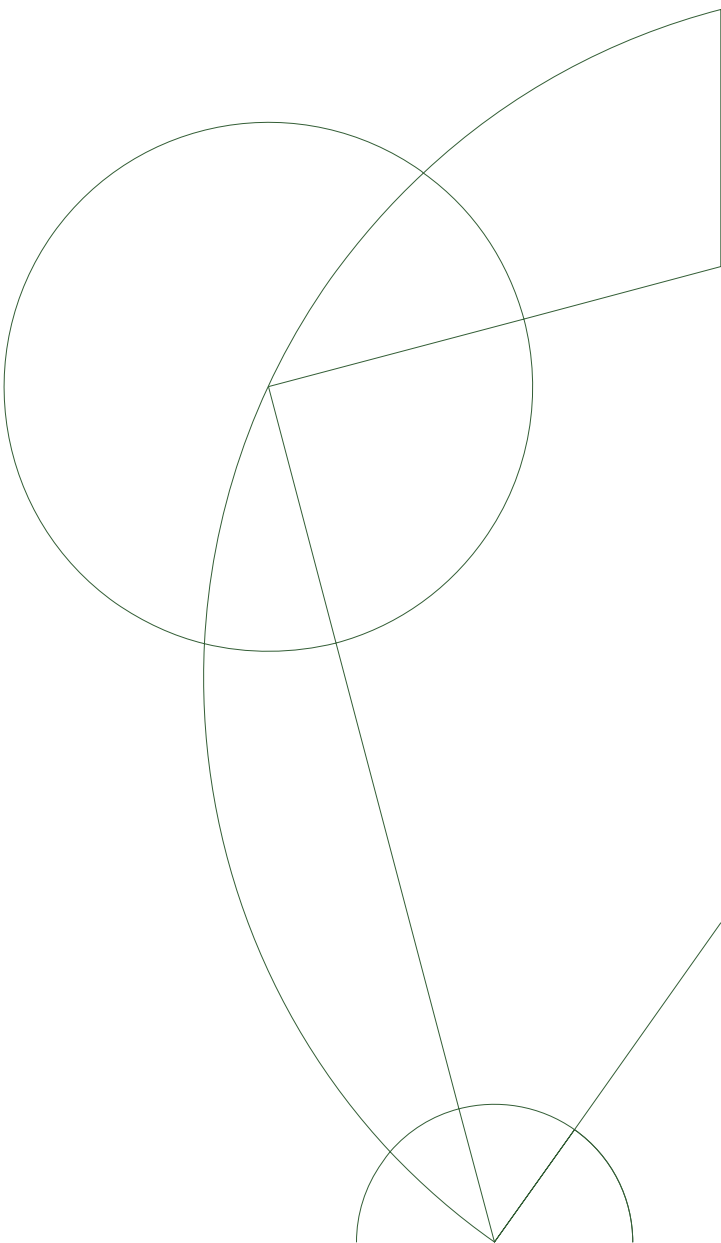
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Bachelor Thesis in Mathematics

Sune Kristian Jakobsen

Barriers to proving $P \neq NP$



Bachelor Thesis in Mathematics.

Department of Mathematical Sciences,

University of Copenhagen

Bachelorprojekt i matematik.

Institut for matematiske fag,

Københavns Universitet

Advisors: Jakob Grue Simonsen, Søren Eilers.

June 10, 2011

Abstract

This thesis is about the P $\stackrel{?}{=}$ NP problem and why this is so difficult to solve. We consider two different techniques from complexity theory, see how they can be used to separate complexity classes and why they do not seem to be strong enough to prove P $\neq$ NP. The first technique is diagonalization, which we will use to prove the time hierarchy theorem. We will then see why a result of Baker, Gill and Solovay seems to indicate that this technique cannot prove P $\neq$ NP. We will see how the circuit model can be use to prove $PARITY \notin AC^0$ and see a result by Razborov and Rudich, which indicate that this approach is also too weak to prove P $\neq$ NP. Finally, we will see that provability and computability are closely related concepts and show some independence results.

Resumé

Dette projekt handler om P $\stackrel{?}{=}$ NP-problemet, og om hvorfor det er så svært at løse. Vi betragter to forskellige teknikker fra komplexitetsteori, ser hvordan de kan bruges til at adskille kompleksitetsklasser og hvorfor det tilsyneladende ikke er tilstrækkelige til at bevise P $\neq$ NP. Den første teknik er diagonalisering, som vi vil bruge til at vise tidshierarkisætningen. Vi vil derefter se at et resultat af Baker, Gill og Solovay tyder på at denne teknik ikke kan vise P $\neq$ NP. Vi vil herefter se hvordan kredsmodellen kan bruges til at vise $PARITY \notin \mathrm{AC}^0$ og se et resultat af Razborov og Rudich, som tyder på at denne fremgangsmåde heller ikke er tilstrækkeligt til at vise P $\neq$ NP. Endelig vil vi se at beviseligehed og beregnelighed er tæt forbundede og bevise nogle uafhængighedsresultater.

Contents

Preface

Although the subject of this thesis is from theoretical computer science, on the borderline between mathematics and computer science, it was written as a part of my Bachelor in Mathematics. The thesis is about the P = NP problem. Here P is the class of languages that can be decided in polynomial time on a deterministic Turing machine, and NP is the class of languages that can be decided in polynomial time on a non-deterministic Turing machine. Most researchers in the field think that $P \neq NP$ [10] but the problem have now been open for 40 years. The intended reader of the thesis is anyone who knows what a Turing machine is, has heard about the P = NP problem and would like to know more about it. It is my hope that such a reader will achieve an understanding of some of the known techniques of complexity theory and why they do not seems to be strong enough to prove $P \neq NP$.

Comprehension of this thesis requires an understanding of what a Turing machine is and basic knowledge of complexity theory. In this regard, Spiser's "Introduction to the Theory of Computation" [24] may be useful. Still, I have decided to have a chapter with preliminaries. Here we define the notation and show some basic theorems about Turing machines. In Chapter 2 we consider an application of a technique called diagonalization and we see a reason that this does not seem to be strong enough to show $P \neq NP$. In Chapter 3 we do the same with a type of proofs called "natural proofs". In Chapter 4 we will see why there is a close connection between the concepts of computation and provability and we will see some independence results. In order to keep the thesis at a reasonable length, I have decided to put a section about the arithmetical hierarchy in an appendix. The results from this appendix are only used in Chapter 4. It will be useful to have read Chapter 2 in order to understand Chapter 4, but otherwise it should be possible to read Chapter 2, 3 and 4 independently.

Acknowledgements

I want to thank my advisor Jakob Grue Simonsen for suggesting articles to read and theorems to prove and for forcing me to write at lot at the beginning of the project. That made the last month of the project much less tense than it would otherwise have been. My internal adviser, Søren Eilers, helped me through the formalities and I want to thank him for that. I also want to thank Rasmus Nørtoft Johansen for mathematical proof reading, Mathias Bæk Tejs Knudsen for pointing out some small mistakes and Cathryn Moore for general proof reading. They have all helped raising the quality of this thesis.

Chapter 1

Preliminaries

1.1 Notation and definitions

Definition 1. When Σ is a set, Σ^* denotes the set of all finite strings over that set. We let ϵ denote the empty string, and for $x \in \Sigma^*$ the notation $|x|$ denotes the length of x. Whenever X is some structure, $\langle X \rangle$ denotes a representation of the structure in the input alphabet. We will use $\mathbb{N}$ to denote the set $\{0, 1, \dots\}$ of non-negative integers, $\mathbb{N}^+$ to denote the set $\{1, 2, \dots\}$ of positive integers, and $\mathbb{R}^+$ is the set of positive real numbers. We will use $\log(n)$ to denote the function $\log : \mathbb{N} \to \mathbb{N}$ defined by:

$$\log(n) = \begin{cases} 0 & , \text{ if } n = 0 \\ \lfloor \log_2(n) \rfloor & , \text{ otherwise} \end{cases}$$

where $\lfloor \cdot \rfloor$ is the floor function. For a function $f : \mathbb{N} \to \mathbb{R}$ we have the following sets:

- $O(f(n)) = \{g : \mathbb{N} \to \mathbb{R} | \exists c \in \mathbb{R}^+, N \in \mathbb{N} \forall n \geq N : g(n) \leq cf(n)\}$
- $o(f(n)) = \{g : \mathbb{N} \to \mathbb{R} | \forall c \in \mathbb{R}^+ \exists N \in \mathbb{N} \forall n \geq N : g(n) \leq cf(n)\}$
- $\Omega(f(n)) = \{g : \mathbb{N} \to \mathbb{R} | \exists c \in \mathbb{R}^+, N \in \mathbb{N} \forall n \geq N : g(n) \geq cf(n)\}$
- $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

In this document, we will mostly use multi-tape Turing machines. There are many slightly different but equivalent definitions of Turing machines. We will use the following, which is a combination of the definitions given in [19, Definition 2.1] and [24, p. 150]:

Definition 2. A *k-tape Turing machine* is a 5-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0)$ where:

- Q is the finite set of states,
- Σ is the finite input alphabet and does not contain the *blank symbol* $\sqcup$ or the *first symbol* $\rhd$,
- Γ is the finite tape alphabet $\Sigma \cup \{\sqcup, \rhd\} \subseteq \Gamma$,
- $\delta : Q \times \Gamma^k \to (Q \cup \{h, \text{“yes”}, \text{“no”}\}) \times (\Gamma \times \{\leftarrow, \rightarrow, -\})^k$ is the transition function, and if at least one of the $g_i \in \Gamma$ is $\rhd$ then $\delta(q_1, g_1, g_2, \dots g_k) = (q_2, g_1, d_1, g_2, d_2, \dots g_k, d_k)$ where $d_i = \rightarrow$ if $g_i = \rhd$ and $d_i = -$ otherwise
- $q_0 \in Q$ is the start state.

The Turing machine starts in the state q_0 and with $\rhd x$ on the first tape, where x is the input string and $\rhd$ on all the other tapes. The “$\rhd$”s are used to make sure that the heads do not fall off the tape to the left. The Turing machine has k heads, one on each tape. The heads start on the leftmost cell of their tape. If the i'th head reads the letter g_i and the machine is in state q_1 and $\delta(q_1, g_1, g_2, \dots, g_k) = (q_2, h_1, d_1, h_2, d_2, \dots, h_k, d_k)$ it goes to state q_2, overwrites the letter at the i'th head by h_i and move the i'th head in direction d_i where “$-$” means “do not move”. If, at some point, q_2 is “yes” the machine *accepts*, if it is “no” the machine *rejects* and if it is “h” the machine's output is the string y such that the last letter of y is not $\sqcup$ and the k'th tape is “$\rhd y \sqcup \sqcup \dots$”. If this happens on input x we say respectively “M *accepts* x”, “M *rejects* x”, and “$M(x) = y$”.

No Turing machines in this thesis will be explained formally, but hopefully they are all explained in such a way that the reader *could* in principle write down a formal description.

The next definition is standard and can be found in both [19, Definition 2.3] and [24, Definition 3.6].

Definition 3. Let $A \subseteq \Sigma^*$ be a language and M a Turing machine. We say that M *decides* A if M accepts x for every $x \in A$ and M rejects x for every $x \notin A$. When such a Turing machine exists A is *decidable*. We say that M *recognizes* A if M accepts x for every $x \in A$ and M does not accept any $x \notin A$ (it may reject or run forever on such an x). We denote this by $A = L(M)$, and when such a Turing machine exists we say that A is recognizable.[1] If a Turing machine halts on all input, we say that it is *total*.

We are now ready to define the complexity class TIME($f(n)$). Again, there are minor differences from text to text, and again this is a combination of the definitions given in [19, Definition 2.5] and [24, Definition 7.7][2].

Definition 4. If M is a Turing machine that halts on all input, we define its *running time* $g : \mathbb{N} \to \mathbb{N}$ as:

$$g(n) = \max\{\text{numbers of steps taken by } M \text{ when given the input } x \mid |x| = n\}$$

We say that a language L is in TIME($f(n)$) if there is a Turing machine M with running time in $O(f(n))$ that decides L.

The definition of SPACE($f(n)$) is more complicated, because we only want to bound the space used for computation and not the space used for input and output. This too is standard, and the following formulation is partly taken from [19, p. 35].

Definition 5. Let $k \geq 2$ and let M be a k-tape Turing machine. We say that it is a *Turing machine with input and output* if the transition function δ satisfies that whenever $\delta(q, g_1, \ldots, g_k) = (p, h_1, d_1, \ldots h_k, d_k)$ then[3]

1. $g_1 = h_1$
2. If $g_1 = \sqcup$ then $d_1 = \leftarrow$
3. $d_k \neq \leftarrow$

For a Turing machine M with input and output that terminates on all input the *space used by M on input x* is the sum $\sum_{i=2}^{k-1} s_i$ where s_i is the number of positions the head on the ith tape visit when we run M on x.

For a general k-tape Turing machine M, the *space used by M on input x* is just $\sum_{i=1}^{k} s_k$.

A language A is in SPACE($f(n)$) if it is decided by some Turing machine M (possibly a Turing machine with input and output) and for some $g(n) \in O(f(n))$ and any $x \in \Sigma^*$ the space used by M on input x is at most $g(|x|)$.

The classes TIME($f(n)$) and SPACE($f(n)$) are sets of languages, so we can take intersection and unions of them. We are now ready to define some of the most used complexity classes. These are all standard definitions [19, pp. 35, 46, 142, 492][4]:

Definition 6. We define the following complexity classes:

- $\mathrm{P} = \mathrm{TIME}(n^k) = \bigcup_{i=1}^{\infty} \mathrm{TIME}(n^i)$
- $\mathrm{E} = \mathrm{TIME}(2^{O(n)}) = \bigcup_{i=1}^{\infty} \mathrm{TIME}(2^{in})$

[1] Decidable is traditionaly called *recursive* and recognizable is traditionaly called *recursively enumerable*. The notation used here is from [24].

[2] In [24] the author considers the time needed on a single-tape machine, not a multi-tape machine. In [19] the class TIME($f(n)$) is the class of languages decidable in $f(n)$ instead of $O(f(n))$. Due to the linear speedup theorem ([19, Theorem 2.2]), this is equivalent to the definition used here, for all functions f for which there is a $\epsilon > 0$ such that $f(n) > (1 + \epsilon)n + 8$.

[3] Informally we say that the Turing machine is "read only" on the first tape and "write only" on the last, although this does not describe the requirements prefectly. Requirement 2 makes sure that the Turing machine does not use the value of "number of steps the head on first tape is to the right of the input" to store information.

[4] Even though the definitions of TIME and SPACE vary slightly, these five classes do not.

- $\text{EXP} = \text{TIME}(2^{n^k}) = \bigcup_{i=1}^{\infty} \text{TIME}\left(2^{n^i}\right)$
- $\text{L} = \text{SPACE}(\log(n))$
- $\text{PSPACE} = \text{SPACE}(n^k) = \bigcup_{i=1}^{\infty} \text{SPACE}(n^i)$

From the definitions it is clear that $\text{P} \subseteq \text{E} \subseteq \text{EXP}$ and $\text{L} \subseteq \text{PSPACE}$. In the section about the hierarchy theorem we will see a proof that $\text{P} \subsetneq \text{E} \subsetneq \text{EXP}$ and in a similar way, it can be proved that $\text{L} \subsetneq \text{PSPACE}$. Furthermore, these inclusions are well known [24, page 312 and 332]:

$$\text{L} \subseteq \text{P} \subseteq \text{PSPACE} \subseteq \text{EXP}$$

We have now defined one of the two classes in the $\text{P} \stackrel{?}{=} \text{NP}$ problem. To define the other one, we need to define non-deterministic Turing machines. This is also a standard definition, found in both [19, p. 45] and [24, p. 152].

Definition 7. A *non-deterministic Turing machine* is defined as a Turing machine, except that instead of

$$\delta : Q \times \Gamma^k \to (Q \cup \{h, \text{“yes”}, \text{“no”}\}) \times (\Gamma \times \{\leftarrow, \rightarrow, -\})^k$$

we have

$$\delta : Q \times \Gamma^k \to \mathcal{P}\left((Q \cup \{h, \text{“yes”}, \text{“no”}\}) \times (\Gamma \times \{\leftarrow, \rightarrow, -\})^k\right)$$

where $\mathcal{P}$ denotes the power set. At each step, the Turing machine can choose which of the element in $\delta(q, g_1, \ldots g_k)$ to follow. If δ returns the empty set, the machine just terminates. For every sequence of choices we call the resulting computation a *branch* of computation. If M is a non-deterministic Turing machine and $A \subseteq \Sigma^*$ we say that M *decides* A if: for any $x \in A$ there is a branch of computation of M that accepts x and for any $x \notin A$ all branches of computation rejects A. The *running time* of a non-deterministic Turing machine is the running time of longest branch of computation, and the *space used* is the largest amount of space used by any of the branches.

