MASTER'S THESIS

Master of Science in Advanced Mathematics and Mathematical Engineering

Title: On lower bounds for circuit complexity and algorithms for satisfiability

Author: Joaquim Casals Buñuel

Advisor: Albert Atserias Peri

Department: Computer Science

Academic year: 2016/17

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat de Matemàtiques i Estadística

Abstract

This work is devoted to explore the novel method of proving circuit lower bounds for the class **NEXP** by Ryan Williams. Williams is able to show two circuit lower bounds:

- A conditional lower bound which says that **NEXP** does not have polynomial size circuits if there exists better-than-trivial algorithms for `CIRCUIT SAT`
- An inconditional lower bound which says that **NEXP** does not have polynomial size circuits of the class $\mathbf{ACC^0}$

We put special emphasis on the first result by exposing, in as much as of a self-contained manner as possible, all the results from complexity theory that Williams use in his proof. In particular, the focus is put in an efficient reduction from non-deterministic computations to satisfiability of Boolean formulas.

The second result is also studied, although not as thoroughly, and some pointers with regards to the relationship of Williams' method and the known complexity theory barriers are given.

Contents

Introduction

The question of proving circuit complexity lower bounds is one that has been present in complexity theory for a long time. The appeal of circuits, as opposed to the traditional Turing machine model, is that, while computationally equivalent, they look combinatorially simpler. Thus, if we want to show that some function is not computable with a certain amount of resources, it seems that it ought to be easier to prove it using circuits and, since they are computationally equivalent to Turing machines, we would do so without loss of generality. For instance, if we could show that some problem in **NP** does not have Boolean circuits of polynomial size, i.e that it requires superpolynomial size Boolean circuits, then we would settle the $\mathbf{P} =? \mathbf{NP}$ question (with a negative answer). Unfortunately, the hope that the model of Boolean circuits would help towards the goal of settling the main questions of complexity theory was largely unrealized. In the 80's, there was progress in showing circuit lower bounds for *restricted* classes of circuits such as $\mathbf{AC^0}$ or $\mathbf{ACC^0}$. These are classes that are made up of polynomial size and constant depth circuits. It was shown that any circuit that can compute the `PARITY` function cannot be in $\mathbf{AC^0}$ ([Has86, Ajt83]) and it cannot be in $\mathbf{ACC^0}$ either ([Raz87, Smo87]). Another set of important results concerned monotone circuits ([Raz85]). Progress in this line eventually stalled. With regards to unrestricted classes of circuits, it was known since early times that **EXPSPACE** (the class of languages that can be computed in $O(2^{p(n)})$ space where $p(n)$ is a polynomial) required superpolynomial size circuits. The method to prove it was a simple counting argument. Kannan ([Kan82]) managed to prove the same result for a smaller class. He proved that $\mathbf{NEXP^{NP}}$ requires superpolynomial size circuits. The hope was that researchers would be able to keep on restricting the class for which they proved super polynomial circuits lower bounds and eventually prove it for **NP**, thus settling one of the millennium problems. Unfortunately, progress on this front halted with the work of Kannan until very recently with the result by Williams. Williams novelty is that he is able to provide conditional superpolynomial lower bounds for **NEXP**, which makes it the smallest "general" class for which we have evidence of superpolynomial lower bounds. The condition upon which Williams' result rests, is that there exist better-than-brute-force algorithms for `CIRCUIT SAT`. Moreover, in later work Williams is able to prove an unconditional superpolynomial lower bound for **NEXP**, although for the restricted class $\mathbf{ACC^0}$.

Main contributions of this work

Williams' method relies on quite a large number of results from complexity theory, and moreover, some of these results are quite old and the literature is not easily accessible for these results. One of the contributions of this work is an attempt at exposing, in as much of a self-contained manner as possible, all the results used by Williams. In this work the focus is put on a key result on reductions from non-deterministic computations to satisfiability of Boolean formulas.

One of the results used by Williams is the result $\mathbf{NEXP} = \mathbf{MIP}$ ([BFL91]). In this present work it is shown that we can avoid this "hard" result and instead use the more elementary $\mathbf{PSPACE} = \mathbf{IP}$ ([Sha92]). Another important ingredient in Williams' proof is an efficient version of Cook's proof that Boolean formula satisfiability is **NP**-complete. An efficient reduction from non-deterministic computations to Boolean formulas satisfiability is exhibitted. Further-

more, Williams method requires not only an efficient reduction, but also an easily computable one. Unfortunately, for the second condition we do not have a nicely self-contained proof and the literature seems to be silent on the topic. As a matter of fact, all consulted papers refer to prior work as a justification for this reduction, but after thorough literature search we have not been able to find a complete published proof of it.

Overview

The organization of the work is the following: We start by introducing the most fundamental concepts of complexity theory in Chapter 1. Then, in Chapter 2, we study the results used by Williams in [Wil10]. This makes up the most part of this work.

Chapter 3 is devoted to lightly survey the lower bounds for $\mathbf{AC^0}$ and $\mathbf{ACC^0}$ to help put in perspective the novelty of Williams' method. In Chapter 4 the Williams' result $\mathbf{NEXP} \not\subset \mathbf{ACC^0}$([Wil14]) achieved by applying the ideas of Chapter 2 is reviewed. Finally, in Chapter 5 we quickly introduce what are known as *Complexity Barriers*. These are results that shows us that certain methods will not be able to prove results as interesting as $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$. At the end of the chapter we try to give a couple pointers as to why it is thought that Williams' method circumvents all three known complexity barriers.

Part I

Williams' method

Chapter 1

Preliminaries

1.1 Turing Machines

The Turing machine will be the primary computation model. Intuitively a Turing machine is an abstract machine which has states and a transition table that indicates how to move from one state to another. It has tapes, divided by cells, and the machine can interact with the tape via a header, which reads and writes one symbol at a time, and can move to adjacent cells in a single computation step. At each computation step, the machine reads a symbol from a tape's cell and updates the tape and its state according to the transition table. Formally, a one-tape (deterministic) Turing machine M is defined as a tuple $M = \langle \Gamma, Q, \delta, q_{\text{start}}, q_{\text{accept}} \rangle$ where

- Γ is the alphabet. The set of symbols which the machine can read and write on the tape.
- Q is the set of states of the machine. It must contain at least the starting state (i.e $q_{\text{start}} \in Q$), which is the state of the machine at the start of the computation and the accepting state (i.e $q_{\text{accept}} \in Q$), which is the state that the machine must have at the end of an accepting computation.
- $\delta : \Gamma \times Q \mapsto \Gamma \times Q \times H$ is the partial transition function. Given the read symbol and the current state it maps to the symbol to be written, the next state, and the header movement. Here $H = \{-1, 0, 1\}$ means that the header goes to the left (-1), stays the same (0), or goes to the right (1).

A k-tape Turing machine has a transition function of the form $\delta : \Gamma^k \times Q \mapsto \Gamma^k \times Q \times H^k$. Whenever $k \geq 2$, we assume that there exists a read-only tape that contains the input to the machine.

We say that a deterministic Turing machine M accepts an input x if the computation of M on input x, denoted $M(x)$, halts on the accepting state. Otherwise we say that M rejects x or that it does not halt. We say that a Turing machine runs in time $T(n)$ if for inputs of length n the machine halts in at most $T(n)$ steps. For any machine M, the set of strings that are accepted by M is what we call the *language* of the machine, and is denoted by $L(M)$.

The non-deterministic Turing machine differs with its deterministic counterpart in that the transition function is non-deterministic. That is, δ is of the form $\delta : \Gamma \times Q \mapsto \mathcal{P}(\Gamma \times Q \times H)$ where $\mathcal{P}(A)$ denotes the powerset of A. In other words, instead of outputing a *single* pair (symbol, state, header), the non-deterministic δ function outputs arbitrarly many (up to the whole set, of course). This means that at each given time, the non-deterministic Turing machine is in arbitrarly many states, so it can run computations "in parallel" so to speak. Then, on input x we say that M accepts x if there exists some computation path such that $M(x)$ halts on an accepting state. Otherwise we say that M rejects x or does not halt. Alternatively, we could have defined a non-deterministic Turing machine as a machine equipped with a pair of

deterministic transition functions (δ_0, δ_1) that are applied simultaneously at each step. The next proposition shows that the choice is irrelevant.

Proposition 1. *Let M be a non-deterministic Turing machine whose transition function is of the form $\delta : \Gamma \times Q \mapsto \mathcal{P}(\Gamma \times Q \times H)$. Then, there exists a non-deterministic Turing machine M' such that the transition function of M' is a pair of deterministic transition functions (δ_0, δ_1) and $L(M) = L(M')$. Moreover, if M's running time is $T(n)$, then M''s running time is at most $c \cdot T(n)$ for some constant c that only depends on M.*

Proof. Let $\delta(\alpha, q) = X$, where X is a set of triplets of the form (symbol,state, header movement). Without loss of generality, assume that $|X| = 2^k$ for some k. We can assume this, since we can always fill X with copies of $(\alpha, q, 0)$, so that from a computational point of view, this would be dummy transitions that just wastes time. We partition X in two sets of equal size (which we can do since $|X|$ is a power of 2). Repeat this process recursively until the sets are singletons. We can see this process as generating a binary tree, whose leafs are the members of X. Call this tree T_X. It is straightforward to see that we can associate a new state to each internal node of T_X and define δ_0, δ_1 in such a way that one represents branching to the left and the other to the right. This procedure generates as many states as internal nodes has T_X, therefore we get 2^k new states per each original state. Therefore, M' has to perform, at worst, 2^k steps per each step of M. Since k depends on the machine and not on the input, this slowdown only represents a constant slowdown. □

Finally, we will introduce the probabilistic Turing machine. We define a probabilistic Turing machine as a nondeterministic Turing machine (thus having two transitions functions (δ_0, δ_1)). It also has an additional tape with a string of random bits (as long as many computation steps are required). At each step, instead of picking the transition function non-deterministically, the function that is used is determined by the random bits of the random tape. Then, on input x we say that M accepts x if $M(x)$ halts on an accepting state. Otherwise we say M rejects x.

1.1.1 Efficient Turing Machines

We start by introducing the concept of *oblivious* Turing machine. We say that a Turing machine is oblivious if the header position at any given time depends only on the length of the input and not on the contents of the tape. That is, if $h(i, x)$ is a function that outputs the header position at step i with input x, then the machine is oblivious if it holds that $\forall i \; h(i, x) = h(i, 0^{|x|})$.

For the purposes of the following results we will introduce the concept of two-way infinite tapes. Usually, Turing machines tape are assumed to be infinite in only one direction, that is, the header can go indefinitely far to the right (for instance), but its movement to the left is bounded (this could be enforced, for instance, forbidding left movements of the header at cell 0). For some applications, it is best to think of Turing machines tapes as being two-way infinite, meaning that we can go as far as we want in either direction. The following proposition shows that the only penalty that we need to pay is a constant factor slow-down

Proposition 2. *Let M be a Turing machine with two-way infinite tapes. There exists M' with one-way tapes such that $L(M) = L(M')$ and M''s running time only inccurs in a constant factor slow-down with respect to M.*

Proof. Next we describe how M' can simulate one two-way infinite tape of M using a one-way tape. We just need to repeat the process for each tape of M. Let Γ be the alphabet of M, then M' alphabet will be $\Gamma' = \Gamma \times \Gamma$. Let c_0 be an arbitrary cell of M's tape which we set as the origin. Then M' sets c_0 to be the first cell of the tape. Whenever M reads to the left of the origin, since M' symbols are pairs of symbols (α, β) of M's alphabet, M' reads the second component of the symbol, and conversely when M reads to the right of the origin, M' reads the first component of the symbol. The same reasoning applies when writing □

Now we introduce a construction that will help us improve the previous result. The idea is a refinement of theorem **??**. We will set up buffer zones, so whenever we do a shift we don't have to shift the whole tape, and we will do so in a way that the amortized cost of all shifts will only add a logarithmic factor. First, we add a symbol to the alphabet of M', that will fill all the buffer zones, and will be ignored during the simulation. Hence, if Γ is the alphabet of M then, M' has the alphabet $\Gamma \cup \{\diamond\}$, where $\diamond$ will be the buffering symbol. M' will divide its tapes into buffering zones, labeled R_i/L_i depending on whether they fall to the right or left of the origin, which will be position 0 and will not be in any of the buffering zones. M' will keep the following invariant:

1. A zone labeled with index i will contain 2^{i+1} cells
2. The number of non-$\diamond$ symbols in $R_i \cup L_i$ will be 2^{i+1}
3. Each zone will contain exactly one of three possible number of $\diamond$ symbols: $0, 2^i$ or 2^{i+1}
4. All the non-$\diamond$ symbols of a zone can be found at the start of it. By start we mean the cell closest to the origin

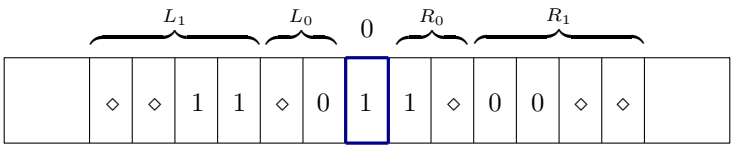

Figure 1.1: A tape of M' divided into buffering zones that are half-full

The following algorithm summarizes the process

$M'(x)$:

1. Simulate one step of $M(x)$
2. If M's header goes to the right (essentially the same process aplies when the header goes to the left): Find the smallest i_0 such that R_{i_0} is empty, move the left-most symbol to the origin, and move the next $2^i - 1$ symbols to the R_j zones with $j < i_0$ such that each R_j becomes half-full.
3. For each L_j with $j < i_0$ move the 2^{i+1} symbols from L_j to L_{j+1}.
4. repeat 1-3 for the number of steps required by $M(x)$

First let's argue that step 2 and 3 can be carried out without problems. Since R_{i_0} is the first non-empty zone, clearly there is enough room for the $2^{i_0} - 1$ symbols, since $\sum_{j=0}^{i_0-1} 2^j = 2^{i_0} - 1$. By the invariant we have that all L_j with $j < i_0$ are full, hence we can have enough symbols to fill each L_j zone. After steps 2-3, the zones $\{R_{i_0}, L_{i_0} \cdots R_1, L_1\}$ are all half-full. If R_{i_0} was half-full, then L_{i_0} was half-full as well, hence after the shifting R_{i_0} is empty, but L_{i_0} is full, preserving the invariant. Likewise if R_{i_0} was full.

Now, let's turn to the cost analysis of the previous operations. In order to perform the right shift with index i_0, we need to half-fill the right side of the tapewith $2^{i_0} - 1$ (remember that one symbol goes to position 0). It is clear that this will take time $O(2^{i_0})$. In the second part, we need to shift $2 \cdot 2^j$ symbols from L_j to L_{j+1}, which clearly takes time $O(2^j)$, and we need to do this for all zones L_j with $j < i$, hence the total cost for this part is $O(\sum_{j=0}^{i_0-1} 2^j) = O(2^{i_0})$. Putting together both parts, we get that a shift with index i_0 costs $O(2^{i_0})$. Now, notice that whenever we perform a shift with index i_0, the following $2^{i_0} - 1$ shifts must have an index smaller than i_0, because we need to consume all the symbols we have just shifted before reaching zone

i_0 again. Therefore, the fraction of shifts with index i_0 is $1/2^{i_0}$. Following this argument, the total cost of the simulation is given by the following expression:

$$O(\sum_{i=0}^{\log t} \frac{t}{2^i} \cdot 2^i) = O(t \cdot \sum_{i=0}^{\log t} 1) = O(t \cdot \log t)$$

Theorem 1. *[AB09] Let M be a machine that runs in time $O(T(n))$. There exists M' with $L(M) = L(M')$ and M' oblivious which runs in time $O(T(n) \cdot \log T(n))$*

Proof. Consider the machine M' described in the previous construction. We cannot say it is oblivious, since at step 2, we look for the first non-empty zone, and this will be determined by the shifting pattern of the execution, which in turn is determined by the contents of the tape. For instance, maybe the machine M simulated, just shifts back and forth one square on input 0^n and just shifts always right on any other input. Therefore, we need to homogenize the headers movement. What we can do is simulate all possible shifts but actually only perform one. That is, The head of M' will first move as to perform a right shift for the indices $\{0 \dots i_{\max}\}$, and afterwards the same for left shifts. Here, $i_{\max}$ is the maximum index required to perform the shift, for instance, if we have simulated 2^k steps of M, then we can only have possibly reached zone k. The fact that for all but one of the shifts, the machine does not modify the tapes contents doesn't affect the cost, which will be the sum of all shifts (left and right) with indices in $\{0, \dots, i_{\max}\}$. We know that a shift with index i has cost $O(2^i)$, hence the cost of the simulating all possible shifts will be

$$O(2 \cdot \sum_{j=0}^{i_0} 2^j) = O(2^{i_0+2}) = O(2^{i_0})$$

So, asymptotically, by the price of one shift we get all of them. We just write into some register which shift must be the real one, and simulate all of them only modifying the tape contents on the registered one. Therefore, the heads movement are completely independent of the tape's content and are the same for all inputs of the same length. □

1.1.2 Universal Turing Machines

One application of the efficient simulation from the previous section is the existence of efficient Universal Turing Machines. Let $\{M_i \mid i\mathbb{N}\}$ be an enumeration of the deterministic Turing machines.

Theorem 2. *There exists a deterministic Turing machine U, such that for every $i \in \mathbb{N}$ and every $x \in \{0,1\}^*$ $U(i,x) = M_i(x)$ where M_i denotes the i-th deterministic Turing machine. Moreover, if M_i on input x halts in t steps, then U on input (i,x) halts in $c_i \cdot t \log t$ where c_i is a constant independent of x and only depends on the machine M_i*

Proof. Note that this construction follows that of theorem 1. We just have made sure that we get a working tape to perform all the shifts of the machine M to be simulated. The special buffer zone is necessary because the machine cannot alter its alphabet depending on the machine to be simulated. Without loss of generality assume that the symbols that denote start of tape and blank cell are common across all alphabets and that the alphabet of U is $\{0,1\}$. If Γ is the alphabet of M, we need to set up a buffer zone of size $\log|\Gamma|$. This construction implies that for each step of $M(x)$, U needs to take $\log|\Gamma|$ steps plus another constant number a that depends on the transition function of M, thus in total $(\log|\Gamma| + c) \cdot t$ steps to simulate M, a constant slow down. Let $b = \log|\Gamma| + a$ and $t' = b \cdot t$. By theorem 1 again we have that we will need $t' \log t' = b\,t \cdot [\log b + \log t] \leq c_i \cdot t \log t$ steps to simulate $M(x)$ where c_i is essentially b but it also hides some constants that appear during the calculations and that are of no importance. But, since it depends on b, we see that indeed the constant of the statement is machine dependant but input independant. □

Next we introduce the non-deterministic universal Turing machine. We we'll see that the non-determinism actually makes a huge difference. It will allow us to only have a constant slowdown instead of the logarithmic factor we get in theorem 2. Let $\{N_i \mid i \in \mathbb{N}\}$ be an enumeration of the non-deterministic Turing machines.

Theorem 3. *There exists a non-deterministic Turing machine U, such that for every $i \in \mathbb{N}$ and every $x \in \{0,1\}^*$ $U(i,x) = M_i(x)$ where N_i denotes the i-th non-deterministic Turing machine. Moreover, if N_i on input x halts in t steps then, U on input (i,x) halts in $c_i \cdot t$ steps where c_i is a constant that is independent of x and only depends on the machine N_i*

Proof. In order to only have a constant slow-down in the simulation we will make use of non-determinism. Roughly, we will guess a computation of the simulated machine and then check that it is correct. The idea is formalized as follows: Let U be the universal nondeterministic Turing machine. Suppose we are given i, x as input and want to simulate the $M_i(x)$, the i-th nondeterministic Turing machine on input x. Supose that $M_i(x)$ halts in t steps. Define R_j a register for the j-th step. R_j is a tuple with the following items

- Current state of M_i on step j
- Next state of M_i for step $j+1$
- Symbols read by M_i on step j
- Symbols written by M_i on step j

We can assume that U has a counter so that we only guess t registers. What U does is instead of reading the work tape, consult M_i's transition table and act accordingly (this would be too slow) we just guess what symbols are read, written and what is the current state and next state. This is written in a tape, and each such guess is what we defined as R_j. We guess t such registers. Now we have t registers $\boldsymbol{R} = \{R_1, \ \ldots \ R_j\}$ which potentially encodes an accepting computation of $M_i(x)$. What is left to do is check that it is indeed the case. $\boldsymbol{R}$ encodes a correct computation if, and only if, the following three points hold:

1. Given R_j, its current state, next state and read/write symbols is consistent with the transition function of M_i
2. Given R_j, R_{j+1}, the next state field for R_j is equal to the current state of R_{j+1}
3. The current state for R_t is q_{accept}

To check **1**, we just need to check each R_j against all possible transitions of M_i. This takes time $c_i \cdot t$. To check **2**, we just need to traverse $\boldsymbol{R}$ and for each consecutive pair of registers do an easy comparison. Again time $c_i \cdot t$. Finally, we just check that for R_t its current state is indeed q_{accept}. For this particular register we don't care about the next state for obvious reasons. Thus in total we needed time $c_i \cdot t$ to guess $\boldsymbol{R}$ and time $c_i \cdot t$) to check that these guesses correctly encode an accepting computation path of $M_i(x)$. In total $c_i \cdot t$ where the c_i is a constant that will depend on M_i. □

1.2 Boolean Circuits

Another model of computation relevant for our purposes is the *Boolean Circuits*. A boolean circuit C_n is a Directed Acyclic Graph (DAG), with n especial vertices called *Input* with indegree 0, and a special vertex called *Output* with indegree 1. A circuit is defined over some basis $\mathcal{B}$, which is the set of types of gates that it can have. For instance, define the canonical base to be $\mathcal{B}_0 = \{\wedge, \vee, \neg\}$. So in this case a circuit over $\mathcal{B}_0$ is one where the gates that are not input nor

output are either OR, AND or NOT gates. As a convention we consider the "bottom" layer (or level 1) of a circuit to be the set of gates that only have the inputs as input and the "top" layer (or level depth(C)) the one composed of the gates whose output is the circuits output. The level or layer to which a gate belongs is defined as the maximum level of its inputs + 1, where the inputs are at level 0. The justification for this "bottom" and "top" vision, is that if we restrict the fanout of the gates to be 1, and the circuit to have only one output, then the circuit can be represented as a tree with the output gate as the root, and the inputs as leaves We say that a family of Boolean Circuits $\{C_n \mid n \in \mathbb{N}\}$ is *uniform* if there exists a Turing machine M such that for all n, $M(1^n)$ outputs a description of C_n. Moreover, if M takes time $T(n)$ to output the circuit description, then we say that $\mathcal{C}$ is $T(n)$-uniform. If left unspecified will assume that all families of circuits are *non-uniform*. So we can think of non-uniformity as just a way to not specify wether the family is uniform or not. Therefore, in some sense, the two conditions are not mutually exclusive. All uniform families can be regarded also as non-uniform. The way to think about non-uniformity is that there may be a Turing machine capable of describing the family but we don't know or we don't care As an example consider the halting language, $L_{\text{halt}} = \{1^n \mid M_n(1^n) \text{ halts}\}$. We know that there is no Turing machine that accepts this language, but there is a family of circuits that does. Indeed, for each n we can hardcode into C_n a 1 or a 0, depending on whether $M_n(1^n)$ halts or not. Hence, $\{C_n \mid n \in \mathbb{N}\}$ is a non-uniform (by default) family that we know is not uniform. We say that a family $\mathcal{C} = \{C_n \mid n \in \mathbb{N}\}$ has size $S(n)$ if for infinitely many n, $|C_n| \leq S(n)$. Where $|C_n|$ is defined as the number of gates of C_n

Theorem 4. *Let $\mathcal{C} = \{C_n \mid n \in \mathbb{N}\}$ be a a $T(n)$-uniform circuit family of size $S(n)$ over $\mathcal{B}_0$ (the canonical base). There exists a Turing machine M that computes the same function as $\mathcal{C}$ in time $O(T(n) + S(n))$*

Proof. Since $\mathcal{C}$ is a uniform family, there exists a Turing machine $M_{\mathcal{C}}$ such that $M_{\mathcal{C}}(1^n)$ outputs a description of $C_n \in \mathcal{C}$. Consider the following Turing machine:

$T(x)$:
1. simulate $M_{\mathcal{C}}(1^{|x|})$. Call C the resulting circuit
2. return $C(x)$

Step 1 takes, by assumption of $T(n)$ uniformity, $O(T(n))$ time. Since the circuit is over $\mathcal{B}_0$, clearly we can simulate any of the gates for this base in constant time. Hence, we can simulate $C(x)$ in time $O(S(n))$ and the result follows □

Theorem 5. *Given a 3-CNF propositional formula $F(x_1 \ldots x_n)$, there is a circuit C_n of size $O(|F|)$ such that $F(x_1 \ldots x_n)$ is true if, and only if, $C(x_1 \ldots x_n) = 1$*

Proof. Given any clause of F, it is clear that there is a circuit that computes it. Just take the OR of the three variables appearing in the clause. Now, define C_n to be the AND of the $O(|F|)$ clauses □

Corollary 6. *Let M be a Turing machine that runs in time $O(T(n))$. There exists a family of circuits $\{C_n \mid n \in \mathbb{N}\}$ of size $O(T(n)^2)$ that computes the characteristic function of $L(M)$.*

Proof. **main idea** As we will see in Chapter 2 (theorem 13), the resulting formula can be easily implemented as a circuit. And the formula has the property that it is satisfiable if, and only if, the given input x belongs to the language. Then, it can be shown that fixing some auxiliary variables we can make a circuit that given x computes the value of the boolean formula from theorem 13. □

Remark 1. *Through out this section we have assumed that Turing machines and Circuits can be encoded by some string in an "efficient" manner. We will not give a formal proof but since*

Turing machines like circuits are finite (family of circuits are infinite, but each single member of the family is a finite object) it is easy to see that we can indeed work with strings that represent them

Remark 2. *All of the functions $T(n)$ that we'll consider to quantify execution time for TM's and size for circuits, are assumed to be of the form $T(n) : \mathbb{N} \mapsto \mathbb{N}$ and monotone*

1.3 Oracle machines

Finally, although we will only discuss briefly results with oracles, we introduce here the definition of oracle machines. An oracle Turing machine, is a Turing machine as defined in the preliminares section with the following additions:

1. A new tape, called the *query* tape, where the Turing machine will write the string which it wants to query and will read there the answer (the answer being YES or NO).

