over $n_H$ variables if $H$ is reducible to an instance of $F$ over $n_F$ variables.

A $P_F$-proof of $H$ consists of the decision trees that make such reduction possible. The *size* of such proof is given by $n_F$, while the *degree* is given by the maximum depth among the involved decision trees. Hence, the $P_F$ complexity of $H$ is given by:

$$P_F(H) := \min\{\log \text{size}(\Pi) + \text{depth}(\Pi) : \Pi \text{ is a } P_F\text{-proof of } H\}$$

This definition directly implies that given $S_F \in \mathsf{TFNP}^{dt}$, the **characterizing proof system** of $S_F^{dt}$ is equivalent to the canonical proof system $P_F$. Canonical proof systems are *sound*, since by construction any valid substitution of an unsatisfiable CNF formula is also unsatisfiable, and also *efficiently verifiable*, since it suffices to check that each of the clauses of $F_H$ is either tautological or a weakening of a clause in $H$, which can both be done in polynomial time compared to the size of the proof.

The following theorem plays a crucial role in $\mathsf{TFNP}^{dt}$ characterization through proof complexity, stating that the proof system $P_F$ has a short proof of $H$ if and only if $S_H$ efficiently reduces to $S_F$. In other words, an efficient proof of a formula in a characterizing proof system automatically gives an efficient reduction to the corresponding complete search problem.

**Theorem 5.** *Let $S_F \in \mathsf{TFNP}^{dt}$ and let $H$ be an unsatisfiable CNF formula. The two following results hold:*

1. *If $H$ has a size $s$ and depth $d$ proof in $P_F$ then $S_H$ has a size $O(s)$ and depth $d$ reduction to $S_F$*

2. *If $S_H$ has a size $s$ and depth $d$ decision tree reduction to $S_F$ then $H$ has a size $s2^{O(d)}$ and depth $d$ proof in $P_F$*

*In particular, this implies that $S_F^{dt}(S_H) = \Theta(P_F(H))$.*

*Proof.* Suppose that $T = (T_i)_{i \in [n_F]}$ and $T' = (T_j^o)_{j \in [m_F]}$ is a $P_F$ proof of $H$ of size $s$ and depth $d$. Given any assignment $\alpha$ such that $(\alpha, i) \in S_F$, let $C_i$ be the clause of $F$ falsified by $T_1(\alpha), \ldots, T_{n_F}(\alpha)$ and let $p$ be the path followed by $T_i^o(\alpha)$. It's easy to see that a clause of the formula $C_i(T) \vee \neg p$ must be falsified by $\alpha$. In particular, such clause is also the weakening of the $T_i^o(\alpha)$-th clause of $H$, concluding that $(\alpha, T_i^o(\alpha)) \in S_H$. In other words, the $P_F$ proof of $H$ corresponds to a reduction from $S_H$ to $S_F$ of size $n_F = O(s)$ and depth $d$.

Vice versa, suppose that $T = (T_i)_{i \in [n_F]}$ and $T' = (T_j^o)_{j \in [m_F]}$ is a decision tree reduction from $S_H$ to $S_F$ of size $s$ and depth $d$. Then, we can construct $F_H$ as previously described through the use of these decision trees. Let $L$ be a clause of $C_i(T)$ for some $i \in [m_F]$ and let $p$ be any path in $T_i^o$. If the formula $C_i(T) \vee \neg p$ is tautological, then it can be ignored since $F_H$ is a CNF. Otherwise, let $\alpha$ be an assignment that falsifies $L \vee \neg p$. Then, it holds that $T_1(\alpha), \ldots, T_{n_F}(\alpha)$ falsifies $C_i(T)$ and that $T_i^o(\alpha)$ follows path $p$. Thus, the $T_i(\alpha)$-th clause of $\neg H$ must also be false, implying that $L \vee \neg p$ is a weakening of such clause. This concludes that $F_H$ is a $P_F$-proof of $H$ of depth at most $d$ (due to how $F_H$ is constructed) and thus that the size is at most $s2^{O(d)}$. $\square$

# Chapter 4

# Parity in black-box **TFNP**

## 4.1 Parity decision trees

The concept of parity is extensively studied in computer science. In our case, we are interested in exploring parity through the lens of *linear forms modulo 2*, i.e. linear combinations defined on $n$ variables over the algebraic field $\mathbb{F}_2$. In this field, each term can either be a 0 or a 1, with the defining characteristic that $1 + 1 = 0$.

**Definition 32.** Given $n$ variables $x_1, \ldots, x_n$, we define a linear form as a linear combination over $\mathbb{F}_2$. In general, a linear form can be expressed as $\sum_{i=1}^{n} \alpha_i x_i$, where $\alpha_1, \ldots, \alpha_n \in \mathbb{F}_2$

Intuitively, each sum in a linear form is nothing more than an application of the **XOR operator**: the linear form $x_1 + x_2$ is equal to 1 if and only if $x_1$ is *different* from $x_2$ (i.e. if $x_1 = 1$ and $x_2 = 0$ or if $x_1 = 0$ and $x_2 = 1$). Additionally, in $\mathbb{F}_2$ the concepts of addition and subtraction are equivalent: since $1 + 1 = 0$, we easily get that $1 = -1$. Through these properties, parity can be used to determine if two or more objects are equal or not. For example, consider the following system of linear forms:

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

By simplifying the linear system we get that:

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

which tells us that $x_2 = 1$ and $x_1 = x_4 \neq x_3$ must hold.

But what happens if we apply the concept of parity in decision trees? What if, instead of querying variables to know their value, we ask the parity of a set of values by querying linear forms? This idea gives rise to the extended model of **parity decision trees**.

Instead of being labeled by single variables, the nodes of a parity decision tree (PDT for short) are labeled by a linear form $f$. Each node has two outgoing edges, one labeled by $f = 0$ and the other by $f = 1$. Every path from the root of the PDT to one of its nodes defines a system of linear forms given by all the labels of the edges on the path. In general, given the PDT $T$ and a node $v$, we denote this system with $\Phi_v^T$. Given an assignment $\alpha(x_1, \ldots, x_n)$, the output of a PDT is dictated by the parity queries made by each node.
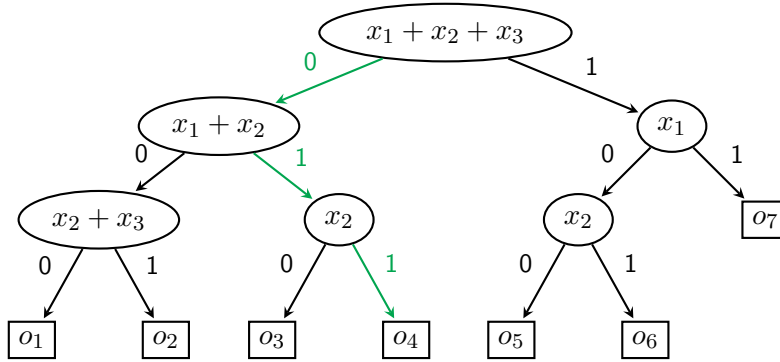


**Figure 11.** An example of a parity decision tree of size 13 and depth 3.

In the above example, the green path defines the following system of linear forms:

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

which once simplified corresponds to the assignment $x_0 = 0, x_2 = 1, x_3 = 1$. We define the class $\mathsf{FP}^{pdt}$ as the set of $\mathsf{TFNP}^{dt}$ problems that are efficiently solvable by a PDT, where the complexity measures are defined as in normal decision trees.

**Definition 33.** We define $\mathsf{FP}^{pdt}$ as the set of query search problems $R = (R_n)_{n \in \mathbb{N}}$ for which there exists a polylogarithmic depth PDT $T_n$ such that $T_n(x) = y$ if and only if $(x, y) \in R_n$.

It's easy to see that $\mathsf{FP}^{dt} \subseteq \mathsf{FP}^{pdt}$ since any decision tree is just a PDT with all the queries defined only on one variable. Any PDT can be converted into a normal decision tree simply by "splitting" each linear query. Given a node $v$ labeled with the linear form $f + x_i$, let $u$ and $w$ be the children of $v$ respectively given by $f + x_i = 0$ and $f + x_i = 1$. Let $T_u$ and $T_w$ be the two subtrees with root $u$ and $w$.
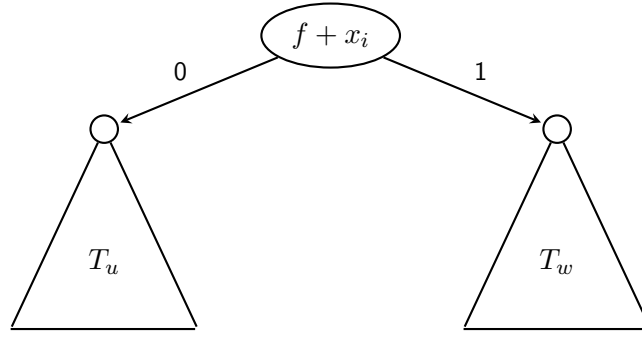
**Figure 12.** The initial subtree of a parity decision tree.

