Sapienza
Università di Roma

Introduction of parity-based computational models in total search problems

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

Simone Bianco
ID number 1986936

Advisor
Prof. Nicola Galesi

Co-Advisor
Prof. Massimo Lauria

Academic Year 2023/2024

TODO.

Contents

1

Introduction

Chapter 1

Preliminaries

1.1 Computational models and Turing machines

Throughout history humans have been solving problems through a wide variety of models capable of computing a valid result, ranging from their own intellect to mechanical devices capable of solving problems. In particular, each computational model can be described as a list of sequential operations. Given the same initial conditions, these lists are expected to yield the same exact result each time the computation is executed.

In modern mathematics, this is described through the concept of **algorithm**, that being 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 straight forward definition for an algorithm, 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 kinds of algorithms. In 1950, Alan Turing was able to define a theoretical computational model capable of capturing the concept of computation itself through a simple - but sufficient - theoretical machine, i.e. the now called **Turing machine**.

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 $\sqcup$, 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 is able to shift left and right one and 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.

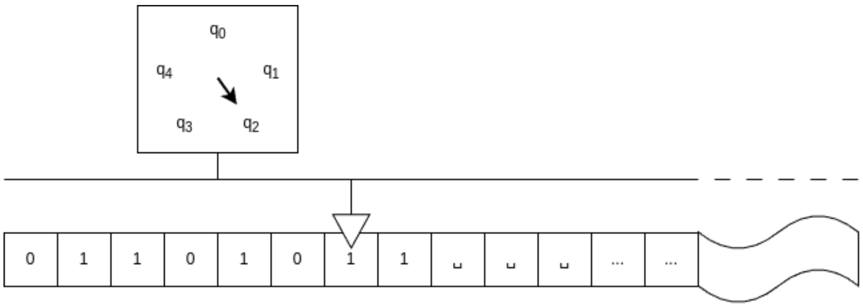

Figure 1.1. A Turing Machine

The *Church-Turing thesis* states that it suffices to restrict attention to this single model since it can compute all computable mathematical functions with only a little loss of efficiency. This result is able to characterize the concept of algorithm through Turing machines themselves: for each algorithm there is an equivalent TM and vice versa. Furthermore, Turing proved the existence of an *universal Turing machine*, a TM that given an encoded string that describes another Turing machine is capable of simulating that very same machine. This fact shouldn't be a surprise: modern computers are nothing more than universal TMs that can execute any given algorithm described by machine code, which is just a string of zeros and ones.

We give the following formal definition of Turing machine.

Definition 1.1. 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.
- Γ is a finite set of symbols, usually called the tape alphabet. The tape alphabet always contains the symbol $\sqcup$.
- Σ 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 L and 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 then it transitions to the state p, replaces the symbol a with b and moves the head to the left.

1.2 Complexity measures

After being able to give a mathematically stable definition computation through Turing machines, researchers shifted their focus on understanding what problems are computable. In particular, they showed that some problems are **uncomputable** by proving that there cannot exist a Turing machine capable of carrying out their computation without going in infinite loops and never halting, such as Turing's famous *Halting problem*, i.e. determining if a machine will halt or not for a given input.

A "good" algorithm (or TM) should be able to solve the associated problem in an efficient way. But what does it mean for a TM to be *efficient*? To formally define 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 1.2. Given a Turing machine M, 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 initially blank cells used by M to compute an input string of length n.

Time and space complexity are intrinsically related one to the other: time limits the amount of space and vice versa. 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 a low amount of used space then the computation will require more steps. These reasons are enough to make both time and space an interesting measure to evaluate the efficiency of an algorithm.

Usually, 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 1.3. 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 values $c, N \in \mathbb{N}_{>0}$ 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 values $c, N \in \mathbb{N}_{>0}$ such that $\forall n \geq N$ it holds that $f(n) \geq cg(n)$.
3. f is in Theta of g, written as $f(n) = \Theta(g(n))$, if there are three values $c, d, N \in \mathbb{N}_{>0}$ 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 $g(n)$. 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 $f(n) = \Omega(g(n))$.

Efficiency dictates wether a problem is actually feasible or not in the real world: if a problem is computable by a TM but it requires a immense amount of time or space to get to the result, then the computation is practically unachievable. This kind of problems are called **intractable problems**.

Complexity is generally measured in terms of asymptotic behavior. An algorithm is considered time efficient if it is able to compute the answer 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 is is able to compute the answer in at most a logarithmic amount of space, i.e. in $O(\log^k n)$ space for some $k \in \mathbb{N}$.

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 through 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 can clearly be expressed as a real multiple of the value m. This means that $k, x, y = O(m)$ and thus they can be encoded with $O(\log m)$ bits (asymptotic notation allows us to drop the subscript of the logarithm), thus our requiring $3 \cdot O(\log m)$ cells, which is asymptotically equivalent to $O(\log m)$ cells. We conclude that the space complexity of such TM is $O(\log m) = O(n)$.

To copy a string of length ℓ, the Turing machine needs copy ℓ 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 steps of the loop 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 $O(m \log m)$ cells. Thus, we conclude that the time complexity of such TM is $O(m \log m) = O(2^n n)$.

This implies that the example algorithm defined above is inefficient in both time and space, making repeated addition one of the worst ways to solve this task. But does this mean that the problem is intractable? Modern computers are able to square a number in milliseconds, so the answer to this question should clearly be no. In fact, even by implementing the column method for multiplying numbers usually taught to kids we could solve the problem in a very low amount of steps but a not-so efficient amount of space.

Efficiency is the lingering question that torments 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 even after more than 70 years of research the question still persists in the mind 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 a simple question with a «*yes*» or «*no*» answer, such as asking if some input object has got some property or not. Each decision problem can be described as a subset of given language Σ^*, 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. Usually, a positive «*yes*» answer is represented by a 1, while a negative «*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 $= \{n \in \mathbb{N} \mid n \text{ is prime}\}$.

Definition 2.1. A decision problem for a property P is a subset L of a language Σ^* such that $L = \{x \in \Sigma^* \mid P(x) = 1\}$.

Any decidable problem can be *decided* by a Turing machine, meaning that for any input x of the language Σ^* the TM is capable of returning the answer 0 or 1. Decidability theory plays a core rule in math and computer science since most of problems can be modeled by it. However, by their own nature, decision problems are limited. Some problems require a more complex result then 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?*». This type 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 an "harder" type of 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).

Formally, 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 example, the question «*what is the prime factorization of n?*» is modeled by the functional problem FACTORING $= \{(n, (p_1, \dots, p_k)) \in \mathbb{N} \times \mathbb{N}^k \mid n = p_1 \cdot \dots \cdot p_k\}$.

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 $p_1, \dots, p_k$ the prime factorization of n?*» corresponds to the decision problem FACTORIZATION$_n = \{(p_1, \dots, p_k) \in \mathbb{N}^k \mid n = p_1 \cdot \dots \cdot p_k\}$.

Even thought 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 in a different way. For example, the decision problem PRIMES can be converted into the functional problem $\{(n, b) \in \mathbb{N} \times \{0, 1\} \mid b = 1 \text{ if } n \text{ is prime, } b = 0 \text{ otherwise}\}$, but they aren't effectively the same problem even thought they answer the same question.

Another important thing to notice is that even though the name implies a correlation to mathematical functions due to the concept of input-output being involved, the given definition also includes *partial* and *multivalued* functions, that being 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 $\{0, 1\}^k$ for some $k \in \mathbb{N}$, describing all inputs as a sequence of bits. Since each problem could have inputs of different lengths, researches have defined search problems through the use of a sequence of relations rather than a single relation [LNN+95; BCE+98; RGR22; BFI23]. This also allows search problems to have different types of outputs based on the length of the inputs.

Definition 2.2. 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 function 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 in numerous categories, each defining its own subclass of problems. One of the most important subclasses is made 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 (see Chapter 1). Not all decision problems have been shown to be efficiently solvable, in fact some of them have been proven to be outside of P (again, see chapter Chapter 1). However, several problems for which there is currently no answer regardless wether 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 2.3. 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 efficiently verifiable is referred to as NP. This class of problems has been shown to be 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 replace with the verifier definition due to NTMs being only a theoretical computational model that is physically unrealizable. For our purposes, it is sufficient to consider the modern definition of NP.