2. A new state q_{query}. Upon switching to this state, the Turing machine asks the oracle if the string written in the *query* tape belongs to it or not. The oracle answers YES or NO in one computational step.

Likewise, we can extend Boolean circuits to include a gate A which interprets its inputs as a query and outputs according to whether the string described by the inputs belong to the oracle A or not.

1.4 Propositional formulas and 3-CNF

Finally, we will close this first chapter with a result on propositional logic. In this work we will for the most part work with a special kind of propositonal formulas called 3-CNF. For completeness sake, here we introduce briefly the Tseitin transform, a result that allows to efficiently convert an arbitrary propositional formula into a 3-CNF that preserves its satisfiability. We showcase here the main idea. Given φ a propositional formula, for any subformula ψ we introduce an auxiliary variable a_ψ. Then, define the Tseitin transform of φ, denoted by $\Phi(\varphi)$, by considering the following cases:

- $\varphi = \psi_1 \wedge \psi_2$. Then introduce the auxiliary variables a_φ, b_0, b_1 and define the following 3-CNF $\Phi(\varphi) = (\overline{a_\varphi} \vee a_{\psi_1} \vee b_0) \wedge (\overline{a_\varphi} \vee a_{\psi_1} \vee \overline{b_0}) \wedge (\overline{a_\varphi} \vee a_{\psi_2} \vee b_1) \wedge (\overline{a_\varphi} \vee a_{\psi_2} \vee \overline{b_1}) \wedge (\overline{a_{\psi_1}} \vee \overline{a_{\psi_2}} \vee a_\varphi) \wedge \Phi(\psi_1) \wedge \Phi(\psi_2)$.

- $\varphi = \psi_1 \vee \psi_2$. Then introduce the auxiliary variables a_φ, b_0, b_1 and define the following 3-CNF $\Phi(\varphi) = (\overline{a_\varphi} \vee a_{\psi_1} \vee a_{\psi_2}) \wedge (\overline{a_{\psi_1}} \vee a_\varphi \vee b_0) \wedge (\overline{a_{\psi_1}} \vee a_\varphi \vee \overline{b_0}) \wedge (\overline{a_{\psi_2}} \vee \vee a_\varphi \vee b_1) \wedge (\overline{a_{\psi_2}} \vee a_\varphi \vee \overline{b_1}) \wedge \Phi(\psi_1) \wedge \Phi(\psi_2)$.

- $\varphi = \neg\psi$. Then introduce the auxiliary variables a_φ, b_0, b_1 and define the following 3-CNF $\Phi(\varphi) = (\overline{a_\varphi} \vee \overline{a_\psi} \vee b_0) \wedge (\overline{a_\varphi} \vee \overline{a_\psi} \vee \overline{b_0}) \wedge (a_\psi \vee a_\varphi \vee b_1) \wedge (a_\psi \vee a_\varphi \vee \overline{b_1}) \wedge \Phi(\psi)$.

1.5 Complexity notions

In the previous sections we have introduced a couple computational models: the Turing machines and the Boolean circuits. They are not the only ones (for instance there exists partial recursive functions) but are the most used in complexity theory and are the ones that we will be using throughout this work. This models all can compute the same. The notion of computation is defined via the Turing-Church principle, which roughly states that "the notion of computation is captured by that of Turing machines". Thus a function is computable, if and only if is

computable by a Turing machine. With that settled we turn to the problem of determining what is computable and what is not. That is the realm of *Computability Theory*. The *Computational Complexity* theory focuses on the computable problems, and tries to classify them according to the different resources that they need. The main resources considered are time and space. A problem may be "Given a graph G, is it hamiltonian?" or "given n numbers, output them in order". The first type of problems are called *decisional*, and can be summarized as "Given some input, does it have some specific property? Yes/no". On the other hand, the other type of problems are called *functional*, and are of the form "Given some input x, output f(x)" for some function f. Note that functional problems subsume decisional problems, because in the decisional case we just take f(x) to be a function with boolean output. We will for the most part focus on *decisional* problems. In order to formalize this idea we consider sets of strings that encode problems. For instance, we can consider a string $x \in \{0,1\}^{n^2}$ as an encoding of the adjacency matrix of some graph with n vertices and using this representation we can *decide* if the graph has some property (for instance being hamiltonian).

We are interested in the characteristic function of such sets set. That is, given a set of strings A denote it's characteristic function by $\chi_A(x)$ (i.e $\chi_A(x) = 1 \iff x \in A$). Computational complexity is concerned in classifying this sets A by the resources it takes to compute their characteristic function $\chi_A(x)$. A Complexity class is a collection of sets $\{A_1, A_2, A_3, \ldots\}$ such that computing their characteristic functions take a "similar" amount of resources. Usually the being "similar" is defined as "the amount of resources consumed is bounded asymptotically by the same function (or the same type of function)". For instance, **P** is the collection of all sets such that the amount of steps required by a Turing machine to compute their characteristic function is bounded by some polynomial. We say that a *language* L is *decided* by a Turing machine M, if for any x we have that $M(x)$ halts on an accepting state $\iff x \in L$. Finally, we will also need to introduce a modifer for complexity class. Given languages L, L', we say that they agree infinitely often if, and only if, $L \cap \{0,1\}^n = L' \cap \{0,1\}^n$ for infinitely many n. Given **C** a complexity class, we define **io-C** $= \{L' \mid L'$ agrees infinitely often with some $L \in \mathbf{C}\}$

1.5.1 Reductions and complete problems

In order to study the different complexities we use what are known as reductions. The idea is that we want to relate the difficulty of computing the characteristic function of different sets, which we also call problems. For the purposes of this work, given two sets or problems A, B we say that A reduces to B, denoted $A \leq B$, if there exists a polynomially computable function f such that given any input x

- $x \in A \implies f(x) \in B$.
- $x \notin A \implies f(x) \notin B$.

If B belongs to some complexity class **C**, and for any other **A** in **C** we have that $\mathbf{A} \leq \mathbf{B}$, then we say that **B** is a **C**-complete problem. The interesting property is that if **B** belongs also to some other complexity class $\mathbf{C}'$, then automatically we get the complexity inclusion $\mathbf{C} \subseteq \mathbf{C}'$, provided $\mathbf{C}'$ is closed under reductions, meaning that if $\mathbf{A} \leq \mathbf{B}$ and $\mathbf{B} \in \mathbf{C}'$ then we have $\mathbf{A} \in \mathbf{C}'$.

1.5.2 Useful complexity classes

Next we introduce the definition of the complexity classes that we will work with.

Definiton 1. DTIME$(T(n))$ *is defined as the class of languages that can be decided by a deterministic Turing machine running in* $O(T(n))$ *steps. Likewise,* **NTIME**$(T(n))$ *is defined as the class of languages that can be decide by a non-deterministic Turing machine running in* $O(T(n))$ *steps.*

We can also characterize computational classes by the space used in deciding their languages.

Definiton 2. SPACE$(T(n))$ *is the class of languages that can be decided by a Turing machine using at most* $O(T(n))$ *cells. In other words, the Turing machine tapes' headers stay within distance* $O(T(n))$ *from their origin.*

In the case of Boolean circuits, we usually charactherize them by their size. Note that we can translate the notion of accepting of a Turing machine to any other computational model, in particular we can do it to Boolean circuits.

Definiton 3. SIZE$(T(n))$ *is defined as the class of languages that can be decided by a family of Boolean circuits of size* $O(T(n))$

Finally we introduce what are known as Interactive Protocols. Given an input x with $|x| = n$, an interactive protocol is an interaction between two players which ends up with the acceptance of x or its rejection. The two players are the following:

- Prover: this player can be regarded as an all powerful oracle, in the sense that it can compute any function whose output is of polynomial size (with respect to its input). Therefore, the prover is an oracle that computes some function $f : \{0,1\}^n \mapsto \{0,1\}^{\text{poly}(n)}$
- Verifier: this player is a polynomial probabilistic Turing machine which exchanges a number of messages with the prover, performs some computation for each message and finally accepts or rejects x

Denote by $(V \leftrightarrow P)(x) = 1$ an interaction which on input x ends up accepting and $(V \leftrightarrow P)(x) = 0$ and interaction which on input x ends up rejecting.

Definiton 4. *Let* L *be a language. We say that* L *has a* $T(n)$*-round interactive protocol, if for every* x *with* $|x| = n$ *there exists a polynomial probabilistic Turing machine (the Verifier) such that*

$$x \in L \implies \exists\ P \ \ Pr[(V \leftrightarrow P)(x) = 1] \geq \frac{2}{3}$$

$$x \in L \implies \forall\ P \ \ Pr[(V \leftrightarrow P)(x) = 1] \leq \frac{1}{3}$$

and the number of messages exchanged between the prover and the verifier is bounded by $T(n)$ *with certainty (with probability 1).*

We define the class $\mathbf{IP}(T(n))$ as the class of languages that have a $T(n)$-round interactive protocol. We define $\mathbf{MA}(T(n))$ as a particular case of the previous settings, where the protocol just consists of two steps:

1. The Prover starts by sending a message to the verifier
2. The Verifier decides whether to accept x or not, based on a deterministic polynomial time computation that depends on the Verifiers randomness , the input and the Prover's message

Next we list some of the most widely used classes:

- $\mathbf{P} = \bigcup_{c>0} \mathbf{DTIME}(n^c)$.
- $\mathbf{NP} = \bigcup_{c>0} \mathbf{NTIME}(n^c)$.
- $\mathbf{E} = \bigcup_{c>0} \mathbf{DTIME}(2^{cn})$.

- $\mathbf{NE} = \bigcup_{c>0} \mathbf{NTIME}(2^{cn})$.
- $\mathbf{EXP} = \bigcup_{c>0} \mathbf{DTIME}(2^{n^c})$
- $\mathbf{NEXP} = \bigcup_{c>0} \mathbf{NTIME}(2^{n^c})$.
- $\mathbf{P}/poly = \bigcup_{c>0} \mathbf{SIZE}(n^c)$.
- $\mathbf{PSPACE} = \bigcup_{c>0} \mathbf{SPACE}(n^c)$.
- $\mathbf{IP} = \bigcup_{c\geq 1} \mathbf{IP}(n^c)$.
- **MA**.

Chapter 2

Lower bounds via circuit satisfiability

2.1 Overview

William's theorem states that the following two possibilities are incompatible:

(A) $\mathbf{NEXP} \subseteq \mathbf{P}/poly$

(B) `CIRCUIT SAT` has better-than-trivial algorithms

In this context, better-than-trivial algorithms for `CIRCUI SAT` means algorithms that run in time $O(2^n \cdot m^c/f(n))$ on circuits with n inputs and m gates where c is a constant that does not depend on the input circuit and $f(n)$ is superpolynomial (i.e $n^{\omega(1)}$).

The proof can be split into two parts

(a) If (A) holds, then "small" witness circuits exist

(b) If (B) holds, then "small" witness circuits do not exist

In order to make precise this conditions we need the notion of a verifier for a language $L \in \mathbf{NTIME}(T(n))$. A verifier V for L is a polynomial time algorithm such that if $x \in L$ then there exists $y \in \{0,1\}^{T(n)}$ such that $V(x,y) = 1$. Otherwise, if $x \notin L$, we have that for all $y \in \{0,1\}^*$ it is the case that $V(x,y) = 0$. With this notion introduced we say that a language $L \in \mathbf{NTIME}(T(n))$ has $S(n)$ universal witness circuits if for every polynomial time verifier V of L, it is the case that there is a circuit C of size at most $S(n)$ such that for every $x \in \{0,1\}^n \quad V(x,y) = 1 \Leftrightarrow x \in L$ where $y_i = C(x,i)$ for $i \in \{1,\ldots,T(n)\}$.

In this case, "small" witness circuits means witness circuits of polynomial size, that is $S(n) = poly(n)$. In order to prove (a) and (b), we will require several cornerstone results of complexity theory:

$$(a)\begin{cases} 1. & \mathbf{PSPACE} = \mathbf{IP} \\ 2. & \text{pseudorandom generators from circuit lower bounds} \\ 3. & \text{fixed polynomial circuit lower bounds for } \mathbf{PSPACE} \end{cases}$$

$$(b)\begin{cases} 4. & \text{efficient reductions to 3-SAT} \\ 5. & \text{tight time-hierarchy results for non-deterministic computations} \end{cases}$$

We will now give an idea of how (A) and (B) are used to obtain the desired lower bounds. Suppose that given x we can generate a propositional formula Φ_x which is satisfiable if, and only if, $x \in L$, for any $L \in \mathbf{NTIME}(2^n)$. Then, under condition (A), there exists "small" circuits

that encode a satisfying assignment to Φ_x (if there is one). Next, given any input x the idea will be to construct a circuit D (making use of Φ_x) such that D will be unsatisfiable if, and only if, Φ_x is satisfiable and, additionally, D will have much fewer variables than Φ_x (exponentially many fewer). D is essentially composed of two parts. First, a circuit that computes the i-th clause of Φ_x . Next, D has "integrated" another circuit, called W, that encodes an assignment of Φ_x (which exists if condition (A) holds). This circuit is what we call the "witness circuit". It is called a witness, because if the assignment encoded by W satisfies Φ_x, then it "witnesses" the fact that $x \in L$. Finally the D is built such that if on input i the i-th clause of Φ_x is satisfied under the assignment encoded by W, then D outputs "0". Therefore, if D outputs "0" for every inputs (and thus is unsatisfiable), it means that under the assignment encoded by W all clauses of Φ_x are satisfied and thus $x \in L$. If D outputs "1" for some input, then some clause is not satisfied under the assignment of W and therefore Φ_x is not satisfied and $x \notin L$. So we give decided the membership of x via deciding the satisfiability of a circuit, namely D which has much fewer variabels than Φ_x.

Now condition (B) comes into play. Condition (B) implies that the satisfiability of the circuit D can be computed in time $o(2^n)$. Now, if we pick L such that $O(2^n)$ time is required to decide it, applying the previous reasoning we can do it faster, in time $2^{n-\omega(\log n)}$. Indeed, suppose that $L \in \mathbf{NTIME}(2^n) \setminus \mathbf{NTIME}2^{n-\omega(\log n)}$ (which is guaranteed to exist by the non-deterministic time hierarchy). Given x, a non-deterministic algorithm can first guess the witness circuits W, then build D and decide the if x belongs to L in time $2^{n-\omega(\log n)}$ reaching a contradiction. All the results used in constructing D are unconditional, except for one. The fact that "small" witness circuit exists is conditioned to (A). Therefore we conclude that (A) cannot hold (thus yielding a lower bound for **NEXP**) if (B) is true.

In the next sections we will inspect the difference pieces needed to prove a result such as

Theorem 7. *Let $f(n)$ be a super polynomial function. If* `CIRCUIT SAT` *on circuits of n inputs and m gates can be solved in (co-non)deterministic time $O\Big(\frac{2^n \cdot m^c}{f(n)}\Big)$ then* $\mathbf{NEXP} \not\subset \mathbf{P}/poly$

2.2 PSPACE = IP

The first result needed is part of the **PSPACE** = **IP** ([Sha92]). We are interested in the **PSPACE** $\subseteq$ **IP** direction. This part requires what is known as the *arithmetization* of Boolean formulas.

2.2.1 IP $\subseteq$ PSPACE

To see this consider a protocol in **IP**. In this protocol, by definition, the Prover and the Verifier exchange at most a polynomial number of messages (of polynomial length). Consider a Turing machine that guesses the string sent by the prover, guesses the randomness used by the Verifier, then runs the Verifier and repeats the process recursively with the Verifier's answer. This way the Turing machine tries all possible combinations of exchanged messages, tries all possible randomness for the Verifier and, counts how many accepting paths there are. Since one full simulation requires polynomially many messages of polynomial size and the verifier is polynomially bounded, it is clear that simulating one possible interaction is in **PSPACE**. But note that once we simulate one interaction, we can forget about it, we only need to remember whether the Verifier accepted or not. Thus this Turing machine is in **PSPACE** and can compute the optimal prover for the protocol. It follows then that **IP** $\subseteq$ **PSPACE**.

2.2.2 PSPACE $\subseteq$ IP

Here we introduce the key technique used in proving **PSPACE** $\subseteq$ **IP**. The proof relies on the fact that the true quantified boolean formulas problem (`TQBF`) is **PSPACE**-complete. Thus we

only need to exhibit an **IP** protocol for that problem and the result will follow.

Definiton 5. *We define a Quantified Boolean Formula (QBF) as*

$$\Phi(x_1 \dots x_n) = Q_1 x_1 \dots Q_k x_k \, \Psi(x_1, \dots, x_n)$$

where Ψ is a 3-CNF on the Boolean variables $x_1, \dots, x_n$ and $Q_i \in \{\exists, \forall\}$.

The key idea will be that of *arithmetization*([LFKN90]). The idea is to "lift" boolean formulas to polynomials over some (finite) field and then work with the polynomials as representatives of this formulas. We will see that polynomials give us extra power, in particular it gives us self-correcting methods, which play a key role in the proof of **PSPACE** = **IP**. As a warmup we will give an **IP** protocol for $\#\mathbf{SAT_d}$ in which we count the number of satisfying assignments of a 3-CNF formula.

Definiton 6. *We define the decision problem $\#\mathbf{SAT_d}$ as the set of pairs $(\varphi(x_1, \dots, x_n), a)$ where $\Phi(x_1 \dots x_n)$ is a 3-CNF formula in the variables $x_1 \dots x_n$ and $a \in \mathbb{N}$ denotes the number of satisfying assignments of $\varphi(x_1 \dots x_n)$. The problem consists in given the pair $(\varphi(x_1, \dots, x_n), a)$ decide if the number of satisfying assignments of $\varphi(x_1, \dots, x_n)$ is a*

Given a 3-CNF formula $\Phi(x_1 \dots x_n)$ we will define a polynomial $p_\Phi(x_1 \dots x_n)$ with the property that $\Phi(a_1 \dots a_n) = 1 \iff p_\Phi(a_1 \dots a_n) = 1$ and $\Phi(b_1 \dots b_n) = 0 \iff p_\Phi(b_1 \dots b_n) = 0$

Proposition 3. *Given a 3-CNF formula $\varphi(x_1 \dots x_n)$ with m clauses, there exists a polynomial $p_\varphi(x_1 \dots x_n)$ such that $\varphi(a_1 \dots a_n) = 1 \iff p_\varphi(a_1 \dots a_n) = 1$ and $\varphi(b_1 \dots b_n) = 0 \iff p_\varphi(b_1 \dots b_n) = 0$. Moreover, the degree of $p_\varphi(x_1 \dots x_n)$ is bounded by $3m$*

Proof. Let ℓ_i be a literal of the 3-CNF, that is $\ell_i = x_i$ or $\ell_i = \overline{x_i}$. Then trivially $p_{\ell_i} = x_i$ is itself a polynomial which agrees with ℓ_i in the first case, and $p_{\ell_i} = 1 - x_i$ is a polynomial which agrees with ℓ_i in the second case. For a clause $C = (\ell_i \vee \ell_j \vee \ell_k)$ we just define $p_C(x_i, x_j, x_k) = p_{\ell_i} + p_{\ell_j} + p_{\ell_k}$. Finally, $\Phi(x_1 \dots x_n) = \bigwedge_{i=1}^m C_i(\ell_{i_1}, \ell_{i_2}, \ell_{i_3})$ which amounts to the product of polynomials $p_\Phi(x_1 \dots x_n) = \prod_{i=1}^m p_{C_i}(x_{i_1}, x_{i_2}, x_{i_3})$. It is immediate to see that the degree can be at most $3m$ □

Next we introduce the key protocol in proving **PSPACE** ⊆ **IP**.

Lemma 1. ***The Sumcheck protocol*** *Suppose that $g(x_1 \dots x_n)$ is a degree d polynomial, p a prime number with $p > d$ and k an integer and suppose that*

$$k = \sum_{b_1 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} g(b_1, \dots, b_n) \mod p \tag{2.1}$$

and moreover suppose that there is a polytime algorithm that given integers $a_1, \dots, a_n$ computes $p_\varphi(a_1, \dots, a_n)$. Then there exists a protocol which given $\widehat{k}$, if $\widehat{k} = k$ then the prover can make the verifier accept with probability 1 and if $\widehat{k} \neq k$ then any prover can only make the verifier accept with probability at most $1 - \left(1 - \frac{d}{p}\right)^n$

Proof. [AB09] Consider the following polynomial

$$q(x) = \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} g(x, b_2, \dots, b_n) \tag{2.2}$$

Note that $q(x)$ is an univariate polynomial of degree d. It is clear that if $\widehat{k} = k$ then the prover can make the verifier accept with probability 1, as stated. Suppose that $\widehat{k} \neq k$. Suppose that the prover can make the verifier accept with probability at most $1 - (1 - d/p)^{n-1}$ holds for polynomials of degree d on $n - 1$ variables. We prove the case for n variables by induction.

If $n = 1$ the verifier only needs to check that $g(1) + g(0) = k$, and it follows easily that the hypothesis is true.

Now suppose that $n \geq 2$. The protocol starts by the Prover sending a polynomial $h(x)$ to the verifier. If $h(x) = q(x)$ clearly the verifier will accept with probability 1. Therefore, suppose that the prover sends a polynomial $h(x) \neq q(x)$. As before, the verifier computes $h(0) + h(1) = \widehat{k_1}$. If $\widehat{k_1} \neq k$ the verifier rejects. Suppose that $h(0) + h(1) = \widehat{k_1} = k$, so the verifier cannot reject yet. Consider the polynomial $s(x) = h(x) - q(x)$. Since $h(x), q(x)$ have degree d, so does $s(x)$. Therefore $s(x)$ has at most d roots, and it follows that $h(x)$ and $q(x)$ can agree on at most d values. Next, the verifier picks $a \in GF(p)$ uniformly at random and asks the prover to recursively prove

$$h(a) = \widehat{k_2} = \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} g(a, b_2, \ldots, b_n) \tag{2.3}$$

By the previous obseravtion, the probability that eq. (2.3) holds is $\frac{d}{p}$. Therefore, making use of the inductive hypothesis, the probability that the Prover succeeds in fooling the Verifier is at most$(1 - d/p) \cdot (1 - d/p)^{n-1} = (1 - d/p)^n$ □

Equipped with this proposition we can now show the following result

Theorem 8. *[AB09]* $\#\mathbf{SAT_d} \in \mathbf{IP}$

Proof. We have a pair $(\Phi(x_1 \ldots x_n), a)$ where $\Phi(x_1 \ldots x_n)$ is a 3-CNF formula and $a \in \mathbb{N}$. We need a protocol for the prover P to convince the verifier V that indeed $(\Phi(x_1 \ldots x_n), a) \in \#\mathbf{SAT_d}$. The number of satisfying assignments can be expressed as

$$\sum_{b_1 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} p_\Phi(b_1 \ldots b_n)$$

Therefore, the prover wants to convince the verifier that this sum is exactly a. First, the prover sends a prime $p \in (2^n, 2^{2n}]$. The verifier can verify in polynomial time that indeed p is a prime. Next, applying lemma 1 we get that the Prover can make the Verifier accept with probability 1 if $(\Phi(x_1 \ldots x_n), a) \in \#\mathbf{SAT_d}$ and if $(\Phi(x_1 \ldots x_n), a) \notin \#\mathbf{SAT_d}$ then the verifier rejects with probability at least $(1 - 3m/p)^n$ and the prover can pick a suitable p in the range stated so that the result follows. □

Theorem 9. PSPACE $\subseteq$ IP

Proof. [AB09]**Sketch main idea** In order to proof this inclusion, it is enough to proof that `TQBF` (i.e the set of true quantified boolean formulas) has a protocol in **IP**, since `TQBF` is a **PSPACE**-complete problem. Then, given a formula $\Psi = \forall x_1 \exists x_2 \cdots \exists x_n \psi(x_1 \ldots x_n)$ where ψ is quantifier free, thus using the idea from the previous case we have that $\Psi \in$ `TQBF` if we have

$$0 \neq \prod_{b_1 \in \{0,1\}} \sum_{b_2 \in \{0,1\}} \prod_{b_3 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} p_\psi(b_1 \ldots b_n) \tag{2.4}$$

The first idea would be to reuse the sumcheck protocol used in for the $\#\mathbf{SAT_d}$ case but note that the products increase the degree of the polynomial, and so if we try to use a polynomial similar to that in eq. (2.2) the degree could go as high as 2^n which would be intractable for the polynomial verifier. The solution is to note that we only care for those values of $p_\psi(z)$ for $z \in \{0,1\}^n$. Therefore we introduce an operator L_{x_i} which on a polynomial $h(x_1 \ldots x_i \ldots x_m)$ is defined as

$$L_{x_i} h(x_1 \ldots x_i \ldots x_m) = x_i \cdot h(x_1 \ldots 1 \ldots x_m) + (1 - x_i) \cdot h(x_1 \ldots 0 \ldots x_m)$$

We think of the quantifiers $\exists x_i$ and $\forall x_i$ as operators also in the following way:

$$\exists x_i h(x_1 \dots x_i \dots x_m) = h(x_1 \dots 0 \dots x_m) + h(x_1 \dots 1 \dots x_m)$$

$$\forall x_i h(x_1 \dots x_i \dots x_m) = h(x_1 \dots 0 \dots x_m) \cdot h(x_1 \dots 1 \dots x_m)$$

Finally, because we are only interested in the case were the variables are $\{0,1\}$ we are free to linearize at will. Therefore we will use the following expression where $\exists, \forall$ and L are regarded as operators that act on polynomials where the order of application is from the right (i.e we start to apply by the "end" of the quantifier stack)

$$\forall x_i L_{x_1} \exists x_2 L_{x_1} L_{x_2} \forall x_3 L_{x_1} L_{x_2} L_{x_3} \cdots \exists x_n L_{x_1} \cdots L_{x_n} p_\psi(x_1 \dots x_n)$$

to make sure that the intermediate polynomials that appear in the sumcheck protocol have low degree.