We replace $v$ with the node $v'$ labeled with the linear form $x_i$ and introduce two new nodes $u', w'$ such that $u'$ is the child of $v'$ when $x_i = 0$ and $w'$ is the child of $v'$ when $x_i = 1$. We label $u'$ with the linear form $f$ and let a copy of $T_u$ be the children of $u'$ when $f = 0$, while a copy of $T_w$ is the children of $u'$ when $f = 1$. Symmetrically, we label $w'$ with the linear form $f$ and let a copy of $T_w$ be the children of $w'$ when $f = 0$, while a copy of $T_u$ is the children of $w'$ when $f = 1$.
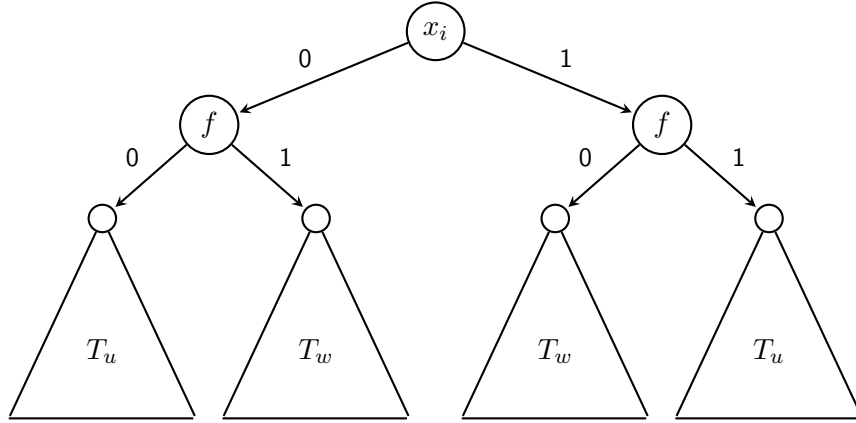


**Figure 13.** The subtree after the splitting process.

By repeating this process until all queries are defined on a single variable, we obtain a decision tree equivalent to the original PDT. This final decision tree has exponential size and polynomial depth, which *may not* be the smallest possible decision tree that solves the search problem solved by the initial PDT. However, we can easily prove that parity decision trees are indeed much stronger than decision trees.

**Theorem 6.** $\mathsf{FP}^{dt} \subsetneq \mathsf{FP}^{pdt}$

*Proof.* Any decision tree is also a parity decision tree, thus we trivially get that $\mathsf{FP}^{dt} \subseteq \mathsf{FP}^{pdt}$. Let PARITY be the *n-bit parity search problem*, i.e. the problem of determining the parity of $n$ variables for a given assignment $\alpha$. This problem can be solved by a PDT of size and depth $O(1)$ by making a single query $x_1 + \ldots + x_n$, concluding that PARITY $\in \mathsf{FP}^{dt}$.

**Claim.** Any decision tree solving PARITY on $n$ variables has depth $\Omega(n)$.

*Proof of the claim.* Suppose that PARITY on $n$ variables can be solved by a decision tree $T$ with depth less than $n$. We use an adversarial argument: we think of an execution of $T$, where some adversary answers each query of $T$ on the value of every input bit. The adversary can respond with $x_i = 0$ on all the first $n-1$ bits. Until the last bit $x_n$ is revealed, the tree has no way of determining the final output since it can either be 0 or 1 until the value of $x_n$ is revealed, concluding that it requires at least another query. □

By definition, $\mathsf{FP}^{dt}$ contains all the problems with a decision tree of polylogarithmic depth. Since PARITY requires a decision tree with depth $\Theta(n)$, we get that PARITY $\notin \mathsf{FP}^{dt}$. □

Since a system of linear forms can have multiple solutions, many assignments could be mapped to the same output. However, some systems could also be unsatisfiable, meaning that the node is unreachable by any assignment. When this happens we say that the node is **degenerate**.

Like normal decision trees, PDTs can be used to solve the false clause search problem associated with any unsatisfiable CNF. A parity decision tree for a CNF formula $F$ is a PDT defined on the same variables of $F$ where for each leaf $v$ one of the following conditions holds:

1. The leaf is *degenerate*

2. The leaf *refutes* a clause $C$ of $F$, meaning that the system $\Phi_v^T$ is satisfiable and every one of its solutions falsifies $C$

3. The leaf *satisfies* a clause $C$ of $F$, meaning that the system $\Phi_v^T$ has only one solution and it also satisfies $C$
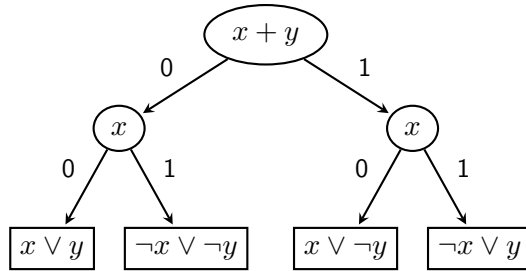


**Figure 14.** A parity decision tree for $(x \vee y) \wedge (\neg x \vee \neg y) \wedge (\neg x \vee y) \wedge (x \vee \neg y)$.

We observe that if a node doesn't meet any of the previous conditions then it cannot be a leaf node. Moreover, we also observe that the system associated with the root of any PDT is always satisfiable due to it containing no linear forms. Since we are interested in studying PDTs for refusing unsatisfiable CNF formulas, the third case will never be true for any leaf. However, we still need a way to exclude the first case since an unsatisfiable system cannot be associated with any assignment. Luckily, each degenerate PDT can be conveniently converted into a non-degenerate one through a very simple process [IS20].

**Proposition 5.** *Let $F$ be an unsatisfiable CNF formula. If $S_F$ can be solved with a degenerate PDT of size $s$ and depth $d$, it can also be solved with a non-degenerate PDT of size at most $s$ and depth at most $d$.*

*Proof.* Let $T$ be a degenerate PDT of size $s$ and depth $d$ that solves $S_F$. Let $U$ be the set of degenerate nodes of $T$. Notice that since $\Phi_r^T$ is empty, thus always satisfiable, we know that $r \notin U$. Consider the node $u \in U$ with the minimal distance from the root $r$. Since $u$ is not the root of $T$, there must be two vertices $p$ and $s$ such that $p$ is the parent of $u$ and $s$ is the sibling of $u$.

We notice that $\Phi_s^T$ must be satisfiable: if we assume that this is not true then both $\Phi_s^T$ and $\Phi_u^T$ would be unsatisfiable, which can be true only if $\Phi_p^T$ is also unsatisfiable, but we chose $w$ as the node in $U$ with minimal distance. Since $\Phi_s^T$ is satisfiable, the label $f = \alpha$ on the edge $(p, s)$ must be already implied $\Phi_p^T$, meaning that each assignment that satisfies $\Phi_p^T$ also satisfies $\Phi_s^T$.

We construct a new PDT $T'$ by removing the subtree $T_u$ with root $u$ from the initial PDT $T$ and by contracting the edge $(p, s)$, merging the two nodes $p$ and $s$ into a single node $v$. In other words, the subtree $T_u$ gets removed and the children of $s$ become the new children of $p$. Each assignment that satisfies $\Phi_p^T$ also satisfies $\Phi_s^{T'}$, concluding that $T'$ also solves $S_F$. By repeating the process until $U$ is empty, we get a non-degenerate PDT that solves $S_F$ of size at most $s$ and depth at most $d$. $\qquad\square$

## 4.2   Linear Resolution over $\mathbb{F}_2$

Once we have defined the class $\mathsf{FP}^{pdt}$, we are interested in finding a proof system that characterizes it. Consider a system $\Phi$ of linear forms defined on $\mathbb{F}_2$. This system can be viewed as the conjunction of the linear forms that it describes:

$$
\begin{cases}
f_1 = \alpha_1 \\
f_2 = \alpha_2 \\
\quad\vdots \\
f_k = \alpha_k
\end{cases}
\iff (f_1 = \alpha_1) \wedge (f_2 = \alpha_2) \wedge \ldots \wedge (f_k = \alpha_k)
$$

We can rewrite these conjunctions as a negation of a disjunction:

$$
\bigwedge_{i=1}^k (f_i = \alpha_i) \iff \neg \bigvee_{i=1}^k \neg(f_i = \alpha_i) \iff \neg \bigvee_{i=1}^k (f_i = 1 + \alpha_i)
$$

which implies that the negation of the system is equivalent to a set of disjunctions:

$$
\neg \bigwedge_{i=0}^k (f_1 = \alpha_1) \iff \bigvee_{i=0}^k (f_1 = 1 + \alpha_1)
$$

We define such a set of disjunctions as a **linear clause**. More generally, a *linear CNF formula* over $\mathbb{F}_n$ is a conjunction of linear clauses defined on $\mathbb{F}_n$.

**Definition 34.** A linear CNF formula is a conjunction of $m$ disjunctions of linear equations over $\mathbb{F}_n$.

$$\bigwedge_{i=1}^{m} \bigvee_{j=1}^{k_i} (f_j = \alpha_j)$$

Linear CNF formulas can assume a complex structure such as the following:

$$((x_1 + x_2 = 0) \vee (x_1 = 1)) \wedge ((x_2 + x_3 + x_4 = 3) \vee (x_2 + x_4 = 0))$$

We define **Linear Resolution over** $\mathbb{F}_n$ (or $\mathsf{ResLin}(\mathbb{F_n})$), an extension of standard Resolution (see Chapter 3) based on the following two rules:

1. *Resolution rule*: given two linear clauses $(f = 0) \vee C$ and $(f = 1) \vee D$ defined on $\mathbb{F}_n$, we can derive the linear clause $C \vee D$

2. *Weakening rule*: given a linear clause $C$, we can derive any linear clause $D$ such that $C \implies D$.