By definition of these two classes, its easy to see that $\mathsf{P} \subseteq \mathsf{NP}$ since every problem that is efficiently solvable can also be efficiently verified. However, it is currently not known wether $\mathsf{P} = \mathsf{NP}$ or not. The answer to this question is considered to be one of the most important questions in mathematics. In fact, if $\mathsf{P} = \mathsf{NP}$ 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 $\mathsf{P} \neq \mathsf{NP}$. For example, cryptography assumes that for some cryptographic schemes it's easy to check that an encrypted string is the result of scheme being applied on an original message, which works as a certificate, and very hard to actually find the original message only through the encrypted string. If $\mathsf{P} = \mathsf{NP}$ 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 2.4. 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 $\exists w \in \{0, 1\}^{\mathrm{poly(n)}}$ 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 thought any decision problem can be transformed ina search problem with only two possible output, since they are defined on two different types of problems it doesn't really make sense to say that $\mathsf{P} \subseteq \mathsf{FP}$ or that $\mathsf{NP} \subseteq \mathsf{FNP}$. However, an important result shows that it can hold that $\mathsf{P} = \mathsf{NP}$ if and only if $\mathsf{FP} = \mathsf{FNP}$ [BG94; DK14]. This implies that, even though search problems are by definition more complex than decision problems, answering one of the two questions would answer both of them.

Theorem 2.1. *$\mathsf{P} = \mathsf{NP}$ if and only if $\mathsf{FP} = \mathsf{FNP}$*

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

Suppose now that $\mathsf{P} = \mathsf{NP}$. We already know that $\mathsf{FP} \subseteq \mathsf{FNP}$, so we have to show that $\mathsf{FP} \subseteq \mathsf{FNP}$. Let $R = (R_n)_{n\in\mathbb{N}} \in \mathsf{FNP}$ 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 $L_n = \{(x, y) \mid \exists z \in \{0, 1\}^k, k \leq n \text{ s.t. } (x, zw) \in R_n\}$. It's easy to see that $L_n \in \mathsf{NP}$ since each pair (x, z) is certified by the string zw itself and the correctness of this certificate can be polynomially verified given that $R \in \mathsf{FNP}$.

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

```
function Solve_n($x$)
    $y = \varepsilon$                                  ▷ $\varepsilon$ 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
```

□

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 worlds 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 2.5. 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 $\mathsf{FP} \subseteq \mathsf{TFNP} \subseteq \mathsf{FNP}$, 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 $\mathsf{FP} = \mathsf{FNP}$ 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 that can certificate that the solution is correct.

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

Proposition 2.1. $\mathsf{TFNP} = \mathsf{F}(\mathsf{NP} \cap \mathsf{coNP})$

Proof. If $R = (R_n)_{n \in \mathbb{N}} \in \mathsf{TFNP}$ 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 $\overline{R} \in \mathsf{FNP}$. Hence, we conclude that $R \in \mathsf{F}(\mathsf{NP} \cap \mathsf{coNP})$.

Vice versa, if $S \in \mathsf{F}(\mathsf{NP} \cap \mathsf{coNP})$ then trivially we have that $S \in \mathsf{FNP}$. Moreover, since $S \in \mathsf{F}(\mathsf{NP} \cap \mathsf{coNP})$ 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 $S \in \mathsf{TFNP}$.

□

2.3 The TFNP hierarchy

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 know 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 trough *many-to-one mappings*, computable functions that map instances of the original problem to instances of the reduced problem.

Definition 2.6. A decision problem A is many-to-one reducible to a decision problem B, written as $A \leq_m B$, if there is a computable function f such that $x \in A$ if and only if $f(x) \in A$.

When a reduction can be efficiently computed by a TM with a time (or space) complexity that is in the order of to the complexity of B, the problem A can be solved by a machine that first computes the reduction and then solves the problem B, implying that the complexity of A is "as hard as" B, meaning that its complexity is in the order of the complexity of B. For example, if B is in P and the reduction $A \leq_m B$ can be computed in polynomial time then A also lies in P.

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 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 for such search problem. But how could we map inputs without solutions to other inputs without solution? What if one of the two problems involved is total and the other isn't? This clearly doesn't make sense. Even if it did make sense, we are only interested in finding solutions. We give the following definition of search problem reduction:

Definition 2.7. 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 $S = (S_n)_{n \in \mathbb{N}}$, written as $R \leq_m S$, 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 function $f : \{0,1\}^m \to \{0,1\}^n$ and a 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.

Reductions play a critical role in computer science. In particular, they allow us to define the concept of **completeness**, the property of an entire class of problems to be reduces into one specific problem from that very same class.

Definition 2.8. A problem B is said to be complete for a class of problems $\mathcal{C}$ if $B \in \mathcal{C}$ and $\forall A \in \mathcal{C}$ it holds that $A \leq B$.

Under some circumstances, if a complete problem is proven to have a specific property then that property gets automatically inherited by all the problems of the class. For example, if an NP-Complete problem is proven to be solvable in polynomial time, then every single problem inside NP would inherit this property, making the entire class collapse and giving an answer to the question $\mathsf{P} = \mathsf{NP}$. However, in order for this inheritance ability to hold, these reductions must still be efficient with respect to the complexity of the class. For example, for a problem B to be NP-Complete then any NP problem A must be reducible to B in polynomial time, since otherwise there would be no way of using the reduction to efficiently obtain solutions of A. The same reasoning also holds for the concept of completeness in the class FNP.

The most famous NP-Complete problem is the SAT problem, which asks «*does this formula have an assignment that satisfies it?*», first 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, that being FSAT, modelling 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].

Proposition 2.2. *The decisional problem A is NP-Complete if and only if the functional problem FA is FNP-Complete.*

However, 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 there is no output assignment that satisfies them. Researchers believe that the existence of such problem is very unlikely since this total problem would be able to give a solution to problems that aren't total.

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 who have a lot of TFNP problems reducible to them. These important problems form additional subclasses of TFNP.

Definition 2.9. Given TFNP problem R, we define the class R as the subset of TFNP problems efficiently reducible to the problem R in polynomial time, formally $R = \{S \in \mathsf{TFNP} \mid S \leq_m R \text{ in polynomial time}\}$

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 are actually 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:

Maybe write problem names in italics

- PLS (Polynomial Local Search): the class of search problems designed to model the process of finding the local optimum of a function or alteratively 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 $\mathsf{SOPL} = \mathsf{PLS} \cap \mathsf{PPADS}$ [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 CONTINUOUS-LOCALPOINT problem. It has been proven that $\mathsf{CLS} = \mathsf{EOPL} = \mathsf{PLS} \cap \mathsf{PPAD}$ [FGH+22; GHJ+22a], where EOPL is the class of search problems that are polynomial-time reducible to the END-OF-POTENTIAL-LINE problem.

$$\begin{tikzcd}
 & \mathsf{FNP} & \\
 & \mathsf{TFNP} \arrow[u] & \\
\mathsf{PLS} \arrow[ur] & \mathsf{PPP} \arrow[u] & \mathsf{PPA} \arrow[ul] \\
 & \mathsf{PPADS} \arrow[u] & \\
\mathsf{SOPL} \arrow[uu] \arrow[ur] & & \mathsf{PPAD} \arrow[uu] \arrow[ul] \\
 & \mathsf{CLS} \arrow[ul] \arrow[ur] & \\
 & \mathsf{FP} \arrow[u] &
\end{tikzcd}$$

Figure 2.1. 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 very simple combinatorial principles. 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 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 $\mathsf{FP} \neq \mathsf{TFNP}$, answering the question P vs. NP. By 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).

2.4 White-box TFNP

In computer science and engineering, systems and models are divided in two categories: white-box systems and black-box system. A system is said to be a **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: in a **white-box TFNP** problem we're interested in how the problem gets verified (or computed if it's also in FP), while a **black-box TFNP** problem we're interested only in the verifiability (or computability) of the problem. In recent years, these two TFNP models have been formalized through the use of *protocols* and *decision trees*. In this section we will briefly discuss protocols and the white-box model, while decision trees and the black-box model will be extensively discussed in the following chapter.

A *protocol* is an algorithm that generates communications between two parties, namely Alice and Bob, who cooperate in order to achieve a common objective, like computing a function.

Definition 2.10 ([RYM+22])**.** Let X be Alice's input set and let Y be Bob's input set. A protocol π is a rooted directed binary tree whose leaves are associated to outputs and internal nodes are owned by either Alice or Bob, where the owner of v is noted by $\text{owner}(v)$. 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 $\text{owner}(v) = \text{A}$ and $Z = Y$ if $\text{owner}(v) = \text{B}$. 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 by $\pi(x, y)$.

A function f is said to be computed by the protocol π if $f = \pi$. By their own definition, protocols are also Turing complete since they are nothing more then an algorithm computed by two parties instead of one. A protocol encodes all possible messages that may be sent by the parties during any potential conversation, producing a conversation only when it is executed using a particular input. Since a protocol always returns an answer for all possible inputs, any protocol is *total*.

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 inputs.