We now just need to make some adjustments to the sumcheck protocol, and the correctness follows as in the theorem 8. For a (proper) proof the reader is directed to [AB09] (Chapter 8, proof of theorem 8.19) or [Sha92]. □

2.3 Pseudorandom generators from circuit lower bounds

We will need the following result of [KM02]. This result is based on work from [NW94] and essentially guarantees the existance of a pseudorandom generator if given access to a hard function.

Theorem 10. *[NW94],[KM02] For every $\epsilon > 0$ there exists $\delta < \epsilon$ and $e \in \mathbb{Z}$ such that given random access to a boolean function on n^δ with circuit complexity at least $n^{\delta \cdot e}$, there is a pseudorandom generator $G : \{0,1\}^\epsilon \to \{0,1\}^n$ computable in time $2^{O(n^\epsilon)}$ which fools circuits of size n.*

We will introduce a few definitions and prove the lemma that is key in proving theorem 10. First we need to introduce some new concepts. Let $f : \{0,1\}^* \mapsto \{0,1\}$ be a Boolean function. The hardness of f on inputs of length n denoted $H_f(n)$ is the largest integer k such that for any circuit C with $|C| \leq k$ the following holds:

$$\left| \Pr_{x \in \{0,1\}^n} [C(x) = f(x)] - \frac{1}{2} \right| < \frac{1}{k}.$$

The complexity of f on inputs of length n denoted $C_f(n)$ is the smallest integer k such that there exists a circuit C with $|C| = k$ such that $\forall x \in \{1,0\}^n \; C(x) = f(x)$. We will allow ourselves to use languages in place of functions. That is, given a language L decided by some Turing machine M (i.e, $L(M) = L$), the complexity of L on inputs of length n (which we will denote by L_n) is defined as the complexity of the restriction of the boolean function computed by M to inputs of length n. Likewise, we define the notion of hardness of L. A way to make sense of the above two concepts is the following: The circuit complexity of a function is the minimum size required by a circuit such that on input x the circuit produces the same answer as $f(x)$. For instance, if given a function f any circuit C of size n fails on at least one input (i.e there exists x such that $C(x) \neq f(x)$) then f has circuit complexity at least n. Therefore, its a notion of worst-case hardness. On the other hand, the hardness H_f as defined here, is a notion of average hardness. If $H_f \geq n$ we expect that circuits of size n disagree with f often.

The work of Nisan and Widgerson ([NW94]) gives us a way to construct a pseudorandom generator given access to a function f with $H_f \geq n^2$. In Williams method, the hypothesis we can use is that there exists a language L such that for any constant k we have that $C_L(n) \geq n^k$ for infinitely many n. Thus, we need a way extract a function from L that has the required hardness. What we need is transformation that from a function with worst-case hardness (i.e) The following theorem, which is extracted from [BFNW93, RW00], gives us the required tool.

Theorem 11.

$$\forall c' \, \exists c \, \forall L \in \mathbf{E} \, \exists L' \in \mathbf{E} \text{ such that } \overset{\infty}{\forall} n \, C_L(n) \geq n^c \implies \overset{\infty}{\forall} n \, H_{L'}(n) \geq n^{c'}$$

In words: For any polynomial $n^{c'}$ we desire, we can obtain a function with hardness $n^{c'}$ for infinitely many n if given access to a n^c worst-case hard function for infinitely many n, where c depends on c'. The hypothesis we use in Williams' method is that we have a language with worst-case hardness n^k for any k and for infinitely many n. Thus it follows that we will be able to achieve the necessary hardness.

Let's focus now in proving the key lemma required in theorem 10 just to give an idea of the kind of results needed.

We say that a collection of boolean functions $G = \{g_n \mid n \in \mathbb{N}\}$ with $g_n : \{0,1\}^{s(n)} \mapsto \{0,1\}^n$ is a pseudorandom generator if for any circuit C with $|C| = n$ we have that

$$\left| \Pr_{x \in \{0,1\}^{s(n)}} [C(G(x)) = 1] - \Pr_{y \in \{0,1\}^n} [C(y) = 1] \right| \leq \frac{1}{n}.$$

A collection of sets $S_1 \ldots S_n$ with $S_i \subseteq \{1 \ldots s\}$ is called a (k, m)-design over the universe $\{1 \ldots s\}$ if the following holds:

- For all i: $|S_i| = m$.
- For all $i \neq j$: $|S_i \cap S_j| \leq k$.

Let $S = \{S_1, \ldots, S_n\}$ be a (n, ℓ)-design over the universe $\{1, \ldots, s\}$. Given $x \in \{0,1\}^s$ and a boolean function $f : \{0,1\}^\ell \mapsto \{0,1\}$ we denote by $f_S(x)$ the n bit string resulting from concatenating $f(x|_{S_i})$ for all $1 \leq i \leq n$, where $x|_{S_i}$ denotes the restriction of x to the subset indexed by S_i.

Theorem 12. *[NW94],[KM02] Let $f : \{0,1\}^\ell \mapsto \{0,1\}$ be such that $H_f(\ell) \geq n^2$ and let S be a $(\log n, \ell)$-design over the universe $\{1, \ldots, s\}$. Then, $G\{0,1\}^s \mapsto \{0,1\}^n$ defined as $G_n(x) = f_S(x)$ is a pseudorandom generator.*

Proof. The proof works by contradiction. Suppose that G is not a pseudorandom generator. Then for some circuit C with $|C| = n$ the following holds

$$\Pr_{x \in \{0,1\}^s} [C(G(x)) = 1] - \Pr_{y \in \{0,1\}^n} [C(y) = 1] > \frac{1}{n}. \tag{2.5}$$

Note the lack of absolute value. We will assume that in this case eq. (2.5) holds. For the simmetric case essentially the same reasoning applies, in one case we predict the i-th bit from the previous ones and in the other from the posterious ones. First we will show that these implies that the some bit of $f_A(x)$ can be predicted from the previous ones. Define for all $0 \in \{1 \ \ldots \ l\}$ a distribution D_i over $\{0,1\}^n$ as follows:

$$D_i = f(\alpha|_{S_1}) \circ \cdots \circ f(\alpha|_{S_i}) \circ \beta_1 \circ \cdots \circ \beta_{n-i}$$

where α is uniformly distributed over $\{0,1\}^s$ and β is uniformly distributed over $\{0,1\}^{n-i}$. Note that what we are doing is some kind of "continuous" transformation, from the distribution defined using the pseudorandom generator, to the totally random distribution wihout using the pseudorandom generator. The i-th distribution consists of strings that agree with $f_S(x)$ up to the i-th position and then pick the remaining bits at random. We can rewrite eq. (2.5) left hand side as

$$\Pr_{z \in D_0} [C(z) = 1] - \Pr_{y \in D_n} [C(y) = 1] = \sum_{i=1}^{n} \Pr_{z \in D_{i-1}} [C(z) = 1] - \Pr_{z \in D_i} [C(z) = 1]. \tag{2.6}$$

It follows from eq. (2.5) and eq. (2.6) that for some $1 \leq j \leq n$

$$\Pr_{y \in D_{j-1}}[C(y) = 1] - \Pr_{z \in D_j}[C(z) = 1] \geq \frac{1}{n^2}. \tag{2.7}$$

We can set up a predictor $P(x)$ for $f(x)$:

1. Set $\beta|_{M^j} = x$ and pick the rest of β uniformly at random.
2. Set up a binary string $z = \rho_1 \circ \cdots \circ \rho_j \circ f(\beta|_{M^{j+1}}) \circ \cdots \circ f(\beta|_{M^n})$ where $\rho_1 \ldots \rho_j$ are picked uniformly at random from $\{0,1\}$ and β is picked uniformly at random from $\{0,1\}^\ell$.
3. Let $w = C(z)$. Output $w \oplus \overline{\rho_j}$.

Thus, to have a compact definition, this predictor for the j-th bit is nothing more than $P(x) = C(z) \oplus \overline{\rho_j}$. We claim that:

$$\Pr_{\rho,\beta}[P(x) = f(x)] - \frac{1}{2} = \Pr_{y \in D_{j-1}}[C(y) = 1] - \Pr_{z \in D_j}[C(z) = 1]. \tag{2.8}$$

Let $\overline{y}$ denote y with the j-th bit flipped. Consider now eq. (2.8) conditioned on ρ and β:

- If $C(y) = C(\overline{y})$ then $C(y)$ and $C(z)$ are equally likely to accept, thus the right side vanishes. With respect to the left hand side, the probability of having the equality $P(x) = f(x)$ comes down to hitting the right j-th bit which is taken at random in $P(x)$. Thus, the left hand side vanishes as well.
- If $C(y) \neq C(\overline{y})$ then we have that $C(z)$ accepts with probability $\frac{1}{2}$, and it is the case that $P(x) = g(x)$ whenever $C(y) = 1$.

This justifies eq. (2.8). From eq. (2.7) and eq. (2.8) follows

$$\Pr_{\rho,\beta}[P(x) = f(x)] \geq \frac{1}{2} + \frac{1}{n^2}. \tag{2.9}$$

Since $x = \beta|_{S_i}$ is distributed uniformly, by an averaging argument we can conclude that there exists ρ, β such that

$$\Pr_{x \in \{0,1\}^\ell}[P(x) = f(x)] \geq \frac{1}{2} + \frac{1}{n^2}. \tag{2.10}$$

Since we have $|S_i \cap S_j \leq \log n|$ for $i \neq j$ each S_k $j < k \leq n$ can only depend on at most $\log n$ bits of x and therefore can be computed by circuits of size at most n on inputs x. There are at most $n-1$ components and since we have fixed ρ and $|C| \leq n$ it follows that $P(x)$ can be computed by a circuit of size $(n-1)n + n = n^2$. This fact in conjunction with eq. (2.10) yield $H_f < n^2$ contradicting the hypothesis. □

Finally, the following lemma guarantees that construction such a design as the one required for the previous result is possible.

Lemma 2. *[NW94] Given n, m with $\log n \leq m \leq n$ there exists a $(\log n, m)$-design over a universe of size $O(m^2)$. Moreover, this design is computable by a Turing machine using $O(\log n)$ space.*

2.4 Fixed polynomial circuit lower bounds for PSPACE

Next, we need a result by [Kan82] which gives a lower bound on the size of circuits computing **PSPACE**. The argument used is a diagonalizing one plus a counting argument.

Lemma 3. *For any* $q \in \mathbb{N}$ $\mathbf{PSPACE} \nsubseteq \mathbf{io\text{-}SIZE}(n^q)$

Proof. First we will show that for some n, there exists a boolean function $f : \{0,1\}^n \to \{0,1\}$ that doesn't have circuits of size n^q.
For any given n, there are 2^{2^n} different boolean functions on n variables. On the other hand, with n^q gates we have at most $3^{n^q} \cdot n^{2q \cdot n^q}$ different circuits, which asymptotically is $2^{n^{q+1}}$ at most, and $2^{n^{q+1}} << 2^{2^n}$. We can find a tightest bound by considering the set S consisting of just the first n^{2q} binary strings of length n (in lexicographical order) instead of the set of all possible binary string of length n. There are $2^{n^{2q}}$ possible boolean functions in n variables whose domain is a subset of S, and $2^{n^{q+1}} < 2^{n^{2q}}$.

Next, we will show that for any given input length n, any boolean function $f : \{0,1\}^n \to \{0,1\}$ has a circuit that computes it.
Let $D \subseteq \{0,1\}^n$. There is a circuit C^* of size $O(|D| \cdot n)$ deciding D. Consider the following formula $\Phi_D(x) = \bigvee_{\alpha \in D} \bigwedge_{i=1}^{n} x_i = \alpha_i$. Seeing $\Phi_D(x)$ as a circuit its size is $O(|D| \cdot n)$.

We can encode each gate using $O(log\, s + log\, n)$ bits, where s is the total number of gates of the circuit and n is the length of the input. Therefore, any circuit with input length n and size s can be encoded using $O(s \cdot (log\, n + log\, s))$ bits. Hence, C_D requires $O(|D| \cdot n \cdot (log\, n + log\, (|D| \cdot n)))$ bits to be encoded.

From the previous point follows that a circuit C^* whose domain is a subset of S will require at most $O(n^{2q} \cdot log\, n)$ bits, which is asymptotically upper bounded by n^{2q+1} bits. Also, any circuit of size $O(n^q)$ can be encoded using $O(n^q \cdot log\, n)$ bits, which is asymptotically upper bounded by n^{q+1}.

The following algorithm Turing machine finds a circuit that computes a function not computable using $O(n^q)$ gates. We go through all possible circuit representations using $n^{2(q+1)}$ bits and find the one whose associated function cannot be computed using $O(n^q)$ gates, which we are guaranteed to find for large enough n.

$M(x)$:
1. Pick a string i with $|i| \leq 2^{|x|^{2q+1}}$
2. Pick a string z with $|z| \leq 2^{|x|^{q+1}}$
3. If $\exists y\ C_i(y) \neq C_z(y)$ return C_i otherwise back to 1

The idea is that we have a first loop (step 1) that iterates over all circuits whose description takes at most $2^{|x|^{2q+1}}$ bits as long as we have not verified that all circuits whose description is smaller compute a different boolean function. Fix this circuit C_i. In the second loop, we look for a circuit whose description is smaller than that of C_i but computes the same function (the third loop is responsible for checking this equality). If for each smaller circuit C_j, we find that for some $z \in 0, 1^n \qquad C_i(z) \neq C_j(z)$ then we let $C^* = C_i$ and output $C^*(x)$
The looping part of M has running time $2^{|x|^{2q+1}} \cdot 2^{|x|^{q+1}} \cdot 2^{|x|} = 2^{O(|x|^{3q})}$. Moreover, we only need polynomial space, since we only need to consider ,at a given time, two circuits whose description is of polynomial size. To output $C^*(x)$ we only require polynomial time, therefore, $L(M)$ is in **PSPACE**. By construction, C^* requires size greater than $O(n^q)$, and therefore the function computed by M must have circuits of size greater than $O(n^q)$ for large enough n, because M is just simulating C^*. Since the bounds we have given are asymptotic, we have that for all $n > n_0$ for some constant n_0, we are guaranteed to find such a circuit C^* that requires more than $O(n^q)$ gates. Therefore, $L(M)$ could only have circuits of size $O(n^c)$ for finitely many input lengths, hence $L(M)$ cannot be in $\mathbf{io\text{-}SIZE}(n^q)$, but $L(M)$ is in **PSPACE**.

□

2.5 Efficient reductions to 3-SAT

Next we discuss one of the key ingredients in Williams' method, the efficient reduction from non-deterministic computations to satisfiability to Boolean formulas. The main idea is based on the proof of Cook-Levin's theorem ([Coo71]) which shows that satisfiability of Boolean formulas is **NP**-complete. We will see that the size of the formula that this method yields is quadratic. Later on this chapter we will discuss what implications this have and why do we need an even more efficient formula. But first to get an idea we discuss this inefficient version.

Theorem 13. *[Coo71] Let L be a language in* $\mathbf{NTIME}(T(n))$. *Deciding $x \stackrel{?}{\in} L$ can be reduced to the satisfiability of a 3-CNF formula Φ_x of size $O(T(|x|)^2)$. Moreover, given i, we can compute the i-th clause of Φ_x in time $O(log^d(T(|x|)))$ for a fixed d.*

Proof. For the rest of the proof we will assume that the Turing machine has only one tape. It is straightforward to extend this proof to the multitape case, but the notation would get too clutered.

Let the following tableau represent the computation of the Turing machine M that decides L. Let x be an input with $|x| = n$ and $T(n) = t$. From this tableau, we can define a propositional

	Y	0	$j-1$	j	$j+1$	n			t-1	State
0	y_0	x_0		$\ldots$		x_{n-1}	$\triangleright$	$\ldots$	$\triangleright$	q_{start}
i	y_i			α						q
$i+1$	y_{i+1}		$\rho_{y_{i+1}}$	$\sigma_{y_{i+1}}$	$\gamma_{y_{i+1}}$					$q_{y_{i+1}}$
$t-1$	y_{t-1}	1	$\triangleright$	$\ldots$					$\triangleright$	q_{accept}

Table 2.1: The Turing's machine tableau of computation. Y is the non-deterministic bit

Φ_x, such that it is satisfiable if, and only if, there exists an accepting path in M on input x. Let δ_i for $i \in \{0, 1\}$ be the non-deterministic transition function. Define the following propositional variables:

- $S_\alpha^{(i,j)}$ meaning that symbol α is at cell j on step i.
- E_q^i meaning that the machine is on state q on step i.
- $H^{(i,j)}$ meaning that the machine's header is at cell j on step i.
- C^i meaning that at step i we choose δ_1 (if false, means that we choose δ_0 instead).

Let δ_i be of the following form:

- $\delta_0(\alpha, q) = (\beta_0, q_0, \Delta_0)$.
- $\delta_1(\alpha, q) = (\beta_1, q_1, \Delta_1)$.

Where $\Delta_i \in \{-1, 0, 1\}$ is where should the machine's head move represented as an offset from the actual position. Then for $i \in \{1, \ldots, t-1\}$ We define the propositional formulas that describe the tableau of M. We start by the "non-degenerate"case, which is the case for the cells that fit the "T shape", i.e, for those that $j \in \{1, \ldots, t-2\}$. For these cells we define the

following formulas:

$$\forall(i,j)\,\forall(\alpha,q)\in \mathrm{dom}(\delta_0) \begin{cases} H^{(i,j)} \wedge S_\alpha^{(i,j)} \wedge E_q^i \wedge C_0^i \to S_{\beta_0}^{(i+1,j)} \\ H^{(i,j)} \wedge S_\alpha^{(i,j)} \wedge E_q^i \wedge C_0^i \to E_{q_0}^{i+1} \\ H^{(i,j)} \wedge S_\alpha^{(i,j)} \wedge E_q^i \wedge C_0^i \to H^{(i+1,j+\Delta_0)} \end{cases} \tag{2.11}$$

$$\forall(i,j)\,\forall(\alpha,q)\in \mathrm{dom}(\delta_1) \begin{cases} H^{(i,j)} \wedge S_\alpha^{(i,j)} \wedge E_q^i \wedge C_1^i \to S_{\beta_1}^{(i+1,j)} \\ H^{(i,j)} \wedge S_\alpha^{(i,j)} \wedge E_q^i \wedge C_1^i \to E_{q_1}^{i+1} \\ H^{(i,j)} \wedge S_\alpha^{(i,j)} \wedge E_q^i \wedge C_1^i \to H^{(i+1,j+\Delta_1)} \end{cases} \tag{2.12}$$

$$\begin{cases} \neg S_\alpha^{(i,j)} \vee \neg S_\beta^{(i,j)} & \forall(i,j)\,\forall(\alpha,\beta) \quad \alpha \neq \beta \\ \neg E_q^i \vee \neg E_r^i & \forall i\,\forall(q,r) \quad q \neq r \\ \neg H^{(i,j)} \vee \neg H^{(i,k)} & \forall i\,\forall(j,k) \quad j \neq k \\ \neg C_0^i \vee \neg C_1^i & \forall i \\ C_0^i \vee C_1^i & \forall i \end{cases} \tag{2.13}$$

Note that if $j = 0$ and the transition has $\Delta_i = -1$, then we will not define the implications in (2.11) (since we would be moving the header out of bounds). Likewise, if $j = t$ then we will not define the implications in (2.12) if the transition has $\Delta_i = 1$.

Finally, we miss the cases $i \in \{0, t\}$. When $i = 0$ we just set to true the variables $\{S_{x_j}^{0,j}\}$ for $j \in \{0, \ldots, n-1\}$ and the rest of symbol variables with index $(0, \_)$ are set to false, to ensure that the only thing in the tape at the start is the input. Finally, the case $i = t$ correspond to the end of the execution, and so these variables are defined by the previous formulas concerning the correct execution of M.

We need to ensure that the computation accepts, hence we set $E_{q_{\text{accept}}}^t$ to true. We also need that the initial conditions of the computation are correct. Therefore, on input x of length n, we set as true $H^{(0,0)}$ because the machine's head must start at position 0, $S_{x_i}^{(0,i)}$ for all $i \in \{0, \ldots, n-1\}$ to correctly represent the contents of the input tape and finally $E_{q_{\text{start}}}^0$ because the very first state must be the starting one.

All the formulas are easily transformed into a 3-CNF with only increasing the formulas size by a constant factor. Note that the formula has a very regular structure. The formula has three main parts. We can assume that the first d clauses, for some constant d that depends on the length of the input, correspond to the initial and final conditions. The remaining clauses correspond to "packs" of clauses. Each pack corresponds to the previously defined formulas for each possible pair (i, j). Therefore, given and index of a clause k, we can easily output the clause by determining to what part of the first d clauses or if it belongs to the other part of the formula, to what pair (i, j) it corresponds. Once this is known (which we can know by just doing arithmetic with the index) we can output the appropriate clause. Thus we only require time proportional to doing arithmetic with the index, thus time $O(log^{O(1)} t)$.

□

The problem of theorem 13 is that we need to ensure the consistency of the tape contents at all steps. This ends up making the formula have quadratic size. The idea will be to delay this checking. We will first ensure that the headers and the state transitionsa are valid, and after we will ensure that the tape contents are consistent. Instead of taking snapshots of the whole tape, we will just take snapshots of what each header can "see". Then, to enforce the consistency of the tapes we just need to make sure that the accesses to the tape cells are consistent. This will be made formal in theorem **??**

The key theorem that yields this efficient propositional formula, which is also efficiently computable is the following

Theorem 14. *Let L be such that $L \in \mathbf{NTIME}(T(n))$. Let M be a non-deterministic Turing machine such that $L(M) = L$. Then given x deciding if $x \overset{?}{\in} L$ can be reduced to the satisfiability of a propositional formula Φ_x with $|\Phi_x| = O(T(n)\log^{O(1)} T(n))$. Moreover, given $i \in \{1, \ldots, \log(c \cdot T(n)\log^{O(1)} T(n))\}$ we can compute the i-th clause of Φ_x in time $O(\log^{O(1)} T(n))$.*

In [Wil10] this result is credited to [Tou01] and [FLvMV04]. We have not been able to find a published complete proof of theorem 14. The best we have managed is to follow the line of work suggested by [Tou01] and reuse a result on the simulation of random access Turing machines by [Rob91]. Unfortunately, we have only been able to proof the first part of theorem 14. Namely, the fact that there exists such formula of the given size, but we haven't been able to give a formal proof of its efficient computability. It *seems* like it is indeed true, because the Φ_x is very "regular". Therefore, for the rest of the work we will work with **??**, although at the present time we cannot offer a complete poroof.

Next, we will proof the first half. We will proof that such Φ_x exist and that indeed it has size quasi-linear. First we will study what is known as a permutation network. For the purposes of this section, n will be assumed to be of the form $n = 2^k$ for some $k \in \mathbb{N}$.

Definiton 7. *A* permutation network *can be defined as a circuit, with inputs $\boldsymbol{u} = (u_1 \ldots u_n)$, outputs $\boldsymbol{v} = (v_1 \ldots v_n)$ and a set of auxiliary variables, with the property that we can force the outputs to be any permutation of the inputs by some setting of the auxiliary variables.*

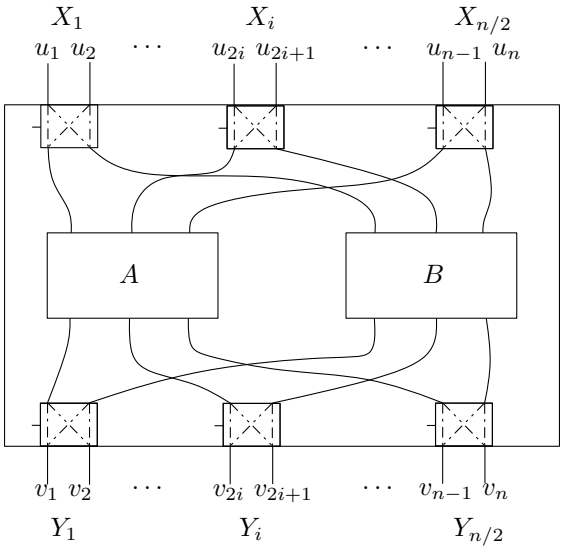

Figure 2.1: Depiction of a permutation network

In the picture, A, B are permutation networks for $n/2$ elements, and the boxes with dotted lines connected to the pairs of inputs/outputs are "switching gates". The "switching gates" are circuits with auxiliary inputs that allow to flip the inputs/outputs. In order to build a a permutation network, we will use a couple lemmas on permutations by [Wak68]. This lemmas will give us a "decomposition" of the permutation that will allows us to build the permutation network. To develop an intuition, what we need is some kind of "decomposition" of the permutation that is "rigid enough" so as to be implemented as a circuit. To that end, we will prove a theorem stating, roughly, that any permutation can be "rigidly" decomposed.