We can now define non-deterministic versions of all the complexity classes. The most interesting ones are the following [19, pp. 46,142]:

Definition 8. By $\text{NTIME}(f(n))$ and $\text{NSPACE}(f(n))$ we denote the class of languages decidable in time respectively space $O(f(n))$. Now

- $\text{NP} = \bigcup_{i=1}^{\infty} \text{NTIME}(n^i)$
- $\text{NL} = \text{NSPACE}(\log(n))$
- $\text{NPSPACE} = \bigcup_{i=1}^{\infty} \text{NSPACE}(n^i)$

It is known that $\text{L} \subseteq \text{NL} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE} = \text{NPSPACE} \subseteq \text{EXP}$, but we do not know if any of these inclusions are really equalities [24, p. 312 and 332], we only know that $\text{L} \subsetneq \text{PSPACE}$ and $\text{P} \subsetneq \text{EXP}$. The question “is the inclusion $\text{P} \subseteq \text{NP}$ strict” was first asked by Cook in [6]. In this thesis, we will see why this is a difficult problem.

1.2 Some basic theorems

First a very simple proposition that we will use later:

Proposition 1. *Let $A, B \subseteq \Sigma^*$ be two languages with $|A \triangle B| < \infty$, where*

$$A \triangle B = (A \setminus B) \cup (B \setminus A)$$

If $A \in TIME(f(n))$ then $B \in TIME(f(n))$.

Proof. Assume that A and B are as in the hypothesis and let M_A be a Turing machine that decides A in time $O(f(n))$. Let n_0 be the length of the longest string in $A \triangle B$. Now we construct a Turing machine M_B as follows: we give it a state for every string in Σ^* of length at most n_0. Now it can read and remember the n_0 first letters of the input (or just the entire input if it is shorter than n_0) and see if there is more than n_0 letter in the input. If not we make M_B accept if and only if the input is in B. If the input is longer than n_0, we know that it is in B if and only if it is in A, so we let the head return to the leftmost symbol and simulate M_A. This Turing machine decides B and runs in time $O(f(n) + n_0) = O(f(n))$. □

It turns out that anything that can be calculated on a multi-tape Turing machine can also be calculated on a single-tape Turing machine. The multi-tape machine might be faster than the best single-tape machine, but the following theorem gives us a bound on how much faster it can be. The theorem and proof is taken from [19, Theorem 2.1].

Theorem 2. *For any k-tape Turing machine M that runs in time $f(n)$, we can construct a single-tape Turing machine that runs in time $O(f(n)^2)$.*

Proof. If $f(n_0) < n_0$ for some n_0 then the machine does not read further than $f(n_0)$ if the input has length n_0. This implies that the machine never reads further than $f(n_0)$, so the output only depends on the $f(n_0)$ first letters. We can now make a single-tape Turing machine that just reads and remembers the $f(n_0)$ first letters (by having a state for each string of elements in Σ of length at most $f(n_0)$) and output what M would give as output. This runs in constant time, $f(n_0)$, and we are done. In the following we can assume $f(n) \geq n$.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0)$ be a k-tape Turing machine. We will now define a single-tape Turing machine M' that simulates it. To do this, we will keep all the used parts of Ms tapes on the tape, one after another. Of couse we cannot separate the tapes with $\triangleright$s, so instead we substitute all these with a new symbol $\triangleright'$. We mark the end of each of Ms tapes by another new symbol $\triangleleft$, and the end of the last tape by two $\triangleleft$s. We also need a way to mark where the heads are on each tape: to do this, we add an underlined version of each element in Γ to the tape alphabet. Now on input x we start by having $\triangleright x$ on the tape. To simulate M we need to move x one to the right, add a $\triangleright'$ to the left of it, add $\triangleleft(\underline{\triangleright'}\triangleleft)^{k-1}\triangleleft$ to the right of it and underline the first letter in x. Then we begin the simulation. In each step the head goes through the tape twice. The first time it reads what is at the underlined squares of the tape. To remember this, while remembering its state, we simply give M' a state for each $(q, g_1 g_2 \dots g_i)$, where $q \in Q$ and $g_j \in \Gamma$ and $0 \leq i \leq k$. Then it goes back to the beginning of the tape. After this it must update the tape. Again we can do this by giving M' a state for each possible update instruction, because k is fixed. The only problem comes when the head of a tape goes further to the right, than it has been before, because then we have to move the $\triangleleft$ and everything to the right of that further to the right. In this case, it will mark the $\triangleleft$ so we get a new symbol $\triangleleft'$, go to the rightmost square and move everything to the right of $\triangleleft'$ one square to the right and insert a $\underline{\sqcup}$. After updating the tape, it goes back to the first square to start the simulation of the next step of M.

The initialization is done in $O(n+k)$, where $n = |x|$. If $f(n)$ is the running time of M, then each of the tapes will have at most $f(n)$ used squares (remember that $f(n) \geq n$, so the input tape is no exception). Thus we use at most $k(f(n)+2)+1$ squares on the tape. So when it adds a new square, it only needs to move at most $O(kf(n))$ squares and it only have to do this $O(kf(n))$ times. This gives $O(k^2 f(n)^2)$ step. Going through the tape twice for each step of M takes $O(kf(n)^2)$ altogether. Thus everything runs in $O(k^2 f(n)^2)$ and as k is fixed, this is $O(f(n)^2)$. □

In the above theorem the simulation time is quadratic in the time taken by the original machine, but we cannot do better than that: Consider the language of palindromes in $\{0,1\}^*$. We can easily decide this in linear time on a 2-tape Turing machine: first we copy the input string to the second tape, and then we let the head on the first tape go back to the leftmost position on the tape. Then the first head goes from left to right while the second head goes from right to left and the machine compares what the two heads are reading. The input is a palindrome if and only if in each step the two heads read the same letter. However, on a single-tape Turing machine, we cannot do better than $\Omega(n^2)$. This can be proven using Kolmogorov complexity [17, Lemma 6.1.1][5]. The idea is: Assume that we have a single-tape Turing machine M that can decide this language in $o(n^2)$. Then consider what this machine would do on input $x0^n x^R$, where $n = |x|$ and x^R denote the reverse of x. As the machine runs in $o(n^2)$ there must be some point in the 0's that the head crosses a sublinear number of times. However, if we know M, n, a position the head crosses only in a small number of times, and the states the machine is in when it crosses that point we can deduce x, because if this was the same for x and y, then the machine would have to accept $y0^n x^R$. This would imply that we could describe any sufficiently long string x of length n in less than n bits, but this is not true.

The next result is form the first article about Turing machines [25].

Theorem 3. *There exists a Turing machine U such that if M is a Turing machine and x is a string over the input alphabet of M then $U(\langle M, x\rangle)$ simulates $M(x)$: if $M(x)$ accepts then $U(\langle M, x\rangle)$*

[5] This lemma is about the language $L = \{xx^R | x \in \{0,1\}^*\}$, which is the language of *even-length* palindromes. The proof for the language of all palindromes is exactly the same.

accepts, if $M(x)$ rejects then $U(\langle M, x\rangle)$ rejects, if $M(x)$ halts with output y then $U(\langle M, x\rangle)$ halts with output $\langle y\rangle$, and if $M(x)$ does not halt then $U(\langle M, x\rangle)$ does not halt.

Proof. We will not give a proof here. See [25] or [19, Section 3.1]. □

Such a Turing machine U is called universal. Usually we will not just simulate another Turing machine, but also use the result of the simulation. To do this, we can define a Turing machine that on input $\langle M, x\rangle$ first simulate U and then uses the output.

The following surprising result is usually proven as a corollary of Kleene's second recursion theorem [19, Problem 3.4.8], but we will instead prove it directly as in [24, Theorem 6.3].

Theorem 4. *For any Turing machine M that takes input on the form $\langle A, x\rangle$, where A is a Turing machine and x is any string, there is a Turing machine M' such that $M'(x)$ halts if and only if $M(\langle M', x\rangle)$ does, and if they halt then $M'(x) = M(\langle M', x\rangle)$*

Intuitively, this implies that we may assume that a Turing machine knows its own description: If we want a Turing machine that given an input x does some computations that involve its own description, we first define a Turing machine M that takes input $\langle A, x\rangle$ and assume that A is M itself. Now this theorem gives us a Turing machine M' that on input x gives that same result as $M(\langle M', x\rangle)$. In other words, $M'(x)$ performs the computations we want on $\langle M', x\rangle$.

Proof. When A and B are Turing machines, we let $B \circ A$ denote the Turing machine that on input x first simulates A on x, then erases everything but the output $A(x)$ and moves all heads back to the leftmost position, and then simulate B on input $A(x)$. If $A(x)$ or $B(A(x))$ does not halt, then $B \circ A$ does not halt. This is computable: there is a Turing machine that given $\langle A, B\rangle$ outputs $\langle B \circ A\rangle$.

When C is a Turing machine, we let A_C denote the Turing machine that on input x outputs $\langle C, x\rangle$. This is also computable.

Given M we define a Turing machine B that on input $\langle C, x\rangle$ outputs

$$M(\langle C \circ A_C, x\rangle).$$

Now we consider what $B \circ A_B$ does. On input x, it first simulates $A_B(x)$, so it writes $A_B(x) = \langle B, x\rangle$ on the output tape. Then B computes $M(\langle B \circ A_B, x\rangle)$, so $B \circ A_B$ is the Turing machine M' we wanted. □

In complexity theory, the concept of completeness is important. The definition is standard [24, Definition 7.34].

Definition 9. A language A is *NP-complete* if

1. $A \in \mathrm{NP}$, and
2. For any $B \in \mathrm{NP}$ there is a polynomial time Turing machine M such that $x \in B \Leftrightarrow M(x) \in A$.

This is useful, because now we only need to find a polynomial time Turing machine that decides a single NP-complete language to show that P = NP. On the other hand, we know that if a language is NP-complete, then it is probably impossible to find a Turing machine that decides the language in polynomial time. Of course, to make use of the definition, we need to know that there exists NP-complete languages.

Theorem 5. *The language 3-SAT is NP-complete.*[6] *In particular, NP-complete problems exists.*

Proof. See [24, Corollary 7.42]. □

[6] For a definition of 3-SAT see [24, p. 277-278].

Chapter 2

Diagonalization and Relativization

Diagonalization is a technique used to separate complexity classes. This chapter argues that it is not strong enough to separate P and NP. The method is used in Cantor's diagonal proof that the set of real numbers are uncountable: in Cantors proof, we assume for contradiction that we have a sequence $x_1, x_2, \ldots$ that contains all the real numbers and then we construct another real number y, by making sure that x_i and y disagree on the i'th decimal place. In complexity theory, we have a list of Turing machines $M_1, M_2, \ldots$ that works within some bounds and then we can find a language L that is not decided by any of the machines, by making sure that $L(M_i)$ and L disagree on input $\langle M_i \rangle$, or possibly on an infinite sequence of inputs $\langle M \rangle 10^j$. First we will see an application of diagonalization.

2.1 The time hierarchy theorem

What can we say about the relationship between TIME($f(n)$) and TIME($g(n)$), if we know that f grows faster than g? It is clear that if we give a Turing machine more time it will at least be able to solve the same problems as before, so TIME($g(n)$) $\subseteq$ TIME($f(n)$). We might also expect that if f grows much faster than g the Turing machine will be able to use the extra time to solve more difficult problems. This is not true in general,[1] but in many cases it is. This is known as the time hierarchy theorem. Here we will see a weak version of the theorem. The same technique can be used to prove a much stronger version of the theorem [24, Theorem 9.10].

First we need a standard definition that can be found in [24, Definition 9.8].

Definition 10. A weakly increasing function $f : \mathbb{N} \to \mathbb{N}$ is called *time constructible* if the function that maps the string 1^n to the binary representation of $f(n)$ is computable in time $O(f(n))$.

So what functions are time constructible? It turns out that most reasonable functions are time constructible. Here are some of the most important:

Proposition 6. *The functions $n, n\log(n), n^2$, and 2^n are all time constructible.*

Proof. To find the binary representation of n given 1^n we simply count the number of 1's. When we count to n we need to change the least significant bit n times, the next $\lfloor \frac{n}{2} \rfloor$ times, the next $\lfloor \frac{n}{4} \rfloor$ times and so on. Altogether this gives:

$$\sum_{i=0}^{\infty} \lfloor \frac{n}{2^i} \rfloor \leq \sum_{i=0}^{\infty} \frac{n}{2^i} = 2n$$

so the obvious counting algorithm runs in $O(n)$ time.

When we have counted n, we can also count the number of digits in the binary representation of n. This gives us $\log(n)$. To multiply, we use the algorithm we all learned in school: to multiply two numbers of at most k bits we need to do $k-1$ additions of number with at most $2k$ bits. This can be done in $O(k^2)$ time. So $n\log(n)$ can be calculated in $O(n + \log(n)^2) = O(n) \subseteq O(n\log(n))$. Similarly, to find n^2 we first count n and then multiply with itself. This is also $O(n) \subseteq O(n^2)$. To find the binary representation of $O(2^n)$ we change all the 1s to 0s and put a 1 in the end. Again this is $O(n) \subseteq O(2^n)$. □

[1] See the gap theorem [19, Theorem 7.3].

However, there is at least one important function that is not time constructible, $\log(n)$: the Turing machine does not have time to look at the entire input. More generally:

Proposition 7. *If $f(n)$ is a time constructible function then either $f(n) \in \Omega(n)$ or f is eventually constant.*

Proof. Assume that f is time constructible and M_f is a Turing machine that on input 1^n returns $f(n)$ in binary and that $t(n) \in O(f(n))$ is the running time. Let c and N be such that $t(n) \leq cf(n)$ for $n \geq N$. We assume that $f(n) \notin \Omega(n)$ and we will then prove that f is constant from some point. Since $f(n) \notin \Omega(n)$ we have

$$\forall d \in \mathbb{R}^+, N \in \mathbb{N} \exists n \geq N : f(n) < dn$$

We insert $d = \frac{1}{c}$ and N in this and get: $\exists n_0 \geq N : t(n_0) \leq cf(n_0) < n_0$. So on input 1^{n_0}, the machine does not have time to reach the end of the input. Thus it does not read the blank after 1^{n_0}, so it would do exactly the same, if there were more 1's in the input. This shows that for all $n > n_0$ we have $f(n) = f(n_0)$. □

We can use this proposition to get a new way of showing that functions are time constructible:

Proposition 8. *If $f(n)$ and $g(n)$ are both time constructible functions, then so is $f(g(n))$*

Proof. Since both f and g are weakly increasing, then so is $f(g(n))$. If one of them is constant from some point, then so is the composition, and we can calculate it in time $O(1)$, so it must be time constructible. Because of the above proposition, we can now assume that $f, g \in \Omega(n)$.

On input 1^n we can calculate $g(n)$ in $O(g(n))$, which we know is contained in $O(f(g(n)))$, as $f \in \Omega(n)$. From the binary representation of $g(n)$ we can get $1^{g(n)}$ in time $O(g(n))$. Finally we use the time constructability of f to get $f(g(n))$ in time $O(f(g(n)))$. □

It might look like all fast growing functions are time constructible, but this is not true: the functions

$$f(n) = \begin{cases} 2 & \text{, if } n = 0 \\ 2^n & \text{, otherwise} \end{cases}$$

and $g(n) = 2^n + I_A(n)$ where I_A is the indicator function of a hard-to-compute set are counterexamples. However, this is cheating. The only problem with the first is that it is decreasing from $n = 0$ to $n = 1$ and in the second, the difficult part of the function is negligible. The following proposition however, shows that there are genuine counterexamples. It follows from the gap theorem [19, Theorem 7.3] and the proof given here is similar to the proof of that theorem. Strictly speaking this is not a diagonalization proof, but the idea is similar.

Proposition 9. *For any computable function $f : \mathbb{N} \to \mathbb{N}$ there is a weakly increasing computable function $g(n) \geq f(n)$ such that no function in $\Theta(g(n))$ is time constructible.*

Proof. We define $g(n)$ for one value at a time, beginning with $g(0) = f(0)$. When $g(n-1)$ is defined, we define $g(n)$ as: the least integer $a \geq \max\{g(n-1), f(n)\}$ such that no Turing machine with description length at most n given input 1^n returns a number between $\frac{a}{n}$ and an in less than na steps.

This function is well-defined, as for any n there is only a finite number of Turing machines with description length at most n. It is even computable, because we can make a Turing machine that on input n calculates all values $g(i)$ with $0 \leq i \leq n$. For fixed i, it tries each a starting from $a = \max\{g(i-1), f(i)\}$ and simulate each of the Turing machines with description shorter than i on 1^i for ai steps and test if the output is between $\frac{a}{i}$ and ai.