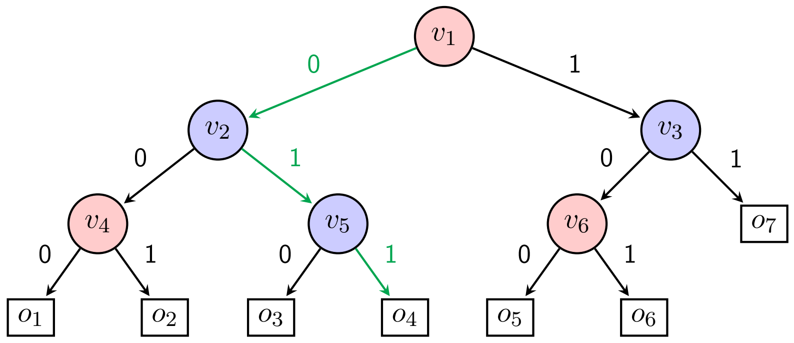

Figure 2.2. An example of a protocol of size 13 and depth 3 where the red nodes are owned by Alice and the blue nodes are owned by Bob. The computation, shown by the green path, for the inputs (x, y) is given by $f_{v_1}(x) = 0$, $f_{v_2}(y) = 1$ and $f_{v_5}(y) = 1$

Protocols are clearly white-boxes capable of solving search problems since every step of the computation is known, even though the way they are defined implies that for a given protocol any input will always return the same output due to them being a deterministic computation, losing the idea of an input being capable of having more solutions. However, they are still a valid way to solve search problems, making them a valid tool to model white-box TFNP [RGR22; BFI23].

Definition 2.11. 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".

In particular, given a TFNP problem R, we denote with R^{cc} the equivalent TFNP^{cc} problem, where *cc* stands for *communication complexity*.

Definition 2.12. 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 π_n such that $\pi_n(x, y) = z$ if and only if $((x, y), z) \in R_n$. Likewise, 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 $((x, y), z) \in R_n$.

Additionally, the concept of reduction also applies for communication search problems, even though it requires a pre-fixed value t of maximum amount of bits usable in the reduction, i.e. the maximum depth of the reduction protocol.

Definition 2.13. 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 reducible into a search problem $S = (S_n)_{n\in\mathbb{N}}$, namely $R \leq S$, 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.

Nonetheless, this allows us to define the TFNP^{cc} hierarchy in the same way as the standard one, making communication complexity a valid system capable of characterizing white-box TFNP problems.

One of the most interesting properties of communication search problems is the ability to be characterized by a single type of search problem: the *monotone Karchmer-Widgerson game.*

Definition 2.14. 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 $x_i \neq y_i$.

Add description for [n] in preliminaries

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 $x_i < y_i$.

In fact, it has been proven that any communication search problem is equivalent to the monotone KW game of some Boolean function [Gál02; GKR+19].

Lemma 2.1. *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 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.*

This result implies that TFNP^{cc} actually coincides with the **study of the monotone Karchmer-Widgerson game**. Moreover, in 1990 Karchmer and Widgerson [KW88] showed that the complexity of these games is equal to the complexity of Boolean circuits solving the associated function, i.e. a circuit capable of computing a the function through logic gates.

Definition 2.15 ([RYM+22])**.** A Boolean circuit is a directed acyclic graph whose nodes, called gates, are associated with either input variables or Boolean operators. Each gate has an out-degree equal to 1 (except for the output gate who has out-degree 0) and in-degree equal to either 0 or 2. All 0 in-degree gates correspond to input variables, the negations of input variables or constant bits, while all 2 in-degree gates compute the logical AND or the logical OR of its given input variables or Boolean function. Each gate v is associated with the Boolean function f_v computed by it.

In particular, Boolean circuits have been proven to be *Turing-complete* [Sip96], meaning that they are capable of processing any computable function. In fact, every modern computer is actually just a bunch of Boolean circuits wired together. Moreover, Turing machines and circuits are capable of simulating each other up to a polynomial factor, so any polynomial complexity computation achievable through one can also be achieved through the other, making circuits a valid white-box system for analyzing TFNP problems.

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. The **circuit complexity** of function f is defined as the size of the smallest Boolean circuit that computes it.

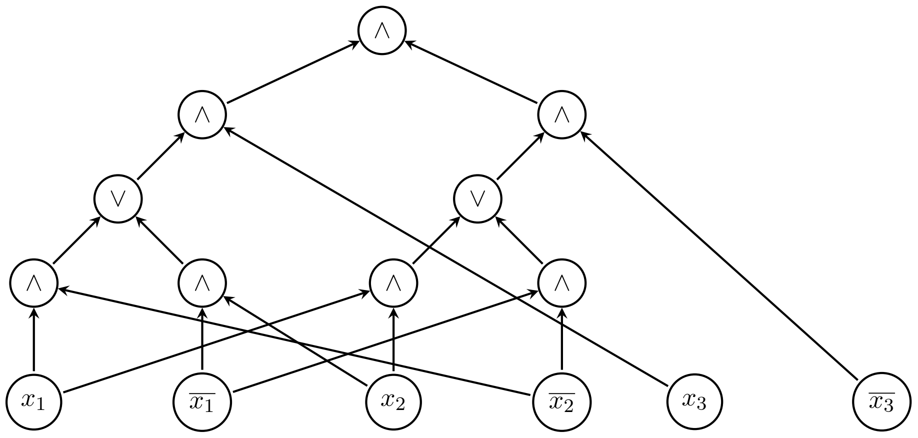

Figure 2.3. A Boolean circuit of size 15 and depth 4 computing $x_1 \oplus x_2 \oplus x_3$. The top gate is the output gate.

Theorem 2.2. *Given a function $f : \{0,1\}^n \to \{0,1\}$, there exists a circuit of depth d that computes f if and only if there exists a protocol of depth d that solves* $\mathrm{KW}(f)$. *Moreover, if f is monotone, the circuit is monotone and the protocol solves* $\mathrm{mKW}(f)$

Should be proven?

With this result, we conclude that TFNP^{cc} can be described in three ways: the study of communication search problems, the study of the monotone Karchmer-Widgerson game and the study of monotone Boolean circuits. This further extends the well-known connections between search problems, communication complexity and circuit complexity, establishing that any result obtained in one of these fields can be in some way extended to the others.

Add TFNP^{cc} hierarchy relative to circuits

Chapter 3

Black-box TFNP

3.1 Oracles and decision trees

In the previous chapter, we have discussed how TFNP subclasses are defined in terms of basic existence principles which can be converted into white-box total search problems solvable by protocols that are reducible to the Karchmer-Widgerson game. In this chapter, instead, we will extensively discuss the **black-box model**.

The difference between the white-box and black-box TFNP models can be formally described as the difference between problems verifiable (or computable) by a simple Turing machine or by a Turing machine equipped with an **oracle**.

Definition 3.1. An oracle for a problem A is an external device that is capable of instantaneously verifying a solution for of such problem. An oracle Turing machine is a Turing machine provided with the ability of querying an oracle. We write M^A to describe an oracle Turing machine provided with an oracle for the problem A.

By definition, it's easy to see that an oracle is nothing more than a black-box device. In particular, an oracle for a decision problem is capable of determining if an input object has the the required property, while an oracle for a search problem is capable of determining if an output is the solution for the associated problem with the given input. Any query to the oracle made by the Turing machine has a complexity of $\Theta(1)$, meaning that it doesn't affect the cost of the computation. This allows us to give the following definition of query search problem [RGR22; BFI23].

Definition 3.2. 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".

Additionally, oracles provide a simple yet effective way to generalize the concept of reduction, 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 the problem B. The idea behind this kind of reductions is simple: if M^B can solve A then any query to the oracle can be replaced with a call to a subroutine that solves B. Many-to-one reductions can be seen as restricted variants of Turing reductions where the number of calls made to the subroutine of problem B is exactly one and the value returned by the reduction is the same value as the one returned by the subroutine.

More generally, given a class $\mathcal{C}$ and an oracle for a problem A, the *relativized version* of the class $\mathcal{C}$ is the set of all problems of $\mathcal{C}$ verifiable (or solvable) with access to the oracle of A. Obviously, this definition implies that $\mathcal{C} \subseteq \mathcal{C}^A$ for all oracles A since any problem that is already in $\mathcal{C}$ can just ignore the oracle. In the particular case of TFNP, it was proven that the relation between each total search problem is strictly connected to the relation of the relativized versions of their classes [BCE+98].

Theorem 3.1. *Given two search problems $R, S \in$ TFNP and their relative classes it holds that $R \subseteq S$ if and only if $R^A \subseteq S^A$ for all oracles A.*

This result states 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. Through these relativized separations, many TFNP subclasses have been proven to be different.

Moreover, the use of oracles allows us to model total search problem through the lens of **decision trees**.

Definition 3.3 ([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 and how they do it, we only care about the result being either a 0 or a 1 in order to proceed with the computation. In fact, like their white-box counterpart, a decision tree encodes 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*.