Like in normal Resolution, in $\mathsf{ResLin}(\mathbb{F_n})$ any derivation of a linear clause $C$ from a linear CNF $F$ is a sequence of linear clauses that ends with $C$, where every clause is either an axiom of $F$ or it can be derived from previous clauses through one of the two derivation rules. A linear CNF is unsatisfiable if and only if the empty linear clause can be derived from it.

Any standard CNF formula can be described as a linear CNF formula over $\mathbb{F}_2$ simply by treating each clause as a disjunction of linear forms made of a single term. For example, the CNF $(x_1 \vee \neg x_2) \wedge (\neg x_3 + x_1)$ can be written as the following linear CNF formula:

$$((x_1 = 1) \vee (x_2 = 0)) \wedge ((x_3 = 0) \vee (x_1 = 1))$$

We call this the *linear encoding* of a CNF. From now on, we will restrict our interests to Linear Resolution over $\mathbb{F}_2$, also called *parity Resolution* (or $\mathsf{Res}_\oplus$).

The weakening rule makes this proof system powerful thanks to how semantical implications can be used as "shortcuts". For example, consider the following linear CNF:

$$(x = 1) \wedge (x + y = 1) \wedge ((x = 0) \vee (y = 1))$$

By rewriting the last linear clause as a negation of a conjunction, we notice that:

$$(x = 0) \vee (y = 1) \equiv \neg((x = 1) \wedge (y = 0))$$

By simple substitution, we get that:

$$\neg((x = 1) \wedge (y = 0)) \implies \neg((x = 1) \wedge (x + y = 1))$$

which is equivalent to:

$$\neg((x = 1) \wedge (x + y = 1)) \equiv (x = 0) \vee (x + y = 0)$$

concluding that $(x = 0) \vee (y = 1) \models (x = 0) \vee (x + y = 0)$. Proceeding with the resolution rule, we get the following Tree-like refutation.
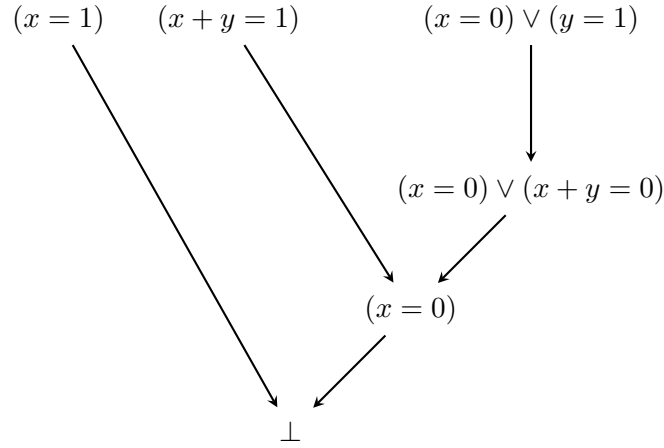
$$(x = 1) \qquad (x + y = 1) \qquad (x = 0) \vee (y = 1)$$

$$(x = 0) \vee (x + y = 0)$$

$$(x = 0)$$

$$\perp$$

**Figure 15.** TreeRes$_\oplus$-proof of the previous linear CNF formula.

It was shown that the weakening rule can be simulated through these simple three rules [IS20]:

1. *Simplification rule*: given a linear clause $C \vee (0 = 1)$, we can derive the linear clause $C$

2. *Syntactic weakening*: given a linear clause $C$, we can derive the linear clause $C \vee (f = \alpha)$

3. *Addition rule*: given a linear clause $C \vee (f = \alpha) \vee (g = \beta)$, we can derive the linear clause $C \vee (f = \alpha) \vee (g = \beta)$

**Proposition 6.** *Any clause obtainable through the weakening rule can also be obtained through a sequence of applications of the previous three rules and vice versa.*

This result makes working with the weakening rule easier: any clause $D$ derived through $k$ applications of these three rules starting from a clause $C$ is automatically a weakening of $C$, implying that we can replace those $k$ applications with one single use of the weakening rule.

## 4.3 Characterization of FP$^{pdt}$ through TreeRes$_\oplus$

TreeRes$_\oplus$ proofs and parity decision trees can be viewed as two sides of the same coin. Any tree-like Res$_\oplus$ refutation of a linear CNF $F$ can be used to construct an (almost) equivalent PDT that solves $S_F$ and vice versa [IS20].

**Lemma 2.** *Let $F$ be a linear CNF formula. If there is a TreeRes$_\oplus$ refutation of $F$ with size $s$ and depth $d$, there also is a PDT of size at most $s$ and depth at most $d$ that solves $S_F$.*

*Proof.* Let $T$ be the proof tree that refutes $F$. We label each edge of $T$ whose associated clauses involve a resolution rule, while all the other weakening edges remain unlabeled. In particular, if a resolution rule is applied to the clauses

$(f = 0) \lor D_1$ and $(f = 1) \lor D_2$ obtaining the clause $D_1 \lor D_2$, we label the edge from the first to the third with $f = 1$, while the other edge is labeled with $f = 0$.

By induction on the depth of a node of $T$, we show that the clause written in $v$ contradicts the system $\Phi_v^T$. The root node contains the empty clause and is labeled by an empty system, making the statement trivially true. Assume now that the statement holds for a generic node $v$. We have to show that the hypothesis also holds for its children $u$ and $w$.

Suppose that $v$ is the result of a resolution rule application, where $D_1 \lor D_2$ is the clause inside $v$. Assume that $u$ is the node that contains $(f = 0) \lor D_1$ while $w$ contains $(f = 1) \lor D_2$. By inductive hypothesis, we know that $D_1 \lor D_2$ contradicts the system $\Phi_v^T$. This means that the set of equalities in $D_1$ contradicts $\Phi_v^T$. Moreover, we know that $\Phi_u^T = \Phi_v^T \land (f = 1)$, concluding that $(f = 0) \lor D_1$ contradicts $\Phi_u^T$. Likewise, we can show that $(f = 1) \lor D_2$ contradicts $\Phi_w^T$.

Suppose now that $v$ is the result of a weakening rule, where $u$ is the only child. Since $(v, u)$ is unlabeled, we get that $\Phi_v^T = \Phi_u^T$. Furthermore, since $v$ is the result of a weakening applied to $u$, we know that the clause in $u$ semantically implies the clause in $v$, but by inductive hypothesis we know that the clause in $v$ contradicts the system $\Phi_v^T$, meaning that $u$ must also contradict the system $\Phi_v^T = \Phi_u^T$. Finally, if $v$ is a leaf then the statement is trivially true since it refutes a clause of $F$.

By contracting all the unlabeled edges given by the weakening rules, we get a parity decision tree that solves $S_F$. Due to this final step, the size of the PDT is at most $s$ and its depth is at most $d$. $\qquad\square$
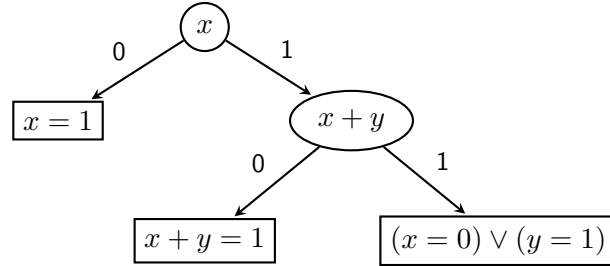


**Figure 16.** The PDT obtained from the proof shown in Figure 15.

**Lemma 3.** *Let $F$ be a linear CNF formula. If there is a PDT of size $s$ and depth $d$ that solves $S_F$, there also is a TreeRes$_\oplus$ refutation of $F$ with size at most $2s$, depth at most $d + 1$ and the weakening rule applied only to the axioms.*

*Proof.* Let $T$ be a PDT of size $s$ and depth $d$ that solved $S_F$. By Proposition 5, we assume that $T$ is non-degenerate. We label every node $v$ of $T$ with the negation of its associated linear system. In other words, every node $v$ is labeled with the linear clause $\neg\Phi_v^T$. Every node is the result of the resolution rule being applied to its children, where the root node is the empty clause.