Before we prove this results on permutations we need a few definitions:

Definiton 8.
Define $X_i = \{2i, 2i+1\}$ and $Y_j = \{2j, 2j+1\}$ for $i, j \in \{1, \ldots, \frac{n}{2}\}$.
Given a permutation $\pi \in S_n$, define $K_i = \{\pi(2i), \pi(2i+1)\}$

Essentially, what we do is "pair up" consecutive inputs and consecutive outputs and give it a name (X_i, Y_j). What we will see is that given π we can find a restriction of it which permutes the X_i. More formally, we will construct a permutation $\sigma \in S_{n/2}$ of the X_i's with the property that if $\sigma(i) = j$ then there exists $a \in X_i$ and $b \in Y_j$ such that $\pi(a) = b$. So, we have a permutation of the "pairings" that is embedded in the original permutation.

Let's state and prove formally this theorem:

Theorem 15. *[Wak68] For every $\pi \in S_n$ there exists $\sigma \in S_{n/2}$ such that if $\sigma(i) = j$ then $\exists a, b \in \{0, 1\}$ such that $\pi(2i + a) = 2j + b$*

Proof. Consider the family of sets $\mathcal{K} = \{K_1, \ldots, K_{n/2}\}$. We need to pick a set of representatives for this family. Suppose that $\mathcal{A} = j_1, \ldots, j_{|K|}$ is a set of representatives for $\mathcal{K}$. Then, we can construct σ as follows:
Fix i. Let $j_i \in A$ be the representative for K_i. We define $\sigma(i) = j_i$. By definition of K_i there exists $i_0 \in X_i$ with $\pi(i_0) = j_0$ with $j_0 \in Y_j$, therefore we have a function that maps each X_i to some Y_j while having the property of "preserving" π in the sense above descrived. We now need to see that σ is indeed a bijection between $\boldsymbol{X} = \bigcup_i X_i$ and A and thus a permutation. Consider two X_p, X_q. Suppose $p \neq q$ and let $\ell = \sigma(p) = \sigma(q)$. This would mean that p and q are represented by the same ℓ, which is a contradiction with the definition of representative. Let's see now that σ is surjective. This follows easily from the fact that σ is injective and its image (the set A) has the same cardinality as its domain, which is $\boldsymbol{X}$.

The only thing left to show is that indeed there exists a set A of representatives for $\mathcal{K}$. To do that, we will show that $\mathcal{K}$ fulfills the *marriage* condition and thus by Hall's theorem we will have the desired set A.
Let $W \subseteq \mathcal{K}$, we want to see that

$$|W| \leq \Big| \bigcup_{K_i \in W} K_i \Big| \tag{2.14}$$

To see it, first note that K_i either has one element or exactly two. Fix X_i and consider the image of its elements under the permutation, that is consider $(\pi(2i), \pi(2i + 1))$. Either $\sigma(2i), \sigma(2i + 1))$ both belong to the same Y_j or not. If they belong to the same Y_j, then $K_i = \{j\}$. Otherwise, $K_i = \{j_1, j_2\}$ for j_1, j_2 according to σ. Let's distinguish both cases when considering the union of K_i's. Define $S = \{K_i \mid K_i \text{ is a singleton}\}$ and $D = W \setminus S$. Rewrite $\bigcup_{K_i \in W} K_i$ as $\bigcup_{K_i \in S} K_i \cup \bigcup_{K_i \in D} K_i$

- If $K_i \in S$, thus of the form $K_i = \{j\}$, then it will contribute with j to the union because for any other $i' \neq i$ $j \notin K_{i'}$
- If $K_i \in D$, thus of the form $K_i = \{j_1, j_2\}$, in principle we don't know if it contributes with some element to the union, because it could overlap with some other $K_{i'}$ in D. But we know for any arbitrary j', it can only belong to two different sets K_i. Therefore, if we are taking the union of sets with exactly two elements, and each element can at most appear twice, we have that each set of the union contributes at least one element to the union.

It follows then that eq. (2.14) holds for the family $\mathcal{K}$ and thus by *Hall's* theorem the result follows. □

Corollary 16. *Given $\pi \in S_n$ there exists $\sigma_0, \sigma_1 \in S_{n/2}$ such that*

$$\bigcup_i \Big\{(i, \pi(i))\Big\} = \Big\{\bigcup_j \Big\{(j, \sigma_0(j))\Big\}\Big\} \cup \Big\{\bigcup_j \Big\{(j, \sigma_1(j))\Big\}\Big\}$$

Proof. First, note that the construction in theorem 15 has the property that for each X_i only one of the two elements belong to the domain of σ and for each of the Y_j only one of the two elements belong to the image of σ. Now, let $\sigma_0 = \sigma$. We will construct σ_1. For each Y_j let v_b denote the unused element (i.e $b \notin im(\sigma)$). Then, there is some X_i such that $a \in X_i$, and $a \notin dom(()\sigma)$. Thus, define $\sigma_1(i) = j$. It is straightforward to see that we can keep pairing them off with the same reasoning, and we get a permutation of $\sigma_1 \in S_{n/2}$ such that

$$\bigcup_j \Big\{ \ (j, \sigma_1(j)) \Big\} = \bigcup_i \Big\{ (i, \pi(i)) \Big\} \setminus \bigcup_j \Big\{ (j, \sigma_0(j)) \Big\}$$

□

The theorem 15 and corollary 16 are the foundation for the construction of a *permutation network*. As before $\boldsymbol{u} = (u_1, \ldots, u_n)$ and $\boldsymbol{v} = (v_1, \ldots, v_n)$ will be the inputs and ouputs of the circuit. By corollary 16 we can split any permutation into two permutations that permute the X_i. Therefore, we only that for each $u_{2i}, u_{2i+1} \ \in X_i$ we need to choose which elements goes to π_1 and which one goes to π_2. The same must be done for each $v_{2j}, v_{2j+1} \ \in Y_j$. Once this is done, we just apply corollary 16 recursively. This gives raise to the circuit depicted in fig. 2.1. Where A, B are permutation networks for $n/2$ inputs. The depth of the circuit is easy to see that is $O(\log n)$. If $S(n)$ is the size of the permutation network for n inputs, it is easy to see that $S(n) = 2 \cdot S(n/2) + c$ where c is the constant size of the "switching" gates. Thus, $S(n) = O(n \ \log n)$

2.5.1 A formula for permutations

Based on the permutation network introduced before, Stearns and Hunt ([SHH86]) devise a formula which fixed the inputs $\{u_1, \ldots, u_n\}$ and outputs $\{v_1, \ldots, v_n\}$ is satisfiable if, and only if, there exists an assigment to the auxiliary variables corresponding to some permutation $\pi \in S_n$ such that for all $z \in \{1, \ldots, n\}$ $u_z = v_{\pi(z)}$. Here we will reproduce their main result. First some notation. **Bold** face letters will indicate a vector. Moreover $\boldsymbol{x}^k$ will indicate a vector with 2^k vectors as components, that is $\boldsymbol{x}^k = (x_1, \ldots, x_{2^k})$. Moreover, let $\boldsymbol{x}_i = (x_{i,1}, \ldots, x_{i,m})$.

Lemma 4. *[SHH86] Let $a_{i,j}$ and $b_{i,j}$ be boolean variables with $1 \leq i \leq 2^k \quad 1 \leq j \leq m$ for some m, k. There exists a formula F_k such that:*

1. *F_k has $a_{i,j}$, $b_{i,j}$ as variables along with $2 \cdot m(k-1) \cdot 2^k$ auxiliary extra variables*
2. *$|F_k| = 2c(2k-1) \cdot 2^k$ where c is the size of the "switching" gate*
3. *For all possible assignaments α of the variables $(a_{i,j}$, $b_{i,j})$, $F|_\alpha$ is satisfiable if, and only if, there exists a permutation π such that:*

$$a_{i,j} = b_{\pi(i),j}$$

for all i, j

In vector notation this translates to $\boldsymbol{a}_i = \boldsymbol{b}_{\pi(i)}$. Note that the permutation is defined at the vector "level". That is we permute the whole vector leaving its components as is.

Proof. Define $\boldsymbol{x}_i = (x_{i,1} \ \ldots \ x_{i,m})$. If we only had 2 vectors to permute then the formula

$$P(\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{y_1}, \boldsymbol{y_2}) = ((\boldsymbol{x_1} = \boldsymbol{y_1}) \wedge \boldsymbol{x_2} = \boldsymbol{y_2})) \vee ((\boldsymbol{x_1} = \boldsymbol{y_2}) \wedge \boldsymbol{x_2} = \boldsymbol{y_1}))$$

Equality between vectors is defined as equality component wise as one could expect. We will work by induction on k. First, if $k = 1$, then the formula

$$F_1(\boldsymbol{x}^1, \boldsymbol{y}^1) = P(\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{y}_1, \boldsymbol{y}_2)$$

For $k = i + 1$, assume that $F_i(\boldsymbol{x}^i, \boldsymbol{y}^i)$ is satisfiable if, and only if, $\boldsymbol{y}^i$ is a permutation of the $\boldsymbol{x}^i$. Define:

$$F_{i+1}(\boldsymbol{x}^{i+1}, \boldsymbol{y}^{i+1}) = \bigwedge_{m=1}^{2^i} P(\boldsymbol{x}_{2m}, \boldsymbol{x}_{2m+1}, \boldsymbol{p}_m, \boldsymbol{r}_m) \bigwedge_{m=1}^{2^i} P(\boldsymbol{q}_m, \boldsymbol{s}_m, \boldsymbol{y}_{2m}, \boldsymbol{y}_{2m+1}) \wedge F_i(\boldsymbol{p}^i, \boldsymbol{q}^i) \wedge F_i(\boldsymbol{r}^i, \boldsymbol{s}^i)$$

Notice that the formula P corresponds to the "switching gates" of the permutation network. That is, P is true, if and only if, the assigments to its parameters behave like the "switching gate". Let's verify the correctness of this construction, in other words, let's verify that property 3 is fulfilled. Suppose that we have an assigment to the a's and b's vectors that follows a given permutation π, in other words we have

$$a_{i,j} = b_{i,\pi(j)} \quad \forall i, j$$

First, suppose that $k = 1$. Suppose that we have an assignment to $(\boldsymbol{a}_1, \ \ldots \ , \boldsymbol{a}_m)$ and $(\boldsymbol{b}_1, \ \ldots \ , \boldsymbol{b}_m)$ such that each $\boldsymbol{b}_i$ is a permutation of the corresponding $\boldsymbol{a}_i$. By definition, $F_1(\boldsymbol{a}^1, \boldsymbol{b}^1) = P(\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{b}_1, \boldsymbol{b}_2)$ and clearly we have a permuting assignment for the $\boldsymbol{a}$'s and $\boldsymbol{b}$'s if, and only if, $P(\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{b}_1, \boldsymbol{b}_2)$ is true. Suppose that $K = i+1$ and that property 3 holds for $K = i$. Let α be an assigment to F_{i+1} fulfilling property 3 for $\boldsymbol{a}^{i+1}, \boldsymbol{b}^{i+1}$. By induction, we have that $F_i(\boldsymbol{p}^i, \boldsymbol{q}^i)$ and $F_i(\boldsymbol{r}^i, \boldsymbol{s}^i)$ are satisfiable if, and only if, the assigments to the vectors $\boldsymbol{p}^i, \boldsymbol{q}^i$ and $\boldsymbol{r}^i, \boldsymbol{s}^i$ are permutations, which is plausible since we are free to extend α for these vectors because $F_i = F_i|_\alpha$ since α only assigns $\boldsymbol{a}$'s and $\boldsymbol{b}$'s. Therefore we can extend α to whatever assignment that fullfils property 3 for $\boldsymbol{p}^i, \boldsymbol{q}^i$ and $\boldsymbol{r}^i, \boldsymbol{s}^i$, which is easy to see that it exists. Hence, what is left to see is that under the assignment α

$$F'|_\alpha(\boldsymbol{a}^{i+1}, \boldsymbol{b}^{i+1}) = \bigwedge_{m=1}^{2^i} P(\boldsymbol{a}_{2m}, \boldsymbol{a}_{2m+1}, \boldsymbol{p}_m, \boldsymbol{r}_m) \bigwedge_{m=1}^{2^i} P(\boldsymbol{q}_m, \boldsymbol{s}_m, \boldsymbol{b}_{2m}, \boldsymbol{b}_{2m+1})$$

is true if, and only if, α indeed fulfills property 3. Note that the pairs (z_{2m}, z_{2m+1}) are the Z_m defined in theorem 15. Therefore, $\boldsymbol{a}^{i+1}, \boldsymbol{b}^{i+1}$ are permuted under α if, and only if, each of the P formula is true because of corollary 16. Therefore, if all conjunctants are true, $F'|_\alpha$ is true and the result follows.

Let's see now how many *new* variables (i.e those that are not $\boldsymbol{a}, \boldsymbol{b}$) does the F_k construction need. It is easily verifiable that the number of new variables in F_k responds to the recurrence

$$N(k) = 2 \cdot m \cdot 2^k + 2 \cdot N(k-1)$$

and $N(1) = 0$ since we don't need auxiliary variables for the base case. It can be shown that the solution to this recurrence is $N(k) = 2^k \cdot 2m \cdot (k-1)$ as property 1 states. Finally, let's see what is the length of F_k ($|F_k|$). First, note that $|P| = 4 \cdot c$ where c is the size of the formula that asserts equality between two vectors which is easily constructed. Therefore, $|F_k| = 2 \cdot 2^{k-1} \cdot 4c + 2 \cdot |F_{k-1}|$ and $|F_1| = 4c$ since $F_1 = P$. We have that the solution to this recurrence is $|F_k| = 2^k \cdot 2c(2k-1)$ □

Finally we will introduce encodings for three predicates that will be of key importance in the reduction to satisfiabiliy that we will use. Suppose that $\boldsymbol{x} = (x_1, \ldots, x_m)$ and $\boldsymbol{y} = (y_1, \ldots, y_m)$ are Boolean vectors. We will show how to encode the predicates $A_=(\boldsymbol{x}, \boldsymbol{y}), A_<(\boldsymbol{x}, \boldsymbol{y}), A_{+1}(\boldsymbol{x}, \boldsymbol{y})$ that indicate that $\boldsymbol{x} = \boldsymbol{y}, \boldsymbol{x} < \boldsymbol{y}$ and $\boldsymbol{x} = \boldsymbol{y} + 1$ respectively.

Lemma 5. *[SHH86] Let $x_1, \ldots, x_m$ and $y_1, \ldots, y_m$ be Boolean variables. Let $\alpha = \sum_{i=1}^{m} x_i 2^{i-1}$ and $\beta = \sum_{i=1}^{m} y_i 2^{i-1}$ be the integers they represent in binary code, then there exist propositional formulas F such that*

1. *F satisfiable if, and only if, $\alpha = \beta$ with $|F| = 4m$*
2. *F satisfiable if, and only if, $\alpha + 1 = \beta$ with $|F| = m^2 + 5m - 4$*
3. *F satisfiable if, and only if, $\alpha < \beta$ with $|F| = 6m - 4$*

Proof. **of 1** We have the obvious formula

$$F(\boldsymbol{x}, \boldsymbol{y}) = \bigwedge_{i=1}^{m} ((\overline{x_1} \vee y_1) \wedge (\overline{y_1} \vee x_1))$$

□

Proof. **of 2** Let $F_1(x, y) = \overline{x} \wedge y$. Then build inductively

$$F_m(\boldsymbol{x}, \boldsymbol{y}) = (x_m \leftrightarrow y_m \wedge F_{m-1}(x_{m-1}, \ldots, x_1, y_{m-1}, \ldots, y_1)) \vee ((\overline{x_m} \wedge y_m) \wedge \bigwedge_{i=1}^{m-1} (x_i \wedge \overline{y_i}))$$

The above formula says that either the most significant bits are equal and for the rest of bits it is the case that they differ by one, or we have the case where $\alpha = 2^m - 1$ and $\beta = 2^m$. We have that $|F_1| = 2$ and $|F_m| = 6 + 2(m-1) + |F_{m-1}|$. Thus, $|F_m| = m^2 + 5m - 4$ □

Proof. **of 3** Let $F_1(x, y) = \overline{x} \wedge y$. Then build inductively

$$F_m(\boldsymbol{x}, \boldsymbol{y}) = (x_m \leftrightarrow y_m \wedge F_{m-1}(x_{m-1}, \ldots, x_1, y_{m-1}, \ldots, y_1)) \vee (\overline{x_m} \wedge y_m)$$

Similar to the previous case, either the most significant bits are equal an the property is fulfilled by the other $m-1$ bits or the most significant bit of $\boldsymbol{y}$ is 1 and that of $\boldsymbol{x}$ is 0. We have that $|F_1| = 2$ and $|(|F_m) = 6 + |F_{m-1}|$. It is easy to verify that this yields $|F_m| = 6m - 4$ □

Theorem 17. *Let L be such that $L \in \mathbf{NTIME}(T(n))$. Let M be a non-deterministic Turing machine such that $L(M) = L$. Then given x deciding if $x \stackrel{?}{\in} L$ can be reduced to the satisfiability of a propositional formula Φ_x. Moreover, $|\Phi_x| = O(T(n) \log^{O(1)}(T(n)))$.*

Proof. Let x be the input with $|x| = n$ and let $T(n) = t$. We will describe a propositional formula which will be satisfiable if, and only if, $M(x)$ has at least one accepting computation path.

Define the following propositional variables:

- $I^i_{(j,k)}$ to encode (in binary) the position of the header of the k-th tape at step i.
- $S^i_{(\alpha,k)}$ meaning that the cell being read by the header at step i contains symbol α at the start of step i.
- E^i_q meaning that the machine is at state q at the start of step i.
- C^i_j meaning that the machine picks δ_j as a transition function at step i.

With this variables we will encode each computational step. For each step i and tape k we will have a register, call it R^k_i which will describe, using the previous variables, the state of the k-th tape of the machine M when computing $M(x)$. Therefore, R^k_i will describe:

1. The position of the header of the k-th tape, i.e, the address of the cell being read on the k-th tape.

2. The state of the machine at the start of step i.

3. the content, at the beginning of step i, of the cell that is being addressed.

Now let's describe in more detail how are the previous items described using the variables introduced before. Keep in mind that this registers are just an abstraction, a way of grouping the propositional variables, that we do in order to reason about them.

$$R_i^k = \begin{cases} \{I^i_{(j,k)} \mid j \in \{1, \dots, \lceil \log t \rceil\}\ \} & \text{encoding the address of the cell being read by the header} \\ \{S^i_{(\alpha,k)} \mid \alpha \in \Gamma\} & \text{All possible symbols that can be on a tape} \\ \{E^i_q \mid q \in Q\} & \text{All possible states that the machine can have at step i} \end{cases}$$

First, we impose the obvious restrictions. Such as that we can only have one symbol read per step, only one state true per step, that we can only pick one transition function per step, etc... We call the conjunction of all this formulas α_x.

$$\begin{cases} \neg S^i_{(\alpha,k)} \vee \neg S^i_{(\beta,k)} & \forall (i,k)\, \forall (\alpha, \beta) \quad \alpha \neq \beta \\ \neg E^i_q \vee \neg E^i_r & \forall i\, \forall (q,r) \quad q \neq r \\ \neg C^i_0 \vee \neg C^i_1 & \forall i \\ C^i_0 \vee C^i_1 & \forall i \end{cases} \tag{2.15}$$

After the obvious conditions are imposed, we will make sure that this registers encode valid headers movement, and valid transitions between the machine states. We will postpone for now to check the consistency of the symbols read/written on the tape. Therefore, we just want to make sure that between register i and $i+1$ the header movement and the state transition is valid according to M's transition functions. To that effect we will make use of lemma 5. Define (where X, Y denote registers):

- $A_=(X,Y)$ which is true if, and only if, the address encoded in the register X is equal to that of Y.

- $A_{+1}(X,Y)$ which is true if, and only if, the address encoded in the register X, when added 1 (in binary) yields address encoded in register Y.

- $A_<(X,Y)$ which is true if, and only if, the address encoded in register X is less than that of Y.

- $T_<(X,Y)$ which is true if, and only if, the execution step encoded in register X is less than that of Y.

In order to have a more compact notation, suppose that the transition functions are of this form (for $i \in \{0,1\}$) $\delta_i(\alpha, q) = (\beta_i, q_i, h_i)$ where $\alpha, \beta_i \in \Gamma$ $q, q_i \in Q$ and $h_i \in \{-1, 0, 1\}$. Define also

$$A_h(X,Y) = \begin{cases} A_=(X,Y) & \text{if } h = 0 \\ A_{+1}(X,Y) & \text{if } h = 1 \\ A_{+1}(Y,X) & \text{if } h = -1 \end{cases}$$

We can think of this predicate as a "macro" to get a bit more compact notation. Note that all of this is fixed per machine so all this "macros" can be thought of being expanded when defining the Turing machine that computes the reduction.

Using this predicates we impose that the headers movement and the states transitions are correct. We call the conjunction of all this formulas β_x.

$$\forall i \, \forall (\alpha, q) \in \mathrm{dom}(\delta_0) \begin{cases} S^i_{(\alpha,k)} \wedge E^i_q \wedge C^i_0 \rightarrow A_{h_0}(R^i, R^{i+1}) \\ S^i_{(\alpha,k)} \wedge E^i_q \wedge C^i_0 \rightarrow E^{i+1}_{q_0} \end{cases} \tag{2.16}$$

$$\forall i \, \forall (\alpha, q) \in \mathrm{dom}(\delta_1) \begin{cases} S^i_{(\alpha,k)} \wedge E^i_q \wedge C^i_1 \rightarrow A_{h_0}(R^i, R^{i+1}) \\ S^i_{(\alpha,k)} \wedge E^i_q \wedge C^i_1 \rightarrow E^{i+1}_{q_1} \end{cases} \tag{2.17}$$

Now we have a set of registers $\{R^k_i\}$ to which we have imposed the condition of encode valid headers movements and valid states transitions. We will know proceed to enforce the consistency of the tape contents. Once this is done it will follow easily that the resulting formula will be satisfiable, if and only if there exists an accepting computation path in $M(x)$. Now we introduce a "copy" of the registers, denoted R^*_i (we drop the k superscript. We could keep it but we want to keep somewhat light notation). This registers are a copy of the previous ones. So we introduce $I^{i^*}_j$, $S^{i^*}_\alpha$, $E^{i^*}_q$ and $C^{i^*}_j$ for $j \in \{0, 1\}$.

The meaning is the same, but we will use the formula defined in **??**, and this $*$ registers will be the output to the permutation network, and the non $*$ registers will be the input. Let $R = \{R_1, \ldots, R_t\}$ and $R^* = \{R^*_1, \ldots, R^*_t\}$. Thus, we will use $F_{\lceil \log t \rceil}(R, R^*)$. Note that in order to use **??** we need to pad the variables to be a power of 2, but we can make the machine loop as many steps as necessary on the accepting state. At this point we can assume that R^* is a permutation of R. Lets first assume more, and assume that R^* is not only a permutation of R but an ordering with respect to the cell address in each register. Moreover, suppose that within each subsequence sharing the same cell address, the registers R^{j*} are ordered by execution step. Then, we can impose the correctness of the tape contents. In order to do so first let $\Delta(R^{i^*}, R^{(i+1)^*})$ be defined as the conjunction of following formulas:

$$\forall i \, \forall (\alpha, q) \in \mathrm{dom}(\delta_0) \left\{ S^{i^*}_\alpha \wedge E^{i^*}_q \wedge C^{i^*}_0 \rightarrow S^{(i+1)^*}_{\beta_0} \right. \tag{2.18}$$

$$\forall i \, \forall (\alpha, q) \in \mathrm{dom}(\delta_1) \left\{ S^{i^*}_\alpha \wedge E^{i^*}_q \wedge C^{i^*}_1 \rightarrow S^{(i+1)^*}_{\beta_1} \right. \tag{2.19}$$

Intuitively, $\Delta(R^{i^*}, R^{(i+1)^*})$ is true if, and only if, the symbol true in $R^{(i+1)^*}$ follows from a valid transition of M, considering the variables true in R^{i^*}. With this predicate we can finally impose the consistency of the tape contents.