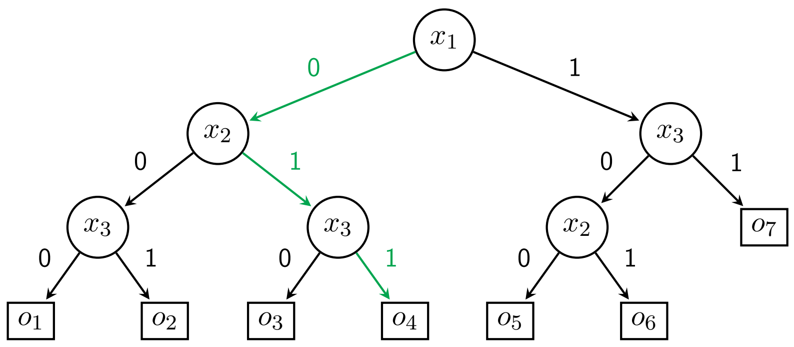

Figure 3.1. An example of decision tree of size 13 and depth 3. The computation, shown by the green path, for the input $x \in \{0,1\}^n$ is given by $x = 011$

Decision trees give an easier way to describe the computation of an oracle Turing machine: if M^B verifies (or solves) 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 is actually a decision tree.

Proposition 3.1. *If there is an oracle Turing machine M^B that verifies (or solves) a problem A then there is a decision tree that verifies (or solves) A.*

Not sure if it's an if and only if, possibly yes

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.* As in the communication complexity formulation described for the white-box model, given a TFNP problem R, we denote with R^{dt} the equivalent TFNP^{dt} problem, where dt stands for *decision tree* [RGR22; BFI23].

Definition 3.4. 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 T_y such that $T_y(x) = 1$ if and only if $(x, y) \in R_n$.

Furthermore, we also introduce the concept of decision tree reductions. Differently from classic and communication search problems, these 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 $\{0, 1\}$. This definition will allow us to prove some following results in a more convenient way. An analogous formulation can be obtained by computing this function through a simple decision tree.

Definition 3.5. 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 $S = (S_n)_{n\in\mathbb{N}}$, namely $R \leq S$, 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 family of decision trees $T_i : \{0,1\}^m \to \{0,1\}$ for each $i \in [n]$ 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 $T(x) := (T_1(x), \ldots, T_n(x))$. In other words, the decision trees $T_1, \ldots, T_n$ map inputs of R into inputs of S, while the decision tree g maps solutions of S into solutions of R.

The difference in notation between $T_1, \ldots, T_n$ and T^o_y underlines the fact that the former give a $\{0, 1\}$ output, while the latter gives a more 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, including each T_i and each T^o_y. The **complexity of the reduction** is given by $\log n + d$. Moreover, we denote as $S^{dt}(R)$ the minimum complexity of any decision tree reduction from R to S.

Using this definition, we can define complexity classes of total query search problems via decision tree reductions: given a total query search problem $S = (S_n)_{n\in\mathbb{N}}$, we re-define the subclass of problems **efficiently reducible to** S as:

$$S^{dt} := \{R : S^{dt}(R) = \text{polylog}(n)\}$$

where $R = (R_m)_{n\in\mathbb{N}}$.

3.2 Connections to Proof Complexity

Like in the white-box case, the black-box TFNP model can be studied under multiple lenses. In particular, a well known result extensively discusses even before the rise of the black-box model is the connection between total query search problems and **propositional proof complexity** [BFI23; GHJ+22b; RGR22].

Add proof complexity discussion here or in preliminaries

It's easy to see that any CNF formula gives rise to an associated search problem: finding an unsatisfied clause inside the formula (if there is any).

Definition 3.6. Given the CNF $F = C_1 \wedge \ldots \wedge C_m$ over n variables, we define $\text{Search}(F)$ as the following associated search problem: given an input assignment $\alpha(x_1, \ldots, x_n)$, return the index of an unsatisfied clause (if there is any).

In particular, when F is an unsatisfiable CNF formula, $\text{Search}(F)$ is clearly a *total search problem* since for any input assignment there will always be an unsatisfied clause. In a similar fashion, 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, denoted with 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. We notice that, by De Morgan's theorem, $\overline{D_T}$ is a CNF.

Proposition 3.2. *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 $R_n = \text{Search}(F_n)$. 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 $\text{polylog}(n)$-depth decision tree T_y that verifies R_n. Consider the CNF $F_n := \bigwedge_{y \in O_n} \overline{D_{T_y}}$. 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 $D_{T_y}(x) = 1$ and therefore $\overline{D_{T_y}}(x) = 0$, concluding that F_n is unsatisfiable. Moreover, this formulation also concludes that:

$$(x, y) \in R_n \iff T_y(x) = 1 \iff \overline{D_{T_y}}(x) = 0 \iff (x, y) \in \text{Search}(F_n)$$

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

□

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

Through this connection, Göös et al. [GKR+19] showed that many important proof systems are characterized by an associated 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:

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

where size(Π) is the *size* of Π in P, i.e. the sum of the sizes of all the formulas inside it or in other words the total number of symbols in Π, and deg(Π) is the *degree* of Π associated to P, which varies from proof system to proof system. This degree measure will be specified for the proof systems used in following sections.

Since each TFNP^{dt} problem is equivalent to the false clause search problem of a family F of formulas, the complexity parameter defined above can be used to define another characterization of TFNP^{dt} problems.

Definition 3.7. We say that a proof system P **characterizes** a TFNP^{dt} problem R (and reflexively that R characterizes P) if it holds that

$$R^{dt} = \{\text{Search}(F) : P(F) = \text{polylog}(n)\}$$

where $F = F_1, F_2, \dots$ is a family of formulas. In a stronger sense, it holds that $R^{dt}(\text{Search}(F)) = \Theta(P(F))$.

Most of the TFNP subclasses discussed in previous sections has been shown to have a characterizing proof system. References and proofs to these characterizations can be found in [GHJ+22b; BFI23].

- $\text{PLS}^{dt}(\text{Search}(F)) = \Theta(\text{TreeRes}(F))$
- $\text{PPA}^{dt}(\text{Search}(F)) = \Theta(\mathbb{F}_2\text{–NS}(F))$
- $\text{PPADS}^{dt}(\text{Search}(F)) = \Theta(\text{unary} - \text{NS}(F))$
- $\text{PPAD}^{dt}(\text{Search}(F)) = \Theta(\text{unary} - \text{SA}(F))$
- $\text{SOPL}^{dt}(\text{Search}(F)) = \Theta(\text{RevRes}(F))$
- $\text{CLS}^{dt}(\text{Search}(F)) = \Theta(\text{RevResT}(F))$
- $\text{FP}^{dt}(\text{Search}(F)) = \Theta(\text{TreeRes}(F))$

3.3 Canonical Proof Systems

In an intuitive way, this characterization also shows that *any* TFNP^{dt} problem can be transformed into a proof system for refuting unsatisfiable CNF formulas of polylogarithmic width: since any 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 $T_i : \{0,1\}^{n'} \to \{0,1\}$. We refer to $C(T)$ as the CNF formula obtained by substituting each variable x_i in C with the associated tree T_i and rewriting the result as a CNF, or more conveniently:

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

Definition 3.8. 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 $T' = (T^o_j)_{j\in[m_F]}$, where $T_i : \{0,1\}^{n_H} \to \{0,1\}$ and $T^o_j : \{0,1\}^{n_H} \to [m_H]$, such that given the following formula:

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

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 $\overline{p}$ inside $C_i(T)$. Each clause $C_i(T) \vee \overline{p}$ must be either tautological (since it could contain a variable and its negation) or a weakening of the corresponding clause of H 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 Search(H) to Search(F) induces the search problem Search(F_H).

Reductions between CNF formulas imply that reductions between search problems reduction are actually a proof system. In particular, given a problem Search$(F) \in \mathsf{TFNP}^{dt}$, the **canonical proof system** of such problem, denoted with P_F proves 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*, or *depth* in this case, is given by the maximum depth among the involved decision trees. Hence, the P_F complexity of H is given by:

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

These 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 be done polynomially in size of the proof.

From this definition of canonical proof system, the following theorem is given for free. This theorem plays a crucial role in TFNP^{dt} characterization through proof complexity, stating that P_F has a short proof of H if and only if $\mathrm{Search}(H)$ efficiently reduces to $\mathrm{Search}(F)$.

In particular, we observe that **any characterizing proof system** is actually the canonical proof system of the associated search problem, a result that follows from the two definitions. In other words, proving a formula in a characterizing proof system automatically gives a reduction to the corresponding complete search problem.