Since $T$ is a PDT that solves $S_F$, each leaf refutes a linear clause of $F$. Hence, for each leaf $u$ we have that $\Phi_u^T \implies \neg C$ for some linear clause $C$ of $F$, which equivalently means that $C \implies \neg\Phi_u^T$, concluding that the linear clause of each leaf is actually the weakening of a clause of $F$. Then, for each leaf $u$ we can add a new

neighbor node $w$ and label it with the clause $C$, where the edge $(w, u)$ becomes an application of the weakening rule. This process increases the depth of the tree by 1 and increases the size by at most $s$. □
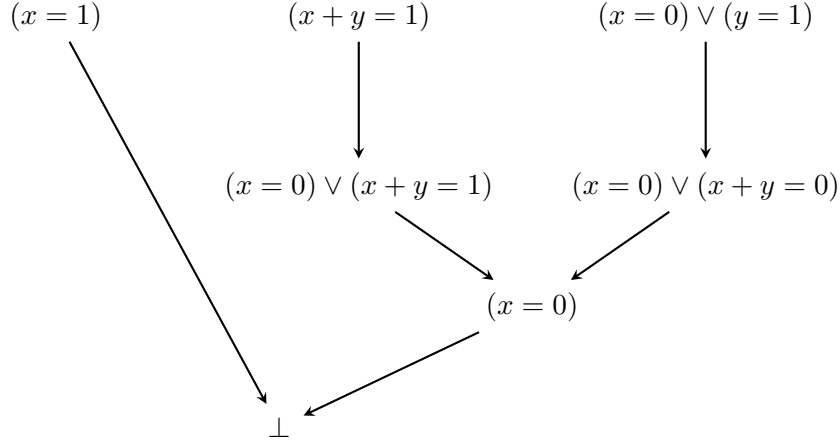
$$(x = 1) \qquad (x + y = 1) \qquad (x = 0) \vee (y = 1)$$

$$(x = 0) \vee (x + y = 1) \qquad (x = 0) \vee (x + y = 0)$$

$$(x = 0)$$

$$\bot$$

**Figure 17.** $\mathsf{TreeRes}_\oplus$-proof obtained from the PDT shown in Figure 16.

We conclude that problems efficiently solvable parity decision trees are indeed characterized by Tree-like Linear Resolution over $\mathbb{F}_2$.

**Theorem 7.** $\mathsf{FP}^{pdt}(S_F) = \Theta(\mathsf{TreeRes}_\oplus(F))$

After defining the class $\mathsf{FP}^{pdt}$ and proving that $\mathsf{TreeRes}_\oplus$ characterizes it, we're interested in studying where this class lies in the $\mathsf{TFNP}^{dt}$ hierarchy. It's a well-known fact that $\mathsf{TreeRes}_\oplus$ can efficiently simulate $\mathsf{TreeRes}$ but the reverse doesn't hold due to the hardness of simulating weakening rules. This result also follows more naturally from our results since $\mathsf{FP}^{dt} \subsetneq \mathsf{FP}^{pdt}$. Parity makes PDTs stronger than decision trees, but how much stronger?

We know that Tree-like Linear Resolution over $\mathbb{F}_2$ is based on linear forms defined on $\mathbb{F}_2$. Since Nullstellensatz over $\mathbb{F}_2$ also works with polynomials over $\mathbb{F}_2$, our intuition was to show that these two proof systems are somehow related to one another. Initially, our first hypothesis was that $\mathsf{TreeRes}_\oplus$ is a very powerful tool even capable of efficiently simulating $\mathbb{F}_2$-NS. We tried to prove this result by showing that $\mathsf{PPA}^{dt} \subseteq \mathsf{FP}^{pdt}$, which appeared to be out of reach.

In a seminal paper [IS20], Itsykson and Sokolov discussed how $\mathsf{TreeRes}_\oplus$ cannot efficiently simulate regular Resolution (or $\mathsf{RegRes}$), a restricted proof system derived from Resolution. Due to any regular Resolution proof also being a Resolution proof, this result also implies that $\mathsf{TreeRes}_\oplus$ cannot efficiently simulate $\mathsf{Res}$. Thanks to Theorem 5 and the fact that $\mathsf{PLS}^{dt}(S_F) = \Theta(\mathsf{Res}(F))$, we conclude the following black-box separation.

**Proposition 7.** $\mathsf{PLS}^{dt} \not\subseteq \mathsf{FP}^{pdt}$

This separation rings a bell: looks like PDTs aren't actually that strong. We quickly shifted our perspective on our previous study on relationships with Nullstellensatz, trying to show that the simulation holds in the other direction. Indeed, we were capable of proving that any $\mathsf{TreeRes}_\oplus$ can be converted into a small $\mathbb{F}_2$-Nullstellensatz proof, providing us a **black-box inclusion** for our new class.

## 4.4    Nullstellensatz over $\mathbb{F}_2$ and $\mathsf{PPA}^{dt}$

In 1893, the mathematician Hilbert proved a theorem that established the basis of algebraic geometry, a field that studies the relations between algebra and geometry. This theorem is now known as Hilbert's **Nullstellensatz** (german for *zero-locus theorem*).

The *weak Nullstellensatz*, a corollary of the stronger theorem, states that given $m$ polynomials $p_1, \ldots, p_m$ defined on $F[x_1, \ldots, x_n]$, where $\mathbb{F}$ is a generic algebraic field, the system $p_1(x) = p_2(x) = \ldots = p_m(x) = 0$ is unsolvable if and only if there are $m$ polynomials $g_1, \ldots, g_m$ defined on $F[x_1, \ldots, x_n]$ such that $\sum_{i=1}^m g_i p_i = 1$.

This weaker version of the theorem has been used to define an *algebraic* proof system, that being a proof system based on polynomial algebra. Intuitively, these proof systems are based on the idea of showing that a set of polynomials $p_1, \ldots, p_m$, called *axioms*, doesn't share a common root, which serves as a proof for the polynomials. In this case, a Nullstellensatz proof is given by the set of polynomials $g_1, \ldots, g_m$ through which we get that $\sum_{i=1}^m g_i p_i = 1$ [BIK+94].

Any CNF formula can be translated to an *algebraic encoding*, a set of polynomials $p_1, \ldots, p_m$ for which the CNF formula is unsatisfiable if and only if there is a Nullstellensatz proof for $p_1, \ldots, p_m$. Given the clause $C = \bigvee_{i=1}^k x_i \vee \bigvee_{j=1}^h \neg y_j$, the algebraic encoding of $C$, written as $p_C$, is given by $p_C := \prod_{i=1}^k x_i \cdot \prod_{j=1}^h (1 - y_j)$.

To clear things up, we notice that through this formulation the concept of truthfulness is **inverted**: the boolean values 0 and 1 respectively correspond to the algebraic values 1 and 0. For example, the boolean clause $C$ evaluates to 1 when at least a literal inside it evaluates to 1, while an algebraic clause evaluates to 0 when at least a literal inside it evaluates to 0.

The algebraic encoding of a CNF formula $F = C_1 \wedge \ldots \wedge C_m$ is given by the set of polynomial equations $p_F = \{p_{C_1} = 0, \ldots, p_{C_m} = 0, x_1^2 - x_1 = 0, \ldots, x_n^2 - x_n = 0\}$. These last polynomials are necessary to impose that the values of $x_1, \ldots, x_n$ are either a 0 or a 1. A Nullstellensatz *refutation* for $F$ is given by the polynomials $g_1, \ldots, g_m, h_1, \ldots, h_n$ such that:

$$\sum_{i=1}^m g_i p_{C_i} + \sum_{j=1}^m h_j (x_j^2 - x_j) = 1$$

In order for the CNF $F$ to be satisfied by an assignment $x$, each clause must evaluate to 1, while in Nullstellensatz the polynomials inside $p_F$ must all evaluate to 0.

$$
\begin{aligned}
0 &\longrightarrow 1 \\
1 &\longrightarrow 0 \\
x_i &\longrightarrow x_i \\
\neg x_i &\longrightarrow 1 - x_i \\
C \vee D &\longrightarrow C \cdot D \\
C \wedge D &\longrightarrow C + D
\end{aligned}
$$

**Figure 18.** Mappings from boolean encoding to algebraic encoding.

When a polynomial $q$ can be derived from a set of axioms $P$, we write $P \overset{\mathsf{NS}}{\vdash} q$. If $F$ is a CNF formula and $P \overset{\mathsf{NS}}{\vdash} 1$ then we get a Nullstellensatz refutation. Consider the following CNF formula:

$$x_1 \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_3 \vee x_4) \wedge x_4$$

The algebraic encoding is given by $p_1 = x_1$, $p_i = (1 - x_{i-1})x_i$ when $2 \leq i \leq 4$ and $p_5 = 1 - x_4$. To refute this CNF, we must find the polynomials $g_1, \ldots, g_5, h_1, \ldots, h_4$ through which

$$\sum_{i=1}^{5} g_i p_i + \sum_{j=1}^{4} h_j (x_j^2 - x_j) = 1$$

To simplify things, we let $h_1, \ldots, h_4 = 0$ in order to have $\sum_{j=1}^{4} h_j(x_j^2 - x_j) = 0$. Let $g_1, \ldots, g_5$ be equal to:

$$g_1 = x_2 x_3 x_4 \qquad g_2 = x_3 x_4 \qquad g_3 = x_4 \qquad g_4 = g_5 = 1$$

We easily get that:

$$
\begin{aligned}
\sum_{i=1}^{5} g_i p_i &= x_1 x_2 x_3 x_4 + (1 - x_1)x_2 x_3 x_4 + (1 - x_2)x_3 x_4 + (1 - x_3)x_4 + (1 - x_4) \\
&= x_2 x_3 x_4 + (1 - x_2)x_3 x_4 + (1 - x_3)x_4 + (1 - x_4) \\
&= x_3 x_4 + (1 - x_3)x_4 + (1 - x_4) \\
&= x_4 + (1 - x_4) \\
&= 1
\end{aligned}
$$

concluding that $P_F \overset{\mathsf{NS}}{\vdash} 1$ and thus proving that the CNF is unsatisfiable. In Nullstellensatz, the *size* of a proof is the total number of monomials of the polynomials that make the proof, i.e. the total number of terms in the sum once fully expanded without simplifying any addition (or subtraction). The *degree* of the proof is the maximum degree of any polynomial $g_i p_i$ or $h_j(x^j + x_j)$. For example, the polynomial $(1 - x_1)(1 - x_2)x_2 x_3$ has size 4 and degree 4 since $(1 - x_1)(1 - x_2)x_2 x_3 = x_2 x_3 - x_2^2 x_3 - x_1 x_2 x_3 + x_1 x_2^2 x_3$. The previous proof has size $1 + 2 + 2 + 2 + 2 = 9$ and degree 4.