However, this function is not time constructible: Assume for contradiction that M is a Turing machine that decides $h(n) \in \Theta(g(n))$ in time $t(n) \in O(h(n)) = O(g(n))$. Let N_t and c_t be such that $t(n) \leq c_t g(n)$ for all $n \geq N_t$, let N_h and c_h be such that $h(n) \leq c_h g(n)$ for $n \geq N_h$ and let N_g and c_g be such that $g(n) \leq c_g h(n)$ for $n \geq N_g$. Finally define $N = \max\{N_t, c_t, N_h, c_h, N_g, c_g, |\langle M \rangle|\}$, and consider what happens when we run $M(1^N)$. By definition it should return $h(N)$ and we see that $\frac{g(N)}{N} \leq h(N) \leq Ng(N)$ and that it terminates within $t(N) \leq Ng(N)$, but this contradicts the definition of g. □

Using the same technique we can even show something stronger: If f_1 and f_2 are two time constructible functions and $f_1(n) \leq f_2(n)$ and $f_1(n) \in o(f_2(n))$ we can find a function $f_1(n) \leq g(n) \leq f_2(n)$ such that $\Theta(g(n))$ does not contain any time constructible function.[2]

We will now use time constructible functions to state and prove a time hierarchy theorem. The approach used here, including the following definition and Lemma 10 is taken from [19, Section 7.2].

Definition 11. For any time constructible function $f(n) \leq n$ we define

$$A_f := \{\langle M, x\rangle | M \text{ accepts } x \text{ in at most } f(|x|) \text{ steps}\}$$

where M is a multi-tape Turing machine.

We will now use the language A_f to show that two complexity classes are different.

Lemma 10. *If $f(n) \geq n$ is time constructible, then $A_f \in TIME(f(n)^3)$.*

Proof. We will describe a Turing machine U_f that decides A_f in $O(f(n)^3)$ steps. First we mark the input length on a *clock tape* and calculate $f(|x|)$ on some tapes used only for this, and then write $f(|x|)$ on the clock tape. All this takes $O(f(n))$. We then want to simulate M so first we copy the description of M to a *description tape* and the start state of M to a *state tape*. Finally we have a *tape tape* that at any step in the simulation contains an encoding of what would have been on Ms tapes in one tape, like in the proof of Theorem 2. So we begin by writing $\triangleright x \triangleleft (\underline{\triangleright}'\triangleleft)^{k_M-1}\triangleleft$ on the tape tape, where k_M is the number of tapes in M. Unlike in the proof of Theorem 2, k_M is not fixed, so we use the description of M when we generate this string. We now simulate M similar to the way we did in the proof of Theorem 2. In each step the Turing machine moves its head on the description tape to its current state, goes over the entire tape tape to read what the heads read and find out what state to go to. It writes down this state on the state tape, and goes through the tape tape again to update it. Then it decreases the clock by one and is ready to go to the next step in the simulation. If the clock hits zero it rejects, if M terminates, U_f just returns what M returns.

The initialization of the clock tape takes $O(f(n))$ and the rest on the initialization can be done in $O(n)$. Let l_M be the length of the description of each symbol used by M. Now the space used by the tape tape is $O(l_M k_M f(|x|))$, as the space used by M is $O(k_M f(|x|))$. In the simulation of a single step it first needs to find the current state, which can be done in time linear in the length of the description of M. Then it goes over the entire tape tape, and find the rule of what to do next on the description tape. All this is in $O(n + l_M k_M f(n)) = O(l_M k_M f(n))$. Finally it goes through the tape tape again to update, and when it does this, it might need to extent some of the (representations of) tapes to the right. This can be done in $O(l_M k_M^2 f(|x|))$. Altogether the time used to simulate a single step is $O(l_M k_M^2 f(|x|))$. Both l_M and k_M are in $O(\log(|\langle M\rangle|))$, so the total time used by U_f is in $O(f(|x|)^2 \log(n)^3) \subseteq O(f(n)^3)$. □

Lemma 11. *Let $f(n) \geq n$ be a time constructble function and let $g : \mathbb{N} \to \mathbb{N}$ be a function with $\lim_{n\to\infty} \frac{g(n)}{f(\lfloor \frac{n}{2} \rfloor)} = 0$, where $\lfloor\cdot\rfloor$ denotes the floor function. Now $A_f \notin TIME(g(n))$.*

This is where we see the diagonalization: For each Turing machine M, we will find an input where it disagrees with A_f.

Proof. Assume for contradiction that there is a Turing machine M_g that decides A_f in time $t(n) \in O(g(n))$. Consider the Turing machine D_g that on input $\langle M\rangle 10^i$ simulates $M_g(\langle M, \langle M\rangle 10^i\rangle)$ and returns the opposite. This can be done in time $2n + t(2n + 1)$ and thus in $2t(2n + 1)$. This implies that D_g runs in $o(f(n))$ so for sufficiently large values of i we see that $D_g(\langle D_g\rangle 10^i)$ terminates within time $f(n)$.

Let us see if $D_g(\langle D_g\rangle 10^i)$ should return true or false? If it returns false, then $\langle D_g, \langle D_g\rangle 10^i\rangle \notin A_f$ by definition of A_f so $M_g(\langle D_g, \langle D_g\rangle 10^i\rangle)$ must return false. But then $D_g(\langle D_g\rangle 10^i)$ should return true, presenting a contradiction. If instead $D_g(\langle D_g\rangle 10^i)$ returns true we must have $\langle D_g, \langle D_g\rangle 10^i\rangle \in A_f$ as we know that it terminates within time $f(n)$. So then $M_g(\langle D_g, \langle D_g\rangle 10^i\rangle)$ returns true and $D_g(\langle D_g\rangle 10^i)$ returns false and again we have a contradiction. Hence, $A_f \notin \text{TIME}(g(n))$. □

These two lemmas together give the following.

[2] To show this, you just need to let the number of Turing machines considered and the "avoidance factor" grow slowly as a function of n.

Theorem 12 (Time hierarchy theorem (weak version[3]))**.** *If $f(n)$ is time constructible and $f(n) \geq n$ then TIME($f(n)$) $\subsetneq$ TIME($f(2n+1)^6$).*

Proof. Since $f(n)$ is time constructible, then so is $n \mapsto f(2n+1)^2$. Now we can define $h : \mathbb{N} \to \mathbb{N}$ by $h(n) = f(2n+1)^2$. Lemma 11 gives us $A_h \notin \mathrm{TIME}(f(n))$ but Lemma 10 tells us that $A_h \in \mathrm{TIME}(h(n)^3) = \mathrm{TIME}(f(2n+1)^6)$. □

This theorem can be used for separating some complexity classes. E.g.

Corollary 13. *$P \subsetneq E \subsetneq EXP$.*

Proof. Both $n \mapsto 2^n$ and $n \mapsto 2^{n^2}$ are time constructible so

$$\mathrm{P} \subseteq \mathrm{TIME}(2^n) \subsetneq \mathrm{TIME}((2^{2n+1})^6) = \mathrm{TIME}(2^{12n}) \subseteq \mathrm{E}$$
$$\mathrm{E} \subseteq \mathrm{TIME}(2^{n^2}) \subsetneq \mathrm{TIME}((2^{(2n+1)^2})^6) \subseteq \mathrm{TIME}(2^{25n^2}) \subseteq \mathrm{EXP}$$

□

2.2 Relativization

We have just seen that we can use diagonalization to separate complexity classes, so is there any hope that it could be used to separate P and NP? The answer seems to be no, because these proofs "relativize". To understand what this means, we need to define what an oracle Turing machine is. The definition of Turing machine varies from author to author, so naturally the definition of oracle Turing machine must vary too, but they are essentially the same [19, Definition 14.3][7, page 60-61][24, Definition 9.17].[4]

Definition 12. For $k \geq 2$ a k-tape oracle Turing machine is a 8-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_?, q_Y, q_N)$ where

- Q is the set of states,
- Σ is the input alphabet and does not contain $\sqcup$ or $\triangleright$,
- Γ is the tape alphabet $\Sigma \cup \{\sqcup, \triangleright\} \subseteq \Gamma$,
- $q_0, q_?, q_Y, q_N \in Q$
- δ is the transition function,

  $$\delta : (Q \setminus \{q_?\}) \times \Gamma^k \to ((Q \cup \{h, \text{“yes”}, \text{“no”}\}) \setminus \{q_Y, q_N\}) \times (\Gamma \times \{\leftarrow, \rightarrow, -\})^k$$

  and if at least one of the $g_i \in \Gamma$ is $\triangleright$ then

  $$\delta(q_1, g_1, g_2, \ldots g_k) = (q_2, g_1, d_1, g_2, d_2, \ldots g_k, d_k)$$

  where $d_i = \rightarrow$ if $g_i = \triangleright$ and $d_i = -$ otherwise.

The first tape is the *input tape* and the k'th tape is called the *oracle tape*.

When $A \subseteq \Sigma^*$ we define M^A as the machine that does the following: on input x the setup is exactly as for a Turing machine, and as long as the state is not $q_?$ it uses the transition function, like a usual Turing machine does. When in state $q_?$ it goes to state q_Y if the content of the oracle tape is a string in A and to q_N otherwise and then erases the oracle tape, all in one step.

A non-deterministic oracle Turing machine is just the obvious combination of oracle Turing machines and non-deterministic Turing machines. When in state $q_?$ the next step is deterministic, as the content of the oracle tape is either in A or not.

If $B \subseteq \Sigma^*$ is a language decided by an oracle Turing machine M^A that is deterministic and uses only polynomial time, we say that $B \in \mathrm{P}^A$. Similarly, if B is decided by a non-deterministic polynomial time Turing machine we say that $B \in \mathrm{NP}^A$.

[3] A stronger version says that if t_2 is time-constructible and $t_2(n) \geq t_1(n) \geq n$ and

$$\liminf_{n \to \infty} \frac{t_1(n) \log(t_1(n))}{t_2(n)} = 0$$

then $\mathrm{TIME}(t_1(n)) \subsetneq \mathrm{TIME}(t_2(n))$. In [7, Theorem 1.23] Du and Ko proofs essentially this statement, but there definitions are slightly different from the definitions used here.

[4] There are however definitions of weaker machines that use oracles [1, page 436-437][7, Section 4.6], but in both [1] and [7] an "oracle Turing machine" would refer to something essentially equivalent to the definition given here.

We will need the following simple proposition in a later chapter

Proposition 14. *Let $A, B \subseteq \Sigma^*$ be two languages with $|A \triangle B| < \infty$. Then $P^A = P^B$ and $NP^A = NP^B$.*

Proof. This is similar to Proposition 1. Given A as an oracle, we can decide B in linear time by a Turing machine that stores $A \triangle B$. A polynomial time computation can only quiry the oracle a polynomial number of times, so the polynomial time computation using B as an oracle can be simulated in polynomial time using A and an oracle. Thus $\mathrm{P}^B \subseteq \mathrm{P}^A$ and $\mathrm{NP}^B \subseteq \mathrm{NP}^A$. By symmetry, $\mathrm{P}^A \subseteq \mathrm{P}^B$ and $\mathrm{NP}^A \subseteq \mathrm{NP}^B$. □

The following is the first half of a theorem by Baker, Gill and Solovay[1].[5]

Theorem 15. *There exists an $A \subseteq \Sigma^*$ with $P^A = NP^A$. In particular, $P^{TQBF} = NP^{TQBF}$, where $TQBF$ is the language of true quantified Boolean formulas.*

Proof. Clearly, $\mathrm{P}^A \subseteq \mathrm{NP}^A$ for all A. We have

$$\mathrm{NP}^{TQBF} \subseteq \mathrm{NPSPACE} = \mathrm{PSPACE} \subseteq \mathrm{P}^{TQBF}$$

Here the first inclusion holds, because $TQBF \in \mathrm{NPSPACE}$[6], so if $L \in \mathrm{NP}^{TQBF}$ and M^{TQBF} is a non-deterministic oracle Turing machine that decides L, we can simulate M^{TQBF} by a non-deterministic Turing machine: Each time M^{TQBF} queries $TQBF$, we use the polynomial space algorithm to see if the string is in $TQBF$. We know that M^{TQBF} uses only polynomial time, so both the length and the number of the queries must be polynomial, so the simulation machine uses only polynomial space. Thus $L \in \mathrm{NPSPACE}$. From Savitch's Theorem [24, Theorem 8.5] we known that NPSPACE = PSPACE. Finally, we know[7] that $TQBF$ is PSPACE-complete, that is: any problem in PSPACE can be reduced to $TQBF$ in polynomial time. Thus $\mathrm{PSPACE} \subseteq \mathrm{P}^{TQBF}$. □

The existence of such an oracle A might suggest that P = NP. However, the other half of the theorem by Baker, Gill, and Solovay points in the other direction. First we need a lemma.

Lemma 16. *Let $L \in P^B$ for some language $B \subseteq \Sigma^*$. Then there exists an oracle Turing machine M such that M^B decides L and a polynomial p such that for all oracles $A \subseteq \Sigma^*$ the machine M^A on input x terminates in at most $p(|x|)$ steps.*

Proof. Let $L \in P^B$. We know there is a polynomial time oracle Turing machine N^B that decides L. Let p be a polynomial bound on the time used by N^B, such that $p(n)$ can be calculated in at most $p(n)$ steps. Now we get M by modifying N: we add a clock, such that M rejects, if it has not accepted in time $p(n)$. Now M^A always terminates in at most $p(n)$ steps and $L(M^B) = L(N^B) = L$, as N^B accepts in at most $p(n)$ steps. □

Now we are ready to prove the second half of the Baker-Gill-Solovay theorem [7, Theorem 4.19].

Theorem 17. *There exists a language B such that $P^B \neq NP^B$.*

Proof. For any language B we define

$$L_B = \{0^n | \exists x \in B, |x| = n\}$$

It is easy to see that $L_B \in \mathrm{NP}^B$: a non-deterministic Turing machine with B as oracle can on input 0^n just guess a string of length n and ask the oracle if it is in B. We want to find a B such that $L_B \notin \mathrm{P}^B$.

Consider the set of oracle Turing machines M that have a polynomial p such that $M^A(x)$ terminates within time $p(|x|)$ for all oracles A and input x. Let $M_1, M_2, \ldots$ be a sequence of all such Turing machines, and let $p_1, p_2, \ldots$ be the polynomials that bound their running time. By lemma 16 we only need to show that none of these Turing machines decide L_B to show that $L_B \notin \mathrm{P}^B$. We now define an increasing sequence $B_0, B_1, \ldots$ of oracles. We begin by setting $B_0 = \emptyset$ and in step i, starting from $i = 1$ we do the following:

[5] However, the proof here is from [24, Theorem 9.20].
[6] [24, Theorem 8.9].
[7] [24, Theorem 8.9].

First choose an n_i such that $2^{n_i} > p_i(n_i)$ and $n_i > 2^{n_{i-1}}$, and let $x_i = 0^{n_i}$. Then we simulate $M_i^{B_{i-1}}(x)$. If this accepts, we set $B_i = B_{i-1}$. If it rejects, it does so in at most $p_i(n_i) < 2^{n_i}$ steps, so we can find a string y_i if length n_i that M_i did not query. Now define $B_i = B_{i-1} \cup \{y_i\}$.

Define $B = \bigcup_{i=1}^{\infty} B_i$. As $n_i > 2^{n_{i-1}}$ the sequence n_i is increasing, so by induction we see that the strings in B_i all have length at most n_i. Furthermore we have $2^{n_i} > p_i(n_i)$ for all i so for $j > i$ we see that $n_j > 2^{n_{j-1}} \geq 2^{n_i} > p_i(n_i)$, so M_i running on x_i does not have time to ask if a string for length n_j is in the oracle. But we can get B from B_i by adding strings y_j of length n_j with $j > i$ so $M_i^{B_i}(x_i)$ and $M_i^B(x_i)$ gives the same result. If $M_i^{B_{i-1}}(x_i)$ accepts we defined $B_i = B_{i-1}$ so $M_i^{B_i}(x_i)$ also accepts, but in this case B_i, and thus B, does not contain a string of length n_i, so M_i^B does not decide L_B. If $M_i^{B_{i-1}}(x_i)$ rejects we defined $B_i = B_{i-1} \cup \{y_i\}$ and since $M_i^{B_{i-1}}(x_i)$ did not query y neither does $M_i^{B_i}(x_i)$ and so this also rejects. But in this case B_i and hence B does contain a string y_i of length n_i, so again we see that M_i^B does not decide L_B, and we are done. □

Remark 1. The above two theorems are very important, because they rule out some ways of deciding the P vs. NP problem. Intuitively, you can even think of it as an independence result: if all the "axioms" about computations you are using are on the form "this can be computed quickly on a Turing machine" you cannot hope to prove P = NP or P ≠ NP. You will have to use propositions on the form "this cannot be computed quickly on a Turing machine" too. Otherwise you would also have proven that $\mathrm{P}^B = \mathrm{NP}^B$ and $\mathrm{P}^A \neq \mathrm{NP}^A$ of all oracles A and B, contradicting the above.