Theorem 3.2. *Let* $\mathrm{Search}(F) \in \mathsf{TFNP}^{dt}$ *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* $\mathrm{Search}(H)$ *has a size $O(s)$ and depth d reduction to S_F*
2. *If* $\mathrm{Search}(H)$ *has a size s and depth d decision tree reduction to* $\mathrm{Search}(F)$ *then H has a size $s2^{O(d)}$ and depth d proof in P_F*

In particular, this implies that $\mathrm{Search}(F)^{dt}(\mathrm{Search}(H)) = \Theta(P_F(H))$.

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

Vice versa, suppose that $T = (T_i)_{i \in [n_F]}$ and $T' = (T^o_j)_{j \in [m_F]}$ is a decision tree reduction from $\mathrm{Search}(H)$ to $\mathrm{Search}(F)$ 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^o_i. If the formula $C_i(T) \lor \overline{p}$ is tautological, then it can be ignored since F_H is a CNF. Otherwise, let x be an assignment that falsifies $L \lor \overline{p}$. Then, it holds that $T_1(x), \dots, T_{n_F}(x)$ falsifies $C_i(T)$ and that $T^o_i(x)$ follows path p. Thus, the $T_i(x)$-th clause of $\overline{H}$ must also be false, implying that $L \lor \overline{p}$ 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 $s2^{O(d)}$.

□

Chapter 4

Parity in black-box TFNP

4.1 Parity decision trees and Tree-like Res($\oplus$)

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 being linear equations 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 4.1. Given n variables $x_1, \dots, x_n$, we define a **linear form** as a linear equation over $\mathbb{F}_2$. In general, a linear form can be expressed as $\sum\limits_{i=1}^{n} \alpha_i x_i$, where $\alpha_1, \dots, \alpha_n \in \mathbb{F}_2$

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 this 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 implicitly tells us that $x_2 = 1$ and that $x_1 = x_4 \neq x_3$.

But what happens if we apply the concept of parity in decision trees? What if, instead of querying variables in order 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 labeled 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 Φ_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.

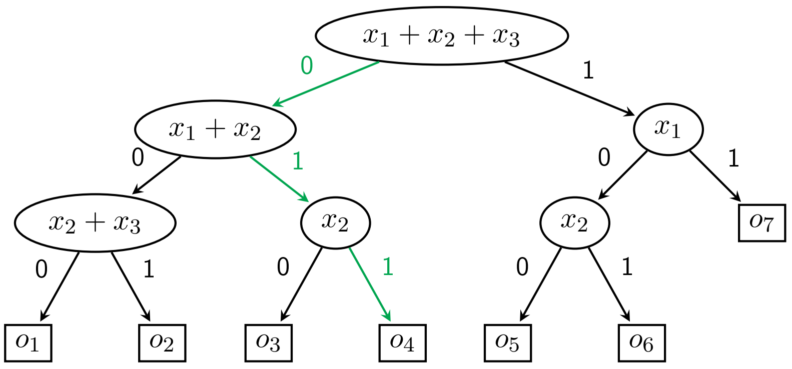

Figure 4.1. An example of parity decision tree of size 13 and depth 3.

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 $x_0 = 0, x_2 = 1, x_3 = 1$. Since a system of linear forms can have multiple solutions, many assignments could actually be mapped to the same output. However, some systems could also be unsatisfiable, meaning that the node cannot be reached by any assignment. When this happens we say that the node is **degenerate**.

Intuitively, every PDT can be converted into a normal decision tree simply by "splitting" each linear query in more queries, a process that exponentially increases the size of the tree. In fact, PDTs tend generally to be more compact that normal decision trees, even though this isn't usually true for simple problems. We define the class FP^{pdt} as the set of TFNP^{dt} problems that are efficiently solvable by a PDT.

Definition 4.2. We define 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 $(x, y) \in R_n$.

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 Φ_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 Φ_v^T has only one solution and it also satisfies C

We observe that if a node doesn't meet any of these 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 4.1. *Let F be an unsatisfiable CNF formula. If* $\mathrm{Search}(F)$ *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 $\mathrm{Search}(F)$. Let U be the set of degenerate nodes of T. Notice that since Φ_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 Φ_s^T must be satisfiable: if this wasn't true then both Φ_s^T and Φ_u^T would be unsatisfiable, which can only be true if Φ_p^T is also unsatisfiable, but we chose w as the node in U with minimal distance. Since Φ_s^T is satisfiable, the label $f = \alpha$ on the edge (p, s) must be already contained inside the system Φ_p^T, meaning that each assignment that satisfies Φ_p^T also satisfies Φ_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 the 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 Φ_p^T also satisfies $\Phi_v^{T'}$, concluding that T' also solves $\mathrm{Search}(F)$.

By repeating the process until U is empty, we get a non-degenerate PDT that solves $\mathrm{Search}(F)$ of size at most s and depth at most d. $\square$

Now, we are interested in finding a canonical proof system that can characterize our brand new class of problems. Consider a generic system of linear forms Φ. This system can be viewed as the conjunction of the linear forms that it describes:

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

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 say that such set of disjunction is a **linear clause**. More generally, a *linear CNF formula* is a conjunction of linear clauses.

Definition 4.3. A linear CNF formula is a conjunction of m disjunctions of linear equations over $\mathbb{F}_2$.

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

Generally, linear CNF formulas can assume a complex structure, such as the following:

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

Moreover, any standard CNF formula can be described as a linear CNF formula simply by treating each clause as a disjunction of linear forms made of a single term. For example, the CNF $(x_1 \vee \overline{x_2}) \wedge (\overline{x_3} + x_1)$ 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. Once we have defined a way to treat CNF formulas as linear forms, we are now ready to define a new proof system. We define the **parity resolution** proof system, noted with $\mathsf{Res}(\oplus)$, by the following two rules:

- *Cut*: given two linear clauses $(f = 0) \vee C$ and $(f = 1) \vee D$, we can derive the linear clause $C \vee D$
- *Weakening*: given a linear clause C, we can derive any linear clause D such that $C \implies D$.

Similarly to normal resolution, in Res(⊕) 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. Furthermore, each clause in a derivation is used at most once we say that the derivation has a *tree-like* structure.

By definition, the weakening rule makes this proof system powerful since semantical implications can be used in many forms. 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 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 cut rule, we get the following tree-like refutation:

$$\begin{tikzcd}
(x = 1) \arrow[dddr, dash] & (x + y = 1) \arrow[ddr, dash] & & (x = 0) \vee (y = 1) \arrow[d, dash] \\
 & & & (x = 0) \vee (x + y = 0) \arrow[dl, dash] \\
 & & (x = 0) \arrow[dl, dash] & \\
 & \perp & &
\end{tikzcd}$$

Parity decision trees and tree-like parity resolution can be viewed as two sides of the same coin. In fact, any PDT can be used to construct a tree-like Res(⊕) refutation and vice versa while maintaining (almost) the same size and depth **??**.

Lemma 4.1. *Let F be an unsatisfiable linear CNF formula. If there is a PDT that solves* Search(F) *of size s and depth d, there also exists a tree-like* Res(⊕) *refutation of F of size at most $2s$, depth at most $d + 1$ and weakening rule applied only to the leaves.*

Proof. Let T be a PDT of size s and depth d that solved $\text{Search}(F)$. By Proposition 4.1, 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 $\neg\Phi_v^T$. Clearly, every node is a result of the cut rule being applied on it's children, where the root node is the empty clause.

Since T is a PDT that solves $\text{Search}(F)$, 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 a 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.

□

Lemma 4.2. *Let F be an unsatisfiable linear CNF formula. If there is a tree-like* $\mathsf{Res}(\oplus)$ *refutation of F of size s and depth d, there also exists a PDT that solves* $\text{Search}(F)$ *of size at most s and depth at most d.*

Proof. Let T be the proof tree that refutes F. We label each edge of T whose associated clauses involve a cut rule, while all the other weakening edges remain unlabeled. In particular, if a resolution rule is applied to the clauses $(f = 0) \vee D_1$ and $(f = 1) \vee D_2$ obtaining the clause $D_1 \vee D_2$, 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 vertex of T, we show that the clause written in v contradicts the system Φ_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 cut rule application, where $D_1 \vee D_2$ is the clause inside v. Assume that u is the node that contains $(f = 0) \vee D_1$ while w contains $(f = 1) \vee D_2$. By inductive hypothesis, we know that $D_1 \vee D_2$ contradicts the system Φ_v^T and equivalently that the system $\neg(\neg D_1 \wedge \neg D_2)$ contradicts Φ_v^T. This means that se of equalities in D_1 contradict Φ_v^T. Moreover, we know that $\Phi_u^T = \Phi_v^T \wedge (f = 1)$, concluding that $(f = 0) \vee D_1$ contradicts Φ_u^T. Likewise, we can show that $(f = 1) \vee D_2$ contradicts Φ_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 Φ_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 $\text{Search}(F)$. Due to this final step, the size of the PDT is at most s and its depth is at most d.