Nullstellensatz's degree measure vaguely resembles Resolution's width measure. For example, the algebraic encoding of a CNF clause $C$ of width $w$ clearly has degree $w$. Moreover, it's easy to see that a degree upper bound $d$ for the Nullstellensatz

refutation of a CNF formula defined on $n$ variables implies a size upper bound of $n^{O(d)}$. This result enables us to restrict our interest to the degree of the proof.

**Proposition 8.** *Given a CNF formula $F$ defined on $n$ variables, if $P_F \overset{\text{NS}}{\vdash} 1$ with degree $O(d)$ then the size of the proof is $n^{O(d)}$.*

A common result shows that in Nullstellensatz proofs we can assume that polynomials are *multilinear* (short for *multivariate and linear*), meaning that each variable of each term has algebraic multiplicity equal to at most 1. For example, the polynomial $xy + yz$ is multilinear, while $x^2y$ isn't. This assumption affects the size and the degree of the proof only by a constant factor, which is negligible, allowing us to work easier.

As shown in Figure 10, $\mathbb{F}_2$-Nullstellensatz characterizes the black-box version of the class PPA, the class of total search problems that are reducible to the **Polynomial Parity Argument (PPA)**, first defined by Papadimitriou [Pap94].

Suppose we are given a polynomial-size circuit $C$ with $n$ input bits and $n$ output bits. For each possible input $x \in \{0,1\}^n$, the circuit provides a pair $(p, s) \in \{0,1\}^n \times \{0,1\}^n$ where $p$ and $s$ are the predecessor and successor of $x$. This circuit generates a graph $G_C$ with $2^n$ nodes, where $(u, v) \in E(G_C)$ if and only if $u$ is the predecessor of $v$ and $v$ is the successor of $u$. By construction, each node has at most two neighbor nodes and some nodes may even be disconnected.

The canonical PPA-Complete problem is LEAF: given a specific node - usually assumed to be the $0 \ldots 0$ node - with an odd number of neighbors inside the graph $G_C$ generated by $C$, we are required to find another odd-degree node. The solution of this problem is guaranteed to exist thanks to the **Handshaking lemma**, which states that every undirected graph has an even number of odd-degree nodes. The hardness of the question comes from the impossibility of using common path-traversing algorithms to find the solution: the graph has exponential size with respect to the input size!
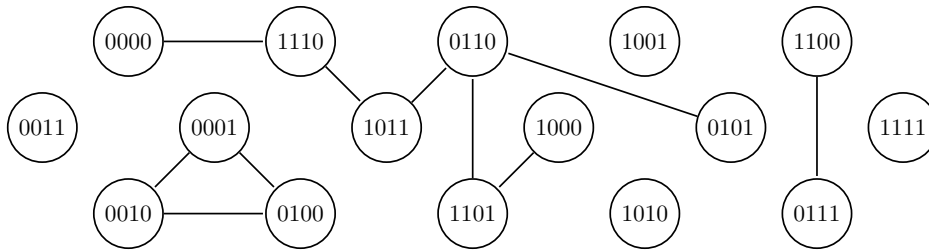


**Figure 19.** The circuit input is made of 4 bits and the generated graph has $2^4$ vertices.

Through Theorem 5, we know that an efficient proof for a formula $F$ directly implies that its search problem $S_F$ is actually efficiently reducible to the black-box version of the LEAF problem. In the following section, we show how to convert an efficient TreeRes$_\oplus$ into an efficient $\mathbb{F}_2$-Nullstellensatz proof, proving that each problem efficiently solvable through a PDT is reducible to an instance of LEAF$^{dt}$, which implies that FP$^{pdt} \subseteq$ PPA$^{dt}$.

## 4.5 From TreeRes$_\oplus$ to $\mathbb{F}_2$-Nullstellensatz

We prove that Nullstellensatz over $\mathbb{F}_2$ is capable of efficiently simulating TreeRes$_\oplus$. Given the linear clause $C = \bigvee_{i=1}^k (f_i = \alpha_i)$, the algebraic encoding of $C$, written as $p_C$, is given by $p_C := \prod_{i=1}^k (f_i + a_i)$. The algebraic encoding of a linear CNF formula $F = C_1 \wedge \ldots \wedge C_m$ is given by the set of polynomial equations $p_F = \{p_{C_1} = 0, \ldots, p_{C_m} = 0, x_1^2 - x_1 = 0, \ldots, x_n^2 - x_n = 0\}$.

To achieve our result, we first convert the TreeRes$_\oplus$ proof into an (almost) equivalent proof through the method shown in Chapter 4 and then balance the degree of the $\mathbb{F}_2 - \mathsf{NS}$ proof obtained from the tree-like proof. The following result follows from Lemma 2 and Lemma 3.

**Corollary 1.** *Every* TreeRes$_\oplus$ *proof of size $s$ and depth $d$ can be converted to a* TreeRes$_\oplus$ *proof of size $O(s)$, degree $O(d)$ and with the weakening rule applied only to the axioms.*

Next, we prove some basic Nullstellensatz inspired by the ones shown in [CEI96], which allow us to manipulate and combine different types of proofs. We will use these properties to inductively simulate the resolution rule.

Given a formula $F$, we denote with $F_{|f=\alpha}$ the formula obtained by substituting $f$ with $\alpha$ inside $F$. This is usually called **restriction of $F$ on $f = \alpha$**. In particular, we notice that if $F$ is a CNF formula then by restricting it on $f = 0$ any clause $C \vee (f = 0)$ becomes $C \vee (0 = 1)$, which is equivalent to $C$ through weakening. If we restrict $F$ on $f = 1$, any clause $C \vee (f = 0)$ becomes $C \vee (1 = 1)$, which directly evaluates to 1. In a similar fashion, given the polynomial $p$ we denote with $p_{|f=\alpha}$ the restriction of $p$ on $f = \alpha$, i.e. the polynomial obtained by setting $f$ equal to $\alpha$ in $p$.

**Lemma 4.** *Given the multilinear polynomials $p_1, \ldots, p_k$ over $\mathbb{F}_2$ of degree at most $d$ and a linear form $f$, it holds that:*

1. *If $p_1, \ldots, p_k, f \vdash 1$ with degree $d$ then $p_1, \ldots, p_k \vdash 1 - f$ with degree $d + 1$.*

2. *If $p_1, \ldots, p_k, 1 - f \vdash 1$ with degree $d$ then $p_1, \ldots, p_k \vdash f$ with degree $d + 1$.*

3. *$p_i, f \vdash p_{i|f=0}$ with degree $d$*

4. *$p_i, 1 - f \vdash p_{i|f=1}$ with degree $d$*

5. *If $p_{1|f=0}, \ldots, p_{k|f=0} \vdash 1$ with degree $d$ and $p_{1|f=1}, \ldots, p_{k|f=1} \vdash 1$ with degree $d + 1$, then $p_1, \ldots, p_k \vdash 1$ with degree $d + 1$*

6. *If $p_{1|f=0}, \ldots, p_{k|f=0} \vdash 1$ with degree $d + 1$ and $p_{1|f=1}, \ldots, p_{k|f=1} \vdash 1$ with degree $d$, then $p_1, \ldots, p_k \vdash 1$ with degree $d + 1$*

*Proof.* We proceed claim by claim:

1. Since $g_{k+1}f + \sum_{i=1}^k g_i p_i = 1$, we easily get that:

$$(1 - f)g_{k+1}f + (1 - f)\sum_{i=1}^k g_i p_i = 1 - f$$

Moreover, since $(1 - f)f = 0$ due to $f$ being defined on $\mathbb{F}_2$, we can remove $f$ from the proof, concluding that $(1 - f)\sum_{i=1}^{k} g_i p_i = 1 - f$ with degree incremented by one due to $f$ being a linear form.

2. Similar to the first claim of this lemma: multiply the whole proof by $f$ instead of $1 - f$.

3. Let $p = f p_1 + p_0$, where $p_1$ and $p_0$ are independent of $f$. By restricting $p$ on $f = 0$ we get that:

$$p_{|f=0} = 0 \cdot p_1 + p_0 = p_0 = p - f p_1$$

Since $p$ has degree at most $d$ and $f$ has degree 1, $p_1$ must have degree at most $d - 1$, thus $p - f p_1 = p_{|f=0}$ is a proof of degree at most $d$.