This seems to rule out the possibility of proving P ≠ NP using diagonalization: in the proof of Lemma 11, that states that $A_f \notin \mathrm{TIME}(g(n))$ under some assumptions about f and g, we assume that there exists a Turing machine M_g that decides A_f in time $O(g(n))$ and use this to define a new Turing machine, that on input x simulates M_g on some input y, that depends on x. But if we define

$$A_f^A = \{\langle M, x\rangle | M^A \text{ accepts } x \text{ in at most } f(|x|) \text{ steps}\}$$

where A is an oracle and M is an oracle Turing machine, then we could use the proof of Lemma 11 to show that $A_f^A \notin \mathrm{TIME}(g(n))^A$. We could do the same for Lemma 10 and thus we can prove the time hierarchy theorem *relatively* to any oracle, by just writing M^A instead of M for all Turing machines M. When this is the case, we say that the proof method *relativizes* [7, Section 4.3]. This tells us that any proof that P ≠ NP would have to use a different method than the one we use to prove Lemma 10 and 11. However, we do not have a definition of "diagonal proof", so we cannot prove that any diagonal proofs relativize. It might be that more complicated forms of diagonalization could be used to prove P ≠ NP and Kozen has even claimed that any proof of P ≠ NP must be a diagonal proof [8, Section 2.4].

Remark 2. From the above proofs it seems to be much easier to find an oracle A with $\mathrm{P}^A = \mathrm{NP}^A$ than to find an oracle B with $\mathrm{P}^B \neq \mathrm{NP}^B$, but if you include the proof that $TQBF$ is PSPACE-complete and the proof that NPSPACE = PSPACE, the proof of the existence of A is much longer. It also turns out that $\mathrm{P}^C \neq \mathrm{NP}^C$ for "almost all" oracles C. To make this statement well-defined, we identify each oracle A with its characteristic sequence: $I_A(\epsilon)I_A(0)I_A(1)I_A(00)\ldots$, where

$$I_A(x) = \begin{cases} 1 & \text{, if } x \in A \\ 0 & \text{, otherwise} \end{cases}$$

Except from the countable set of sequences that ends with an infinite string of 1's, these are in an obvious one-to-one correspondence with the set of real numbers in $[0, 1)$, so now we can indentify A with $0.I_A(\epsilon)I_A(0)I_A(1)I_A(00)\ldots$ in binary. Then the Lebesgue measure on the real numbers gives us a measure μ on the set $\mathcal{C}$ of oracles. For this measure we have.

Theorem 18. *Let $\mathcal{B} \subseteq \mathcal{C}$ be the set of oracles B such that $P^B \neq NP^B$. Then $\mu(B) = 1$.*

Proof. Sketch: For an oracle B we define

$$L_B = \{0^n | (\exists x, |x| = n), x1, x1^2, \ldots x1^n \in B\}$$

We see that $L_B \in \mathrm{NP}^B$ and we want to show that the probability that $L_B \in \mathrm{P}^B$ is 0. As there are only countably many Turing machines, it is enough to show that for a fixed oracle Turing machine running in polynomial time, the probability that it decides L_B is zero. However, a fixed Turing machine can on input 0^n, only try polynomially many strings x of length n and the number of x's increases exponentially and the probability that a fixed x works decreases exponentially. Thus the

probability that M finds a x that work in polynomial time goes to zero, while the probability that such an x exists stays above 0, so M will have to guess and will have positive probability of guessing wrong. Moreover, the correctness of its guess on 0^{n_1}, $0^{n_2}, \ldots 0^{n_k}$ will be almost independent of the correctness of its guess on $0^{n_{k+1}}$ when n_{k+1} is much larger than $n_1, n_2 \ldots n_k$. So we can get the probability that M guess correct below ϵ for any $\epsilon > 0$ by considering sufficiently many inputs on the form 0^n. A formal proof can be found in [7, Theorem 4.24]. □

This was proven by Bennett and Gill [2], who also conjectured that whenever two complexity classes were different relative to most oracles, then they were different. However, this *random oracle hypothesis* turned out to be false [3].

Remark 3. In the proof of the existence of the oracle B such that $\mathrm{P}^B \neq \mathrm{NP}^B$ we are using that the non-deterministic Turing machine can query exponentially many strings, because it can branch into exponentially many computational paths. It would seem fairer, if we only allowed the non-deterministic Turing machine to make polynomially many queries in total. Let us denote the class of languages decidable by such a Turing machine by NP_b^B. With this restriction we have:

Theorem 19. *$P = NP$ if and only if $\forall A : P^A = NP_b^A$.*

Proof. The 'if'-part is easy, because $\mathrm{NP}_b^\emptyset = \mathrm{NP}$. The 'only if'-part is proven in [7, Theorem 4.22]. □

Chapter 3

Natural Proofs

Until now we have only considered one model of computation - the Turing machine. Another important model is Boolean circuits [7, Chapter 6][19, Section 11.4].

3.1 Boolean circuits

Definition 13. A *Boolean circuit* is a finite directed acyclic graph with labeled nodes such that

- Nodes with no incoming edges are labeled by a variable, x_i, or a Boolean constant, 0 or 1.
- Nodes with incoming edges are labeled either "$\neg$" (negation), "$\wedge$" (and), or "$\vee$" (or).
- Nodes labeled "$\neg$" have only one incoming edge.

The nodes with no outgoing edges are the *output nodes*, and these are ordered.[1] Nodes labeled with variables are called *input edges*, nodes labeled "$\neg$", "$\wedge$", or "$\vee$" are called *logical gates*. For a logical gate, the number of incoming edges is the *fanin* and the number of outgoing edges is the *fanout*. Both of these are unbounded in general. A computation is made by assigning values 0 and 1 to the input variables and then recursively assigning 0s and 1s to the logical gates in the obvious way. The *size* of a Boolean circuit is the number of logical gates in it and the *depth* is the length of a longest path from a variable to an output gate.

We say that a language $L \subseteq \{0,1\}^*$ is decided by a *family of circuits* $\mathcal{C} = \{C_0, C_1, \dots\}$ if C_i is a circuit with i input nodes and $C_{|x|}(x)$ returns 1 whenever $x \in L$ and 0 otherwise. The *circuit complexity*[2] of L is the function $f : \mathbb{N} \to \mathbb{N}$ such that $f(i)$ is the size of the smallest circuit C_i where $C_i(x) = 1 \Leftrightarrow x \in L$ for all x of length i. We say that L has polynomial circuit complexity if f is bounded above by a polynomial.

This model is in some ways simpler than the Turing machine: in the Turing machine we can have an unbounded number of states, and this makes it difficult to prove that some function cannot be calculated fast. In the Boolean circuit model, we only use three kinds of computation, so it is easier to use "combinatorial methods" to show lower bounds on the complexity of languages. Furthermore, there are relations between the two models, so a result about circuit complexity can then be translated to a result about Turing machines. There are however important differences between Turing machines and families of circuits:

Proposition 20. *Any language $L \subseteq \{0,1\}^*$ is decided by a family of circuits.*

Proof. Let $L \subseteq \{0,1\}^*$. To show that L is decided by a family of circuits, it is enough to construct for any fixed n, a circuit C_n with n variables such that $C_n(x) = 1 \Leftrightarrow x \in L$ for all $x \in \{0,1\}^n$: For each $x \in L$ of length n we make a $\wedge$-node called "x" that is 1 if and only if the input is x. We do this by creating an edge from the input variable x_i to the node x if the i'th bit in x is 1, and from $\overline{x_i}$ to x if it is 0. Here $\overline{x_i}$ is a negation node with an incoming edge from x_i. When this is done, we let the output node be a $\vee$-node with input from all nodes "x" with $x \in L$. □

[1] Often there is only one output node. This corresponds to Turing machines that either accept or reject. Sometimes we need to compute a function that takes values in $\{0,1\}^n$ and then we need n ordered output nodes.

[2] Some authors only allow fanin to be at most 2 in the definition of circuit complexity and circuit size [7, p. 196 and 198]. The set of languages with polynomial circuit complexity is the same in both definitions [7, Proposition 6.3].

By letting L be an undecidable set where the proposition "$x \in L$" only depends on $|x|$, we have an undecidable set with circuit complexity $f(n) = 0$! In particular, there are languages with polynomial circuit complexity, that are not in P. It might seem strange at first, that undecideable languages can be decided by a family of circuits. This fact is due to an important difference between the Turning machines and circuit families: a Turing machine has the same states and transition function on any input, but a circuit family has a circuit for each input length. In some cases it is useful to consider only uniform circuit families. A *uniform* circuit family is a circuit family $\{C_0, C_1, \dots\}$ such that there is a log-space Turing machine M that on input 1^n returns a describtion of C_n [19, p. 269]. However, in this document we will not restrict to uniform circuits.

Although not every language decided by a polynomial size circuit family is in P, circuits might still be used to prove P $\neq$ NP, because we have an implication the other way. The following proof is standard and can be found in [7, Theorem 6.4]:

Theorem 21. *Any language $L \in P$ over the alphabet $\{0, 1\}$ has polynomial circuit complexity.*

Proof. Let $L \in$ P be such a language. We know from Theorem 2 that L can be decided on a single-tape Turing machine M in polynomial time $p(n)$. We can assume that $\{0, 1\}$ is the input alphabet of M, and we will then construct a polynomial size family of circuits that decides the same language.

To prove this theorem we need to use configurations. The *configuration* of a single-tape Turing machine at some point in the calculation is the concatenation of

1. The string of letters strictly before the head,
2. The state of the Turing machine or "yes", "no", or "h", if it has halted
3. The string of letters from the head and onwards, until and including position $p(n) + 1$.

Here we assume that the set of letters and the set of states are disjoint. The evaluation of M on an input x, with $|x| = n$ consists of $p(n) + 1$ configurations each containing $p(n) + 1$ letters, and each of these letters is either an element in Γ or in $Q \cup \{\text{"yes"}, \text{"no"}, h\}$. We will now construct a circuit that uses these configurations to simulate the single-tape Turing machine: we create a node for each element in $(\Gamma \cup Q \cup \{\text{"yes"}, \text{"no"}, h\}) \times \{0, 1, \dots p(n)\} \times \{1, 2, \dots p(n) + 1\}$ and we want the node (g, t, m) to be 1 if and only if the m'th letter in the configuration after step t is g. The nodes $(\text{"1"}, 0, m)$ with $2 \leq m \leq n + 1$ are then the input nodes and $(\text{"0"}, 0, m)$ for $2 \leq m \leq n + 1$ are the negation of $(\text{"1"}, 0, m)$. For $2 \leq m \leq n + 1$ and g not "0" or "1" we set $(g, 0, m) = 0$. Similarly $(g, 0, 1)$ is 1 if $g = \triangleright$ and 0 otherwise, $(g, 0, 0)$ is 1 if g is the start state and 0 otherwise, and $(g, 0, m)$ with $m > n + 1$ is 1 if $g = \sqcup$ and 0 otherwise.

For $t > 0$, the node (g, t, m) is a logical gate: the m'th letter, $0 < m < p(n)+1$ in the configuration at time t is a function of the $m - 1$'th, the m'th, and the $m + 1$'th letter at time $t - 1$. Thus for each $g \in \Gamma \cup Q \cup \{\text{"yes"}, \text{"no"}, h\}$ there is a circuit that determines (g, t, m) given $(h, t - 1, m - 1), (h, t - 1, m), (h, t - 1, m + 1)$ for all $h \in \Gamma \cup Q \cup \{\text{"yes"}, \text{"no"}, h\}$. Similarly for $m = 0$ and $m = p(n) + 1$. For a fixed Turing machine, we only need finitely many different kinds of such circuits, one for each $g \in \Gamma \cup Q \cup \{\text{"yes"}, \text{"no"}, h\}$, so the size of all these circuits is bounded by a constant k. Thus we can represent the entire computation history of $M(x)$ by a $O(k \cdot p(n)^2)$ sized circuit. We then create a $\vee$-node with incoming edges from all the nodes $(\text{"}yes\text{"}, p(n), m)$, so this node is 1 if and only if $M(x)$ accepts. Finally we recursively remove all other nodes with no outgoing edges. This gives us a family of Boolean circuits of size $O(p(n)^2)$ that decides L. □

The set of languages with polynomial circuit complexity is called[3] P/*poly*. So the above theorem says that P $\subseteq$ P/*poly*. To show that P $\neq$ NP it would be enough to show that a problem in NP is not in P/*poly*. It could be that P $\subsetneq$ NP $\subsetneq$ P/*poly* and then this line of attack would not work, but some results suggest that this is unlikely [7, Theorem 6.11][4].

[3] This name is due to a different characterization of the set: it is the set of languages decidable by a Turing machine with polynomial advice [7, Theorem 6.8]. That is: a language L is in P/*poly* if and only if there is a function $h : \mathbb{N} \to \{0, 1\}^*$ where $|h(n)|$ is polynomial in n and a language $L' \in$ P such that $x \in L \Leftrightarrow (x, h(|x|)) \in L'$

[4] If NP $\subseteq$ P/*poly* then "the polynomial hierarchy collapses to the second level", that is: an *alternating Turing machine* is a non-deterministic Turing machine, where each state is either labeled $\wedge$ or $\vee$. Acceptance can now be defined as a game: each time the Turing machine is in a $\vee$ state you decide which of the posible steps it should take, and each time it is in a $\wedge$ state, your opponent decides. If you can make the machine end in an accept state, the computation accepts. We then define Σ_i^{P} respectively Π_i^{P} to be the class of languages decidable by a polynomial time alternating Turing machine, where each computation branch contains at most i runs of $\wedge$ and $\vee$ states, starting with a $\vee$ respectively $\wedge$ state. Now the polynomial hierarchy is $\mathrm{PH} = \cup_{i \in \mathbb{N}} \Sigma_i^{\mathrm{P}} = \cup_{i \in \mathbb{N}} \Pi_i^{\mathrm{P}}$. That the hierachy collapses to the second level means that $\mathrm{PH} = \Sigma_2^{\mathrm{P}}$. There is more about the polynomial hierarchy and alternating Turing machines in [7, Chapter 3].

The following is standard [7, Section 6.5].

Definition 14. For $i \in \mathbb{N}$ we define *nonuniform-AC^i* as the class of languages decideable by a family of circuits of polynomial size and $O(\log(n)^i)$ depth, and *nonuniform-AC* as $\bigcup_{i\in\mathbb{N}}$ *nonuniform*-AC^i. In particular, *nonuniform-AC^0* is that class of languages decideable by a family of constant depth polynomial size circuits.

3.2 PARITY

We are a long way from being able to prove things like "3-SAT does not have polynomial circuit complexity". It seems to be easier to prove non-trivial lower bounds if we consider simpler languages and smaller bounds on the size of the circuits. To see an example of a lower bound on circuit complexity, we define the following simpler problem as in [9].

Definition 15. The language $PARITY \subseteq \{0,1\}^*$ is the language of strings containing an odd number of 1's. We define $p_n : \{0,1\}^n \to \{0,1\}$ by $x_1 + x_2 + \cdots + x_n \pmod 2$ and we say that the functions p_n and $1 - p_n$ are *n-parity functions*. A circuit that computes an n-parity function is call an *n-parity circuit*.

This language is obviously in P: it can even be decided by a two state single-tape Turing machine in linear time. If we use the idea from Proposition 20 we get a circuit family deciding $PARITY$ with size $O(n2^n)$ and depth 3. If we instead use the idea from Theorem 21 we get a family with size $O(n^2)$ and depth $O(n)$. We can do better than that by using divide and conquer: the parity of the string $x_1x_2\ldots x_n$ is just $x_1 + x_2 + \cdots + x_n \pmod 2$, and now

$$x_1 + \ldots x_{\lfloor n/2\rfloor} + x_{\lfloor n/2\rfloor+1} + \cdots + x_n = (x_1 + \cdots + x_{\lfloor \frac{n}{2}\rfloor}) + (x_{\lfloor \frac{n}{2}\rfloor+1} + \cdots + x_n) \pmod 2$$

So to make a circuit that solves the $PARITY$ problem for n bit, we only need two copies of a circuit that solve it for $\lfloor \frac{n}{2}\rfloor + 1$ and a circuit solving it for 2. This idea gives us a circuit family of size $O(n)$ and depth $O(\log(n))$, so $PARITY \in$ nonuniform-AC^1.

Still there is a trade-off: either we can use a constant depth but exponential size circuit family or we can use a family of linear size but unbounded depth. In the following, we will see that $PARITY \notin$ nonuniform-AC^0, that is: any circuit family that solves $PARITY$ in polynomial size must have unbounded depth. To show this, we consider a modified version of the circuit model. The following definition very similar[5] to that in [9] and to distinguish this from usual circuits, we use the term from [7, p. 222].