□

By these two lemmas, it's easy to see that tree-like parity resolution is a proof system capable of characterizing the class FP^{pdt}.

Corollary 4.1. $\mathsf{FP}^{pdt}(\text{Search}(F)) = \Theta(\mathsf{TreeRes}(\oplus)(F))$

4.2 Nullstallensatz over $\mathbb{F}_2$

4.3 $\text{NS}{-}\mathbb{F}_2$ simulates $\text{TreeRes}(\oplus)$

Chapter 5

Notes

Tree-like resolutions for an unsatisfiable CNF formula are strictly connected to the decision trees that solve its associated search problem. In particular, it can be proven that the smallest tree-like refutation has the exact same structure of the smallest decision tree.

Lemma 5.1. *[BGL13] Let F be an unsatisfiable CNF formula. If there is a tree-like refutation of F with structure T, there also exists a decision tree with structure T that solves* Search(F)

Proof. We procede by induction on the size s of the refutation of F.

Let $F = C_1 \wedge \ldots \wedge C_m$. If $s = 1$, then the refutation is made up of only one step that ends with the empty clause, implying that $\exists i \in [m]$ such that $F = C_i = \bot$. Hence, Search(F) can be solved by the decision tree made of only one vertex labeled with i.

We now assume that every formula with a tree-like refutation with a structure of size s there exists a decision tree with the same structure that solves the search problem associated with the formula.

Suppose now that the size s of the refutation is bigger than 1. Let x be the last variable resolved by the refutation and let T_0 and T_1 be the subtrees of T such that x is the root of T_0 and $\overline{x}$ is the root of T_1.

Consider now the formulas $F\restriction_{x=0}$ and $F\restriction_{x=1}$, respectively corresponding the formula F with the value 0 or 1 assigned to x. It's easy to see that the subtrees T_0 and T_1 are valid refutations of the formulas $F\restriction_{x=0}$ and $F\restriction_{x=1}$: if $b = 0$, then x evaluates to 0, otherwise if $b = 1$ then $\overline{x}$ evaluates to 0.

Since T_0 and T_1 have size $s - 1$, by inductive hypothesis there exist two decision tree with structure T_0 and T_1 that solve Search$(F\restriction_{x=0})$ and Search$(F\restriction_{x=1})$.

Finally, the search problem Search(F) can be solved by the decision tree that queries x and proceeds with the decision tree T_b based on the value $b \in \{0, 1\}$ such that $x = b$.

□

Definition 5.1. Given two rooted trees T and T', we say that T is embeddable in T'

if there exists a mapping $f : V(T) \to V(T')$ such that, for any vertices $u, v \in V(T)$, if u is a parent of v in T then $f(u)$ is an ancestor of $f(v)$ in T'.

Lemma 5.2. *[BGL13; LNN+95] Let F be an unsatisfiable CNF formula. If there is a decision tree with structure T that solves* Search(F)*, there also exists a tree-like refutation of F with structure T' such that T' is embeddable in T.*

Proof. The main idea is to associate inductively, starting from the leaves, a clause to each vertex of T in order to transform T in a tree-like refutation of F. In particular, each vertex v gets associated to a clause $C(v)$ such that every input of the decision tree that reaches v falsifies $C(v)$.

Let $F = C_1 \wedge \ldots \wedge C_m$. For all $i \in [m]$, we associate the clause C_i to the leaf of T labeled with i. This constitutes our base case.

Consider now a vertex v that isn't a leaf. Let x be the variable that labels v and let u_0, u_1 be the vertices such that the edge (v, u_0) is taken if $x = 0$ and the edge (v, u_1) is taken if $x = 1$. By induction, assume that u_0 and u_1 have already been associated with the clauses C_0 and C_1.

By way of contradiction, suppose that C_0 contains the literal $\overline{x}$. Then, since in a decision tree each variable can be queried only once in every path, there will always be an input with $x = 0$ that reaches v. Since $x = 0$ and since C_0 contains $\overline{x}$, this input would satisfy C_0, contradicting the fact that C_0 was associated to u_0 in a way that it is falsified by every input.

Thus, the only possibility is that C_0 can't contain the literal $\overline{x}$. Similarly, we can show that C_1 can't contain the literal x. This leaves us with only two possibilities: either $C_0 = x \vee \alpha$ and $C_1 = \overline{x} \vee \beta$ or one of C_0, C_1 doesn't contain $x, \overline{x}$.

In the first case, we can simply associate to v the clause $C = \alpha \vee \beta$. In the second case, we associate to v the clause that doesn't contain $x, \overline{x}$ (chose any of them if both clauses do not contain $x, \overline{x}$).

In particular, we notice that the first case directly emulates the resolution rule, while the second case essentially represent "redundant steps". By "skipping" these redundant steps, we can obtain a tree T' that is embeddable in T and that contains only nodes on which the first case was applied. Finally, it's easy to deduce that the root node of T' will always be associated with the empty clause $\perp$, concluding that T' is the structure of a tree-like refutation of F.

□

Theorem 5.1. *Let F be an unsatisfiable CNF formula. The smallest tree-like refutation of F has size s and depth d if and only if the smallest decision tree solving* Search(F) *has size s and depth d.*

Proof. Let s and d be the size and depth of the smallest tree-like refutation of F. Likewise, let x and y be the size and depth of the smallest decision tree solving Search(F).

Then, by Lemma 5.1, we know that there exists a decision tree that solved Search(F) with the same structure of the smallest refutation. Let α and β be the size and depth of this decision tree. It's easy to see that $s = \alpha \geq x$ and $d = \beta \geq y$.

Viceversa, by Lemma 5.2, we know that there exists a tree-like refutation of F such that its structure is embeddable in the one of the smallest decision tree. Let γ and β be the size and depth of this tree-like refutation. Since the latter is embedded in the smallest decision tree, it's structure must be smaller or equal. Hence, it's easy to see that $x \geq \gamma \geq s$ and $y \geq \delta \geq d$. Thus, we can conclude that $s = x$ and $d = y$.

□

Note: [RGR22] says that this theorem should be generalizable to each tree and not only for the smallest trees (doubt this is true)