4. Similar to the third claim of this lemma: by restricting $p$ on $f = 1$ we obtain that:
$$p_{|f=1} = p_1 + p_0 = p_1 + p_0 + f p_1 - f p_1 = p + (1 - f)p_1$$
thus $p + (1 - f)p_1 = p_{|f=1}$ is a proof of degree at most $d$.

5. Through the previous claims of this lemma $p_1, \ldots, p_k, f \vdash p_{1|f=0}, \ldots, p_{k|f=0} \vdash 1$ with degree $d$. Then, by the first claim of this lemma, we know that $p_1, \ldots, p_k, f \vdash 1$ with degree $d$ implies that $p_1, \ldots, p_k \vdash 1 - f$ with degree $d + 1$. Likewise, we have that $p_1, \ldots, p_k, 1 - f \vdash p_{1|f=1}, \ldots, p_{k|f=1} \vdash 1$ with degree $d + 1$.

   Finally, since $p_1, \ldots, p_k \vdash 1 - f$ and $p_1, \ldots, p_k, 1 - f \vdash 1$ both with degree $d + 1$, by concatenating the two proofs we get that $p_1, \ldots, p_k \vdash 1$ with degree $d + 1$.

6. Similar to the third claim of this lemma: invert the roles of the two initial proofs and use the second claim of this lemma.

$\square$

**Lemma 5.** *Given two disjoint axiom sets $P_1, P_2$ of multilinear polynomials over $\mathbb{F}_2$, if $P_1, p \vdash 1$ with degree $d_1$ and $P_2, 1 - p \vdash 1$ with degree $d_2$ then $P_1, P_2 \vdash 1$ with degree $d_1 + d_2$.*

*Proof.* Assume that $P_1 = \{p_1, \ldots, p_m\}$ and $P_2 = \{q_1, \ldots, q_k\}$ and let $p_{m+1} = p$ and $q_{k+1} = 1 - p$. Since $\sum_{i=1}^{m+1} g_i p_i + \sum_{j=1}^{n} a_j(x_j^2 - x_j) = 1$ for some $g_1, \ldots, g_{m+1}$, we that:

$$\sum_{i=1}^{m} g_i p_i + \sum_{j=1}^{n} a_j(x_j^2 - x_j) = 1 - g_{m+1} p_{m+1} = 1 - g_{m+1} p$$

Likewise, since $\sum_{i=1}^{k+1} h_i q_i + \sum_{j=1}^{n} b_j(x_j^2 - x_j) = 1$ for some $h_1, \ldots, h_{k+1}$, we get that:

$$\sum_{i=1}^{k} h_i q_i + \sum_{j=1}^{n} b_j(x_j^2 - x_j) = 1 - h_{k+1} q_{k+1} = 1 - h_{k+1}(1 - p)$$

We notice that:

$$(1-p)\left(\sum_{i=1}^{m} g_i p_i + \sum_{j=1}^{n} a_j(x_j^2 - x_j)\right) = (1-p)(1 - g_{m+1}p)$$
$$= 1 - p - g_{m+1}p(1-p)$$
$$= 1 - p$$

with degree $d_1 + d_2$. In the last step, we used the fact that $p(1-p) = 0$ holds in $\mathbb{F}_2$. Similarly, we get that:

$$p\left(\sum_{i=1}^{k} h_i q_i + \sum_{j=1}^{n} b_j(x_j^2 - x_j)\right) = p$$

again with degree $d_1 + d_2$. By summing these two proofs, we get that

$$(1-p)\left(\sum_{i=1}^{m} g_i p_i + \sum_{j=1}^{n} a_j(x_j^2 - x_j)\right) + p\left(\sum_{i=1}^{k} h_i q_i + \sum_{j=1}^{n} b_j(x_j^2 - x_j)\right) = 1$$

concluding that $P_1, P_2 \vdash 1$ with degree $d_1 + d_2$. $\qquad\square$

Next, we prove that a Tree-like Linear Resolution over $\mathbb{F}_2$ can be efficiently converted into a small $\mathbb{F}_2$-Nullstellensatz proof. The very same proof can be used to also show that standard Tree-like Linear Resolution can be simulated by $\mathbb{F}_2$-Nullstellensatz, an already known result.

**Lemma 6.** *Let $F$ be an unsatisfiable CNF. If $T$ is TreeRes$_\oplus$ refutation of $F$ of size $s$ and width $w$ that uses only the resolution rule then there is $\mathbb{F}_2$-NS refutation of $F$ of degree $w + \log_2 s$.*

*Proof.* Let $F = C_1 \wedge \ldots \wedge C_k$. We proceed by strong induction on the size $s$ of $T$. When $S = 1$, one of the axioms of $F$ must be the empty clause, hence $p_{C_i} \vdash 1$ for some $i$ and by extension that $p_{C_i}, \ldots, p_{C_k} \vdash 1$.

Suppose now that $s > 1$. Let $f = 0$ and $f = 1$ be the two clauses that derive the empty clause $\bot$ in $T$. These two clauses are derived through two subtrees $T_0, T_1$ of sizes $s_0, s_1$, where $s = s_0 + s_1 + 1$.

By restricting $T_0$ on $f = 1$, the tree $T_{0|f=1}$ is a TreeRes$_\oplus$ proof of the formula $C_{1|f=1} \wedge \ldots \wedge C_{k|f=1}$ of size $s_1$. Since $T$ uses only the resolution rule, $T_0$ also does. Then, my induction, we have that $p_{C_{1|f=1}}, \ldots, p_{C_{k|f=1}} \vdash 1$ with degree $w + \log_2 s_1$. Using a similar argument, the tree $T_{1|f=0}$ produces a proof $p_{C_{1|f=0}}, \ldots, p_{C_{k|f=0}} \vdash 1$ with degree $w + \log s_2$.

If $s_1 < \frac{s}{2}$, then $w + \log_2 s_1 \leq w + \log_2 s - 1$. Thus, the proof $p_{C_{1|f=1}}, \ldots, p_{C_{k|f=1}} \vdash 1$ is also a proof of degree $w + \log_2 s - 1$, while the proof $p_{C_{1|f=0}}, \ldots, p_{C_{k|f=0}} \vdash 1$ is also a proof of degree $w + \log_2 s$. By the fifth claim of Lemma 4, we conclude that $p_{C_i}, \ldots, p_{C_k} \vdash 1$ with degree $w + \log_2 s$.

If $s_2 < \frac{s}{2}$, instead, we get a symmetric proof, where the role of the two proofs is swapped. By the sixth claim of Lemma 4, we again conclude that $p_{C_i}, \ldots, p_{C_k} \vdash 1$ with degree $w + \log_2 s$. $\qquad\square$
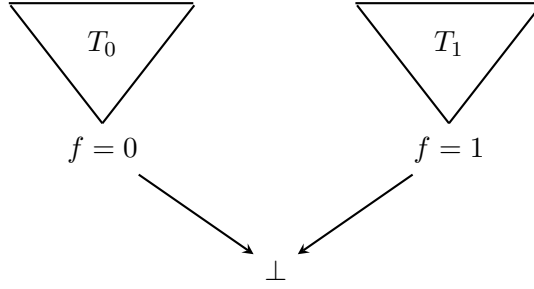
**Figure 20.** Representation of the idea behind Lemma 6.

The weakening rule is generally hard to simulate through $\mathbb{F}_2$-Nullstellensatz. However, if $D$ is derived through weakening from an axiom clause $C_i$ of a CNF formula $F$, we can easily simulate this rule. This result is enough for our purposes but it can also be extended to derivations from a non-axiom clause with a little blow-up in degree.

**Lemma 7.** *Let $F = C_1 \wedge \ldots \wedge C_m$ be a CNF formula and let $D$ be a linear clause. If $C_i \implies D$ then $p_{C_i} \vdash p_D$ with degree $d + k$, where $d$ is the width of $D$ and $k$ is the width of $C_i$.*

*Proof.* Let $C := C_i$. Assume $C = \bigvee_{i=1}^{k}(x_i = \alpha_1)$ and $D = \bigvee_{j=1}^{d}(f_j = \beta_j)$. We notice that any polynomial $q(1 + q)$ can be derived with degree 2 from axioms.

$$(y_1 + \ldots + y_t)(y_1 + \ldots y_t + 1) = \sum_{i=1}^{t} y_i^2 + \sum_{i=1}^{t} y_i + 2 \sum_{i \neq j} y_i y_j = \sum_{i=1}^{t} y_i^2 + y_i$$

since $2 = 0$ in $\mathbb{F}_2$. This implies that for each $j \in [d]$ we can derive $f_j + \beta_j + 1$ with degree $d + 1$.