For all i define the following formula, whose conjunction for all i we call γ_x:

$$(\neg A_=(R^{i^*}, R^{(i+1)^*}) \vee \Delta(R^{i^*}, R^{(i+1)^*})) \wedge (A_=(R^{i^*}, R^{(i+1)^*})) \vee (S^{(i+1)^*}_\square \wedge E^{(i+1)^*}_{q_{\mathrm{start}}})) \tag{2.20}$$

The first disjunctant is clear, if they have the same address, then we need the predicate $\Delta(R^{i^*}, R^{(i+1)^*})$ to be true, which will ensure that the contents that were written at step i were consistent with the transitions of M. If they don't have the same address, since they are adjacent in the ordering, this means that here we change subsequence, and we start a new subsequence with another address. Therefore, the $i+1$ register must correspond to a one with a blank symbol (since it would be the first time we access it). Recall that each subsequence with the same address they are ordered by execution step. The correctness of this formulas follow easily, keepin in mind the idea of theorem 13. Finally, in order to enforce the ordering of the R^* we add the following formulas:

$$\begin{cases} A_=(R^{i^*}, R^{(i+1)^*}) \vee A_<(R^{i^*}, R^{(i+1)^*}) \\ \neg A_=(R^{i^*}, R^{(i+1)^*}) \vee T_<(R^{i^*}, R^{(i+1)^*}) \end{cases} \tag{2.21}$$

Which force the permutation to become an ordering. Call the conjunction for all i of 2.21 formulas ζ_x. Recall that the size of the formula in lemma 4 was $|F_m| = 2c(2m-1)\cdot 2^m = O(m2^m)$ where m was the logarithm of the input, thus $|F_{\log t}| = O(t \log t)$. Therefore, the final formula of the reduction is

$$\Phi_x = \alpha_x \wedge \beta_x \wedge \gamma_x \wedge \zeta_x \wedge F_{\log t}(R_1 \ \ldots \ R_{2^{\lceil \log t \rceil}})$$

and $|\Phi_x| = O(t) + O(t \log^{O(1)} t) + O(t) + O(t \log^{O(1)} t) + O(t \log t) = O(t \log^{O(1)} t)$. □

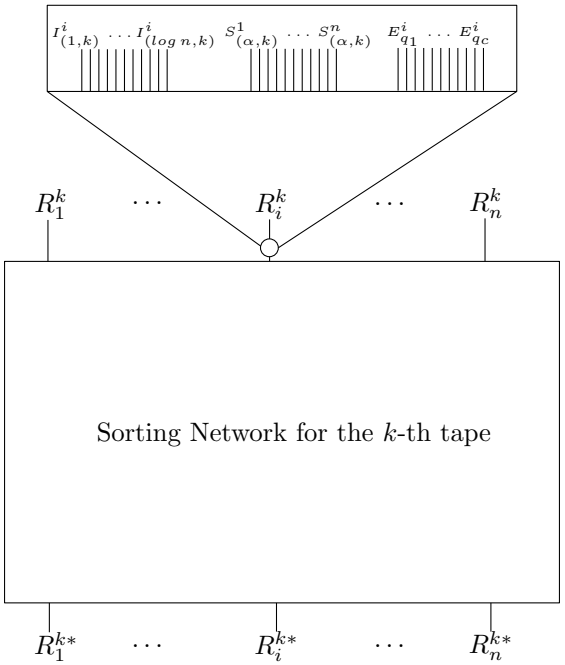

Figure 2.2: Depiction of a permutation network for the k-th tape with $c = |Q|$

It is worth pointing out that we could have used a sorting network instead of a permutation network. The proof is roughly the same, the sorting network guarantees us the ordering, so we can get rid of ζ_x and $F_{\log t}$, but to get efficient sorting networks one must resort to the more involved proof of [AKS83] to get an $O(n \log n)$ sorting network. A proof using "easy" sorting networks can be found in [vM07]. It would be worth to check in detail both methods and assess, in detail, which are the benefits of one and the other, since there seems to be no detailed proof of either method in the literature. Unfortunately, this is something we had to skip during this work but it is an interesting question.

2.6 Time-hierarchy for non-deterministic computations

An important thing to discuss is that of time hierachies. In this subsection we will first see the deterministic time hierarchy and afterwards the nondeterministic time hierarchy. The former will not be specially relevant for this work, but it is easily proven and the main idea used to proof it is then tweaked to get the nondeterministic one.

Theorem 18. *Let $f(n), g(n)$ be time-constructible functions and $f(n) \log f(n) = o(g(n))$. Then*

$$\mathbf{DTIME}(f(n)) \subset \mathbf{DTIME}(g(n))$$

Proof. Let $\{M_j \mid j \in \mathbb{N}\}$ be an enumeration of the deterministic Turing machines such that $L(M_j) \in \mathbf{NTIME}(f(n))$ for any j. We will build a diagonalizing machine D. We will make use of the efficient universal Turing machine U built in theorem 2.

$D(x)$:

1. run $U(M_x, x)$ for $g(|x|)$ steps

2. return 1-$U(M_x, x)$

It is clear that $L(D) \in \mathbf{DTIME}(g(n))$. Suppose for a contradiction that there exists D' such that $L(D') = L(D)$ and $L(D') \in \mathbf{DTIME}(f(n))$. In order to do the opposite of machine M_x we need that M_x have enough time to compute, which means that the following inequality must hold

$$g(|x|) > c_x \cdot f(|x|) \log f(|x|)$$

where c_x is the constant dependant on M_x as seen in theorem 2. By assumption, for fixed c_x there exists n_x such that the previous inequality holds for $n \geq n_x$. Since each machine is represented by infinitely many strings, pick a representation $d = \lfloor D' \rfloor$ with $d \geq n_d$. Therefore, we have that $D(d) = D((D', d)) = 1 - D'(d)$ by construction, and by assumption $D(d) = D'(d)$ a contradiction □

The analogous theorem for the non-deterministic setting is a bit different. The first idea one could have is just to replace the **DTIME** in theorem 18 with **NTIME**. But this runs in the problem of doing the opposite of a non-deterministic machine. To do that, we need to see what is the behaviour in all possible execution paths, which would take time $O(2^{f(n)})$, assuming the machine runs in time $O(f(n))$. Even with this exponential, we could get a hierarchy, but it would be a very coarse separation (exponential). In order to get an interesting result we will apply what is called "lazy diagonalization" or "delayed diagonalization", and it will consist in instead of diagonalizing for each input, we will just restrict ourselves to a smaller subset, which will give us enough time to explore all paths. This "delayed diagonalization" idea origins from [Coo72].

Theorem 19. ***nondeterministic time hierarchy*** *[Zak83] Let f, g be time constructible functions satisfying $f(n+1) = o(g(n))$ then*

$$\mathbf{NTIME}(f(n)) \subset \mathbf{NTIME}(g(n))$$

Proof. [AB09] The general proof is a bit involved, so we will showcase the idea as in [AB09] and show the case $f(n) = n^{1.1}, g(n) = n^{1.5}$. As stated, for this proof we will use "delayed diagonalization". Let $\{M_j \mid j \in \mathbb{N}\}$ be an enumeration of the nondeterministic Turing machines such that $L(M_j) \in \mathbf{NTIME}(g(n))$ for any j. Define $h(n)$ to be:

$$h(n) = \begin{cases} 1 & \text{if n = 0} \\ 2^{h(n-1)^{1.1}} & \text{otherwise} \end{cases}$$

$D(z)$:

1. check that $z = 1^{|z|}$, let $|z| = n$; reject otherwise.

2. find i such that $h(i) < n \leq h(i+1)$.

3. if $n < h(i+1)$ return $U_N(M_i, 1^{n+1})$ (if it has not halted in $n^{1.1}$ steps halt and accept).

4. else if $n = h(i+1)$ return $1 - M_i(1^{h(i)+1})$ by simulating M_i on all computation paths.

The machine D runs in time $O(n^{1.5})$. First, notice that we can find i in time $O(n^{1.5})$. If we run the non-deterministic universal Turing machine (step 3) then by theorem 3 it runs in time $O(n^{1.1}) \leq O(n^{1.5})$. For step 4, to return the contrary of M_i we need to explore all paths to see if it accepts or rejects. Thus, we need time $O(2^{(h(i)+1)^{1.1}}) = O(h(i+1)) = O(n)$ thus $D(z)$ runs in time $O(n^{1.5})$. Now suppose for a contradiction that $L(D) \in \mathbf{NTIME}(n^{1.1})$. Then it follows that for some i_0, $L(M_{i_0}) = L(D)$. Since each machine is represented by infinitely many indices, pick i_0 such that $U_N(M_i, 1^{n+1})$ takes less than $n^{1.1}$ steps. By assumption, we have that $M_{i_0}(1^a) = D(1^a)$ for $h(i_0) < a \leq h(i_0 + 1)$ and by construction we have that $M_{i_0}(1^{h(i_0)+1}) \neq D(1^{h(i_0+1)})$ which contradicts the previous statement. □

2.7 Existance of small witness circuits

Recall that at the start of this chapter, Williams' method was split into two:

(a) If (A) holds, then "small" witness circuits exist

(b) If (B) holds, then "small" witness circuits do not exist

We are now ready to prove both implications. We will first prove (a), namely that if $\mathbf{NEXP} \subseteq \mathbf{P}/poly$ then there exist "small" witness circuits and after we will prove (b), which said that if `CIRCUIT SAT` has better-than-trivial algorithms then "small" witness circuits do not exist.

2.7.1 Small witness circuits exist

Lemma 6. *If* $\mathbf{PSPACE} \subseteq \mathbf{P}/poly$ *then* $\mathbf{PSPACE} \subseteq \mathbf{MA}$

Proof. To show that **PSPACE** is in **MA**, it suffices to show that the *True Quantified Boolean Formulas(TQBF)* problem is in **MA**, since TQBF is **PSPACE**-complete. Hence, we must show that given a true quantified boolean formula there exists an **MA** protocol which convinces Arthur of the truth of the formula. . The protocol goes as follows:

- Merlin guesses the polynomial representation of the polynomial circuit of the optimal prover for TQBF and sends it to Arthur
- Arthur probabilistically verifies the validity of the boolean formula using the circuit sent by Merlin, without any more interaction.

Clearly, this is a **MA** protocol. Since $\mathbf{PSPACE} \subseteq \mathbf{IP}$ by the construction in section 2.2.1, there exists an **IP** protocol for TQBF. By hypothesis $\mathbf{PSPACE} \subseteq \mathbf{P}/poly$, therefore there must be a family of polynomial circuits which computes the optimal prover for the TQBF **IP** protocol. Hence, Merlin can send this circuit to Arthur, and Arthur can probabilistically verify the truth of the QBF using the prover sent by Merlin. □

Lemma 7. *If* $\mathbf{NEXP} \subset \mathbf{P}/poly$*, then there is* $q \in \mathbb{N}$*, such that* $\mathbf{NTIME}(2^n)/n \subset \mathbf{SIZE}(n^q)$

Proof. Fix some enumeration of the non-deterministic Turing machines $\{M_j | j \in \mathbb{N}\}$ and consider the following non-deterministic Turing machine:

$U(z)$:
1. check that z is of the form (i, x) where i codes a natural number. Reject otherwise
2. run $M_i(x)$ for $2^{2|x|}$ steps
3. accept iff $M_i(x)$ halts and accepts in the time given

By theorem 2, if M_i on input x runs in time $t(|x|)$, then U simulates M_i in time $c_i \cdot t(|x|)^2$, where c_i is a constant depending on M_i. Hence, U runs in time at most $O((2^{2|z|})^2) = O(2^{4|z|})$, which makes the language decided by U be in **NEXP**

By hypothesis, since $L(U)$ is in **NEXP** there is a family of circuits $\{C_n | \, n \in \mathbb{N}\}$ of size $O(n^q)$, for some constant q, that decides $L(U)$. We show that every $L \in \mathbf{NTIME}(2^n)/n$ is also decided by a family of circuits $\{C'_n | \, n \in \mathbb{N}\}$ of size $O(n^q)$.
Let M and $\{y_n | n \in \mathbb{N}\}$ be the non-deterministic Turing machine and the sequence of advice strings that witness that L is in $\mathbf{NTIME}(2^n)$/n. That is, M runs in time $O(2^n)$, and for every $n \in \mathbb{N}$ and every $x \in \{0,1\}^n$ it holds that $x \in L$ if and only if M accepts (x, y_n), and y_n is a binary string of length n. Let i be such that $M = M_i$.

By definition the running time of M on input (x, y_n) is bounded by $a \cdot 2^{|(x,y_n)|}$ for some constant a as long as $|(x, y_n)| \geq n_0$ for some constant n_0. Now, for every n and x, with x

having length n, we have that x is in L if, and only if, M accepts on input (x, y_n). In turn, for sufficiently large n, this happens if, and only if, M accepts (x, y_n) in $a \cdot 2^{|(x,y_n)|}$, and hence in time at most $2^{2|(x,y_n)|}$. Therefore, x is in L if, and only if, U accepts $(i, (x, y_n))$.

Next, we show that L can be decided by a family of circuits $\{C'_n | \, n \in \mathbb{N}\}$ of size $O(n^q)$. For inputs x of length n, define $C'_n(x) = C_m((i, (x, y_n)))$ where m is the length of $(i, (x, y_n))$. We claim that the family of circuits $\{C'_n | n \in \mathbb{N}\}$ decide L and have size $O(n^d)$. By definition, C'_n decides L. Next we show that C'_n has size $O(n^q)$. Indeed, If $n \geq n_0$, by definition we have that $C_n(x) = C_m(i, (x, y_n))$ with $m = (i, (x, y_n))$. Now, as we have previously stated, $\{C_m | m \in \mathbb{N}\}$ has circuits of size $O(n^q)$, therefore C'_n has size $O((|(i, (x, y_n))|)^q) = O((2n)^q) = O(n^q)$ since i is the constant size description of M_i and $|x| = |y_n| = n$.

Therefore $\{C'_n | \, n \in \mathbb{N}\}$ is a family of circuits that decide L and have size $O(n^q)$. □

Lemma 8. *If* **NEXP** $\subseteq$ **P**/*poly then for any* $L \in$ **NEXP**, L *has universal witness circuits of polynomial size*

Proof. We will assume that **NEXP** $\subseteq$ **P**/poly and that for some $L \in$ **NEXP**, L does not have universal witness circuits of polynomial size and derive a contradiction. Since **NEXP** $\subseteq$ **P**/*poly* we have *classPSPACE* $\subseteq$ **P**/*poly* which implies that **PSPACE** $\subseteq$ **MA** by lemma 6. We use this to reach a contradiction with lemma 3. From this contradiction will follow that L must have universal witness circuits of polynomial size.

Thus, let $L \in$ **NEXP** be such that it does not have universal witness circuits of polynomial size. Therefore, for some correct verifier V of L, there exists an infinite sequence $\{x_i | i \in \mathbb{N}\}$ such that for every x_i, and for every y of length $2^{|x_i|^c}$, $V(x_i, y) = 1$ implies that y_k (the k-th bit of y) cannot be computed by any circuit of size $O(n^d)$ for any constant d.

Let P be a **MA** protocol. Let Arthur's computation in P have circuit size n^a for some constant a. We can simulate P infinitely often as follows: For inputs z of length n, set as advice some x_m of length n. If there is no such x_m of the appropriate length, set as advice the string 0^n.

$M(z)$:

1. guess a witness w of length $2^{|x_m|^c}$, and check that $V(z, w) = 1$. Reject otherwise.
2. guess Merlin's polynomial string
3. Simulate arthur by evaluating G on all seeds of length $n^{\epsilon a}$ and take the majority vote.

Let $\epsilon = \dfrac{1}{a}$, then $n = n^{\epsilon a}$. Since $|w| = 2^{n^{\epsilon ac}}$, we can treat it as a boolean function on $n^{\epsilon ac}$ variables. By hypothesis, w has circuit complexity at least $n^{\epsilon ad}$ for any d. Therefore, w has circuit complexity at least $n^{\epsilon ad'} > n^{\delta e}$ for appropriate d'. By theorem 10, G evaluated on seeds of length $n^{\epsilon a}$ along with random access to w, fool circuits of size n^a, hence they fool Arthur. The simulation takes $O(2^{n^{\epsilon ac}})$ to guess the witness w and verify that $V(z, w) = 1$, polynomial time to guess Merlin's string, and $O(2^{n^{\epsilon a}})$ to evalute Arthur on all possible seeds of length $n^{\epsilon a}$, hence, the total runtime is $O(2^{n^{\epsilon ac}} + 2^{n^{\epsilon a}}) = O(2^{n^{\epsilon ac}})$ We have shown that we can simulate any **MA** protocol, for inifinitely many input lengths, in time $O(2^{n^\beta})$ with advice of length n^β where $\beta = \epsilon ac$. Hence, **MA** $\subseteq$ **io-NTIME**$(2^{n^\beta})/n^\beta$ for $\beta > 0$, since ϵ is not fixed. Now we can show the following classes inclusions for a fixed q

$$\textbf{PSPACE} \subseteq \textbf{MA} \subseteq \textbf{io-NTIME}(2^{n^\beta})/n^\beta \subseteq \textbf{io-SIZE}(n^q)$$

Which by lemma 3 is false, thus we reach a contradictions, meaning that indeed L does have polynomial universal witness circuits. The class inclusions are justified in the following way: The first inclusion is given by lemma 6 and the third inclusion is given by lemma 7. □

2.7.2 Small witness circuits do not exist

A key part in the lower bound method exposed [Wil10] is the fact that given a language $L \in \mathbf{NTIME}(2^n)$, on input x we can generate a propositional formula (call it Φ_x) such that deciding its satisfiability is equivalent to deciding the membership of x to L. The method proceeds by exhibiting an algorithm that assuming that "small" witness circuits exists, decides the satisfiability of Φ_x in time $o(2^n)$, thus reaching a contradiction with the non-deterministic time hierarchy. In order to get this algorithm to run in time $o(2^n)$ we need to exhibit an algorithm that decides`CIRCUIT SAT` algorithm in the same time (or less). Now we will argue why the quadratic size of the formula yielded by theorem 13 is not enough. The inputs of this circuit will be determined by the number of clauses that the formula has. Thus, if $|\Phi_x| = n^2$ we essentially need that the `CIRCUIT SAT` can be solved in time $o(2^{n/2})$ which is a very strong condition, and it is unlikely to exist such algorithms. After seeing the general construction, we will introduce the necessary tools to generate Φ_x with $|\Phi_x| = O(n \log^{O(1)(n)})$ which only forces the `CIRCUIT SAT` algorithm to run in time $o(2^n)$, a more feasible endeavour.

Theorem 20. *Let $c \geq 1$ and d be a constant that only depends on the computation model.If* `CIRCUIT SAT` *on circuits of n inputs and m gates can be solved in (co-non)deterministic time $O\Big(\frac{2^n \cdot m^c}{f(n)}\Big)$ with $f(n) = \Omega(n^d a(n) \cdot (S(n) + n^{2d})^c)$, and $a(n)$ strictly positive monotone and $n \leq S(n) \leq \frac{2^n}{n \cdot a(n)}$, then $\mathbf{NTIME}(2^n)$ does not have $S(n)$ universal witness circuits.*

Proof. The proof works by contradiction. We fix a language $L \in \mathbf{NTIME}(2^n)$ and consider a verifier $V(x, y)$ for L which on input (x, y) calculates Φ_x and checks if y encodes a satisfying assignment of Φ_x. Since we will assume that L has universal witness circuits of size $S(n)$, it will be the case that y can be encoded by circuits of this size. Finally, we will proof from this that we can decide L in time $o(2^n)$ contradicting the non-deterministic hierarchy theorem.

Let x be the input with $|x| = n$. By theorem 14, we have that Φ_x is satisfiable if, and only if, $x \in L$. Moreover, we have that the i-th clause of Φ_x can be computed in time $O(n^d)$, therefore there is a circuit that computes the i-th clause with size $O(n^{2d})$. In particular, given the index i of a clause, this circuit outputs the index of its three literals, and their sign (whether they are negated or not). Call this circuit C_x. Next, we can guess a circuit W of size $S(n)$ that witnesses the satisfiability of Φ_x. This circuit W exists by hypothesis. Now, with these two circuits we construct a third circuit, in such a way that it will be unsatisfiable if, and only if, $x \in L$. We call this circuit $D_{(x,W)}$. $D_{(x,W)}$ is built as follows: Its input will be the index of a clause of Φ_x. Note that Φ_x has at most $O(2^n \cdot n^d)$ clauses, therefore, $D_{(x,W)}$ has $log(c2^n \cdot n^d) = n + d \log n$ inputs.

We set up three copies of W and we feed to each copy one of the variables appearing in the i-th clause. Finally, we evaluate the circuit H with the inputs shown in the figure. H is 0 if, and only if, there is some literal satisfying the i-th clause. Therefore, $D_{(x,W)} = 0 \iff x \in L$. Denote by $A(x)$ the algorithm that solves `CIRCUIT-SAT` in $O(2^n \cdot m^c / f(n))$ co-nondeterministic time, which exists by hypothesis. The following machine decides L:

$M(x)$:
1. Guess W
2. Build $D_{(x,W)}$
3. Return $A(D_{(x,W)})$

Let $V(x, y)$ be a verifier for L that given x constructs Φ_x and checks if y encodes an assigment making Φ_x true. Also, denote by Φ_x^k the k-th clause of Φ_x and let $R(x, y)$ be a verifier for

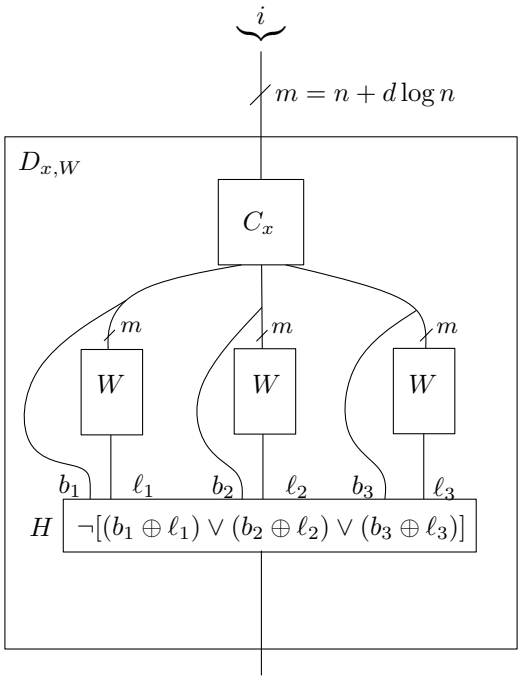

Figure 2.3: Depiction of circuit $D_{(x,W)}$

`CIRCUIT-SAT`. The correctness of M follows from the following chain of double implications:

$$x \in L \iff \exists y \text{ s.t } V(x,y) = 1 \iff \exists W\, |W| \leq S(|x|) \wedge y_i = W(i)$$
$$\iff \exists W \text{ s.t } \forall \gamma \quad \Phi_x^\gamma = z_k \vee \ldots \vee z_{k+l} \to \Phi_x^\gamma(W(z_k)\ldots W(z_{k+l})) = 1 \iff$$
$$\exists D_{x,W} \text{ s.t } \forall z\, D_{x,W}(z) = 0 \iff$$
$$\exists W \text{ s.t } D_{x,W} \text{ is } \mathbf{UNSAT} \iff \exists W \exists \omega \quad |\omega| = O\Big(\frac{2^n \cdot m^c}{f(n)}\Big) \wedge R(D_{(x,W)}, \omega) = 0$$

Let's analyze the running time of $M(x)$. To guess W we need to guess a representation of W. We can represent a circuit of size $S(n)$ with $O(S(n) \cdot log(S(n)))$ bits, hence, since $S(n) \leq \frac{2^n}{n \cdot a(n)}$ we have that $O(S(n) \cdot log(S(n))) \leq \frac{2^n}{n} \cdot (n - log\, n) = O(\frac{2^n}{a(n)})$ is an upper bound for the number of bits needed for a representation of W. For step 2, we have that $|D_{(x,W)}| = n^{2d} + 3S(n) + H$ which certainly is bounded by $O(\frac{2^n}{a(n)})$. Finally, for step 3, we have by hypothesis that $A(D_{(x,W)})$ takes time

$$O\Big(\frac{2^n \cdot (n^{2d} + 3s(n) + |H|)^c}{f(n)}\Big) = O\Big(\frac{2^n}{a(n)}\Big)$$

where this last equality follows by the hypothesis on $f(n)$. The running time of $M(x)$ in total is $O\Big(\frac{2^n}{a(n)} + \frac{2^n}{a(n)}\Big) = O\Big(\frac{2^n}{a(n)}\Big)$. Since $M(x)$ decides L and L was an arbitrary language in $\mathbf{NTIME}(2^n)$, we conclude that $\mathbf{NTIME}(2^n) \subseteq \mathbf{NTIME}(2^n/a(n))$, but since $\dfrac{2^{n+1}}{a(n+1)} = o(2^n)$ this inclusion contradicts the non-deterministic time hierarchy theorem. Thus, we conclude that $\mathbf{NTIME}(2^n)$ cannot have $S(n)$-size universal witness circuits. □

2.7.3 Extending the method

Here we give a way to generate witnesses if we want to prove circuit lower bounds for another class $\mathcal{C}$ of circuits which may be more restrictive than just $\mathbf{P}$/poly. This method requires stronger conditions, but it is interesting to briefly study it because it will make an appearanece in chapter 4 when we go over Williams' proof that $\mathbf{NEXP} \not\subseteq \mathbf{ACC^0}$.