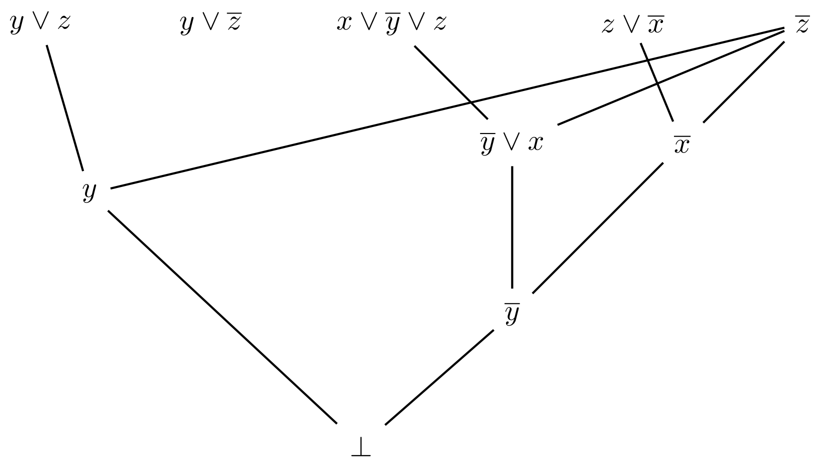

Figure 5.1. Dag-like refutation of the previous formula

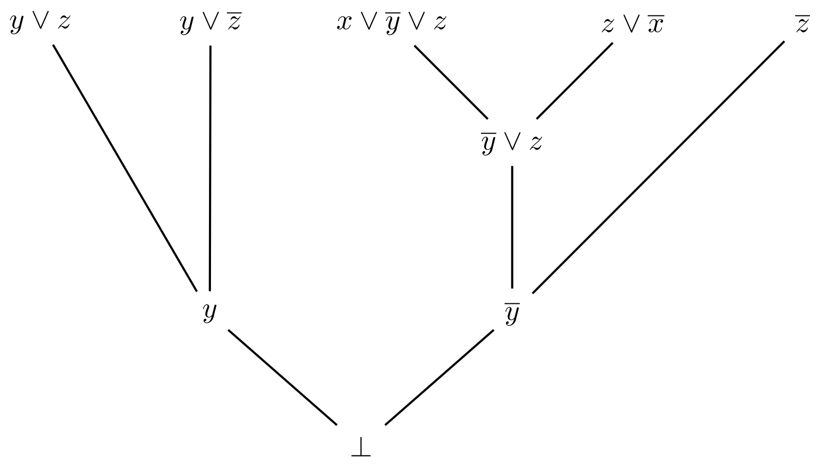

Figure 5.2. Tree-like refutation of the previous formula

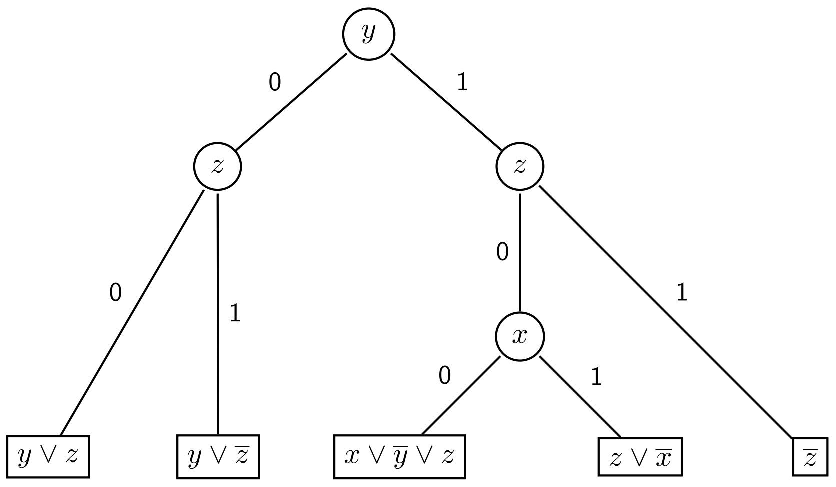

Figure 5.3. Decision tree for the previous formula

5.1 The $\frac{1}{3}, \frac{2}{3}$ lemma

Definition 5.2. Given a tree T and a node v, we denote as T_v the subtree of T having v as its radix.

Lemma 5.3 (Lewis' $\frac{1}{3}, \frac{2}{3}$ lemma [**1-3_2-3**])**.** *If T is a binary tree of size $s > 1$ then there is a node v such that the subtree T_v has size between $\left\lfloor \frac{1}{3}s \right\rfloor$ and $\left\lceil \frac{2}{3}s \right\rceil$.*

Proof. Let r be the radix of T and let ℓ be a leaf of T with the longest possible path $r \to \ell$. Let $v_1, \dots, v_k$ be the nodes of such path, where $r = v_1$ and $\ell = v_k$. For each index i such that $1 \leq i \leq k$, let $a_i b_i$ be the two children of v_i.

Claim 5.3.1. For any index i, if T_{v_i} has size at least $\left\lfloor \frac{1}{3}s \right\rfloor$ then for some index j, where $i \leq j \leq k$, it holds that T_{v_j} has size between $\left\lfloor \frac{1}{3}s \right\rfloor$ and $\left\lceil \frac{2}{3}s \right\rceil$.

Proof of the claim. If T_{v_i} has also size less than $\left\lceil \frac{2}{3}s \right\rceil$ then we are done. Otherwise, since $T_{v_i} = \{v_i\} \cup T_{a_i} \cup T_{b_i}$, one between the subtrees T_{a_i}, T_{b_i} must have size at least $\frac{1}{2} \lceil 2 \rceil 3s - 1$, meaning that it has size at least $\left\lfloor \frac{1}{3}s \right\rfloor$. If this subtree has also a size at most $\left\lceil \frac{2}{3}s \right\rceil$ then we are done. Instead, if this doesn't hold for both subtrees, we can repeat the process (assuming that $v_{i+1} := a_i$ without loss of generality) since we know that $T_{v_{i+1}}$ has size greater than $\left\lfloor \frac{1}{3}s \right\rfloor$.

By way of contradiction, suppose that this process never finds a subtree with size at most $\left\lceil \frac{2}{3}s \right\rceil$. Then, this would mean that it also holds for $v_k = \ell$. However, since ℓ is a leaf, we know that T_{v_ℓ} must have size 1, which is definitely at most $\left\lceil \frac{2}{3}s \right\rceil$ for any value of s, giving a contradiction. Thus, there must be a node that terminates the process.

□

Since $T_{v_1} = \{r\} \cup T_{a_1} \cup T_{b_1}$, we know that for both of these subtrees must have at least $\left\lfloor \frac{1}{3}s \right\rfloor$. Thus, assuming that $a_1 = v_2$, the claim directly concludes the proof.

□

5.2 Nullstellensatz

Definitions taken from [**Nullstellensatz**]

Definition 5.3 (Hilbert's Nullstellensatz)**.** Given the polynomials $p_1, \dots, p_m \in \mathbb{F}[x_1, \dots, x_n]$, the equation $p_1 = \dots = p_m = 0$ is unsolvable if and only if $\exists g_1, \dots, g_m \in \mathbb{F}[x_1, \dots, x_n]$ such that $\sum\limits_{i=1}^{m} g_i p_i = 1$.

Hilbert's Nullstellensatz can be used to define the following proof system:

Definition 5.4 (Nullstellensatz Refutation)**.** Given the set of polynomial equations $P = \{p_1 = 0, \dots, p_m = 0\}$ over $\mathbb{F}[x_1, \dots, x_n]$, where $\mathbb{F}$ is any field, a Nullstellensatz refutation is a set of polynomials $\pi = \{g_1, \dots, g_n\} \subseteq \mathbb{F}[x_1, \dots, x_n]$ such that $\sum_{i=1}^{m} g_i p_i = 1$.

The set of polynomials $P = \{p_1, \dots, p_n\}$ is called the axiom set and the set $\pi = \{g_1, \dots, g_n, h_1, \dots, h_m\}$ is called proof of P.

By also adding the polynomial equations $x_1^2 - x_1 = 0, \dots, x_n^2 - x_n = 0$ to the set of axioms, the NS proof system is sound and complete for the set of unsatisfiable CNF formulas. Thus, in general, given the set of axioms $P = \{p_1 = 0, \dots, p_m = 0, x_1^2 - x_1 = 0, \dots, x_n^2 - x_n = 0\}$, we say that $\pi = \{g_1, \dots, g_m, h_1, \dots, h_n\}$ is a CNF proof of P if:

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

For any proof $\pi = \{g_1, \dots, g_n, h_1, \dots, h_m\}$ of the axioms $P = \{p_1, \dots, p_n\}$, we define the *degree of* π as:

$$\deg(\pi) = \max\{\deg(g_i p_i), \deg(h_j) + 2 \mid 1 \leq i \leq n, 1 \leq j \leq m\}$$

If P has a proof π of degree $\deg(\pi) = d$ then we say that $P \vdash_d^{\mathsf{NS}} 1$.

Proposition 5.1. *Given a set of axioms P, if $P \vdash_d^{\mathsf{NS}} q$ then $P, 1 - q \vdash_d^{\mathsf{NS}} 1$*

Proof. Since $P \vdash_d^{\mathsf{NS}} q$, we know that $\exists g_1, \dots, g_m, h_1, \dots, h_n \in \mathbb{F}[x_1, \dots, x_n]$ such that:

$$\sum_{i=1}^{m} g_i p_i + \sum_{j=1}^{n} h_j (x_j^2 - x_j) = q$$

where $\deg(q) = d$.

Let $p_{m+1} := 1 - q$ and $P' = P \cup \{p_{m+1} = 0\}$. We define $g'_1, \dots, g'_m, g'_{m+1}$ as:

$$g'_i = \begin{cases} 1 & \text{if } i = m + 1 \\ g_i & \text{otherwise} \end{cases}$$

With simple algebra we get that:

$$\sum_{i=1}^{m+1} g'_i p_i + \sum_{j=1}^{n} h_j (x_j^2 - x_j) = g'_{m+1} p_{m+1} + \sum_{i=1}^{m} g'_i p_i + \sum_{j=1}^{n} h_j (x_j^2 - x_j) = (1 - q) + q = 1$$

thus $\pi = \{g'_1, \dots, g'_{m+1}, h_1, \dots, h_n\}$ is a proof of P. Moreover, since $\deg(q) = d$ implies that $\deg(g'_{m+1} p_{m+1}) = d$, it's easy to see that $\deg(\pi) = d$ holds, concluding that $P, 1 - q \vdash_d^{\mathsf{NS}} 1$

□

Lemma 5.4. *Given two disjoint axiom sets P_1, P_2, if $P_1, p \vdash_{d_1}^{\mathsf{NS}} 1$ and $P_2, 1 - p \vdash_{d_2}^{\mathsf{NS}} 1$ then $P_1, P_2 \vdash_{d_1 + d_2}^{\mathsf{NS}} 1$.*

Proof. Suppose that $P_1 = \{p_1, \ldots, p_m\}$ and $P_2 = \{q_1, \ldots, q_k\}$. Let $p_{m+1} = p$ and let $q_{k+1} = 1 - p$. By hypothesis, we know that