Definition 16. A 0-circuit is a *literal*, that is x_j or $\overline{x_j}$ for some j. A i-circuit is a non-empty finite set $C = \{C_1, C_2, \ldots C_k\}$ of $i-1$-circuits. The value v_C of a i-circuit is then defined as

$$v_C = \begin{cases} \bigwedge_{j=1}^{k} v_{C_j} & i \text{ is odd} \\ \bigvee_{j=1}^{k} v_{C_j} & i \text{ is even} \end{cases}$$

An i-circuit is called a *$\wedge$-circuit* if i is odd, and a *$\vee$-circuit* if i is even. A *levelable circuit* is any i-circuit, $i \in \mathbb{N}$, or a constant circuit, 0 or 1.

We say that a levelable circuit B *belongs to* C if there is a (possibly empty) sequence $B_1, \ldots, B_n$ such that $B \in B_1 \in \ldots B_n \in C$ or if $B = C$. The *depth* of a i-circuit C is i and its *size* is the number of levelable circuits of depth > 0 belonging to it.

The levelable circuits differ in two ways from the usual circuits: they do not contain any negations, but "have all the negations at the bottom" and they have levels. However, we can always translate from one type to the other.

Proposition 22. *For any levelable circuit with n variables of size s and depth d, there is a (usual) circuit with size at most $s + n$ and depth at most $d + 1$ that compute the same function. For any (usual) circuit with size s and depth d, there is a levelable circuit with size at most $2sd$ and depth at most $d + 1$.*

[5] The differences are that $\vee$ and $\wedge$ are interchanged and that "size" does not include the 0-circuits, but does include C itself.

Proof. From the levelable circuit, we get a circuit by adding the node $\overline{x_i}$ for each i. This costs us at most n in size and 1 in depth.

Given a circuit C we want to construct an equivalent levelable circuit. First we move all the negation to the bottom of the circuit, in the following way. We make a negated version $\overline{C}$ of C by substituting each input node x_i by $\overline{x_i}$ and each $\wedge$-node by a $\vee$-node and vice versa. Now the value at a node in C will always be the negation of the corresponding node in $\overline{C}$. Thus, instead using negation nodes, we can negate by creating edges between C and $\overline{C}$. All this gives at most double size and at most the same depth.

Next we will "level" the circuit. To order the nodes in levels, we put all the input nodes in level 0 and for all other nodes, we find the length of the longest path from this node to the output node. If this length is i, we put the node in level $d-i$. We then have levels from 0 to d and each node will only have incoming edges from lower levels and outgoing edges to higher levels, but edges might go more than one level up and there might be $\wedge$- and $\vee$-nodes in the same level. We now divide each level > 0 in a $\wedge$-part and a $\vee$-part and order the levels as: $1\wedge$, $1\vee$, $2\vee$, $2\wedge$, ... This will end in $d\wedge$ if d is even and in $d\vee$ if d is odd. Now the point is that we can combine two consecutive levels if they are of the same kind, e.g. $1\vee$ and $2\vee$. The only problem in this would be if there is an edge from $u \in 1\vee$ to $v \in 2\vee$. However, in this case we can simply remove this edge and add an edge from w to v for each w that has an edge to u. This way we end up with $d+1$ level, but we might still have edges going more than one level up. To avoid this, we use $\vee$-nodes with only one input as "wires". We need at most one "wire" for each gate at each level, and none at the level of the gate itself, so we need at most $2ds$ gates. □

From the above theorem we see that changing "circuits" to "levelable circuits" in the defintion of nonuniform-AC^i and nonuniform-AC would give an equivalent definition. With levelable circuits, the approach from Proposition 20 gives a family with depth 2 and exponential size. With depth 2 we cannot do any better [7, Lemma 6.35]:

Lemma 23. *Any levelable circuit that computes an n-parity function and has depth 2 contains at least $2^{n-1}+1$ gates.*

Proof. A depth 2 levelable circuit has a $\vee$-node on top and $\wedge$-nodes in the lowest level. So if one of the $\wedge$'s gives 1, the levelable circuit returns 1. But changing just one bit must always change the value of a parity function, so each of the $\wedge$-nodes, that are true for some input, must contain either x_i or $\overline{x_i}$ for each i. So each $\wedge$-node is true on at most one input, but a parity function is true on 2^{n-1} inputs. Thus the levelable circuit must have at least 2^{n-1} nodes on the lowest level and one on level 2. □

To prove that $PARITY \notin$ nonuniform-AC, we need the following definition from $[9, p.18]$:

Definition 17. A *restriction* is a function $\rho : \{x_1, x_2, \ldots x_n\} \to \{0, 1, *\}$. If k is the number of i's with $\rho(x_i) = *$ and $f : \{0,1\}^n \to \{0,1\}$ is a function, ρ *induces* a function $f^\rho : \{0,1\}^k \to \{0,1\}$: if $y \in \{0,1\}^k$ then $f^\rho(y) = f(y^\rho)$, where y^ρ is $\rho(x_1)\rho(x_2)\ldots\rho(x_n)$ with the i'th $*$ replaced by the i'th letter in y.

Similarly, for a levelable circuit C we can use ρ to substitute some of the literals x_i and $\overline{x_i}$ with constants in the obvious way. Now some of the levelable circuits in C might be *forced*, that is, constant. If the entire levelable circuit C is forced C^ρ denote the constant circuit with this value. If C is not forced, C^ρ is the levelable circuit you get by removing all forced levelable circuits in C.

We see that if C computes f then C^ρ computes f^ρ and that the restriction of a parity function is itself a parity function [9, Lemma 3.1 and 3.2]. The following theorem and proof is from [9, Theorem 3.3].

Theorem 24. *No family of levelable circuits that has constant depth and polynomial size can compute $PARITY$.*

Proof. Assume for contradiction that there exists a constant depth d and an integer k such that for each $n \geq 2$ there is a depth d levelable parity circuit of size at most n^k. Let d be the smallest value for which there exists such a k. We know from Lemma 23 that $d \geq 3$. Now the idea is to use the distributive law to switch the two lowest levels, so we end up with both level 2 and 3 consisting of $\vee$-nodes and combine these two levels, thereby finding a family of polynomial size levelable circuits with depth $d-1$ that decides the parity problem to reach a contradiction. However, the use of the

distributive law will generally increase the size of the circuit exponentially, so first we will obtain levelable circuits with bounded fanin in the two lower levels. We divide the proof into three steps.

Step 1: Let k be an integer such that for each $n \geq 2$ there exists a depth d levelable parity circuit C_n of size at most n^k. We want to show that for sufficiently large n there is a restriction ρ such that

1. C_n^ρ computes a m-parity function for some $m \geq \frac{\sqrt{n}}{2}$,
2. All the 1-circuits in C_n^ρ have fanin $\leq 6k$.

To prove this, we simply pick a random restriction, by randomly and independently picking the value of each $\rho(x_i)$ such that

$$\begin{aligned}\Pr\left(\rho(x_i) = 0\right) &= \frac{1 - \frac{1}{\sqrt{n}}}{2} \\ \Pr\left(\rho(x_i) = 1\right) &= \frac{1 - \frac{1}{\sqrt{n}}}{2} \\ \Pr\left(\rho(x_i) = *\right) &= \frac{1}{\sqrt{n}}\end{aligned}$$

Let us bound the probability that a given 1-circuit in C_n^ρ has fanin $\leq 6k$. Each 1-circuit in C_n^ρ is the restriction B^ρ of some 1-circuit B in C_n, so consider a 1-circuit B in C_n. If the size of B is $\geq 6k \log n$ we say the B is *wide*, otherwise it is *narrow*. We split into two cases:

B **is wide:** If B^ρ is forced, then it is not part of C_n^ρ and thus not a problem.

$$\begin{aligned}\Pr(B \text{ is not forced}) &= \Pr(\text{no member of } B \text{ is assigned } 0) \\ &\leq \left(\frac{1 + \frac{1}{\sqrt{n}}}{2}\right)^{6k \log n} \\ &\leq \left(\frac{3}{4}\right)^{6k \log n}, \text{ for } n \geq 4 \\ &\leq \left(\frac{3}{4}\right)^{6k \log_2 \frac{n}{2}} \\ &= \left(\frac{n}{2}\right)^{6k \log_2(3/4)}\end{aligned}$$

which is in $o(n^{-2k})$ as $\log_2(3/4) \leq -\frac{1}{3}$.

B **is narrow:** In this case we bound the probability that B^ρ is too large:

$$\begin{aligned}\Pr(B \text{ size of } B^\rho > 6k) &= \Pr(B \text{ contains } > 6k \text{ input } x_i \text{ with } \rho(x_i) = *) \\ &= \sum_{j=6k+1}^{6k \log(n)} \binom{6k \log n}{j} \left(\frac{1}{\sqrt{n}}\right)^j \left(1 - \frac{1}{\sqrt{n}}\right)^{6k \log(n) - j} \\ &\leq (6k \log(n) - 6k) \binom{6k \log n}{6k} \left(\frac{1}{\sqrt{n}}\right)^{6k} \\ &\leq (6k \log(n))^{6k+1} n^{-3k}\end{aligned}$$

which is again in $o(n^{-2k})$.

The size of C_n is at most n^k, so there are at most n^k 1-circuits in C. Thus the probability that requirement 2 is violated is bounded by $n^k \cdot o(n^{-2k}) \subseteq o(1)$. Let X denote the number of variables that ρ assigns an $*$. Then X follows the binomial distribution with parameters n and $p = \frac{1}{\sqrt{n}}$, so its mean is $\mu = \sqrt{n}$ and its variance is

$$\sigma^2 = np(1-p) = n\frac{1}{\sqrt{n}}(1 - \sqrt{n}) \leq \sqrt{n}$$

Now we use Chebyshev's inequality, $\Pr(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2}$ for all $k > 0$. Setting $k = \frac{\sqrt[4]{n}}{2}$ we get:

$$\Pr\left(X < \frac{\sqrt{n}}{2}\right) \leq \Pr\left(|X - \sqrt{n}| \geq \frac{\sqrt[4]{n}}{2}\sigma\right) \leq \left(\frac{2}{\sqrt[4]{n}}\right)^2 = \frac{4}{\sqrt{n}} \in o(1)$$

So the probability that requirement 1 fails is also in $o(1)$. In particular, for sufficiently large n there exists a ρ such that both requirements are fulfilled.

Now we can obtain a polynomially sized family $D_1, D_2, \ldots, D_n, \ldots$ of levelable circuits with depth d and with all 1-circuits' fanin bounded by a number c independent of n: to get D_i, we take C_{4i^2}, and a ρ satisfying the above requirements. Now $C^\rho_{4i^2}$ has depth d, size at most $(4i^2)^k$, the 1-circuits have fanin at most $c = 6k$, and it computes an m-parity function for some $m \geq \frac{\sqrt{4i^2}}{2} = i$. By fixing $m - i$ variables we get D_i.

Step 2: Let $D_1, D_2, \ldots$ be a family of levelable circuits of size at most n^k and depth at most d and with all 1-circuits' fanin bounded by c. In this step we want to bound the size of the 2-circuits, again by choosing a random restriction from the same distribution as in step 1. For a constant b_c we say that a restriction ρ *fails* if:

1. D^ρ_n contains fewer than $\frac{\sqrt{n}}{2}$ unassigned variables, or
2. There is a 2-circuit B^ρ of D^ρ that depends on $\geq b_c$ inputs.

Requirement 1 is true with probability $o(1)$ as in step 1. We will now use induction to show the following:

Claim: For every c there is a constant b_c such that for any 2-circuit A that only contains 1-circuits of size $\leq c$ we have $\Pr(A^\rho \text{ depends on } > b_c \text{ inputs}) \in o(n^{-k})$.

Proof of claim: By induction on c. For $c = 1$ the 1-circuits are just "wires" for the variables, so we can use the same argument as in step 1. The only difference is that a 1-circuit is a $\wedge$-circuit and thus forced if one of the inputs is 0, while a 2-circuit is a $\vee$-circuit, and forced if one of the inputs is 1.

Assume that the claim is proved for $c - 1$. Now we say that A is wide if it has at least $k4^c \log(n)$ disjoint 1-circuits, and otherwise it is narrow. We split into two cases:

A **is wide:** In this case A is likely to be forced. Let S be a set of $k4^c \log(n)$ disjoint 1-circuits of A. For $n \geq 9$ we have:

$$\begin{aligned}
\Pr(A \text{ is not forced}) =& \Pr(\text{no member of } A \text{ is forced to be 1}) \\
\leq& \prod_{B \in S} \Pr(B \text{ is not forced to be 1}) \\
\leq& (1 - 3^{-c})^{k4^c \log(n)} \\
\leq& (1 - 3^{-c})^{k4^c \log_2\left(\frac{n}{2}\right)} \\
=& \left(\frac{n}{2}\right)^{k4^c \log_2(1-3^{-c})} \\
\leq& \left(\frac{n}{2}\right)^{-k4^c 3^{-c}}
\end{aligned}$$

which is in $o(n^{-k})$.

A **is narrow:** Let S be a maximal set of disjoint 1-circuits of A and let H be the set of inputs appearing in S. As A is narrow we have $|S| \leq k4^c \log(n)$ and thus $|H| \leq ck4^c \log(n)$. Let h be the number of inputs in H that ρ assigns to $*$, and let $\rho_1, \rho_2, \ldots \rho_{2^h}$ be the 2^h different restrictions obtainable from ρ by changing these $*$s to 0s and 1s. If A^ρ depends on an input x_i then either x_i is among the h $*$'d inputs in H or A^{ρ_j} depends on x_i for some j, so if we can show that both h and the number of inputs A^{ρ_j} depends on are often small, we are done. First we bound h:

$$\begin{aligned}
\Pr(h > 4k) =& \Pr(\text{more than } 4k \text{ of the inputs in } H \text{ is assigned a } *) \\
\leq& \sum_{j=4k+1}^{ck4^c \log(n)} \binom{ck4^c \log(n)}{j} \left(\frac{1}{\sqrt{n}}\right)^j \left(1 - \frac{1}{\sqrt{n}}\right)^{ck4^c \log(n) - j} \\
\leq& ck4^c \log(n) \binom{ck4^c \log(n)}{4k} \left(\frac{1}{\sqrt{n}}\right)^{4k} \\
\leq& (ck4^c \log(n))^{4k+1} n^{-2k}
\end{aligned}$$

which is in $o(n^{-k})$.

As S was chosen to be maximal, any 1-circuit B in A contains an input from H. Thus for any j, any 1-circuit in A^{ρ_j} depends on at most $c - 1$ inputs and by the induction hypothesis, the probability

that A^{ρ_j} depends on more than b_{c-1} inputs is in $o(n^{-k})$. Set $b_c = 4k + 2^{4k} b_{c-1}$. If A^ρ then depends on more than b_c inputs, then either $h > 4k$ or some A^{ρ_j} depends on more than b_{c-1} input. Thus:

$$\begin{aligned}\Pr(A \text{ depends on more than } b_c \text{ input}) \leq & o(n^{-k}) + 2^{4k} o(n^{-k}) \\ \leq & o(n^{-k})\end{aligned}$$

The claim follows by induction. From this we obtain a restriction ρ such that

1. D^ρ has $m > \frac{\sqrt{n}}{2}$ inputs
2. Any 2-circuit in D^ρ depends on at most b_c inputs.

Any 2-circuit that depends on b_c inputs, can be computed in bounded size. Using the same argument as in the end of step 1, this gives us a family $E_1, E_2, \ldots$ of polynomial size d-depth levelable circuits that compute parity functions and has bounded sized 2-circuits.

Step 3: Any 2-circuit is a $\vee$ of a set of $\wedge$'s, and using the distributive law we can convert this to a $\wedge$ of $\vee$'s. We do this to all the 2-circuits of the E_ns: The size of the 2-circuits of the E_ns are bounded, and so is the size of the resulting "up side down" 2-circuits. We then merge the new level 2 and 3 as we did in the proof of Proposition 22. A levelable circuit has $\wedge$'s in the lowest level, so we have to exchange the $\vee$'s with the $\wedge$'s and at the same time we exchange the x_i's with the $\overline{x_i}$'s. This gives us a circuit computing the negation of a parity function, which is itself a parity function. We now have a family $F_1, F_2, \ldots$ of polynomial size, $d-1$ depth levelable parity circuits, contradicting the minimality of d. □

Parity functions can be characterized as the functions with the following property: for any $x \in \{0,1\}^*$ and any position in the sting, changing the bit at that position changes the function value on the string. We can generalize this:

Definition 18. We say that a function $f : \{0,1\}^n \to \{0,1\}$ is s-sensitive if for any restriction $\rho : \{1, 2, \ldots, n\} \to \{0, 1, *\}$ that assigns $*$ to s inputs, the function f^ρ is non-constant. We say that $f : \{0,1\}^* \to \{0,1\}$ is $s(n)$-sensitive if its restriction (in the usual sense of restricting functions) to $\{0,1\}^n$ is $\lfloor s(n) \rfloor$-sensitive for all n. Finally, a language $A \subseteq \{0,1\}^*$ is $s(n)$-sensitive if its indicator function is $s(n)$-sensitive.