We have given a method which would yield (unrestricted) polynomial circuit lower bounds for **NEXP**. The method can be extended to other (more restricted) classes of circuits, other than **P**/poly. Let $\mathcal{C}$ be a class of circuits such that it is closed under composition and it includes the class of circuits $\mathbf{AC^0}$. Then, [Wil10] gives a method to extend the previous method so that if there exists a slightly more efficient algorithm for $\mathcal{C}$-`CIRCUIT SAT` (i.e satisfiability of $\mathcal{C}$ circuits), then we can prove that **NEXP** does not have polynomial size $\mathcal{C}$ circuits. The argument makes use of the same ingredients as before, but with slight modifications. In particular, since we cannot guarantee that theorem 14 has **AC** circuits we have to use a weaker reduction, which leads to a less sharp result. In the next chapter when we go over the proof that $\mathbf{NEXP} \not\subset \mathbf{ACC}$ we will be able to circumvent this limiting factor by making use of nondeterminism. For the time being, let's see how to extend the method to other classes of circuits $\mathcal{C}$.

First, we need a method to produce $\mathcal{C}$ witness for **NEXP**.

Consider a language L in $\mathbf{NTIME}(2^n)$ and let $V_L(x, y)$ be its verifier.Consider the following $\mathbf{E^{NP}}$ machine

$M_L((x, i))$:

1. Let $w = (x, y)$ where y is a the binary string of all ones, with $|y| = d \cdot 2^{|x|}$ for some constant d
2. Query to the oracle if given w, there exists a binary string z with $z < y$ such that $V_L(x, z)$ accepts
3. If **YES** set $y = z$ and go to step 1, if **NO** output the i-th bit of y

It is straightforward to see that this machine computes the smallest binary string that can witness the membership of x to L. Note that each query can be performed in **NP**: given (x, y) and a candidate z we have time to compute $V(x, z)$ since $|y| = O(2^{|x|})$.

With this in mind we can introduce the following theorem

Theorem 21. *If* $\mathbf{E^{NP}}$ *has* $\mathcal{C}$ *circuits of size* $S(n)$ *then* $\mathbf{NTIME}(2^n)$ *has universal witness circuits from class* $\mathcal{C}$ *of size* $S(n)$

Proof. Fix $L \in \mathbf{NTIME}(2^n)$. Let M_L be the machine previously introduced. By hypothesis there is a family of (non-uniform) $\mathcal{C}$ circuits $\{C_n \mid n \in \mathbb{N}\}$ which computes the same function as M_L. It follows that there is a family of $\mathcal{C}$ circuits of size $O(S(n))$ that witnesses the membership to L □

Theorem 22. *If* $\mathcal{C}$-`CIRCUIT SAT` *can be solved in* $O(\frac{2^n n^c}{S(n)})$ *time, then* $\mathbf{E^{NP}}$ *does not have polynomial* $\mathcal{C}$ *circuits.*

Proof. As before, fix $L \in \mathbf{NTIME}(2^n)$. Given an input x, by **??** we know that there exists a propositoinal formula Φ_x such that Φ_x is satisfiable if, and only if, $x \in L$. We have that $|\Phi_x| = 2^{3n} \cdot n^2$. We need to use the weaker theorem 13 instead of theorem 14 because for the former we can compute the i-th clause with circuits in **AC**. Indeed, given (i, j) that index a cell of the computation tableau, we just need to add them as some offset to the variables. For the latter it is not clear that it is that easy. By theorem 21, if $\mathbf{E^{NP}}$ has polynomial $\mathcal{C}$ then it has polynomial universal witnesses. By the same construction as in theorem 20, we get that if $\mathcal{C}$-`CIRCUIT SAT` on n inputs and $poly(n)$ gates is solvable in $O(\frac{2^{n/3} poly(n)}{S(n)})$ time, then the machine carrying out this process as in theorem 20 runs in time $O(\frac{2^n}{S(n)})$ contradicting again the nondeterministic time hierarchy □

Note that we get 2^{3n} size in the formula which forces us to devise a $\mathcal{C}$-`CIRCUIT SAT` algorithms that kills that 3 factor, so something of the sort $O(2^{n/3})$ which is more difficult than just $o(2^n)$ as in the "general" method. If i-th clause of the formula yield by theorem 14 were to have **AC** circuits, then theorem 20 would apply, and we would avoid this problem.

Part II

Circuit Lower bounds & Complexity Barriers

Chapter 3

Circuit lower bounds

In this chapter we want to go over two techniques used to prove circuit lower bounds to have a feeling for the novelty of the method introduced by [Wil10]. The first technique is what is known as the *switching lemma* by Håstad. This technique is used to give lower bounds on the size of a particular class of circuits that compute the `PARITY` problem. The other technique, due to Razborov and Smolensky, is based on working with polynomials that approximate functions computed by circuits. Again, with this technique we prove lower bounds for `PARITY`. The main complexity classes important for the discussion of this techniques are:

- $\mathbf{AC^0}$ is the class of languages decidable by a family of circuits of polynomial size and constant depth with AND/OR gates of unbounded fan-in and NOT gates only at the first level.
- $\mathbf{ACC^0}$ is an extension of the previous class, where we add a gate that computes the *modulus* function for some integer q.

3.0.4 $\mathbf{AC^0}$ lower bounds

We start by surveying the main ideas of [Has86, FSS84] that show lower bounds against $\mathbf{AC^0}$. We start by introducing the necessary notions regarding Boolean formulas and restrictions over them.

Given a formula f with n variables we call a partial assignment ρ on $n-m$ variables an m restriction on f and we denote by $f|_\rho$ the result of applying ρ to f. Let f be a `k-DNF` formula and α some restriction. Assume that we have an ordering $\{T_i \mid i \in \{1 \dots |f|\}\}$ of the clauses, and within each clause we have some order for the literals. We denote by $C(f|_\alpha)$ the *canonical* decision tree of $f|_\alpha$. This tree is built as follows: Take the first T_{j_1} in the ordering that is not killed by α. Denote by U_1 the clause resulting from taking out the restricted literals of T_{j_1}. Suppose that α only leaves d_1 literals in U_1. Now build the complete d_1-depth decision tree over the variables of U_1. Denote by γ_1 the unique path that leads to U_1 being satisfied and thus makes $f|_{\alpha\circ\gamma_1} = 1$. For all other paths p, repeat the process with $f|_{\alpha\circ p}$ where $\alpha \circ p$ denotes the concatenation of the original restriction plus the one induced by the path p. The resulting tree is the *canonical* decision tree of $f|_\alpha$.

Lemma 9. ***Håstad's switching lemma*** *[Has86] Let f be a `k-DNF` formula over n variables. Let α be a random restriction of $s = \sigma \cdot n$ variables. Then for any $0 \leq d \leq s$ and $\sigma \leq \frac{1}{5}$*

$$\Pr_\alpha[DT_{depth}(f|_\alpha) > d] \leq (10\sigma k)^d.$$

Proof. Let $\mathcal{B}$ be the set of bad restrictions, that is those such that if $\beta \in \mathcal{B}$ then $DT_{\text{depth}}(f|_\beta) > d$. Given $\beta \in \mathcal{B}$ we will generate two things:

- an $s-d$ restriction.
- a binary string A encoding auxiliary information.

The proof goes as follows: Given $\beta \in \mathcal{B}$ we will generate a pair $(r, A) \in \mathcal{R}_{s-d} \times \{0,1\}^c$ where c is a constant which will be made explicit during the proof. Next we will show that from (r, A) we can recover β, thus finding an injection. This will allow us to bound the cardinality of $\mathcal{B}$ via counting that of $\mathcal{R}_{s-d} \times \{0,1\}^c$ which is

$$\binom{n}{s-d} 2^{n-(s-d)} \cdot 2^c$$

This proof of the switching lemma is due to Razborov. In particular this proof is an adaptation of that of [O'D].

As stated, take $\beta \in \mathcal{B}$ and consider $C(f|_\beta)$. Let π^* be the leftmost path with length $> d$, and let π be its prefix of length d. Thus, π is a path in $C(f|_\beta)$ of length exactly d. We let γ_1 be the restriction that makes $U_1 = 1$. By using $d_1 \log k + d_1$ bits we can encode which are the literals used in γ_1 and whether they are set to true or false. Let π_1 be the assignation made by π to the variables appearing in γ_1. We add d_1 additional bits to specify how π_1 sets the variables in γ_1, $d_1 \log k + 2d_1$ in total. Now consider the restriction $\beta \circ \pi_1$, which we know kills T_{j_1}. Now, consider the restriction $\beta \circ \pi_1$. We can again consider the first term T_{j_2} not killed by this restriction and generate γ_2 and π_2. At step i we are considering $f|_{\beta \circ \pi_1 \cdots \circ \pi_{i-1}}$ and we have built a restriction $\beta \circ \gamma_1 \circ \cdots \gamma_{i-1}$. We iterate this procedure until we exhaust π (say it takes ℓ steps). During this process we have generated the pair (r, A) where $r = \beta \circ \pi_1 \circ \pi_2 \circ \ldots \circ \pi_\ell \in \mathcal{R}_{s-d}$ is the $s-d$ restriction and A is the binary string encoding the auxiliary info $\gamma_1 \ldots \gamma_\ell$ and $\pi_1 \ldots \pi_\ell$. Note that in total we needed $d_1 \log k + 2d_1 + d_2 \log k + 2d_1 + \ldots + d_\ell \log k + 2d_\ell$ bits, thus in total we need $d \log k + 2d$ bits. Thus, we have found the unknown constant c that was to be defined. We have $c = d \log k + 2d$. Therefore, we now know the cardinal of the image of the injection we want to construct, it is

$$\binom{n}{s-d} 2^{n-(s-d)} \cdot 2^{2 \log k + 2d}$$

Now we want to see that given (r, A), we can invert the process uniquely to get β proving that this mapping is indeed an injection. So suppose we have f and (r, A). The only way to start is to consider $f|_r$. By construction (while generating the successive π_i we keep the invariant that all the previous clauses in the ordering are killed), we know that the first term killed under this restriction is T_{j_1}. We can lookup γ_1 in A so that we know which variables to unset. Then, by looking up π_1 we can consider $\beta \circ \pi_1 \circ \gamma_2 \circ \gamma_\ell$ and again consider the first clause fixed, which by construction will be T_{j_2}. In this way we keep undoing the restriction r until we recover β. □

Lemma 10. *Given a circuit C of size s and depth k, there exists an equivalent circuit C^* of depth k and size at most $(2s)^k$ such that:*

1. *C^* has all the negation gates at the input level.*
2. *C^* has its gates "layered", meaning that at any level there are only AND or OR gates.*
3. *All gates in C^* have fanout 1 (except the inputs) thus forming a tree.*

Proof. Let $g_i = (a_1 \wedge \ldots \wedge a_k)$ be an AND gate of the circuit (the OR case is just the same). Suppose that its output is negated, thus we have $\neg g_i = \neg(a_1 \wedge \ldots \wedge a_k) = (\neg a_1 \vee \ldots \vee \neg a_k)$ by De Morgan's. What we have done is just push the NOT gate down one level by pushing it to the inputs of the gate. It is clear that we could proceed from the top of the circuit down to the

bottom but since we have unbounded fanout, it could be the case that on of the gates that use g_i as input pushes a negation and the other not. Therefore, in order to solve this "clashing" we just add a negated copy of g_i, thus we have a gate computing $\neg g_i$ and the original gate g_i. This gives us a C' circuit with at most double the number of gates, equivalent to C. Now we want to "layer" C'. Assume that the $i+1$ and above levels are correctly layered, and assume without loss of generality, that the i level is supposed to be an AND level. For any gate in the i-th level, if its an AND gate we do not need to do anything. Therefore, suppose g is an OR gate at the i-th level. We would like to just reassign g's level to $i+1$ and be done. Clearly, g's output can only go to some gate at the level $i+1$ or above. If it goes to some gate above level $i+1$ we can safely reassign g. If g's output goes to some gate g' at the $i+1$ level, since the circuit is in $\mathbf{AC^0}$, it has unbounded fan in, so we can connect directly the inputs of g to g' which is also an OR gate (by virtue of being at the $i+1$ level) and then we can safely reassign g. This modification produces a circuit C'' . Note that this last modification doesn't modify the size or depth of C'. Finally we want to make all non-input gates of C'' have fanout 1. Consider the following process: Start from the $k-1$ level of the circuit. We know that the output gate(s) have fanout 1, thus the k-th level can be left alone. Now, for each gate at the $k-i$-th level take the worst case (very rough bound) and suppose they each have fanout s^i. If we add s copies of each gate, by properly increasing (if needed) the fanout at the $k-(i+1)$ level, we can make each gate at the $k-i$ level have fanout 1. We can repeat the process, and then at level j we may get as many as s^j gates. Thus, when we are done we have at most s^k gates. Since $\mathbf{AC^0}$ has constant depth, this k is a constant, thus applying this transformation yields C^*. Note that the previous properties are preserved by this operation. □

Before we use lemma 9 to prove a lower bound for `PARITY` we will discuss briefly the necessary width of a DNF or CNF that computes the `PARITY` function. We argue that `PARITY` on n variables requires a n-DNF or n-CNF. We start by noting that if the width of a DNF is k, then under a suitable restriction of k variables, the DNF becomes a constant 1. Likewise, if the width of a CNF is k, then under a suitable restriction of k variables, the CNF becomes the constant 0. Suppose we have n variables and we restrict $n-1$ of them. By flipping the last variable we flip the value of `PARITY`, thus `PARITY` requires width n CNF or DNF.

Definiton 9. *The* `PARITY` *problem can be formalized as the problem of given n boolean inputs $x_1, \ldots, x_n$, compute the following function*

$$\texttt{PARITY}(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i \mod 2.$$

Theorem 23. *If C is a circuit of depth k in canonical form with the gates at the second level having at most $20 \log s$ inputs that computes* `PARITY` *then $\log |C| \geq \Omega(n^{\frac{1}{k-1}})$.*

Proof. Suppose that C_n is a family of circuits in canonical form that compute the `PARITY` function. We overload the notation and denote by C_n the circuit of the family that computes `PARITY` for inputs of length n. Suppose that the bottom layer (the one following the inputs) is an AND layer. Suppose further that each gate at layer 2 (that is, each OR gate at level 2) represent a DNF of width $w = 20 \log s$ (for the other case we would just get a CNF of the same width). Apply a random restriction α_1 with $|\alpha_1| = \frac{n}{20w}$. By lemma 9, if we let $d = 20 \log s$ the probability that any of these DNFs is not expressable as a w-CNF is bounded by $(\frac{1}{2})^{20 \log s}$ hence, by the union bound, the probability that some OR gate at the second level is not representable by a w-CNF is at most $\frac{s}{s^{20}} = \frac{1}{s^{19}}$. Therefore we conclude that there exists a nonzero probability (in fact, quite high) that all OR gates at the level 2 are representable by a w-CNF. If we plug this CNFs, note that the top gate of a CNF is an AND, which will

be connected to another AND at the level 3 (where the OR gate was previously connected). Therefore, since the fanin is unbounded, we can merge level 2 and 3, thus reducing the depth of C_n by 1. Now, repeat this process $k-2$ times, until we have a circuit of depth 2. Since at each step the circuit preserves the property of computing PARITY, we end up with a circuit of depth 2, which can be represented as a CNF or DNF that computes PARITY of the inputs (keep in mind that some of this inputs have been restricted along the process).

Now, we want to compute how many inputs are still unset. At each step we restrict a $\frac{1}{20\log s}$ part of the inputs, thus after $k-2$ steps we have $m_{k-2} = \frac{n}{(20\log s)^{k-2}}$ unrestricted inputs. It is easy to see that if a function computes PARITY, then under any restriction of its inputs, it still computes PARITY or it computes its negation. We don't really care since both require m-DNF (m-CNF) on m inputs. Since for m inputs we have that PARITY requires a 2^m size DNF (or CNF). Since after $k-2$ iterations we have that C_n only has m_{k-2} inputs it follows that

$$s \geq 2^{m_{k-2}} \implies \log s \geq \frac{n}{(20\log s)^{k-2}} \implies c\log s^{k-1} \geq n \implies O(\log s) \geq n^{\frac{1}{k-1}}.$$

□

The previous proof showcases the idea behind the proof that PARITY $\notin \mathbf{AC^0}$. Bear in mind that the main goal was to showcase the idea and get the flavour of one technique used in proving circuit lower bounds.

3.0.5 $\mathbf{ACC^0}$ lower bounds

Next, keeping within the theme, we will showcase the idea behind the theorem by Razborov and Smolensky ([Raz87, Smo87]), as in [AB09], that the function $\mathtt{MOD}_\mathtt{p}$ cannot be computed by $\mathbf{ACC^0}$[q] circuits, where p, q are primes. We will look at the special case with $p = 2, q = 3$. Recall that $\mathbf{ACC^0}$[q] denotes the $\mathbf{ACC^0}$ circuits that have the mod q gate.

Theorem 24. *[AB09]* PARITY $\notin \mathbf{ACC^0}[3]$

The idea will be the following: Given a $\mathbf{ACC^0}[3]$ circuit C on n inputs, we will show that by using polynomials of degree $(2\ell)^h$ where h is the depth of C, we can approximate it up to a $1 - \frac{|C|}{2^\ell}$ factor. In particular we will pick ℓ so that $(2\ell)^h = n^{\frac{1}{2h}}$ so that we get a $\sqrt{n}$ degree polynomial. Then we show that no $\sqrt{n}$ polynomial agrees with the PARITY function on more than a 49/50 fraction of the inputs.

Proof. We use induction on h the depth of C. For the base case (inputs) it is clear that the polynomial x_i trivially computes it without error. Suppose that we have built a polynomial that approximates all gates of heigth $h-1$ and let $g(x_1 \dots x_n)$ be the function computed by the gate g at level h. We let $GF(3) = \{0, 1, -1\}$. Thus, to make sure that we have boolean output we just need to square whatever we want to be boolean since $0^2 = 0, 1^2 = 1$ and $(-1)^2 = 1$.

- Suppose $g(g_j)$ is a NOT gate. By inductive hypothesis there exists a polynomial $\widetilde{f_j}$ that approximates g_j. Thus, a polynomial approximating g can be defined as $\widetilde{f} = 1 - \widetilde{f_j}$. The degree of $\widetilde{f}$ is the same as $\widetilde{f_j}$ and we do not introduce any error.

- Suppose $g(g_{j_1}, \dots, g_{j_k})$ is a mod 3 gate. We define $\widetilde{f} = (\sum_{i=0}^{k} \widetilde{f_{j_i}})^2$. By inductive hypothesis the $\widetilde{f_{j_i}}$ have degree $(2\ell)^{h-1}$. Thus, $\widetilde{f}$ has degree $2 \cdot (2\ell)^{h-1} < (2\ell)^h$. We do not introduce any error since $0^2 = 0$ and $(-1)^2 = 1$.

- Suppose $g(g_{j_1}, \ldots, g_{j_k})$ is an OR (if it were an AND we just apply De Morgan's). By inductive hypothesis we have a set of polynomials $\{\widetilde{f_1}, \ldots, \widetilde{f_k}\}$ of degree $(2\ell)^{h-1}$ each, that approximate the functions computed by the gates $\{g_{j_1}, \ldots, g_{j_k}\}$ that are the input to g_i. Consider the following polynomial $p(\widetilde{f_1} \ldots \widetilde{f_k}) = 1 - \prod_{i=1}^{k} 1 - \widetilde{f_i}$. Clearly $p(\widetilde{f_1} \ldots \widetilde{f_k})$ computes the OR function of $\{\widetilde{f_1} \ldots \widetilde{f_k}\}$ but the degree of $p(\widetilde{f_1} \ldots \widetilde{f_k})$ would exceed $(2\ell)^h$. The strategy will be to generate a polynomial $\widetilde{g_i}$ of the required degree, but at a cost of introducing some error. Thus, $\widetilde{g_i}$ will approximate g_i but will miscalculate for a fraction of the inputs. We have that $g_i = f_1 \vee \ldots \vee f_k = 1 \iff \exists i \ f_i = 1$. By the random subsum principle if one element of the set $\{f_1, \ldots, f_k\}$ is 1, then the sum of any random subset is non-zero MOD_2 with probability $\frac{1}{2}$. Since here we sum MOD_3 we have that the sum of a random subset is non-zero with probability *at least* $\frac{1}{2}$ (since it could be either 1 or -1). With this in mind, let $T = (T_1, \ldots, T_\ell)$ be a collection of sets where we have for $1 \leq j \leq \ell$ $T_j \subseteq \{1, \ldots, k\}$ and define ℓ polynomials $(\sum_{i \in T_1} \widetilde{f_i})^2, \ldots, (\sum_{i \in T_\ell} \widetilde{f_i})^2$. Now, we compute the OR in the "naive" way of this ℓ polynomials $\widetilde{g_i}(\widetilde{f_1}, \ldots, \widetilde{f_k}) = 1 - \prod_{T' \in T} (\sum_{i \in T'} \widetilde{f_i})^2$. Note that the degree will be $2\ell \cdot (2\ell)^{h-1} = (2\ell)^h$. Fix $x \in \{0,1\}^n$. When computing $C(x)$ we can find two cases. If $g_i = 0$ then $\widetilde{g_i} = 0$, if $g_i = 1$ then for some $j \in \{1, \ldots, k\}$ $\widetilde{f_j} = 1$. Now if $\forall i \in \{1, \ldots, k\}$ $j \notin T_i$ then it could be that $g_i \neq \widetilde{g_i}$. Denote by $\widetilde{g_i}^s$ the polynomial induced by s where $s \in \{0,1\}^k$ denotes a subset of $\{1, \ldots, k\}$. We have that $\Pr_{s \in \{0,1\}^k}[\widetilde{g_i}^s \neq g_i] \leq \frac{1}{2^\ell}$ by the random subsum principle. Thus, for a given x, there are $2^{k-\ell}$ "bad" subsets. Suppose that "bad" subsets are distributed uniformly on $\{0,1\}^n$. Then, each subset is expected to be "bad" for $2^{k-\ell}$ inputs x, which means that there is some choice of subsets that is "bad" for at most $\frac{1}{2^\ell}$ fraction of the input. We pick this set of subsets as our choice to build $\widetilde{g_i}$.

By the above procedure, we end up generating a polynomial $\widetilde{g_s}$ which approximates the output gate of the circuit. By construction, the degree of $\widetilde{g_s}$ is $(2\ell)^h$ where h is the depth of C. Since each gate introduces an error for at most a $\frac{1}{2^\ell}$ fraction of the input, we have that in total $\widetilde{g_s}$ is wrong in at most $\frac{|C|}{2^\ell}$ inputs.

Let f be a polynomial agreeing with $MOD_2 = \texttt{PARITY}(x_1, \ldots, x_k)$ for all inputs in $G \subseteq \{0,1\}^n$. Suppose that the degree of f is bounded by $\sqrt{n}$. Then, $|G| \leq \frac{49}{50} 2^n$. Let $f(x_1, \ldots, x_k)$ be such polynomial. Consider the change of variable $y_i = 1 + x_i \mod 3$ which takes values on $\{-1, 1\}$. Therefore, we have another polynomial $f'(y_1, \ldots, y_k)$ of degree still $\sqrt{n}$ with $f(x_1, \ldots, x_k) = 1 \iff f'(y_1, \ldots, y_k) = 1$ and $f(x_1 \ldots x_k) = 0 \iff f'(y_1, \ldots, y_k) = -1$. This transforms $G \subseteq \{0,1\}^n$ into $G' \subseteq \{-1,1\}^n$. The interesting thing about this transformation is that the following polynomial $p(y_1 \ldots y_k) = \prod_{i=1}^{k} y_i$ computes the function $\texttt{PARITY}(y_1, \ldots, y_k)$. Therefore for any $z \in G'$ we have that $p(z) = f'(z)$. The last step will be to show that this implies that G' must have small cardinality. Let $F_{G'}$ denote the set of functions $S : G' \mapsto GF(3)$, we will show that $|F_{G'}| \leq 3^{\frac{49}{50} \cdot 2^n}$.

Let $\widetilde{S} : GF(3)^n \mapsto GF(3)$ be any function that agrees with S on G'. Then $\widetilde{S}$ can be written as a polynomial in $y_1 \ldots y_n$. Since we are interested in the case $y_i \in \{-1, 1\}$ and $(y_i)^2 = 1$ we can assume, without loss of generality, that the for any monomial $\prod_{i \in I} (y_i)^{r_i}$ we have $r_i \leq 1$. Now consider any monomial with multidegree $|I| > \frac{n}{2}$. We can rewrite it as

$$\prod_{i \in I} y_i = \prod_{i=1}^{n} y_i \prod_{i \in I^\dagger} y_i = f'(y_1, \ldots, y_k) \prod_{i \in I^\dagger} y_i$$

where $I^\dagger = \{1, \ldots, n\} \setminus I$ and the last equality follows because

$$f'(y_1, \ldots, y_k) = \prod_{i=1}^{n} y_i = \texttt{PARITY}(y_1, \ldots, y_k).$$

Note that the degree becomes $\frac{n}{2} + \sqrt{n}$. Therefore, we have that any function in $F_{G'}$ is equal to some polynomial of degree $\frac{n}{2} + \sqrt{n}$. But how many such polynomials are there? Well, each monomial has to be multiplied by some element of $GF(3)$, thus the number of possible polynomials is $3^{\#\text{monomials}}$. The number of monomials with the above characteristics responds to the following expression

$$|\{I \in \{1, \ldots, n\} \mid |I| \leq \frac{n}{2} + \sqrt{n}\}| = \sum_{i=0}^{\frac{n}{2}+\sqrt{n}} \binom{n}{i}$$

which can be shown to be bounded by $\frac{49}{50} 2^n$. □

Chapter 4

Showing ACC lower bounds for NEXP

We will briefly comment how the ideas of [Wil10] are applied in [Wil14] to get circuit lower bounds for **NEXP**. It is shown that $\mathbf{NEXP} \not\subseteq \mathbf{ACC^0}$.