Since $C \implies D$, this can only happen if each $x_i + \alpha_i$ is a linear combination of $(f_1 + \beta_1 + 1), \ldots, (f_d + \beta_d + 1)$, concluding that each $p_D(x_i + \alpha_i)$ is derivable in $\mathbb{F}_2$-Nullstellensatz with degree $d + 1$. Finally, we notice that:

$$p_D = p_C + p_D(x_1 + \alpha_1 + 1) + p_D(x_2 + \alpha_2 + 1)(x_1 + \alpha_1) + \ldots + p_D(x_d + \alpha_d + 1) + \prod_{i=0}^{d-1}(x_i + \alpha_i)$$

which is a derivation of $p_D$ from $p_C$ with degree $d + k$. $\qquad\square$

**Theorem 8.** *Let $F$ be an unsatisfiable CNF. If $T$ is TreeRes$_\oplus$ refutation of $F$ of size $s$ and width $w$ then there is NS refutation of $F$ of degree $O(w + \log s)$.*

*Proof.* Let $F = C_1 \wedge \cdots \wedge C_m$ and let $T$ be a TreeRes$_\oplus$-proof of $F$ with size $s$ and width $w$. Through Corollary 1 we know that there must also be a TreeRes$_\oplus$-proof of size $O(s)$ with the weakening rule applied only to the leaves. Let $\widehat{C_1}, \ldots, \widehat{C_m}$ be the linear clauses obtained through such weakening rules and let $T'$ be the subtree.

Let $T'$ be the subtree of $T$ with root $\perp$ and leaves $\widehat{C_1}, \ldots, \widehat{C_m}$. Since $T'$ only uses the resolution rule, by Lemma 6 we conclude that $p_{\widehat{C_1}}, \ldots, p_{\widehat{C_m}} \vdash 1$ with degree $O(w + \log s)$. Clearly, this also implies that $\deg(p_{\widehat{C_i}}) = O(w + \log s)$ for all $i$.

Since each $\widehat{C_i}$ is a weakening of $C_i$, by Lemma 7 we know that $p_{C_i} \vdash p_{\widehat{C_i}}$ with degree $O(w + \log s)$. We trivially get that $p_{C_i}, (1 - p_{\widehat{C_i}}) \vdash 1$ with degree $O(w + \log s)$. Moreover, since $p_{\widehat{C_1}}, \ldots, p_{\widehat{C_m}} \vdash 1$ with degree $O(w + \log s)$, by Lemma 5 we get that $p_{C_1}, p_{\widehat{C_2}}, \ldots, p_{\widehat{C_m}} \vdash 1$ with degree $O(w + \log s)$. After repeating this process for each weakening clause, we finally conclude that $p_{C_1}, \ldots, p_{C_m} \vdash 1$ with degree $O(w + \log s)$. $\square$
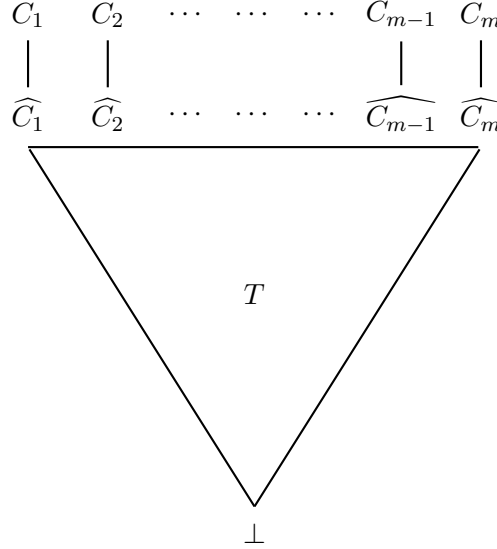


**Figure 21.** Representation of the idea behind Theorem 8.

Finally, thanks to Theorem 5 and the fact that $\mathrm{PPA}^{dt}(S_F) = \Theta(\mathbb{F}_2\text{-NS}(F))$, the last theorem proves that our new class is indeed contained inside $\mathsf{PPA}^{dt}$, meaning that any total search problem efficiently solvable by a parity decision tree can be reduced to an instance of the parity argument problem.

**Theorem 9.** $\mathsf{FP}^{pdt} \subseteq \mathsf{PPA}^{dt}$

*Proof.* Suppose that $R \in \mathsf{FP}^{pdt}$. By definition, there is a PDT that solves $R$ with size $s$ and depth $d$, where $d + \log s = O(\log^k n)$ for some $k \in \mathbb{N}$. We know that each $\mathsf{TFNP}^{dt}$ is equivalent to the false clause search problem of some CNF formula $F$, thus $R = S_F$.

By Lemma 3, we know that there is a TreeRes$_\oplus$ proof of $F$ with size $O(s)$ and depth $d$. Then, by theorem Theorem 8, we know that there must be a $\mathbb{F}_2$-Nullstellensatz refutation for $F$ with degree $O(w + \log s)$ and size $n^{O(w + \log s)}$, where $w$ is the width of the tree-like proof.

We get that the total Nullstellensatz complexity of the proof $F$, given by the sum of the depth and the logarithm of the size, corresponds to $(w + \log s) + \log(n^{O(w + \log s)}) = O(\log^{k+1} n)$, hence $F$ has a polylogarithmic $\mathbb{F}_2$-NS refutation. By Theorem 5, this implies that there is an efficient reduction $S_F \leq_m \mathsf{PPA}^{dt}$, concluding that $R \in \mathsf{PPA}^{pdt}$ $\square$

# Conclusions

The relations between total search problems, protocols, circuits, and proofs make TFNP an interesting and multifaceted theory that holds the potential to capture several key aspects of complexity theory under a single, unified framework. However, despite these promising connections, such universal characterization remains *fuzzy* and incomplete: tools such as *query-to-communication lifting* theorems and *interpolation theorems* do not yet provide a sufficiently strong bridge to unify these approaches.

In our study, we have shown that introducing parity to the black-box model leads to the definition of a new class that used a more powerful computational model, i.e. parity decision trees, providing a new lens through which we can study the complexity of total search problem. We have also showed how parity decision trees are characterized in a natural way by Tree-like Linear Resolution over $\mathbb{F}_2$. Through this connection, we have gained insights into the inherent computational limitations of parity decision trees, suggesting that their applicability may be confined to specific problem classes.
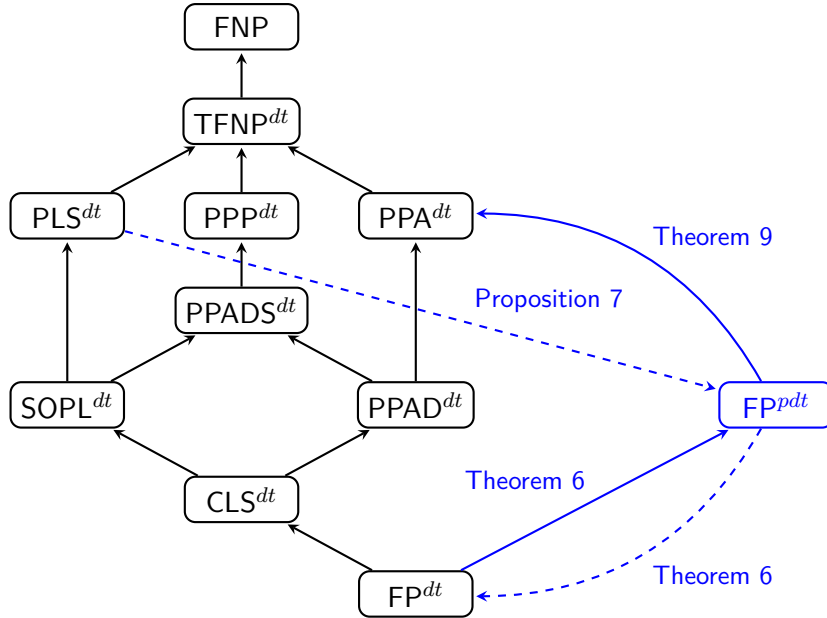


**Figure 22.** TFNP$^{dt}$ hierarchy extended through our results. An arrow $A \to B$ means that $A \subseteq B$ while a dashed arrow $A \dashrightarrow B$ means that $A \nsubseteq B$.

These findings raise several questions for further exploration regarding the use of parity decision trees in the black-box model:

1. *Modeling the TFNP hierarchy with parity decision trees.* We have shown that parity decision trees are computationally stronger than classical decision trees. Can the entire black-box TFNP hierarchy also be effectively modeled using PDTs? Does the introduction of parity decision trees create a fundamentally stronger model? What is the precise relationship between these two hierarchies?

2. *Reductions through Parity Decision Tree.* In the current black-box model, reductions between problems are captured using decision trees. Can such reductions be generalized to parity decision tree reductions? Do the same inclusions and separations between classes also hold in the TFNP$^{pdt}$ hierarchy?

3. *Generalizing to finite fields beyond $\mathbb{F}_2$.* Our results have so far been based on the field $\mathbb{F}_2$. Can these techniques and characterizations be extended to a general finite field $\mathbb{F}_q$? Is Tree-like Linear Resolution over $\mathbb{F}_q$ able to define a broader class of computational models that generalize parity decision trees?

4. *Connection to Communication Complexity.* Parity decision trees have been shown to be linked to communication complexity [Yao15]. Is there an white-box model in TFNP$^{cc}$ analogous to FP$^{pdt}$ that could be characterized by specific types of communication protocols or circuit classes?

# Acknowledgements

This thesis represents the culmination of a long journey, one that would have been impossible to complete without the unwavering support, encouragement, and guidance from the remarkable people around me. I am profoundly grateful to all who played a part in this process and contributed to this achievement.

First and foremost, I would like to express my deepest and most sincere gratitude to my advisors, professor N. Galesi and professor M. Lauria. Your expert guidance, patience and insightful feedback have been fundamental throughout this endeavor.