Useing this terminology, the above theorem tells us that if A is 1-sensitive then it cannot be decided by a family of levelable circuits of constant depth and polynomial size. We can generalize this:

Corollary 25. *If A is $\frac{\sqrt[4^{d-2}]{4n}}{16}$-sensitive for $d \geq 2$, then A cannot be decided by a family of levelable circuits of polynomial size and depth d.*

Proof. By induction on d. For $d = 2$ we use the same approach as in Lemma 23: assume for contradiction that A is $\frac{n}{4}$-sensitive and can be computed by a family of polynomial size depth 2 levelable circuits. A 2-circuit is a $\vee$ of $\wedge$s. Since A is $\frac{n}{4}$-sensitive each satisfiable $\wedge$ must contain either x_i or $\overline{x_i}$ for more than $\frac{3n}{4}$ of the variables x_i. Thus each $\wedge$-node is true for at most $2^{\frac{n}{4}}$ of the 2^n posible inputs. The first $\frac{3n}{4}$ variables can be fixed in $2^{\frac{3}{4}}$ ways and for each of these there is at least one assignment of the last $\frac{n}{4}$ variables such that the resulting string is in A. So A contains at least $2^{\frac{3n}{4}}$ strings and we need at least $\frac{2^{\frac{3n}{4}}}{2^{\frac{n}{4}}} = 2^{\frac{n}{2}}$, gates on the lowest level.

To make the induction step, we do exactly as in Theorem 24. If we can find a $\frac{\sqrt[4^{d-2}]{4n}}{16}$-sensitive language A the can be computed by a polynomial size depth d family of levelable circuits $C_1, C_2, \ldots$, then we construct D, E, and F families as in Theorem 24. We see that the D-family is $\frac{\sqrt[2^{2(d-2)-1}]{4n}}{16}$-sensitive and the E- and F-families are both $\frac{\sqrt[4^{(d-1)-2}]{4n}}{16}$-sensitive, and the F-family has depth $d-1$ contradicting the induction hypothesis. □

In particular we have:

Corollary 26. *If A is $n^{o(1)}$-sensitive[6], then A cannot be computed by a polynomial size constant depth family of levelable circuits.*

[6]Which means that it is $n^{g(n)}$-sensitive for some function $g(n) \in o(1)$.

3.3 Limitations of natural proofs

Proofs like the one in the last section gave hope that it might possible to prove P $\neq$ NP, but in 1994 Razborov and Rudich found a reason that such a proof would be unlikely: They defined "natural proofs", claimed that "all lower bound proofs known to date against nonmonotone Boolean circuits are natural, or can be represented as natural", and showed that under certain widely believed assumptions, no natural proof could show that P $\neq$ NP [21].

The natural way to prove NP $\not\subseteq$ P would be to choose some language $A \in$ NP, and prove that A has some property that no language in P has. This will often be a natural property, as defined here [21][7]:

Definition 19. Let F_n denote the set of Boolean functions on n variables. A *property* $\mathcal{P}$ is a sequence $\mathcal{P}_0, \mathcal{P}_1, \ldots$ where $\mathcal{P}_n \subseteq F_n$. A *description of a function* $f \in F_n$ is just the string $f(00\ldots00), f(00\ldots01), \ldots f(11\ldots11)$ of length 2^n. We now identify $\mathcal{P}_n$ with its indicator function, so that $\mathcal{P}_n : \{0,1\}^{2^n} \to \{0,1\}$ returns 1 if and only if the input is a description of a function in $\mathcal{P}_n$, and we define $\mathcal{P} = \bigcup_{n\in\mathbb{N}} \mathcal{P}_n$. We say that $\mathcal{P}$ is *natural* if it is

- **Constructive:** $\mathcal{P} \in$ P, or equivalently, we can decide if $f : \{0,1\}^n \to \{0,1\}$ has property $\mathcal{P}$ in time $2^{O(n)}$, and
- **Large:** $|\mathcal{P}_n| \in 2^{-O(n)}|F_n|$, or equivalently, a random function is in $\mathcal{P}_n$ with probability $2^{-O(n)}$,

and it is *useful against P/poly* if the following holds:

- **Useful:** If $C_0, C_1, \ldots$ is a family of circuits and C_n computes a function in $\mathcal{P}_n$ for each $n \in \mathbb{N}$, then the size of the circuit family is not bounded above by a polynomial.

If a property is not useful, there is a language $A \in$ P/*poly* with the property, and as the circuit model cannot distingush between P and P/*poly*, any proof that uses a non-useful property to show that $A \notin$ P, most use more complicated ideas. The two other requirements, largeness and constructivness, are more difficult to justify. The best justification is that they seems difficult to avoid.

There is no formal definition of a natural proof, but intuitively it is a proof that somehow uses a natural property. Often combinatorial proofs are not natural, but can be "naturalized". That is, we can modify the proof slightly, and get a natural proof. When this is possible, we say that the proof *naturalizes*. An example is the proof of Theorem 24.

Example 1 (s-sensitive functions). Fix a $s \in \mathbb{N}$ and consider the property "f is s-sensitive". This is constructive: there are only $\binom{n}{s}2^{n-s} \leq 3^n \in 2^{O(n)}$ different restrictions that needs to be checked, so this can be done in time polynomial in 2^n. However, it is not large: for a function f on n variables, there are 2^{n-s} restrictions that assign $*$'s to the first s variables and 0s and 1s to the rest. The probability that a random function is constant on such a restriction is 2^{-2^s+1}, and these probabilities are independent. Thus, the probability that f is non-constant on all of these restrictions is

$$\left(1 - 2^{-2^s+1}\right)^{2^{n-s}} = \left(\left(1 - 2^{-2^s+1}\right)^{2^{-s}}\right)^{2^n} \notin 2^{-O(n)}.$$

However, all s-sensitive functions are non-constant on all these restrictions, so there is only a small probability that a random function is s-sensitive. Thus s-sensitivity, and particular 1-sensitivity, is not a natural property, so the proof of Theorem 24 is not natural.

Example 2 ($(log(n)+3)$-sensitive functions). This property is constructive by the same argument as before. To see that it is also large, we will bound the number of non-s-sensitive functions: such a function is uniquely determined by a $(n-s)$-restriction that makes it constant, the value on that restriction, and the values on the other $2^n - 2^s$ inputs. We can choose a $(n-s)$-restriction in $\binom{n}{s}2^{n-s}$ ways, the value on that restriction in 2 ways and the value on the rest on the input in $2^{2^n-2^s}$ ways, thus there are at most $2\binom{n}{s}2^{n-s}2^{2^n-2^s}$ such functions. So the probability that a random function is not s-sensitive is at most

$$\frac{2\binom{n}{s}2^{n-s}2^{2^n-2^s}}{2^{2^n}} = \frac{2\binom{n}{s}2^n}{2^s 2^{2^s}}$$

[7] The definition here differs slightly from that in [21]: They would call a combinatorial property natural if and only if it has a subset that we call natural.

For $s = \log(n) + 3$ we get:

$$\frac{2\binom{n}{s}2^n}{2^s 2^{2^s}} \leq \frac{2\binom{n}{\log(n)+3}2^n}{2^{2^{\log(4n)+1}}} \leq \frac{2 \cdot 2^n \cdot 2^n}{2^{4n}} \in O(2^{-n})$$

Thus only a vanishing fraction of functions are non-$(\log(n)+3)$-sensitive, so the property is natural. By Corollary 26 the proof of Theorem 24 naturalizes.

In order to state the assumptions in the result of Razborov and Rudich, we will need to define pseudo-random string generators. Intuitively, we want it to be something that outputs strings that "look random", in the sense that we cannot tell the difference between the output of a pseudo-random string generator and a truly random string. To compute something random, it would need a "seed", a random input, and in order to *generate* randomness, it would have to give an output that is longer than the input it receives. So what does it mean that the output "looks random"? We could imagine all kinds of tests: is the output biased toward either 0 or 1, does it contain too long runs of 0s or 1s or too many short runs, or are there some linear relations modulo a large prime, if the string is broken into chunks of length 8, and so on [15, Chapter 3.3]. The answer is, that we want it to be random with respect to every test that can be computed by a not too large circuit family.

In the original article [21], Razborov and Rudich use two different kinds of "pseudo-random generators": ¨both "pseudo-random generators" (sometimes call "pseudo-random number generators") and "pseudo-random function generators". Instead we will use only one general term, that is invented for the purpose.

Definition 20. A $l(k)$ *pseudo-random string generator* for $l(k) > k$ is any family of Boolean function $f_0, f_1, \ldots$ where $f_k : \{0,1\}^k \to \{0,1\}^{l(k)}$. We say that it is $h(k)$-hard against $s(l)$-sized circuits if for any circuit family $C_{l(k)}$ of size $\leq s(l(k))$, where $C_{l(k)}$ takes $l(k)$ inputs, and

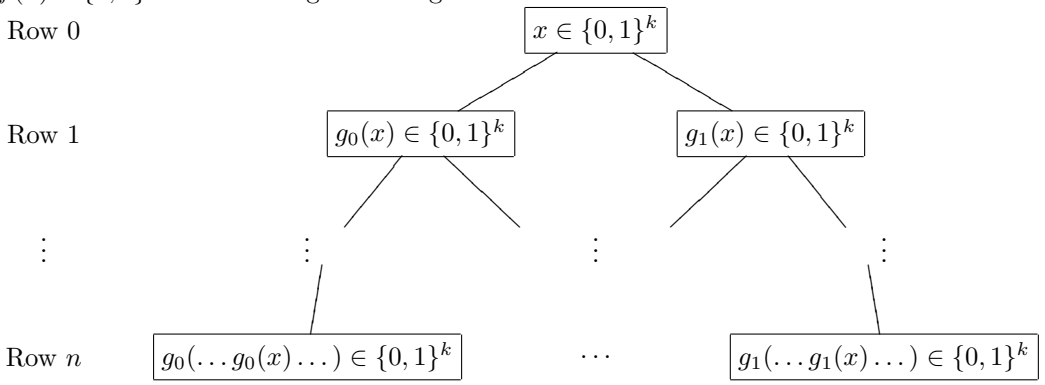
$$\left|P(C_{l(k)}(f(\mathbf{x})) = 1) - P(C_{l(k)}(\mathbf{y}) = 1)\right| < h(k)^{-1}$$

for all sufficiently large k, where $\mathbf{x}$ is taken at random from $\{0,1\}^k$ and $\mathbf{y}$ is taken at random from $\{0,1\}^{l(k)}$. More generally, for classes of functions L, H, and S, an L pseudo-random string generator is H-hard against S-sized circuits, if it is a l pseudo-random string generator that is h-hard against s-sized circuits for a $l \in L$ and all $h \in H$ and $s \in S$.

It is believed [21, p. 5] that there exists polynomial time computable $2k$ pseudo-random string generators that are 2^{k^ϵ}-hard against 2^{k^ϵ}-sized circuits, although this is a stronger assumption than P $\neq$ NP. By a proof similar to that of Theorem 21 this would imply that such a pseudo-random string generator can be computed by polynomial size circuits. The following lemma is extracted from the proof of [21, Theorem 4.1].

Lemma 27. *Assume that there is a $2k$ pseudo-random string generator g that is $2^{k^{\epsilon'}}$-hard against $2^{k^{\epsilon'}}$-sized circuits. Then for any $0 < \epsilon < \epsilon'$ we can construct a $2^{\lceil k^\epsilon \rceil}$ pseudo-random string generator f that is polynomially hard against polynomial sized circuits.*

Proof. Let g be a pseudo-random string generator as given in the assumption and let $0 < \epsilon < \epsilon'$. Set $n = \lceil k^\epsilon \rceil$. Although n is a function of k, we will denote it as n and not $n(k)$, even when k is not fixed. Define $g_0, g_1 : \{0,1\}^k \to \{0,1\}^k$ as the first respectively last k bits of g. We define $f(x)$ as follows: make a binary tree of depth n and label the root by x. Now we fill out the tree, using g_0 and g_1. When a node is labeled $z \in \{0,1\}^k$ we write $g_0(z)$ in its left child and $g_1(z)$ in its right child. Now $f(x) \in \{0,1\}^{2^n}$ is the string consisting of the first bit of each label in the lowest level.



We see that f is on the form $f : \{0,1\}^k \to \{0,1\}^{2^n}$, so now we only need to show that f is polynomially hard against polynomial sized circuits. Assume for contradiction that this is not the case. Then there is a polynomial p such that for arbitrary large k, there is a circuit C_{2^n} of size $\leq p(2^n) \in 2^{O(n)}$ such that

$$|P(C_{2^n}(\mathbf{y}) = 1) - P(C_{2^n}(f(\mathbf{x})) = 1)| \geq p(2^n)^{-1},$$

where $\mathbf{x}$ is taken at random from $\{0,1\}^k$ and $\mathbf{y}$ is taken at random from $\{0,1\}^{2^n}$. Let p be such a polynomial.

Let $v_1, v_2, \ldots v_{2^n-1}$ be the nodes of the tree, starting with the root and reading level by level, from left to right. For $i \in \{1, 2 \ldots, 2^{n-1}\}$ we define $\mathbf{y}_{i,n}$ to be a random variable taking values in $\{0,1\}^{2^n}$ and with the following distribution. Start by labeling the first $2i-1$ nodes with strings in $\{0,1\}^k$, chosen independently and with uniform distribution. Then label the rest of the nodes as we did before, if a node has label z and its children are not labeled, we label them $g_0(z)$ respectively $g_1(z)$. Finally read the first bit in each label in the lowest level to get the value $\mathbf{y}_{i,n}$. We see that $\mathbf{y}_{1,n}$ has the same distribution as $f(\mathbf{x})$ and $\mathbf{y}_{2^{n-1},n}$ has the same distribution as $\mathbf{y}$. We now use telescoping and the triangle inequality:

$$\begin{aligned} p(2^n)^{-1} &\leq \left|P(C_{2^n}(\mathbf{y}_{1,n}) = 1) - P(C_{2^n}(\mathbf{y}_{2^{n-1},n}) = 1)\right| \\ &= \left| \sum_{i=1}^{2^{n-1}-1} P(C_{2^n}(\mathbf{y}_{i,n}) = 1) - P(C_{2^n}(\mathbf{y}_{i+1,n}) = 1) \right| \\ &\leq \sum_{i=1}^{2^{n-1}-1} \left|P(C_{2^n}(\mathbf{y}_{i,n}) = 1) - P(C_{2^n}(\mathbf{y}_{i+1,n}) = 1)\right|. \end{aligned}$$

Thus there is an i such that

$$|P(C_{2^n}(\mathbf{y}_{i,n}) = 1) - P(C_{2^n}(\mathbf{y}_{i+1,n}) = 1)| \geq (p(2^n)2^n)^{-1}.$$

Remember that the construction of $\mathbf{y}_{i,n}$ and $\mathbf{y}_{i+1,n}$ are very similar. In both cases we start by labeling the first $2i-1$ nodes at random, and end by using g_0 and g_1 to label all but the first $2i+1$ nodes. The only difference is that to define $\mathbf{y}_{i,n}$ we use g_0 and g_1 to get the v_{2i} and v_{2i+1} from their parent v_j, and to get $\mathbf{y}_{i+1,n}$ we simply ignore the v_j and chose v_{2i} and v_{2i+1} at random. Thus we have a test that uses randomness and uses g at most 2^{n-1} times, that distinguishes between $g(\mathbf{v_j})$ and $\mathbf{v_{2i}v_{2i+1}}$ with probability $\geq (p(2^n)2^n)^{-1} \in 2^{-O(n)}$, where each $\mathbf{v}$ is chosen at random from $\{0,1\}^k$. We can get rid of the use of randomness. If a random assignment of $v_1, v_2, \ldots, v_{j-1}, v_{j+1}, \ldots v_{2i-1}$ can be used by the distinguisher, then there must be some fixed assignment of these values, that does the job. As k is polynomial in n, and g can be computed by a polynomial size circuit, we can fill out the tree with a circuit of size in $2^{O(n)}$. As $n = \lceil k^\epsilon \rceil \in o(k^{\epsilon'})$, this contradicts the assumptions about g. □

We are now ready to proof the main theorem of [21]:

Theorem 28. *If there exists a polynomial size $2k$ random string generator that is $2^{k^{\epsilon'}}$-hard for $2^{k^{\epsilon'}}$-sized circuits for some $\epsilon' > 0$, there is no natural proof that $P \neq NP$.*

Proof. Assume for contradiction that there is some natural and useful property $\mathcal{P}$ and a random string generator g as in the assumption. We use the construction from Lemma 27 on this random string generator to get a $2^{\lceil k^\epsilon \rceil}$ random string generator f that is polynomially hard for polynomial sized circuits, for some $\epsilon > 0$. Set $n = \lceil k^\epsilon \rceil$. We then consider the value $f(x)$ to be the description of a function in F_n. So $f(x)$ takes input $y \in \{0,1\}^n$ and to calculate $f(x)(y)$ you only need to compute g of some k-length strings n times. We know that k is bounded by a polynomial in n, so $f(x)(y)$ can be computed by circuits of size polynomial in the input length $|y|$. Now each function $f(x)$ is in P, so $\mathcal{P}(f(x)) = 0$ for all k and $x \in \{0,1\}^k$. As $\mathcal{P}$ is natural, it must be large, so $P(\mathbf{f}_n \in \mathcal{P}_n) \geq 2^{-O(n)}$. Thus $\mathcal{P}$ is a polynomial time test that distinguishes $f(\mathbf{x})$ from a random function, contradicting the assumption about f. □

Chapter 4

Independence results

Proofs in mathematical papers are often written in informal English, but we think of proofs as something that could be formulated in a "formal system", usually ZFC. That is, we have some axioms that we assume are true and some rules of inference that tell us how to combine true statements to get other true statements. For example if we know that $A \Rightarrow B$ and we know A then we can deduce B. Now a proof is a sequence of statements such that each statement in the sequence is either an axiom or can be deduced from earlier statements in the proof using a rule of inference.