4.0.6 Overview

The normal course of action would be to apply the general method described in chapter 2. The first problem that Williams' encounters is that the reduction in theorem 14 yields a formula whose i-th clause is computable in polynomial time, but here we only have $\mathbf{ACC^0}$. In chapter 2 when we showed the condition under which there exist witness circuits we saw that we can relax this condition and find a formula whose i-th clause is computable by $\mathbf{AC^0}$ circuits, but it yields a $O(2^{2n})$ formula which translates in that the circuit built in theorem 20 has $2n+O(\log n)$ inputs, thus instead of requiring a `CIRCUIT SAT` algorithm with $o(2^n)$ running time, we need $o(2^{n/2})$ which seems too hard of a condition. In order to solve this problem we will make use of the assumptions that are made to prove the lower bound. Since the proof works by contradiction, we can assume that $\mathbf{P} \subseteq \mathbf{ACC^0}$. This might seem to solve the problem, but recall the concept of a non-uniform circuit family introduced in the preliminaries which meant that although there is a family of circuits that compute a given function, this circuits may not be constructable using Turing machines (as an example we saw the HALT language which has a non-uniform family of circuits). Therefore, under the assumption that $\mathbf{P} \subseteq \mathbf{ACC^0}$ we know that the circuit C_x from theorem 20 is in $\mathbf{ACC^0}$, but we don't know how to construct it. But, we want to proof lower bounds for **NEXP**, which is a *non-deterministic* class. We can use the non-determinism to guess the equivalent circuit to C_x that is in $\mathbf{ACC^0}$. Now, once the circuit is guessed and we are sure that is equivalent to C_x we can just carry on with the construction in theorem 20 and apply the fast algorithm for `ACC-CIRCUIT SAT` to obtain the desired lower bounds.

In the rest of the chapter we will just look at the two parts needed to apply William's method without getting into too much detail: Finding witnesses and exhibiting a better-than-trivial algorithm for `CIRCUIT SAT` (in this case for $\texttt{ACC}^0$`-CIRCUIT SAT`).

4.0.7 Finding the witness

Lemma 11. *Suppose* $\mathbf{P}$ *has* $\mathbf{ACC^0}$ *circuits of depth* d' *and size at most* $S(n)$*. Assume that* `ACC-CIRCUIT SAT` *on circuits with* $cn + \log n$ *inputs, depth* $2d' + O(1)$ *and at most* $O(S(3n) + S(2n)n)$ *size can be solved in* $O(\frac{2^n}{n^c})$ *for sufficiently large* $c > 2d$*. Then it is the case that for every* $L \in \mathbf{NTIME}(2^n)$ *there is a nondeterministic algorithm* $\mathcal{A}$ *such that:*

- $\mathcal{A}$ *runs in* $O(\frac{2^n}{n^c} + S(3n) \cdot poly(n))$ *time.*

- *for every x of length n, $\mathcal{A}(x)$ has some computation path that outputs an* $\mathbf{ACC^0}$ *circuit C'_x with $n + d\log n$ inputs, depth d' and $S(c(n + \log n))$ size, such that $x \in L$ if, and only if, C'_x is the compression of a satisfiable 3-CNF formula of $2^n poly(n)$ size.*

Proof. Fix $L \in \mathbf{NTIME}(2^n)$. Let C_x be the circuit that witnesses $x \stackrel{?}{\in} L$ (by theorem 20). Recall that this circuit can be computed in polynomial time. Thus, the problem

Given x and j output the j-th gate of C_x. That is, output a triplet (g, j_1, j_2) where g is the gate type (OR, AND or NOT), and j_1, j_2 the gates whose output is the input for j (suppose that for the NOT gate, j_2 is set to some value which we agree denotes an "ignore" gate)

is also computable in polynomial time, and thus by assumption there is an $\mathbf{ACC^0}$ circuit $D_i(x, j)$ for $i \in \{1 \dots O(\log n))\}$ which outputs the i-th bit of the gate description with size . Let $D(x, j)$ be the circuit obtained by putting together all $D_i(x, j)$. Thus, $D(x, j)$ outputs the j-th clause of C_x. By construction $D(x, j)$ has size $S(n + d\log n) \cdot O(\log n)$. $\mathcal{A}$ checks that indeed $D(x, j)$ describes C_x, by checking for every possible input z, that $D(x, z)$ outputs a consistent gate description with C_x, otherwise rejects. To perform this checking $\mathcal{A}$ takes time $O(S(n+d\log n)\cdot\log n\cdot\log S(n + d\log n + \log\log n)$ to guess a representation for circuit $D(x, j)$. Then $\mathcal{A}$ takes time $O(n^d) \cdot S(n + d\log n)$ to evalute $D(x, z)$ for all $z \in \{1 \ \dots \ O(n^d)\}$ and check that the output gate is consistent with C_x. Therefore, for this part $\mathcal{A}$ takes time in total

$$O(S(n+c\log n)\cdot\log n\cdot\log S(n + c\log n + \log\log n))+O(n^d\cdot S(n+c\log n) = O(n^d\cdot S(n+c\log n)).$$

Now consider the following problem:

Given (x, i, j) output the value of the output of gate j on $C_x(i)$.

By assumption there is an $\mathbf{ACC^0}$ circuit for this problem also since again this problem is computable in polynomial time. Call this circuit $E(x, i, j)$. E has size $S(n + (n + d\log n) + d\log n) \leq S(3n)$ and depth d'. Now, $\mathcal{A}$ can construct a circuit $\texttt{VALUE}(i, j)$ which outputs the value of the j-th gate of $C_x(i)$. The circuit $\texttt{VALUE}(i, j)$ is made with D and E. On input (i, j), $\texttt{VALUE}(i, j)$ feeds j to $D(x, \cdot)$ which outputs j_1, j_2, g. Then, $\texttt{VALUE}(i, j)$ calculates $v_1 = E(x, i, j_1), v_2 = E(x, i, j_2)$ and $v = E(x, i, j)$. Finally the output value of $\texttt{VALUE}(i, j)$ is defined depending on g:

- If $g = \wedge$ then $\texttt{VALUE}(i, j) = 0 \iff v_1 \wedge v_2 = v$.
- If $g = \vee$ then $\texttt{VALUE}(i, j) = 0 \iff v_1 \vee v_2 = v$.
- If $g = \neg$ then $\texttt{VALUE}(i, j) = 0 \iff v_1 = \neg v$.

The three checks above can be implemented with a constant number of gates and D, E are $\mathbf{ACC^0}$ circuits, thus $\texttt{VALUE}(i, j) \in \mathbf{ACC^0}$. The circuit $\texttt{VALUE}(i, j)$ has $n + d\log n + d\log n + O(1) = n + 2d\log n + O(1)$ inputs, depth $2d' + O(1)$ and size $O(S(3n) + S(n + d\log n) \cdot O(\log n)) \leq O(S(3n) + S(2n)n)$. By assumption, we can run `ACC-CIRCUIT SAT` on $\texttt{VALUE}(i, j)$ in time $O(\frac{2^n}{n^c})$ for $c > 2d$. Hence, $\mathcal{A}$ checks with the efficient `ACC-CIRCUIT SAT` algorithm that $\forall i, j$ $\texttt{VALUE}(i, j) = 0$ which means that it is an unsatisfiable circuit. Finally, we can set up a circuit $\texttt{SAME}(z)$ which just outputs 0 if, and only if, $C_x(z) = E(x, z, \alpha)$ where α denotes the index of the output gate. Then, by the three conditions above, $\texttt{SAME}(z)$ is unsatisfiable if, and only if, $\forall z$ $C_x(z) = E(x, z, \alpha)$, thus $C'_x(z) = E(x, z, \alpha)$. Now, $\mathcal{A}$ runs the efficient circuit sat algorithm for $\mathbf{ACC^0}$ to determine the unsatisfiability of $\texttt{SAME}(z)$ and thus the correctness of $E(x, i, \alpha)$. Since $\texttt{SAME}(z)$ has $n + d\log n$ inputs, d' depth and size $O(S(n + d\log n))$, the satisfiability algorithm again runs in time $O(\frac{2^n}{n^c})$. Now $\mathcal{A}$ just rejects if $\texttt{SAME}(z)$ is satisfiable and otherwise prints the circuit C'_x. □

4.0.8 Fast algorithm for ACC-CIRCUIT SAT

Now we only need an efficient algorithm for ACC-CIRCUIT SAT. We will only give an idea of how it works. For a detailed proof the reader is directed to [Wil14]. The first part involves a reduction from ACC circuits to $\texttt{SYM}^+$ circuits.

Definiton 10. *A $\texttt{SYM}^+$ circuit, is a circuit of depth 2 which at the output level computes a symmetric function, and at the first layer just computes ANDs of the inputs.*

As explained in [Wil14], we can transform any $\texttt{ACC}^0$ circuit into a $\texttt{SYM}^+$. As it turns out, $\texttt{SYM}^+$ circuits allow for a faster method to evaluate them on all inputs, which leads to an efficient CIRCUIT SAT algorithm for this type of circuits. The transformation is rather technical and will be skipped. For a detailed exposition of the transformation the interested reader is directed to [Wil14] Appendix A.

The transformation from $\texttt{ACC}^0$ circuits to $\texttt{SYM}^+$ is formalized with the following theorem:

Theorem 25. *There is an algorithm and function $f : \mathbb{N} \mapsto \mathbb{N}$ with $f(c) \leq 2^{O(c)}$ such that given $\mathbf{ACC^0}$ circuit of depth d and size s, the algorithm outputs an equivalent $\texttt{SYM}^+$ circuit of $s^{O(\log^{f(d)} s)}$ size. The algorithm takes at most $s^{O(\log^{f(d)} s)}$ time. Furthermore, given the number of ANDs in the circuit that evalute to 1, the symmetric function itself can be evaluated in $s^{O(\log^{f(d)} s)}$ time*

Next, we present the method to compute $\texttt{SYM}^+$-CIRCUIT SAT fast.

Lemma 12. *Evaluation lemma ([Wil14]) There is an algorithm that, given a $\texttt{SYM}^+$ circuit of size $s \leq 2^{.1n)}$ and n inputs with a symmetric function that can be evaluated in poly(s) time, runs in $2^n \cdot poly(n)$ time and computes a 2^n bit vector V such that $V[i] = 1$ if, and only if, the $\texttt{SYM}^+$ circuit outputs 1 on the i-th assignment.*

There are two proofs for this lemma. One uses a fast matrix multiplication and the other uses dynammic programming.

Proof. Matrix Multiplication Let C be a $\texttt{SYM}^+$ circuit with $|C| \leq 2^{.1n}$. Partition the inputs of C into two sets A, B of size at most $m = \dfrac{n+1}{2}$ each one. Define two matrices M_A, M_B of dimensions $2^m \times |C|$ and $|C| \times 2^m$ respectively. M_A's rows represent all the possible assigments to the variables in A, and it's columns represent the AND gates of C. The meanings are transposed for M_B. Therefore, we define the contents of both matrices as follows:

$$M_A(i,j) = \begin{cases} 1 \text{ if the i-th assignment does not force the j-th AND to be 0} \\ 0 \text{ otherwise} \end{cases} \tag{4.1}$$

$$M_B(r,s) = \begin{cases} 1 \text{ if the s-th assignment does not force the r-th AND to be 0} \\ 0 \text{ otherwise} \end{cases} \tag{4.2}$$

The time required to produce one of this matrices is $O(2^m \cdot |C| \cdot poly(m)) = O(2^{m+.1n} \cdot poly(m))$. Therefore, in total to produce both matrices we need time $O(2^{n/2+.2n})$. Let $M = M_A \times M_B$. Note that, M is a $2^m \times 2^m$ matrix. Denote by $\alpha_{(i,j)}(k)$ the value of the k-th AND gate under the assignment encoded by the i-th row of M_A with the j-th column of M_B. From the previous definition follows that:

$$M(i,j) = \sum_{k=1}^{|C|} \underbrace{M_A(i,k) \cdot M_B(k,j)}_{=\alpha_{(i,j)}(k)}.$$

Thus, $M(i,j)$ counts for how many indices k we have $\alpha_{(i,j)}(k) = 1$, in other words, how many AND gates are true under the assigment given by i, j. Therefore, the question of the satisfiability of C is the same as asking whether there exists i, j that makes the symmetric function computed at the output of C true.

By [Cop82] we have that we can produce the matrix M in time $O(2^{2m} \cdot poly(m))$. Finally, we need a way to calculate if indeed some of the assignments make the symmetric function at the output equal to 1. In order to do that, initialize a bit vector T with $|T| = |C|$. Then, for each $i \in \{0, \ldots, |C|\}$ set $T[i] = 1$ if, and only if, the symmetric function at the output of C outputs 1 when given i 1's as input. Here we make use of the symmetry of the function, since we only evaluate it for inputs of the form $1^i 0^{|C|-i}$. Since the symmetric function requires time $poly(|C|)$ to be evaluated we need in total $poly(|C|)$ to generate the bit vector T. In total, to generate the matrices M_A and M_B, multiply them to get M and generate the bit vector T we need time

$$O(2^{n/2+.2n} + 2^n \cdot poly(n) + poly(|C|)) = O(2^n \cdot poly(n)).$$

□

Note that to determine the satisfiability of the circuit C, we just loop over all paires i, j with $i, j \in \{0, \ldots, 2^m\}$. If $T[M(i,j)] = 1$ then we can say that C is satisfiable. If we go over all pairs and we don't find the circuit to be satisfiable, then it is unsatisfiable. We need time $O(2^{2m}) = O(2^n)$ to do these procedure.

As noted in [Wil14], it might be the case that the hidden constant in Coppersmith's multiplication algorithm is astronomical. William's presents an alternative proof of the evaluation lemma via dynamic programming. In particular we will use Yate's fast algorithm for the zeta transform.

Lemma 13. ***Yate's fast zeta transform*** *Let $N = \{1, \ldots, n\}$ be a set and let $f : 2^N \mapsto R$ (where R is some ring) . The function $\widehat{f} : 2^N \mapsto R$ defined as*

$$\widehat{f}(A) = \sum_{X \subseteq A} f(X)$$

is called the zeta transform of f. We can compute the set $\{\widehat{f}(A) \mid A \in 2^N\}$ with $O(n2^n + t(n)2^n)$ time where $t(n)$ is the running time of $f(\{1, \ldots, n\})$.

Proof. Let $\rho^i(T)$ denote the collection of subsets that include the subset $\{i+1, \ldots, n\} \cap T$. For instance, if $T = \{1,2,3,4\}$ then $\rho^2(T) = \{\{2,3,4\}, \{1,3,4\}, \{1,2,3,4\}\}$. Then given T we have:

$$\sum_{Z \in \rho^i(T)} f(Z) = \begin{cases} \sum\limits_{X \in \rho^{i-1}(T)} f(X) + \sum\limits_{Y \in \rho^{i-1}(T \setminus \{i\})} f(Y) & \text{if } i \in T \\ \sum\limits_{X \in \rho^{i-1}(T)} f(X) & \text{otherwise} \end{cases}$$

This expression doesn't make much sense in this way, but if we consider the case $i = n$ then we get

$$\sum_{Z \in \rho^n(T)} f(Z) = \sum_{Z \subseteq T} f(Z)$$

because if $n \notin T$ then n is irrelevant and if $n \in T$ then the sum over the subsets of T can be divided by the subsets that contain n and those that do not. Also we have $\rho^0(T) = T$. The interesting thing is that for any i, the result only depends on $i - 1$. Therefore, we define the following function $g_0(T) = f(T)$ and for $i \in \{1 \ldots n\}$:

$$g_i(T) = \begin{cases} g_{i-1}(T) + g_{i-1}(T \setminus \{i\}) & \text{if } i \in T \\ g_{i-1}(T) & \text{otherwise} \end{cases}$$

We have that $g_0(T) = f(T) = \sum_{X \in \rho^0(T)} f(X)$. Suppose that $g_{i-1}(T) = \sum_{X \in \rho^{i-1}(T)} f(X)$, then

$$g_i(T) = \begin{cases} \sum_{X \in \rho^{i-1}(T)} f(X) + \sum_{Y \in \rho^{i-1}(T \setminus \{i\})} f(Y) & \text{if } i \in T \\ \sum_{X \in \rho^{i-1}(T)} f(X) & \text{otherwise} \end{cases} = \sum_{Z \in \rho^i(T)} f(Z).$$

Now consider the following machine:

$M(N = \{1, \ldots, n\})$:
1. Compute $g_0(T) = f(T)$ for $T \subseteq N$ // This takes $O(2^n)$ invocations of f
2. for $i \in \{1, \ldots, n\}$: For all $A \subseteq N$ compute $g_i(A)$ // this takes $O(n \cdot 2^n)$ time
3. output a table with $g_n(A)$ for every $A \subseteq N$

Thus, in total it takes $O(n \cdot 2^n + t(n) \cdot 2^n)$ time where $t(n)$ is the running time of f. □

Note that we assume unit time for the ring operations, but as we will see in our case, f will take polynomial time, thus will dominate the ring operations. We can now prove the evaluation lemma using lemma 13.

Proof. **Dynamic programming** As before we are given a $\texttt{SYM}^+$ circuit C of size s. Define a function $f : 2^n \mapsto \mathbb{N}$ $f(S) =$ "number of gates such that their inputs are exactly S" . Suppose that $T \subseteq \{1, \ldots, n\}$ denotes an assignment of the input variables, where $x_i = 1$ if $i \in T$ and $x_i = 0$ if $x_i \notin T$. Then it can be checked that the number of AND gates that are satissfied by the assignment T are exactly

$$\sum_{X \subseteq T} f(X).$$

We can compute f as a table with 2^n entries. We go over all AND gates (at most s) and update add 1 to each entry corresponding to an assignment that sets all input variables of the AND gate to 1. In total, this takes time $O(2^n \cdot s)$. Note, that the zeta transform of f gives us, for any possible assignment T, how many AND gates are satisfied. We have by lemma 13, that we can compute it in time $O(2^n poly(n))$ because f has been precomputed, thus the computational cost is just looking up the answer. Finally we generate the bit vector evaluating the symmetric function for any possible input of the circuit making use of the zeta transform computed. Since we can compute h in time $poly(s)$, the total time required is $O(2^n poly(n) + poly(s)) = O(2^n poly(n))$. □

Now we lay out the general idea of how the lower bounds are proved. We have not gone over all steps, but from the transformation to $\texttt{SYM}^+$ and lemma 12 we can exhibit a better-than-trivial $\texttt{ACC}^0$-$\texttt{CIRCUIT SAT}$ algorithm. We have the witnesses W so that we can build D_x, but recall that in order to exhibit witnesses of a more restricted class of circuits (other than $\mathbf{P}$/poly) we needed a stronger assumption (theorem 21). We needed the assumption that $\mathbf{E^{NP}}$ had $\mathbf{ACC^0}$, which would yield circuit lower bounds against $\mathbf{E^{NP}}$ and not $\mathbf{NEXP}$. In order to solve this we need the following lemma:

Lemma 14. *[Wil14] Let $\mathcal{C}$ be any circuit class. If $\mathbf{P}$ has non-uniform $\mathcal{C}$ circuits of polynomial size, then every circuit family of size $T(n)$ has an equivalent circuit family of size $poly(T(n))$.*

Proof. Consider the $\texttt{CIRCUIT EVAL}$ problem, which consists of given a description of a circuit C and an input x, return $C(x)$. Clearly, this problem is in $\mathbf{P}$. Thus, there exists a polynomial sized family of circuits $\{E_n \mid n \in \mathbb{N}\}$ that computes $\texttt{CIRCUIT EVAL}$. Let $\{A_n \mid n \in \mathbb{N}\}$ be a family of $\mathcal{C}$-circuits of size $T(n)$. It can be shown that the circuit $G_n(x) = E_{n_0}(A_{|x|}, x)$ is equivalent to A_n where $n_0 \geq n + O(T(n) \cdot \log T(n))$. □

Therefore, with "only" assuming $\mathbf{NEXP} \subseteq \mathbf{ACC^0}$ we also have $\mathbf{P} \subseteq \mathbf{ACC^0}$ and thus we can construct D_x. Finally we lay out how the method goes. We assume that $\mathbf{NEXP} \subseteq \mathbf{ACC^0}$ and we fix $L \in \mathbf{NTIME}(2^n) \setminus \mathbf{NTIME}(2^{n-\omega(\log n)})$ (which again is guaranteed to exist by [Zak83]). Then, from lemma 8 and lemma 14 we conclude that there exists $\mathbf{ACC^0}$ witness circuits. By lemma 11, we generate such witnesses to construct the circuit D_x from theorem 20. Here there is a small technicality. If we were to transform D_x to a $\texttt{SYM}^+$ circuit, the method described to get a fast `CIRCUIT SAT` algorithm would not work. Essentially, for it to work we need to reduce slightly the number of inputs. Define D'_x to be 2^ℓ copies of D_x which have ℓ inputs randomly fixed each and each copy is the input of an OR gate. Clearly this new circuit is satisfiable if, and only if, D_x is. Note that D'_x has $n - \ell$ inputs and size $2^\ell \cdot |D_x|$. We now generate an equivalent circuit D^*_x which is $\texttt{SYM}^+$ and with the help of lemma 12 we determine its satisfiability in time $o(2^n)$. Therefore, we have decided the membership of x in L in time $o(2^n)$, a contradiction, which means that the assumption $\mathbf{NEXP} \subseteq \mathbf{ACC^0}$ must be false, thus it must be the case that $\mathbf{NEXP} \not\subseteq \mathbf{ACC^0}$.

Chapter 5

Complexity barriers

5.1 Overview

In this section we want to introduce the known hurdles that complexity theory is to clear in order to settle questions such as $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$. Most of the interest that Williams' method has drawn is because it is likely that this method is capable of clearing the three known complexity barriers. The first complexity barrier was introduced in the 70's by [BGS75]. This barrier, called *relativization*, showed that there exist oracles A, B such that

- $\mathbf{P^A} = \mathbf{NP^A}$
- $\mathbf{P^B} \neq \mathbf{NP^B}$

What this meant is that any method that were to show $\mathbf{P} = \mathbf{NP}$ or $\mathbf{P} \neq \mathbf{NP}$ could not be sensitive to oracles, that is it could not *relativize*. This meant that although computations with oracles may be impractical, it is a great tool to classify and study the relationships between different complexity classes. After this first barrier, in the 90's, a paper by Razborov and Rudichs ([RR97]) introduced another barrier called *Natural Proofs*. They showed that under the hypothesis that there exist strong enough pseudorandom generators (a plausible hypothesis) most of the known methods to prove circuit lower bounds, such as the ones discussed in chapter 3, could not solve the question $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$. Roughly, this barrier says that if we were to isolate some property of Boolean circuits which make them unable to compute some function, then, this property can be used to distinguish between true random strings and pseudorandom strings "efficiently", thus contradicting the hypothesis that those pseudorandom generators do exist. Finally, the proof that $\mathbf{PSPACE} = \mathbf{IP}$ seemed to circumvent both barriers because it used the *arithmetization* of Boolean formulas. But in 2009, work of Aaronson and Widgerson ([AW08]) "extended" the relativization barrier with what is known as *algebrization*. Then, the arithmetization technique was shown to not relativize but to algebrize.

In this chapter we will briefly introduce the three barriers.

5.2 Natural Proofs

Let $\mathcal{F}_n$ denote the set of all boolean functions on n variables. We define a boolean property as a collection $\{C_n \subseteq \mathcal{F}_n \mid n \in \mathbb{N}\}$ and say that a given boolean function f_n has said property if, and only if, $f_n \in C_n$. We say that the boolean property is *natural* if there exists a collection of subsets $\{C_n^* \subseteq C_n \mid n \in \mathbb{N}\}$ such that the following two criteria are met:

- Constructivity: the predicate $f_n \stackrel{?}{\in} C_n^*$ is computable in time polynomial in the truth table of f_n.

- Largness: $|C_n^*| \geq \frac{|\mathcal{F}_n|}{2^{O(n)}}$.

A combinatorial property is useful against $\mathbf{P}$/poly if for any sequence of functions $\{f_i \mid i \in \mathbb{N} \wedge f_i \in C_i\}$ the circuit size for any f_i is super-polynomial, i.e for any constant c the circuit size of f_n is greater than n^c.

Definiton 11. *[AB09] Let $\{f_k \mid k \in \{0,1\}^*\}$ be a family of boolean functions $f_k : \{0,1\}^{|k|} \mapsto \{0,1\}^{|k|}$ and suppose that there is a polynomial-time algorithm that computes $f_k(x)$ given $k \in \{0,1\}^*$ and $x \in \{0,1\}^{|k|}$. We say that the family is pseudorandom if for every probabilistic polynomial time oracle Turing machine M there is a negligible function $\epsilon : \mathbb{N} \mapsto [0,1]$ such that*

$$\left| \Pr_{k \in \{0,1\}^n} [M^{f_k}(1^n) = 1] - \Pr_{g \in \mathcal{F}_n} [M^g(1^n) = 1] \right| < \epsilon(n)$$

.

A function $\epsilon(n)$ is a *negligible function* if for any polynomial $q(n)$ we have that $|\epsilon(n)| < 1/q(n)$ with $n > n_0$ for sufficiently large n_0.