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

for some $g_1, \ldots, g_{m+1}, a_1, \ldots, a_n$, implying 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, we know that:

$$\sum_{i=1}^{k+1} r_i p_i + \sum_{j=1}^{n} b_j (x_j^2 - x_j) = 1$$

for some $r_1, \ldots, r_{k+1}, b_1, \ldots, b_n$, implying that:

$$\sum_{i=1}^{k} r_i p_i + \sum_{j=1}^{n} b_j (x_j^2 - x_j) = 1 - r_{k+1} q_{k+1} = 1 - r_{k+1}(1-p)$$

We notice that:

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

In the last step, we used the fact that, due to multilinearity, it holds that $p^2 = p$. Proceeding the same way, we find that:

$$\begin{aligned}p\left(\sum_{i=1}^{k} r_i p_i + \sum_{j=1}^{n} b_j (x_j^2 - x_j)\right) &= p\,(1 - r_{k+1}(1-p)) \\ &= p\,(1 - r_{k+1} + r_{k+1} p) \\ &= p - r_{k+1} p + r_{k+1} p^2 \\ &= p\end{aligned}$$

Now, we define $s_1, \ldots, s_{m+k}$

$$s_i = \begin{cases} g_i \cdot (1-p) & \text{if } 1 \le i \le m \\ r_i \cdot p & \text{if } m+1 \le i \le k \end{cases}$$

and $h_1, \ldots, h_n$ as $h_j = a_j \cdot (1-p) + b_j \cdot p$.

At this point, through simple algebra we get that:

$$\sum_{i=1}^{m+k} s_i p_i + \sum_{j=1}^{n} h_j(x_j^2 - x_j) =$$

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

$$(1-p)(1-g_{m+1}p) + p\,(1 - r_{k+1}(1-p)) = p + 1 - p = 1$$

concluding that $\pi_3 = \{s_1, \dots, s_{m+k}, h_1, \dots, h_n\}$ is a proof of $P_1 \cup P_2$. Furthermore, we notice that:

$$\deg((1-p)(1-g_{m+1}p)) = \deg(1-p) + \deg(1-g_{m+1}p) = d_1 + d_2$$

and that:

$$\deg(p\,(1 - r_{k+1}(1-p))) = \deg(p) + \deg(1 - r_{k+1}(1-p)) = d_2 + d_1$$

Finally, we get that:

$$\deg(\pi_3) = \max(\deg((1-p)(1-g_{m+1}p)), \deg(p\,(1 - r_{k+1}(1-p)))) = d_1 + d_2$$

concluding that $P_1, P_2 \vdash^{\mathsf{NS}}_{d_1+d_2} 1$.

$\square$

5.3 Treelike Res and Nullstellensatz

Definition 5.5 ($\mathbb{F}_2$-NS encoding of Res)**.** Given a Res linear clause $C = \bigvee_{i=0}^{k_1} x_i \vee \bigvee_{j=0}^{k_2} \overline{x_j}$, the $\mathbb{F}_2$-NS encoding of C is defined as $\text{enc}(C) := \prod_{i=0}^{k_1} x_i \cdot \prod_{j=0}^{k_2} (1 - x_j)$.

In general, a Res($\oplus$) formula $F = C_1 \wedge \ldots \wedge C_m$ defined on the variables $x_1, \ldots, x_n$ gets encoded in $\mathbb{F}_2$-NS as the set of axioms $P_F = \{\text{enc}(C_i) = 0 \mid 1 \leq i \leq m\} \cup \{x_j^2 - x_j = 0 \mid 1 \leq j \leq n\}$.

Theorem 5.2. *Let F be an unsatisfiable CNF. If T is* Res($\oplus$) *refutation of F of size s then there is* NS *refutation of F of degree $O(\log(s))$.*

Proof. Let $F = C_1 \wedge \cdots \wedge C_n$. We proceed by strong induction on the size s.

If $s = 1$ then the T contains only the empty clause $\perp$, meaning that it also is one of the starting clauses and thus one of the axioms. We notice that $\text{enc}(\perp) = 1$, which easily concludes that $\perp \vdash_0^{\mathsf{NS}} 1$.

Suppose now that $s > 1$. Let $\mathcal{L}$ be axioms of T. Since T is a binary tree, by Lemma 5.3 we know that there is a clause C_k, i.e. a node, of T such that T_{C_k} has size between $\left\lfloor \frac{1}{3} s \right\rfloor$ and $\left\lceil \frac{2}{3} s \right\rceil$.

Let $T' = (T - T_{C_k}) \cup \{C_k\}$. Due to the size of T_{C_k}, we get that T' has size between $\left\lfloor \frac{1}{3} s \right\rfloor + 1$ and $\left\lceil \frac{2}{3} s \right\rceil + 1$. Moreover, we notice that since T is a treelike refutation it holds that T_{C_k} and T' work with different clauses (except C_k), thus their axioms are disjoint. Let $\mathcal{L}_1, \mathcal{L}_2$ be the two sets of axioms respectively used by T_{C_k} and T'.

By construction, we notice that T_{C_k} derives the clause C_k using the axioms $\mathcal{L}_1$, while T_{C_k} derives the clause $\perp$ using the axioms $\mathcal{L}_2, C_k$. Thus, since T_{C_k} and T' have size lower than s, by induction hypothesis we get that $\text{enc}(\mathcal{L}_1) \vdash_{c_1 \cdot \log s}^{\mathsf{NS}} \text{enc}(C_k)$ and $\text{enc}(\mathcal{L}_2), \text{enc}(C_k) \vdash_{c_2 \cdot \log s}^{\mathsf{NS}} 1$ for some constants c_1, c_2. By Proposition 5.1 we easily conclude that $\text{enc}(\mathcal{L}_1), (1 - \text{enc}(C_k)) \vdash_{c_1 \cdot \log s}^{\mathsf{NS}} 1$ and, by Lemma 5.4, that $\text{enc}(\mathcal{L}_1), \text{enc}(\mathcal{L}_2) \vdash_{(c_1 + c_2) \cdot \log s}^{\mathsf{NS}} 1$. Finally, since $\mathcal{L}_1 \cup \mathcal{L}_2 = \mathcal{L}$, we get that $\text{enc}(\mathcal{L}) \vdash_{(c_1 + c_2) \cdot \log s}^{\mathsf{NS}} 1$, meaning that $\mathcal{L}$ has a NS refutation of degree $O(\log s)$.

□

5.4 Treelike $\mathsf{Res}(\oplus)$ and Nullstellensatz

Definition 5.6 ($\mathbb{F}_2$-NS encoding of $\mathsf{Res}(\oplus)$)**.** Given a $\mathsf{Res}(\oplus)$ linear clause $C = \bigvee_{i=0}^{k} (\ell_i = \alpha_i)$, the $\mathbb{F}_2$-NS encoding of C is defined as $\mathrm{enc}_{\oplus}(C) := \prod_{i=0}^{k} (\alpha - \ell_i)$.

In general, a $\mathsf{Res}(\oplus)$ formula $F = C_1 \wedge \ldots \wedge C_m$ defined on the variables $x_1, \ldots, x_n$ gets encoded in $\mathbb{F}_2$-NS as the set of axioms $P_F = \{\mathrm{enc}_{\oplus}(C_i) = 0 \mid 1 \leq i \leq m\} \cup \{x_j^2 - x_j = 0 \mid 1 \leq j \leq n\}$.

Theorem 5.3 ([**res_parity**])**.**

1. *Every tree-like* $\mathsf{Res}(\oplus)$ *proof of an unsatisfiable formula F may be translated to a parity decision tree for F without increasing the size of the tree.*

2. *Every parity decision tree for an unsatisfiable linear CNF may be translated into a tree-like* $\mathsf{Res}(\oplus)$ *proof and the size of the resulting proof is at most twice the size of the parity decision tree (and where the weakening is applied only to the axioms).*

Corollary 5.1. *Every tree-like* $\mathsf{Res}(\oplus)$ *proof of an unsatisfiable formula F can be converted to a tree-like* $\mathsf{Res}(\oplus)$ *proof of at most double the size and with weakening applied only to the axioms.*

Acknowledgements

No idea

Bibliography

[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.

[BGL13] Olaf Beyersdorff, Nicola Galesi, and Massimo Lauria. “A characterization of tree-like Resolution size”. In: *Information Processing Letters* 113.18 (2013), pp. 666–671. ISSN: 0020-0190. DOI: 10.1016/j.ipl.2013.06.002.

[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. URL: https://doi.org/10.1145/1516512.1516516.

[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. URL: https://doi.org/10.1145/800157.805047.

[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. URL: https://doi.org/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. URL: https://www.sciencedirect.com/science/article/pii/S0168007219300855.

[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.

[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.