Instead of making the above precise, we will use another point of view used by [7, Section 4.5].

Definition 21. An *axiomatizable theory* is a triple $F = (\Sigma, W, T)$, where Σ is a finite alphabet, $W \subseteq \Sigma^*$ is a decidable set of *well-formed formulas* and $T \subseteq W$ is a recognizable set of *theorems*. We say that elements in T are *provable* in F, and that F *proves* the elements in T.

Not all formal systems correspond to an axiomatizable theory,[1] but any "reasonable"[2] formal system does. If, for example, the set of axioms are decidable and we have a finite number of inference rules, we can easily make a Turing machine M_T that recognizes the set of theorems. On input x we simply let M_T test each string over $\Sigma \cup \{,\}$ beginning with the shortest, to see if it is a formal proof of x and M_T accept if it finds one. If x is a theorem, it will eventually accept, if x is not a theorem $M_T(x)$ will run forever.

We will assume that Σ contains all the letters we need such as "$\neg$", "$\vee$" and "$\Rightarrow$" and that W contains statements about Turing machines. The standard terminology about formal systems is also used when talking about axiomatizable theories. Parts of the following definition are from [7, Section 4.5]:

Definition 22. We say that an axiomatizable theory is *sound* if all its theorems are "true". We will not formally define what this means, but intuitively, if "M runs in time $O(f(n))$" is a theorem in F, then M indeed runs in time $O(f(n))$. We say that a statement S is *independent* of F if neither S nor $\neg S$ are theorems in F. An axiomatizable theory is *consistent* if it does not prove both P and $\neg P$ for any $P \in W$ and it is *complete* if it does prove either P or $\neg P$ for any $P \in W$.

In 1931 Gödel proved his first incompleteness theorem, that says that any "effectively generated" formal system[3] containing arithmetic that is consistent cannot be complete [11].[4] Later Paul Cohen proved, assuming that ZF is consistent, that Axiom of Choice is independent from ZF [4] and that the continuum hypothesis is independent from ZFC [4][5].

4.1 The halting problem and running times

The following famous theorem from [25] is closely related to Gödel's incompleteness theorem.

[1] Consider the formal system with a non-recognizable set A as the set of axioms. A single statement in A is then a formal proof, and any single statement not in A is not a proof, so no computer can check proofs in this formal system.

[2] It is enough to assume that the set of axioms and the set of inference rules are both recognizable and that each inference rule is computable. The proof of this is only slightly more complicated than the proof given here.

[3] This includes all formal systems that correspond to axiomatizable theories.

[4] Originally he had the extra assumption that the formal system was ω-consistent, but this assumption was removed in [23].

Theorem 29. *The halting language defined as*

$$H = \{\langle M, x\rangle | M \text{ is a Turing machine that halts when run on } x\}$$

is undecidable. Indeed, there is a Turing machine which, given the description $\langle M_H\rangle$ of a Turing machine that supposedly solves the halting problem, returns a counterexample $\langle M, x\rangle$ such that $M_H(\langle M, x\rangle)$ rejects if and only if $\langle M, x\rangle \in H$.

Proof. Assume for contradiction that M_H is a Turing machine that decides H. Define D such that

$$D(\langle M\rangle) = \begin{cases} \text{Rejects} & \text{, if } M_H(\langle M, \langle M\rangle\rangle) \text{ rejects} \\ \text{Loops} & \text{, if } M_H(\langle M, \langle M\rangle\rangle) \text{ accepts} \end{cases}$$

and consider what M_H will do on input $\langle D, \langle D\rangle\rangle$. If it accepts, then $D(\langle D\rangle)$ loops, and M_H should reject $\langle D, \langle D\rangle\rangle$. If M_H rejects $\langle D, \langle D\rangle\rangle$, then $D(\langle D\rangle)$ rejects and M_H should accept. In both cases, we have a contradiction.

We see that the construction of D and thus $\langle D, \langle D\rangle\rangle$ can be done by a Turing machine. □

A slightly stronger version of the same theorem is:

Corollary 30. *The language*

$$H_{bl} = \{\langle M\rangle | M \text{ is a Turing machine that halts when run on blank input}\}$$

is undecidable. Indeed, there is a Turing machine which, given the description $\langle M_{H_{bl}}\rangle$ of a Turing machine that supposedly decides H_{bl}, returns a counterexample $\langle M\rangle$ such that $M_{H_{bl}}(\langle M\rangle)$ rejects if and only if $\langle M\rangle \in H_{bl}$.

Proof. We can reduce H to this problem. We can find a Turing machine R that given $\langle M, x\rangle$ can compute another Turing machine $R(\langle M, x\rangle)$, that first writes x on the input tape and then simulate M. Now $R(\langle M, x\rangle) \in H_{bl} \Leftrightarrow \langle M, x\rangle \in H$, so if there was a Turing machine deciding H_{bl}, we could find a Turing machine deciding H.

The reduction is computable and the problem of finding a counterexample to a Turing machine M_H that supposedly decides H is computable by Theorem 29, so the problem of finding a counterexample to a Turing machine $M_{H_{bl}}$ that supposedly decides H_{bl} is also computable. □

We will now use this to prove a similar theorem about axiomatizable theories. The following was proven in [13, Lemma p. 19]. They proved it directly, without referring to the halting problem, but their proof is essentially equivalent to the proof given here:

Theorem 31. *Given a sound axiomatizable theory F, there is a Turing machine M such that F cannot prove or disprove that M halts when run on empty input. Indeed, there is a Turing machine which, given the description $\langle F\rangle$ of a sound formal system, returns a string $\langle M\rangle$ such "M halts" is independent of F.*

Proof. Let F be a sound axiomatizable theory and assume for contradiction that for any Turing machine M either "M halts" or "M does not halt" is a theorem in F. The set T of theorems in F is recognizable, so we can construct a Turing machine $M_{H_{bl}}$ that simultaneously tests if "M halts" or "M does not halt" is a theorem. If "M halts" is a theorem, $M_{H_{bl}}$ accepts, if "M does not halt" is a theorem, it rejects. By assumption, one of these is a theorem, so $M_{H_{bl}}$ halts, and because F is sound, $M_{H_{bl}}$ computes the halting problem. Contradiction.

We can construct a Turing machine R that computes the above $\langle M_{H_{bl}}\rangle$ given $\langle F\rangle$, and by Corollary 30 R can then compute an input $\langle M\rangle$ such that $M_{H_{bl}}(\langle M\rangle)$ rejects if and only if $\langle M\rangle \in H_{bl}$. Now, if $M_{H_{bl}}(\langle M\rangle)$ halts, this implies that F is not sound. Thus $M_{H_{bl}}(\langle M\rangle)$ does not halt, and there is no proof or disproof in F that M halts on empty input. □

We will now use this to show that even for "very fast" algorithms, it is sometimes difficult to show good upper bounds on the running time. The following is very similar to [13, Theorem p. 21].[5]

[5] In the proof of [13, Theorem p. 22] they assume that F is strong enough to show that if M_N runs in time $< 2^n$ then N terminates (where M_N and N are as in the next proof), in order to simplify the proof. They note that the theorem can be proved without this assumption. Another difference is that the following theorem uses $n + 1$, where they use n^2.

Theorem 32. *Given a sound axiomatizable theory F there exists a Turing machine M with running time $n+1$, but with no proof in F that it runs in time $o(2^n)$.*

Proof. Let F be a sound axiomatizable theory, and assume for contradiction that for any Turing machine M that runs in time $n+1$, the statement "M runs in time $o(2^n)$" is a theorem in F. Given a Turing machine N we can construct a new Turing machine M_N that on input 1^n simulates N on empty input for n steps. If N halts within the first n steps, then M_N halts after 2^n steps. Otherwise M_N halts after $n+1$ steps.

If N does not halt on empty input, then M_N runs in time $n+1$ and by assumption F proves that it runs in time $o(2^n)$. On the other hand, if N does halt, then M_N runs in time 2^n and by the soundness assumption, F cannot prove that M_N runs in time $o(2^n)$. Now we can define a Turing machine $M_{H_{bl}}$ that decides H_{bl}. On input $\langle N \rangle$ it computes $\langle M_N \rangle$ and then it simultaneously simulates N on empty input and tries to test if "M_N runs in time $o(2^n)$" is a theorem in F: If N halts, $M_{H_{bl}}$ accepts, and if "M_N runs in time $o(2^n)$" is a theorem in F, then $M_{H_{bl}}$ rejects. By assumption, one of these must be true, so $M_{H_{bl}}$ halts, and we see that it decides H_{bl}. Contradiction. □

In the above proof we know that if M_N runs in time $o(2^n)$ then N does not halt, but it should be noted that this implication is not necessarily provable in F, and there might even be a Turing machine N for which "M_N runs in time $o(2^n)$" is provable, but "N does not halts' is not. Of course, the above theorem is true for arbitrary fast growing computable functions instead of 2^n. Even the following theorem holds [13, Corollary p. 21].

Corollary 33. *Given a sound axiomatizable theory F there exists a Turing machine M with running time $n+1$, but with no proof in F that M halts on all input.*

Proof. Just as the above proof, except that we define M_N such that is does not halt on input 1^n, if N halts in time n. □

The following easy consequence of the halting problem can be used to prove more independence results about running times.

Lemma 34. *For any sound axiomatizable theory F there is an integer d such that for no $k \in \mathbb{N}$ is the statement S_k="any Turing machine with description length at most d that halts on empty input, halts on empty input within k steps" provable in F*

Proof. Assume for contradiction that this is false, so that for any d there is a k such that the statement S_k is provable in F. From this we can find a Turing machine that decides H_{bl}: Given $\langle M \rangle$, it searches for true statements S_k where $k \in N$ until it finds one. If this is S_{k_0} then it simulates M for k_0 steps, and if M halts within k steps it accepts, otherwise it rejects. □

At the same time we see that S_k is true for sufficiently large k. For each d there is only a finite number of Turing machines with description length at most d, so among the Turing machine with finite running time, there must be a maximal running time. For a similar reason, this maximal finite running time, grows faster than any computable function [20].

We will now use that lemma to prove that sometimes it is possible to prove that a running time is polynomial, but not to give any explicit polynomial bound. The theorem was suggested by my advisor.

Theorem 35. *Given a sufficiently strong sound axiomatizable theory F there exists a Turing machine M such that F can prove "M runs in polynomial time" but for no k can F prove "M runs in time $O(n^k)$".*

Proof. Let d be the integer obtained in Lemma 34. We now define M as follows: On input 1^n simulate each Turing machine with description length at most d for n steps, and remember the largest value $k_n \leq n$ such that one of the Turing machine halts after exactly k_n steps. This can be done in time $O(n)$. Now wait n^{k_n} steps before halting. We now assume that F is strong enough to

- prove $\exists k : S_k$,
- prove $S_k \Leftrightarrow M$ runs in time $O(n^k)$, and
- make substitutions: if F proves $\exists x : P(x)$ and $P(x) \Leftrightarrow Q(x)$ then F proves $\exists x : Q(x)$.

Now F can prove that there exists a k such that M runs in time $O(n^k)$, but by definition of d it cannot prove S_k and for any k, so it cannot prove that M runs in time $O(n^k)$ for any fixed $k \in \mathbb{N}$. □

The above is a theorem about an *algorithm*, and not about the language the algorithm decides. The following corollary, also suggested by my advisor, shows that something similar holds for some languages.

Corollary 36. *Given a sufficiently strong sound axiomatizable theory F there exists a Turing machine M such that F can prove "M runs in polynomial time", but for no Turing machine M' and integer k can F prove "$L(M) = L(M')$ and M' runs in time $O(n^k)$".*

Proof. The proof is similar to that of Theorem 35, but now we let M take input $\langle N, x\rangle$. As before, M computes k_n, where n is the length of the input. Then we let M decide if $\langle N, x\rangle \in A_{n^{k_n}}$, where

$$A_f = \{\langle N, x\rangle | N \text{ accepts } x \text{ in at most } f(|x|) \text{ steps}\}.$$

From Lemma 10 we know that this can be computed in time $O(n^{3k_n})$ and from the proof, we see that it is uniform in k, that is, we can construct a Turing machine that on input $\langle k, N, x\rangle$ can decide if $\langle N, x\rangle \in A_{n^k}$.

In addition to the assumption we had about F in Theorem 35 we will need the assumption: F is strong enough to prove "$A_{n^{k+1}} \notin \text{TIME}(n^k)$"[6] and to prove "$S_k \Rightarrow M$ runs in time $O(n^{3k})$". Now F can prove that M runs in polynomial time, but if it could prove that it runs in $O(n^k)$ for some k, it could also prove S_{k+1}, which is impossible. □

The above theorem says that F cannot prove that $L(M)$ can be decided in time $O(n^k)$, for any k. Nevertheless, there is a Turing machine M' and a k such that $L(M) = L(M')$ and F can prove that M' runs in time $O(n^k)$. We simply add a "clock" to the Turing machine M so that is stops after time kn^k for some sufficiently large k. What the above theorem tells us is, that for any such Turing machine M', we cannot prove $L(M) = L(M')$ in F.

4.2 Is P vs. NP independent?

In this section we will prove the independence of some relativized versions of the P = NP problem. To do this, we need the following definition, taken from [12].

Definition 23. If

$$\{\langle M_1\rangle, \langle M_2\rangle, \dots\}$$

is recognizable set, we say that it is a *recursive presentation* of the family of languages

$$C = \{L(M_i) | i \in \mathbb{N}^+\}.$$

Furthermore, if R is a decidable set of Turing machines, and $\{M_1, M_2, \dots\} \subseteq R$, we say that $\{\langle M_1\rangle, \langle M_2\rangle, \dots\}$ is a *R-presentation* of C.

As we will see, not all classes of languages have a recursive presentation. First, let us see an example of a class that does have a recursive presentation.

Proposition 37. *The class P has a recursive presentation.*

Proof. For any given Turing machine, we can add a "clock" to the machine, such that on input of length n, it halts after at most $k + n^k$ steps. For a $i \in \mathbb{N}^+$ on the form $\langle i\rangle = \langle M, k\rangle$ for some Turing machine M and $k \in \mathbb{N}$, we define M_i as the Turing machine M where we add a $k + n^k$ clock. For $i \in \mathbb{N}^+$ not in that form, we just let M_i be a Turing machine with $L(M_i) = \emptyset$. Now the sequence M_i is computable, so the set $\{\langle M_1\rangle, \langle M_2\rangle, \dots\}$ is recognizable.

To finish the proof, we need to show $\text{P} = \{L(M_i) | i \in \mathbb{N}^+\}$. Clearly all the clocked Turing machines run in polynomial time, so $\{L(M_i) | i \in \mathbb{N}^+\} \subseteq \text{P}$. On the other hand, let $A \in \text{P}$. Now A is decided by some Turing machine M that runs in polynomial time, but then there is a k such that the running time is bounded by $k + n^k$. Now $A = L\left(M_{\langle M,k\rangle}\right)$, so $\text{P} \subseteq \{L(M_i) | i \in \mathbb{N}^+\}$ □

[6] This statement follows from Lemma 11.