To my parents, no words can capture the depth of my gratitude. Thank you for always believing in my abilities and for the countless sacrifices you made to ensure I had every opportunity to succeed. Your unwavering support has been the foundation upon which my confidence and resilience are built. I owe everything to your love and faith in me.

To my brother, thank you for being the dual entity of my life. You have provided me with a different perspective on everything, helping me navigate challenges and see my strengths and weaknesses more clearly. Your presence has been a vital and irreplaceable part of this journey, bringing balance and clarity to my life when I needed it most.

To my closest friends, your loyalty and unwavering support over the years have been invaluable. Your friendship has been a continuous source of joy, laughter, shared experiences and constant growth. Special thanks to those who have been by my side since childhood and adolescence. You have been my pillars through thick and thin, and I am forever grateful for the memories we've created together.

To my university colleagues, thank you for making this academic journey so memorable. The late-night discussions, collaborative problem-solving sessions, shared triumphs and struggles have enriched this experience in ways I will cherish forever.

Finally, to everyone who has contributed to my personal and academic growth along the way - whether through direct support or quiet encouragement - thank you. This achievement is not mine alone; it stands as a testament to your generosity, wisdom and your enduring influence on my life.

# Bibliography

[AB09]      Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. 1st. USA: Cambridge University Press, 2009. ISBN: 0521424267.

[BCE+98]    Paul Beame, Stephen Cook, Jeff Edmonds, et al. "The Relative Complexity of NP Search Problems". In: *Journal of Computer and System Sciences* 57.1 (1998), pp. 3–19. ISSN: 0022-0000. DOI: `10.1006/jcss.1998.1575`.

[BFI23]     Sam Buss, Noah Fleming, and Russell Impagliazzo. "TFNP Characterizations of Proof Systems and Monotone Circuits". In: *14th Innovations in Theoretical Computer Science Conference (ITCS 2023)*. 2023, 30:1–30:40. DOI: `10.4230/LIPIcs.ITCS.2023.30`.

[BG94]      Mihir Bellare and Shafi Goldwasser. "The Complexity of Decision Versus Search". In: *SIAM Journal on Computing* 23.1 (1994), pp. 97–119. DOI: `10.1137/S0097539792228289`.

[BGS75]     Theodore Baker, John Gill, and Robert Solovay. "Relativizations of the P ?= NP Question". In: *SIAM Journal on Computing* 4.4 (1975), pp. 431–442. DOI: `10.1137/0204037`.

[BIK+94]    P. Beame, R. Impagliazzo, J. Krajicek, et al. "Lower bounds on Hilbert's Nullstellensatz and propositional proofs". In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. 1994, pp. 794–806. DOI: `10.1109/SFCS.1994.365714`.

[BKT14]     Samuel R. Buss, Leszek A. Kołodziejczyk, and Neil Thapen. "Fragments of approximate counting". In: *The Journal of Symbolic Logic* 79.2 (2014), pp. 496–525. ISSN: 00224812, 19435886.

[CDT09]     Xi Chen, Xiaotie Deng, and Shang-Hua Teng. "Settling the complexity of computing two-player Nash equilibria". In: *J. ACM* 56.3 (May 2009). ISSN: 0004-5411. DOI: `10.1145/1516512.1516516`.

[CEI96]     Matthew Clegg, Jeffery Edmonds, and Russell Impagliazzo. "Using the Groebner basis algorithm to find proofs of unsatisfiability". In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. STOC '96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 174–183. ISBN: 0897917855. DOI: `10.1145/237814.237860`. URL: `https://doi.org/10.1145/237814.237860`.

[Chu36]     Alonzo Church. "An Unsolvable Problem of Elementary Number Theory". In: *American Journal of Mathematics* 58 (1936), p. 345. URL: https://api.semanticscholar.org/CorpusID:14181275.

[Coo71]     Stephen A. Cook. "The complexity of theorem-proving procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing.* STOC '71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. ISBN: 9781450374644. DOI: 10.1145/800157.805047.

[Dav82]     Martin Davis. "Why Gödel didn't have church's thesis". In: *Information and Control* 54.1 (1982), pp. 3–24. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(82)91226-8. URL: https://www.sciencedirect.com/science/article/pii/S0019995882912268.

[DGP06]     Constantinos Daskalakis, Paul W. Goldberg, and Christos H. Papadimitriou. "The complexity of computing a Nash equilibrium". In: *Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing.* STOC '06. Seattle, WA, USA: Association for Computing Machinery, 2006, pp. 71–78. ISBN: 1595931341. DOI: 10.1145/1132516.1132527.

[DK14]      Ding-Zhu Du and Ker-I Ko. "Models of Computation and Complexity Classes". In: *Theory of Computational Complexity.* 2014. Chap. 1, pp. 1–44. ISBN: 9781118595091. DOI: 10.1002/9781118595091.ch1.

[FGH+22]    John Fearnley, Paul Goldberg, Alexandros Hollender, et al. "The Complexity of Gradient Descent". In: *J. ACM* 70.1 (Dec. 2022). ISSN: 0004-5411. DOI: 10.1145/3568163.

[Gál02]     Anna Gál. "A characterization of span program size and improved lower bounds for monotone span programs". In: *Comput. Complex.* (May 2002), pp. 277–296. DOI: 10.1007/s000370100001.

[GHJ+22a]   Mika Göös, Alexandros Hollender, Siddhartha Jain, et al. "Further collapses in TFNP". In: *Proceedings of the 37th Computational Complexity Conference.* CCC '22. Philadelphia, Pennsylvania: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2022. ISBN: 9783959772419. DOI: 10.4230/LIPIcs.CCC.2022.33.

[GHJ+22b]   Mika Göös, Alexandros Hollender, Siddhartha Jain, et al. "Separations in Proof Complexity and TFNP". In: *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS).* 2022, pp. 1150–1161. DOI: 10.1109/FOCS54457.2022.00111.

[GKR+19]    Mika Göös, Pritish Kamath, Robert Robere, et al. "Adventures in Monotone Complexity and TFNP". In: *10th Innovations in Theoretical Computer Science Conference (ITCS 2019).* 2019, 38:1–38:19. DOI: 10.4230/LIPIcs.ITCS.2019.38.

[IS20]      Dmitry Itsykson and Dmitry Sokolov. "Resolution over linear equations modulo two". In: *Annals of Pure and Applied Logic* 171.1 (2020), p. 102722. ISSN: 0168-0072. DOI: https://doi.org/10.1016/j.apal.2019.102722.

[Kar72]     Richard Karp. "Reducibility Among Combinatorial Problems". In: vol. 40. Jan. 1972, pp. 85–103. ISBN: 978-3-540-68274-5. DOI: `10.1007/978-3-540-68279-0_8`.

[Kle36]     S. C. Kleene. "Lambda-definability and recursiveness". In: *Duke Mathematical Journal* 2.2 (1936), pp. 340–353. DOI: `10.1215/S0012-7094-36-00227-2`.

[KW88]     Mauricio Karchmer and Avi Wigderson. "Monotone circuits for connectivity require super-logarithmic depth". In: *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing.* STOC '88. Chicago, Illinois, USA: Association for Computing Machinery, 1988, pp. 539–550. ISBN: 0897912640. DOI: `10.1145/62212.62265`.

[Lev73]     Leonid A. Levin. "Universal Sequential Search Problems". In: *Problems of Information Transmission* 9.3 (1973). URL: `https://www.mathnet.ru/php/archive.phtml?wshow=paper&jrnid=ppi&paperid=914&option_lang=eng#forwardlinks`.

[LNN+95]     László Lovász, Moni Naor, Ilan Newman, et al. "Search Problems in the Decision Tree Model". In: *SIAM J. Discret. Math.* 8.1 (Feb. 1995), pp. 119–132. ISSN: 0895-4801. DOI: `10.1137/S0895480192233867`.

[MP91]     Nimrod Megiddo and Christos H. Papadimitriou. "On total functions, existence theorems and computational complexity". In: *Theoretical Computer Science* 81.2 (1991), pp. 317–324. ISSN: 0304-3975. DOI: `10.1016/0304-3975(91)90200-L`.

[Pap94]     Christos H. Papadimitriou. "On the complexity of the parity argument and other inefficient proofs of existence". In: *Journal of Computer and System Sciences* 48.3 (1994), pp. 498–532. ISSN: 0022-0000. DOI: `https://doi.org/10.1016/S0022-0000(05)80063-7`.

[RGR22]     Susanna F. de Rezende, Mika Göös, and Robert Robere. "Proofs, Circuits, and Communication". In: *ArXiv* abs/2202.08909 (2022).

[RYM+22]     Anup Rao, Amir Yehudayoff, A Mir, et al. "Communication Complexity and Applications". In: 2022.

[Sip96]     Michael Sipser. *Introduction to the Theory of Computation.* 1st. International Thomson Publishing, 1996. ISBN: 053494728X.

[Tur37]     A. M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2-42.1 (Jan. 1937), pp. 230–265. ISSN: 0024-6115. DOI: `10.1112/plms/s2-42.1.230`.

[Yao15]     Penghui Yao. "Parity decision tree complexity and 4-party communication complexity of XOR-functions are polynomially equivalent". In: *arXiv preprint arXiv:1506.02936* (2015).