For the purposes of showing the limitations of natural proofs, we will need the following theorem:

Theorem 26. *Suppose there exists a pseudorandom generator $G : \{0,1\}^\ell \mapsto \{0,1\}^{2\ell}$. Then, there exists a pseudorandom family of functions.*

Proof. **sketch of the main idea** [GGM86] If $z = G(x)$, let $G_0(x)$ denote the first ℓ bits of $G(x)$, i.e $z_1 \circ \cdots \circ z_\ell$. Likewise let $G_1(x)$ denote the last ℓ bits of $G(x)$. Define then h_k with $k \in \{0,1\}^\ell$ as

$$h_k(x) = G_{x_\ell}(G_{x_{\ell-1}}(\cdots G_{x_\ell}(k)) \cdots)$$

We can visualize h_k as a labeled binary tree of depth ℓ. The root is labeled by k. For any node v with label m, we define its right and left child v_0, v_1 as $G_{m \circ 0}(k)$ and $G_{m \circ 1}(k)$ respectively. Note that each node is naturally labeled by the binary string described by path from the root to it. Thus, for any $x \in \{0,1\}^\ell$ we just need to traverse the path defined by x until we reach a leaf. There we will get $G_{x_\ell}(G_{x_{\ell-1}}(\cdots G_{x_\ell}(k)) \cdots) = h_k(x)$. Regarding the resources needed to compute $h_k(x)$ notice that we need ℓ invocations of G, thus h_k takes polynomial time. We show now that $H_\ell = \{h_k \mid k \in \{0,1\}^\ell\}$ is indeed a pseudorandom family of functions. Suppose for a contradiction that it is not. By definition, there exists a probabilistic polynomial time oracle Turing machine T such that

$$\left| \Pr_{k \in \{0,1\}^\ell} [T^{h_k}(1^\ell) = 1] - \Pr_{g \in \mathcal{F}_\ell} [T^g(1^\ell) = 1] \right| \geq \epsilon(n) \tag{5.1}$$

Using this fact, we will construct a statistical test that will distinguish between a set of truly random strings and a set of strings generated by G, thus arriving at a contradiction because G is assumed to be a pseudorandom generator. Consider the following probabilistic Turing machine A_i^k which we will be restricted to inputs 1^ℓ. Let p_z $z \in \{0,1\}^*$ be the path in the binary tree induced by z

$A_i^\ell(y)$:

1. Let $y' = y_1 \circ \cdots \circ y_i$ be the i length prefix of y
2. If y' has never been processed before select a random string r with $|r| = \ell$, label the node $p_{y'}$ with r and output $G_{y_{i+1} \cdots y_\ell}(r)$
3. Otherwise let w be the label of node $p_{y'}$. Output $G_{y_{i+1} \cdots y_\ell}(w)$

We can finally introduce the statistical test for strings A_T. It first picks $i \in \{0, \ell - 1\}$ uniformly at random. Then, A_T simulates T and answer the queries of T with the corresponding A_ℓ^i machine.

Now, define p_ℓ^i the probability that $T^{A_i^\ell}(1^\ell) = 1$ where $0 \leq i \leq \ell$. Let $p_\ell^H, p_\ell^{\mathcal{F}}$ be the probability that $T^h(1^\ell) = 1$ and $T^f(1^\ell) = 1$ respectively, where h is picked uniformly at random from H and f is picked uniformly at random from $\mathcal{F}_\ell$. By assumption F does not pass the test defined by T, so there exists some polynomial $q(x)$ such that

$$|p_\ell^F - p_\ell^H| > \frac{1}{q(\ell)}$$

Consider now a set U of strings of length $2k$. If U is a set of pseudorandom strings generated by the pseudorandom generator, when A_T is ran feeding it pseudorandom strings, the probability that A_T outputs 1 is

$$\sum_{i=0}^{\ell-1} \frac{1}{k} \cdot p_\ell^i$$

On the other hand, if U is a set of truly random strings, then, when running A_T on this set, the probability that A_T outputs 1 is

$$\sum_{i=0}^{\ell-1} \frac{1}{k} \cdot p_\ell^{i+1}$$

The difference is the p_ℓ^{i+1} we get in the truly random case. Recall, that the idea is that A_T fills the tree with random functions up to the i-th level. But when strings are truly random, when we return the node at level $i+1$ we are returning a truly random string, thus it is equivalent to having filled the $i + 1$ level with random strings.

If we calculate what is the probability difference to output 1 in one case or the other, we get that this is

$$\sum_{i=0}^{\ell-1} \frac{1}{\ell} \cdot p_\ell^{i+1} - \sum_{i=0}^{\ell-1} \frac{1}{\ell} \cdot p_\ell^{i+1} \geq \frac{1}{\ell} \cdot (p_\ell^0 - p_\ell^\ell) > \frac{1}{\ell \cdot q(\ell)}$$

.

□

Next we show that natural proofs are self defeating:

Theorem 27. *There is no natural property against* $\mathbf{P}$*/poly unless* $H_g(k) \leq 2^{k^{o(1)}}$ *for every pseudorandom generator* $g : \{0,1\}^k \mapsto \{0,1\}^{2k}$ *in* $\mathbf{P}$*/poly.*

For the purposes of this proof, C_n will denote $\{C_n \mid n \in \mathbb{N}\}$.

Proof. Suppose for the sake of contradiction that C_n is a natural property against $\mathbf{P}$/poly. Without loss of generality assume that $C_n^* = C_n$. Let $G_k : \{0,1\}^k \mapsto \{0,1\}^{2k}$ be a pseudorandom generator and fix $\epsilon > 0$. Let $n = \lceil k^\epsilon \rceil$. We can generate a family of pseudorandom functions $F = \{f_z \mid z\{0,1\}^k\}$ similar to that of theorem 26. We skip the details, but essentially the difference is that for this proof we are only interested in the first bit of $f_z(x)$. As we have seen in theorem 26 , $f_z(x)$ is computable in polynomial time, thus since C_n is a natural property against $\mathbf{P}$/poly we have that $f_z(x) \notin C_n$, this fact will provide us with an statistical test, thus reaching a contradiction. Consider the following Turing machine with oracle access to some function h which will be either a pseudorandom function or a random function:

$T(z)$:
1. let $n = z^{\epsilon/2}$.
2. Define a function $g : \{0,1\}^n \mapsto \{0,1\}$ as $g(x) = h(x0^{z-n})$.
3. Construct the truth table of $g(x)$.

4. Run the combinatorial property C_n test to $g(x)$ and output the same as the test.

Note that generating the truth table of $g(x)$ takes time $2^{O(n)}$. Now, if the oracle function (h) is really a random function, then T outputs 1 with probability $\frac{1}{n}$ (By the largeness of the combinatorial property C_n). Otherwise, if h is a pseudorandom function, since it can be computed in polynomial time, then $h \notin C_n$ as we stated at the start of the proof. Thus, the probability that T outputs 1 when given oracle access to a pseudorandom function is 0. Therefore, we contradict the fact that $\{f_z \mid z \in \{0,1\}^k\}$ was a pseudorandom family of functions. □

5.3 Relativization

In [BGS75] it is shown that there are oracles A, B such that $\mathbf{P^A} = \mathbf{NP^A}$ and $\mathbf{P^B} \neq \mathbf{NP^B}$. Note that for an arbitrary oracle $\mathbf{C}$, we will be interested in the case $\mathbf{NP^C} \subseteq \mathbf{P^C}$, since it is straightforward to see that $\forall \mathbf{C}\ \mathbf{P^C} \subseteq \mathbf{NP^C}$. Thus we start by showing an oracle that makes them equal.

Theorem 28. *[BGS75] There exists an oracle A such that $\mathbf{NP^A} \subseteq \mathbf{P^A}$.*

Proof. Let A be any **PSPACE** complete language, for instance `TQBF` (True Quantified Boolean formulas). Given a language $L \in \mathbf{NP^A}$, we first show that $L \in \mathbf{PSPACE}$. Indeed, let M^A be a non-deterministic oracle Turing machine such that $L = L(M^A)$. There exists a deterministic machine M' that explores al possible configurations of M^A and whenever M^A makes a query to A. M' answers it itself. Thus $\mathbf{NP^A} \subseteq \mathbf{PSPACE}$. Now, see that given a an instance x we can compute the reduction of it to the **PSPACE** complete problem that we put as oracle in polynomial time and ask then just ask the oracle whether it belongs or not. Thus, we have shown that $\mathbf{NP^A} \subseteq \mathbf{PSPACE} \subseteq \mathbf{P^A}$. □

In order to show an oracle separating both classes, we need a construction a bit more complicated. Given an oracle B define $L_B = \{1^n \mid B \cap \{0,1\}^n \neq \emptyset\}$.

Theorem 29. *There exists an oracle B such that $\mathbf{NP^B} \nsubseteq \mathbf{P^B}$.*

Proof. Let $\{M_i \mid i \in \mathbb{N}\}$ be an enumeration of the deterministic polynomial time Turing machines and let $\{p_i \mid i \in \mathbb{N}\}$ be the corresponding polynomials, i.e M_i on input x with $|x| = n$ runs in time $p_i(n)$. We will build an oracle B inductively. On step i construct B_i as follows:

1. let n_i be such that $2^{n_i} > p_i(n_i)$ and $n_i > p_j(n_j)$ for all $0 \leq j < i$.
2. simulate $M_i(1^{n_i})$ responding with "0" to all queries. Let $M_i(1^{n_i}) = b$.
3. Let $Q = \{q_1, \ldots, q_k\}$ be all the queries done by $M_i(1^{n_i})$. Pick $w \in \{0,1\}^{n_i} \setminus Q$.
4. If $b = 0$ then define $B_i = w \cup B_{i-1}$. Otherwise define $B_i = B_{i-1}$.

For correctnes sake assume that the enumeration starts at 1, and define $B_0 = \emptyset$. Now we check that $B = \{\bigcup B_i \mid i \in \mathbb{N}\}$ is such that $\mathbf{NP^B} \subseteq \mathbf{P^B}$. In particular, it is the case that $L_B \in \mathbf{NP^B}$ but $L_B \in \mathbf{P^B}$. Consider the execution of $M_i^B(n_i)$. First thing to note is that by construction, the only queries that in B that M_i can ask are those in B_i since those in B_j for $j > i$ are too long to be queried by M_i (by construction). Thus we focus on the case $M_i^{B_i}(1^{n_i})$. Pick $x \in B_i$. By construction, if the machine accepts x, then x is not in B and thus not in L_B. Likewise, if the machine rejects x then x is in B and thus in L_B. It follows that L_B cannot be decided by any polynomial time oracle Turing machine, thus $L_B \notin \mathbf{P^B}$. Whereas $L_B \in \mathbf{NP^B}$ since on input 1^n we can guess the string $z \in \{0,1\}^n$ and query the oracle. □

5.4 Algebrization

Here we briefly discuss the barrier of *algebrization* introduced by [AW08].

Given a boolean function $f_m : \{0,1\}^m \mapsto \{0,1\}$ and a finite field $\mathbb{F}$, we say that a polynomial $p(x_1 \dots x_m)$ is an *extension* of f over $\mathbb{F}$ if for every $x \in \{0,1\}^m$ we have $f_m(x) = p(x)$. Such polynomial is denoted by $\widetilde{f}_{m,\mathbb{F}}$. Recall that an Oracle can be regarded as a collection of boolean functions $A = \{A_n \mid n \in \mathbb{N}\}$, where $A_n : \{0,1\}^n \mapsto \{0,1\}$. We define an Oracle extension, denoted by $\widetilde{A}_{\mathbb{F}}$, as a collection of extensions of A_n (each denoted by $\widetilde{A}_{n,\mathbb{F}}$) for $n \in \mathbb{N}$ and for any finite field $\mathbb{F}$ such that there exists a constant c such that the $\text{mdeg}(\widetilde{A}_{n,\mathbb{F}}) \leq c$ for all n and for any finite field $\mathbb{F}$. This way we can extend naturally the concept of relativized class, where if $\mathbf{C}$ is a complexity class and $\mathbf{C}^{\mathbf{A}}$ is the same class where the underlying computational model can query the oracle A, we have that $\mathbf{C}^{\widetilde{\mathbf{A}}}$ is the class where the underlying computational model can query an algebraic extension of A as per defined in **??**.

Also, $\mathbf{C}^{\widetilde{\mathbf{A}}_{[\text{poly}]}}$ is the same as $\mathbf{C}^{\widetilde{\mathbf{A}}}$ except that we only allow to query the extension $\widetilde{A}_{m,\mathbb{F}}$ for $m = poly(n)$ and $|\mathbb{F}| = 2^{O(m)}$.

With this basic concepts we can introduce, as in the last section, the notion of algebrization:

Definiton 12. ***Algebrization*** *[AW08] We say that a complexity classes inclusion $\mathbf{C} \subseteq \mathbf{D}$ algebrizes, if $\mathbf{C}^{\widetilde{\mathbf{A}}} \subseteq \mathbf{D}^{\mathbf{A}}$ for any oracle A and any extension $\widetilde{A}$ of A over any finite field. On the other hand, we say that $\mathbf{C} \subseteq \mathbf{D}$ does not algebrize if there exists $A, \widetilde{A}$ such that $\mathbf{C}^{\mathbf{A}} \not\subseteq \mathbf{D}^{\widetilde{\mathbf{A}}}$.*

Likewise, for a complexity classes separation $\mathbf{C} \not\subseteq \mathbf{D}$ we say that the separation algebrizes if $\mathbf{C}^{\mathbf{A}} \not\subseteq \mathbf{D}^{\widetilde{\mathbf{A}}}$ for any oracle A and any extension $\widetilde{A}$ of A over any finite field. On the other hand, we say that $\mathbf{C} \not\subseteq \mathbf{D}$ does not algebrize if there exists $A, \widetilde{A}$ such that $\mathbf{C}^{\widetilde{\mathbf{A}}} \subseteq \mathbf{D}^{\mathbf{A}}$.

Theorem 30. *[FS88] There exists an oracle A and a language $L \in \textbf{co-NP}^{\mathbf{A}}$ such that $L \notin \mathbf{IP}^{\mathbf{A}}$.*

We just state the theorem since the way it is proved is quite similar to theorem 29.

This theorem implies that the *arithmetization* technique is nonrelativizing since it is able to proof a nonrelativizing result. Suppose we want to relativize the inclusion $\mathbf{PSPACE} \subseteq \mathbf{IP}$, that is we want to show $\mathbf{PSPACE}^{\mathbf{A}} \subseteq \mathbf{IP}^{\mathbf{A}}$. It can be shown that $\texttt{TBQF}^{\texttt{A}}$ is $\mathbf{PSPACE}^{\mathbf{A}}$-complete. Thus, we can proceed and try to adapt the sumcheck protocol (lemma 1) with the inclusion of the oracle A. But what we mean by a formula Φ in $\texttt{TBQF}^{\texttt{A}}$? This formula must be augmented with a special predicate $A(x_1, \dots, x_k)$, which is true if, and only if, the string represented by $x_1, \dots, x_k$ belongs to A. The problem comes when we want to arithmetize Φ. In order to arithmetize Φ, we need a polynomial $p(x_1, \dots, x_k)$ such that $p(x_1, \dots, x_k) = 1 \iff x_1, \dots, x_k \in A$ and it must have low degree. The problem is that A can be arbitrary, which means that we cannot stablish a bound on the degree of $p(x_1, \dots, x_n)$ which could be as high as exponential in n. That's why we can see algebrization as a natural extension of relativization, where we just not only do we assume some oracle, but we also assume some structure for this oracle (namely a set of polynomials which extend it).

In the work of [AW08] where algebrization is introduced, the following theorem is proved:

Theorem 31. *[AW08] There exists $A, \widetilde{A}$ such that $\mathbf{NTIME}^{\widetilde{\mathbf{A}}}(2^n) \subseteq \mathbf{SIZE}^{\mathbf{A}}(n)$.*

Which by a standard padding argument yields the following corollary:

Corollary 32. *There exists $A, \widetilde{A}$ such that $\mathbf{NEXP}^{\mathbf{A}} \subseteq \mathbf{P}^{\widetilde{\mathbf{A}}}/poly$.*

Since [Wil14] shows that $\mathbf{NEXP} \not\subseteq \mathbf{ACC^0}$, if corollary 32 could be improved to show that $\mathbf{NEXP}^{\widetilde{\mathbf{A}}} \subseteq \mathbf{ACC0}^{\mathbf{A}}$, we could conclude that William's method does not algebrize, at least as applied in [Wil14].

There is another important concept related to algebrization introduced in [AW08], and that is the concept of k-algebrizing. Suppose that we have a relativized class inclusion $\mathbf{C}^{\mathbf{A}} \subseteq \mathbf{D}^{\mathbf{A}}$ for alla A. It can be seen easily, that if $\mathbf{D}^{\mathbf{A}} \subseteq \mathbf{F}^{\mathbf{A}}$ for all A, then we have $\mathbf{C}^{\mathbf{A}} \subseteq \mathbf{F}^{\mathbf{A}}$ for all A. In other words, relativization is *transitive*. This does not hold for algebrization because if $\mathbf{C}^{\mathbf{A}} \subseteq \mathbf{D}^{\widetilde{\mathbf{A}}}$ for all $A, \widetilde{A}$ and $\mathbf{D}^{\mathbf{A}} \subseteq \mathbf{F}^{\widetilde{\mathbf{A}}}$ for all $A, \widetilde{A}$, it does not necessarily follow that $\mathbf{C}^{\mathbf{A}} \subseteq \mathbf{F}^{\widetilde{\mathbf{A}}}$ for all $A, \widetilde{A}$. In [AW08] this is left as an open problem, but there is an algebrization notion that exhibits some kind of transitivity. We say that $\widetilde{\widetilde{A}}$ is a double-extension of A and is defined by:

1. Take a low-degree extension $\widetilde{A}$ of A.
2. Define $f(x, i)$ to be the function that returns the i-th bit of the binary representation of $\widetilde{A}(x)$.
3. Take a low-degree extension of $\widetilde{\widetilde{A}}$ of f.

It is clear that we can extend this method to get a k-extension of any oracle. Under this notion we have that for the classes $\mathbf{C}, \mathbf{D}, \mathbf{F}$ if $\mathbf{C}^{\mathbf{A}} \subseteq \mathbf{D}^{\widetilde{\mathbf{A}}}$ and $\mathbf{D}^{\mathbf{A}} \subseteq \mathbf{F}^{\widetilde{\mathbf{A}}}$ then $\mathbf{C}^{\mathbf{A}} \subseteq \mathbf{F}^{\widetilde{\widetilde{\mathbf{A}}}}$ for all $A, \widetilde{\widetilde{A}}$. In [AW08] it is shown that any proof that settles $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ cannot be k-algebrizing for any constant k.

5.5 William's method

We have not been able to study carefully how is Williams' method affected by the complexity barriers previously introduced, but we can briefly discuss how William's method may be affected by the complexity barriers. First let's recall the general idea behind William's method. We had a circuit D_x such that deciding its satisfiability was equivalent to deciding if $x \stackrel{?}{\in} L$ for an arbitrary $L \in \mathbf{NTIME}(2^n)$. Then the proof split in two parts

1. We assume that the separation we want to show is not true. In [Wil10] we assumed $\mathbf{NEXP} \subseteq \mathbf{P}/poly$, and from this assumption we prove the existance of "small" witness circuits.
2. We exhibit a better-than-trivial algorithm for `CIRCUIT SAT`.

First off, it doesn't seem that Williams' method should be affected by Natural proofs. We don't exploit a property of boolean functions to show the circuit lower bounds, we essentially focus on particular functions. We use mainly diagonalizing arguments to get witnesses, which is a technique known to avoid the natural proofs barrier. With respect to exhibitting a better `CIRCUIT SAT` algorithm, clearly, this part is unaffected by the natural proofs barrier.

The method seems to also avoid relativization and algebrization. For the relativization barrier, we have that in order to get the conditional proof of "small" witnesses we make use of the result $\mathbf{PSPACE} = \mathbf{IP}$ by [Sha92] which is known to not relativize. Moreover, a better-than-trivial algorithm for `CIRCUIT SAT` seems to avoid both relativization and algebrization. Consider a circuit augmented with gates for an oracle A or for a low-degree extension $\widetilde{A}$ of it. An oracle can be any arbitrary boolean function, so the oracle gates are substantially more complex than the general basis we use for circuits, i.e the basis over NOT, AND and OR gates. For instance, the better-than-trivial algorithm given to show $\mathbf{NEXP} \not\subseteq \mathbf{ACC^0}$ makes heavy use of the structure of the circuit. In particular, it makes use of the fact that any $\mathbf{ACC^0}$ circuit can be transformed to a $\mathtt{SYM}^+$ which only has AND gates and a symmetric gate at the output.

All this reasons leads to believe that Williams' method indeed is able to circumvent this barriers and hopefully help researchers in complexity theory two produce new results.

Bibliography

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A modern approach*. Cambridge University press, 2009.

[Ajt83] M. Ajtai. $\sum_1^1$-formulae on finite structures. *Annals of Pure and Applied Logic*, 24(1):1 – 48, 1983.

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. An o(n log n) sorting network. *ACM*, 1983.

[AW08] Scott Aaronson and Avi Wigderson. Algebrization: A new barrier in complexity theory. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, STOC '08, pages 731–740, New York, NY, USA, 2008. ACM.

[BFL91] László Babai, Lance Fortnow, and Carsten Lund. Non-deterministic exponential time has two-prover interactive protocols. *computational complexity*, 1(1):3–40, Mar 1991.

[BFNW93] László Babai, Lance Fortnow, Noam Nisan, and Avi Wigderson. Bpp has subexponential time simulations unless exptime has publishable proofs. *Comput. Complex.*, 3(4):307–318, October 1993.

[BGS75] Theodore Baker, John Gill, and Robert Solovay. Relativizations of the $\mathcal{P} =? \mathcal{NP}$ question. *SIAM J. Comput.*, 4(4):431–442, 1975.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. *Proceedings of the Third Annual ACM Symposium. Theory of Computing*, 1971.

[Coo72] Stephen A. Cook. A hierarchy for nondeterministic time complexity. *Journal of Computer and System Sciences*, 1972.

[Cop82] D. Coppersmith. Rapid multiplication of rectangular matrices. *SIAM*, 1982.

[FLvMV04] Lance Fortnow, Richard Lipton, Dieter van Melkebeek, and Anastasios Viglas. Time-space lower bounds for satisfiability, September 2004.

[FS88] Lance Fortnow and Michael Sipser. Are there interactive protocols for co-np languages? *Inf. Process. Lett.*, 28(5):249–251, August 1988.

[FSS84] Merrick Furst, James B. Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, (17):13–27, 1984.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4):792–807, August 1986.

[Has86] J Hastad. Almost optimal lower bounds for small depth circuits. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, pages 6–20, New York, NY, USA, 1986. ACM.

[Kan82] R. Kannan. Circuit-size lower bounds and non-reducibility to sparse sets. *Information and control*, (55):40–56, 1982.

[KM02] Adam R. Klivans and Dieter Van Melkebeek. Graph nonisomorphism has subexponential size proofs unless the polynomial-time hierarchy collapses. *SIAM J. Computing*, 31(5):1501–1526, 2002.

[LFKN90] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. Technical report, Chicago, IL, USA, 1990.

[NW94] Noam Nisan and Avi Wigderson. Hardness vs randomness. *J. Comput. Syst. Sci.*, 49(2):149–167, October 1994.

[O'D] Ryan O'Donnel. Complexity lecture 14. https://www.cs.cmu.edu/ odonnell.

[Raz85] A. A. Razborov. Lower bounds for the monotone complexity of some boolean functions. *Soviet Math. Dol.*, 31(2), 1985.

[Raz87] A. A. Razborov. Lower bounds on the size of bounded depth network over a complete basis with logical addition. *Mathematical Notes of the Academy of Sciences of the USSR*, 4(41):333–338, 1987.

[Rob91] J. M. Robson. An o(t log t) reduction from ram computations to satisfiability. *Theor. Comput. Sci.*, 82(1):141–149, May 1991.

[RR97] Alexander A. Razborov and Steven Rudich. Natural proofs. *Journal of Computer and System sciences*, (55):24–35, 1997.

[RW00] Steven Rudich and Avi Widgerson, editors. *Computational Complexity Theory*, volume 10. American Mathematical Society and IAS/Park city Mathematics Institute, 2000.

[Sha92] Adi Shamir. Ip = pspace. *J. ACM*, 39(4):869–877, October 1992.

[SHH86] R.E Stearns and III H.B. Hunt. On the complexity of the satisfiability problem and the structure of *NP*. Technical report, Computer Science Department, State University of New York at Albany, 1986.

[Smo87] R. Smolensky. Algebraic methods in the theory of lower bounds for boolean circuit complexity. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 77–82, New York, NY, USA, 1987. ACM.

[Tou01] Iannis Tourlakis. Time-space tradeoffs for SAT on nonuniform machines. *Journal of Computer and System Sciences*, (63):268–287, 2001.

[vM07] Dieter van Melkebeek. A survey of lower bounds for satisfiability and related problems. *Foundations and Trends in Theoretical Computer Science*, 2(3):197–303, 2007.

[Wak68] Abraham Waksman. A permutation network. *J. ACM*, 15(1):159–163, January 1968.

[Wil10] Ryan Williams. Improving exhaustive search implies superpolynomial lower bounds. In *Proceedings of the Forty-second ACM Symposium on Theory of Computing*, STOC '10, pages 231–240, New York, NY, USA, 2010. ACM.

[Wil14] Ryan Williams. Nonuniform acc circuit lower bounds. *J. ACM*, 61(1):2:1–2:32, January 2014.

[Zak83] Stanislav Zak. A turing machine time hierarchy. *Theoretical Computer Science*, 26:327–333, 1983.