Each of the machines M_i runs in polynomial time, so we say that P has a recursive presentation by polynomial time Turing machines.

We are now ready to show the following result from [13, Theorem p. 17].

Theorem 38. *Given a sound axiomatizable theory F we can find a Turing machine M such that the statement $P^{L(M)} \neq NP^{L(M)}$ is independent of F.*

Proof. Let B and C be languages such that $\mathrm{P}^B = \mathrm{NP}^B$ and $\mathrm{P}^C \neq \mathrm{NP}^C$. We know from Theorem 15 and 17 that they exist and we see from the proofs that they are decidable, as long as the sequence $M_1, M_2, \ldots$ in the proof of Theorem 17 forms a recursive presentation of P and the sequence $p_1, p_2, \ldots$ is computable. Given F we define M. By Theorem 4 we can assume that M has access to its own description. Now on input x it tries to test if F proves $\mathrm{P}^{L(M)} = \mathrm{NP}^{L(M)}$ or $\mathrm{P}^{L(M)} \neq \mathrm{NP}^{L(M)}$ in $|x|$ steps. If it proves $\mathrm{P}^{L(M)} = \mathrm{NP}^{L(M)}$ in at most $|x|$ steps then M accepts if and only if $x \in C$. If it proves $\mathrm{P}^{L(M)} \neq \mathrm{NP}^{L(M)}$ in at most $|x|$ steps it accepts if and only if $x \in B$. Finally, M has not shown that any of these are theorems in F it rejects.

If $\mathrm{P}^{L(M)} = \mathrm{NP}^{L(M)}$ is a theorem in F, then for sufficiently long inputs x we have $x \in L(M) \Leftrightarrow x \in C$, and by Proposition 14 this would imply that $\mathrm{P}^{L(M)} = \mathrm{P}^C \neq \mathrm{NP}^C = \mathrm{NP}^{L(M)}$, contradiction. On the other hand, if $\mathrm{P}^{L(M)} \neq \mathrm{NP}^{L(M)}$, then for sufficiently long inputs x we have $x \in L(M) \Leftrightarrow x \in B$, and this would imply that $\mathrm{P}^{L(M)} = \mathrm{P}^B = \mathrm{NP}^B = \mathrm{NP}^{L(M)}$, which is again a contradiction. □

We can easily see that $L(M) = \emptyset$, but this does not imply that $\mathrm{P}^\emptyset = \mathrm{NP}^\emptyset$, which is equivalent to $\mathrm{P} = \mathrm{NP}$, is independent. It might not be possible to prove in F that $L(M) = \emptyset$. There is nothing special about the empty set here:

Corollary 39. *Given a sound axiomatizable theory F and a decidable language A, we can find a Turing machine M that decides A such that the statement $P^{L(M)} = NP^{L(M)}$ is independent of F*

Proof. Just like the proof of Theorem 38 except that we also let M accept x if $|M|$ did not find a proof of $\mathrm{P}^{L(M)} = \mathrm{NP}^{L(M)}$ or $\mathrm{P}^{L(M)} \neq \mathrm{NP}^{L(M)}$ and $x \in A$. □

Corollary 39 does not prove that $\mathrm{P}^A = \mathrm{NP}^A$ is independent of F, it only shows that the representation, $L(M)$, of A is "opaque".[7]

The following lemma from [12] shows a connection between axiomatizable theories and Turing machines.[8]

Lemma 40. *A family C of recognizable sets has a recursive presentation if and only if there exists a sound axiomatizable theory F in which for each $L \in C$ there is a Turing machine M such that $L = L(M)$ and the statement "M is total and $L(M) \in C$" is provable in F.*

Proof. "⇒": If C has a recursive presentation $\{\langle M_1\rangle, \langle M_2\rangle, \ldots\}$, we can simply use the set

$$\{\text{"}M_i \text{ is total and } L(M_i) \in C\text{"} | i \in \mathbb{N}\}$$

as our set of theorems.[9]

"⇐": Let F be a axiomatizable theory such that for each $L \in C$ there is a Turing machine, M, such that $L(M) = L$ and the statement "M is total and $L(M) \in C$" is provable in F. We can then define a Turing machine M_F that given $\langle M\rangle$ tests if the statement "M_i is total and $L(M) \in C$" is a theorem in F, and accepts if it is. We know that F is sound, so M_F recognizes C. □

The following similar theorem is also from [12].

Corollary 41. *Let R be a decidable set of total Turing machines. Then a family C of languages has a R-presentation if and only if there exists an sound axiomatizable theory in which for each $L \in C$ there exists a Turing machine $M \in R$ such that $L(M) = L$ and the statement "$L(R) \in C$" is provable in F.*

Proof. As the above. □

[7] A word used by [12].

[8] The results in [12] are about formal systems which, which is a concept similar to axiomatizable theories.

[9] In [12] Hartmanis add the statements "M_i is total and $L(M) \in C$" to the axioms of Peano Arithmetic, and together with the set of well-formed formulas and the rules of inference, the gives a formal system. This formal system can then be turned into an axiomatizable theory. This would be closed under the rules of inference, and thus more "reasonable" than the axiomatizable theory consisting of only the statements "M_i is total and $L(M) \in C$".

The next theorem is also from [12], but there seems to be a gap the proof in that article.[10] The proof given here closes the gap, by using the method from the proof of Theorem 17. First let $M_{p(1)}, M_{p(2)}, \dots$ be a sequence of polynomial time oracle Turing machines such that for any oracle A the set $\{\langle M^A_{p(1)}\rangle, \langle M^A_{p(2)}\rangle, \dots\}$ is a recursive presentation of P^A, and let $\{\langle M_{E(1)}\rangle, \langle M_{E(2)}\rangle, \dots\}$ be a recursive presentation of $\mathrm{SPACE}\left(2^{2^n}\right)$ by Turing machine with space bound $O\left(2^{2^n}\right)$. Furthermore, we will assume that the problem "given a Turing machine with an explicit space bound $k2^{2^n}$ find an equivalent Turing machine $M_{E(i)}$" is computable. The proofs that such recursive presentations exist, are similar to the proof of Proposition 37. The following uses theorems and terminology from Appendix A.

Theorem 42. *The set*

$$D = \{\langle M_{E(i)}\rangle | P^{L(M_{E(i)})} \neq NP^{L(M_{E(i)})}\}$$

is Π^0_2*-complete.*

Proof. Let M be an oracle Turing machine, such that M^A is total and $L(M^A)$ is NP^A-complete for any oracle A. The proof that such an oracle Turing machine exists is almost identical to the proof that NP-complete languages exists. If $\mathrm{P}^A = \mathrm{NP}^A$ then there is a polynomial time Turing machine $M_{p(j)}$ such that $L\left(M_{p(j)}\right) = L\left(M^A\right)$, and if $\mathrm{P}^A \neq \mathrm{NP}^A$ then $L\left(M_{p(j)}\right) \neq L\left(M^A\right)$ for all $j \in \mathbb{N}$. We can now write D as

$$D = \{\langle M_{E(i)}\rangle | \forall j \in \mathbb{N} \exists x \in \Sigma^* : M^{L(M_{E(i)})}_{p(j)}(x) \neq M^{L(M_{E(i)})}(x)\}$$

We can easily encode $j \in \mathbb{N}$ is a string, and $M^{L(M_{E(i)})}_{p(j)}(x) \neq M^{L(M_{E(i)})}$ is decidable for fixed values of i, j, and x. Thus $D \in \Pi^0_2$.

To show that D is Π^0_2-complete, we will reduce ALL to D. That is, we will show that there is a Turing machine that sends a description of a Turing machine M_i to a description of another Turing machine $M_{E(\sigma(i))}$, such that $\langle M_i\rangle \in INF \Leftrightarrow \langle M_{E(\sigma(i))}\rangle \in D$. We know that ALL is Π^0_2-complete, so this will imply that $M_{E(\sigma(i))}$ is Π^0_2-complete.

Let A be the set given in Theorem 15 such that $\mathrm{P}^A = \mathrm{NP}^A$. From the proof of Theorem 15 we see that $A \in \mathrm{SPACE}\left(2^{2^n}\right)$. When M is a Turing machine, we define

$$L_M = \{0^n | \exists x \in L(M), |x| = n\}.$$

We define $M_{E(\sigma(i))}$ by the following pseudo code.

$M_{E(\sigma(i))}$ on input x:

1. Set $n = |x|$
2. Mark of 2^{2^n} squares on the tape
3. Compute the value $M_{E(\sigma(i))}(y)$ for all $|y| < n$
4. Set k_0 =the k-value of input of length $n-1$. If $n = 0$ set $k_0 = 0$
5. Let S be the set of strings $y, |y| < n$ that M accepts
6. Simulate M_i on the $k_0 + 1$'th input until it halts, or $M_{E(\sigma(i))}$ runs out of space. If M_i accepts {
7. Simulate $M^S_{p(k_0+1)}(0^n)$. Let N be the longest string that $M^S_{p(k_0+1)}(0^n)$ asks the oracle
8. Set $M_{E(\sigma(i))}(x) = 1 - M^S_{p(k_0+1)}(0^n)$ for all x, $|x| = n$
9. Set $M_{E(\sigma(i))}(z) = 0$ for all z, $n < |z| \leq N$
10. Set the k-value to $k_0 + 1$} else {
11. Set $M_{E(\sigma(i))}(x) = 1$ if $x \in A$ and $M_{E(\sigma(i))}(x) = 0$ otherwise, keep the k-value at k_0}.

[10] In particular, the argument that $\mathrm{P}^{L(M_{\sigma(i)})} \neq \mathrm{NP}^{L(M_{\sigma(i)})}$ does not seem to work. The fact that the kth step is completed for each $k \geq 1$ only shows that $\mathrm{P} \neq \mathrm{NP}^{L(M_{\sigma(i)})}$.

The value of $M_{E(\sigma(i))}(x)$ might be computed already in the third line, while computing the value of a shorter input. In that case, $M_{E(\sigma(i))}$ terminates already in line 3. If the sequence $M_{p(i)}$ is chosen reasonably, we never run out of space in line 7.

If $L(M_i)$ is finite, we end in line 11 for all but finitely many inputs, so $|L(M_{E(\sigma(i))}) \triangle A| < \infty$ and $\mathrm{P}^{L(M_{E(\sigma(i))})} = \mathrm{P}^A = \mathrm{NP}^A = \mathrm{NP}^{L(M_{E(\sigma(i))})}$.

If $L(M_i)$ is infinite, we go to lines 7 to 10 for each value of $k_0 \in \mathbb{N}$. Because of line 8 we know that $L(M^S_{p(k_0+1)})$ and $L_{M_{E(\sigma(i))}}$ disagree on 0^n. Because of line 9 we know that $M^S_{p(k_0+1)} = M^{M_{E(\sigma(i))}}_{p(k_0+1)}$, so $L(M^{M_{E(\sigma(i))}}_{p(k_0+1)}) \neq L_{M_{E(\sigma(i))}}$. Since this is true for any $k_0 \in \mathbb{N}$, we have $L_{M_{E(\sigma(i))}} \notin \mathrm{P}^{M_{E(\sigma(i))}}$, but for any Turing machine M we have $L_M \in \mathrm{NP}^{L(M)}$, so $\mathrm{P}^{M_{E(\sigma(i))}} \neq \mathrm{NP}^{M_{E(\sigma(i))}}$ as we wanted. □

We will now use this to show a stronger independence result from [12].

Theorem 43. *Given a sound axiomatizable theory F there exists a language $B \in SPACE\left(2^{2^n}\right)$ such that $P^B \neq NP^B$ but for no $M_{E(i)}$ with $L(M_{E(i)}) = A$ can it be proven in F that $P^{L(M_{E(i)})} \neq NP^{L(M_{E(i)})}$.*

Proof. Assume for contradiction that for each $B \in \mathrm{SPACE}\left(2^{2^n}\right)$ such that $\mathrm{P}^B \neq \mathrm{NP}^B$, there is an $M_{E(i)}$ with $L(M_{E(i)}) = B$ such that F proves $\mathrm{P}^{L(M_{E(i)})} \neq \mathrm{NP}^{L(M_{E(i)})}$. Then the set

$$D = \{\langle M_{E(i)}\rangle | \mathrm{P}^{L(M_{E(i)})} \neq \mathrm{NP}^{L(M_{E(i)})}\}$$

would have a recursive presentation $\{\langle M_{i_1}\rangle, \langle M_{i_2}\langle, \dots\}$. We can then write

$$D = \{M_{E(i)} | \exists M_{i_j} \forall x : x \in L\left(M_{E(i)}\right) \Leftrightarrow x \in L\left(M_{i_j}\right)\}$$

and $D \in \Sigma^0_2$. But from Theorem 42 we know that D is Π^0_2-complete, and we have a contradiction. □

Corollary 39 shows that for any decidable set A, we can represent this set in a way that makes $\mathrm{P}^A = \mathrm{NP}^A$ independent, so this is a theorem about representations of sets. The above theorem is more interesting, because it says that the set B is so complicated that $\mathrm{P}^B = \mathrm{NP}^B$ is independent for any representation in $\mathrm{SPACE}\left(2^{2^n}\right)$.

There is nothing special about the function 2^{2^n} here. The same theorem holdsfor any function that grows sufficiently fast, and can be computed without using too much space. However, we cannot generalize to all Turing machine or to all total Turing machine. If we try to generalize it to the set of all Turing machines using a recursive presentation $\{\langle M_{t(1)}\rangle, \langle M_{t(2)}\rangle, \dots\}$, then $x \in L\left(M_{t(i)}\right) \Leftrightarrow x \in L\left(M_{i_j}\right)$ is not decidable, so we cannot prove $D \in \Sigma^0_2$ as in the above proof. We cannot generalize to the set of all total Turing machines either, because the set

$$\{A | \exists M : M \text{ is a total Turing machine and } A = L(M)\}$$

does not have a recursive presentation by total Turing machines. However, it has later been shown that given a sound axiomatizable theory F there exists a decidable language E, such that $\mathrm{P}^E \neq \mathrm{NP}^E$, but for no provable total Turing machine M with $E = L(M)$ can F prove $\mathrm{P}^{L(M)} \neq \mathrm{NP}^{L(M)}$ [22, Theorem 6.9].

We have seen many independence results in this chapter, but they are all in the form "For any given (sufficiently strong) sound axiomatizable theory F, there is a statement on the form ... that is independent of F". Such results does not show anything about the P = NP problem, because there is a sound axiomatizable theory, in which P = NP is decidable. If ZFC is sound then either $ZFC + (\mathrm{P} = \mathrm{NP})$ or $ZFC + (\mathrm{P} \neq \mathrm{NP})$ is such a sound axiomatizable theory. Thus, to show that P = NP is independent from ZFC, you would need to consider the axioms of ZFC.

Appendix A

The arithmetical hierarchy

In order to prove some more independence results, we need to know something about the arithmetical hierarchy. The definition is standard and can be found in [18, IV.1].[1]

Definition 24. A relation $R \subseteq (\Sigma^*)^k$ is *decidable* if the language

$$L_R = \{\langle x_1, \ldots, x_k\rangle | R(x_1, \ldots x_k)\}$$

is decidable. We define Σ_k^0 for $k \geq 0$ as the class of languages L for which there is a decidable $(k+1)$-ary relation R such that

$$L = \{x | \exists x_1 \forall x_2 \ldots Q_k x_k R(x_1, \ldots x_k, x)\}$$

where all quantifiers are over Σ^* and Q_k is $\exists$ if k is odd and $\forall$ if k is even. Similarly we define Π_k^0 as the class of languages L such that there is a decidable $(k+1)$-ary relation R such that

$$L = \{x | \forall x_1 \exists x_2 \ldots Q_k x_k R(x_1, \ldots x_k, x)\}$$

where all quantifiers are over Σ^* and Q_k is $\forall$ if k is odd and $\exists$ if k is even.

A language L is Σ_k^0-*complete* if $L \in \Sigma_k^0$ and for each language $A \in \Sigma_k^0$ there is a total Turing machine M such that $x \in A \Leftrightarrow M(x) \in L$. We define Π_k^0-*complete* similarly.

Some authors define Π_k^0 to be $\mathbf{co}\Sigma_k^0 = \{\overline{A} | A \in \Sigma_k^0\}$ instead[19, 3.4.9]. The following proposition shows that these to definitions are equivalent.

Proposition 44. *For any $n \in \mathbb{N}$ and any language A we have $A \in \Sigma_n^0 \Leftrightarrow \overline{A} \in \Pi_n^0$, where $\overline{A}$ denote the complement of A.*

Proof. For $A = \{x | \exists x_1 \forall x_2 \ldots Q_n x_n R(x_1, x_2, \ldots, x_n)\}$ we have

$$\begin{aligned}\overline{A} =& \{x | \neg \exists x_1 \forall x_2 \ldots Q_n x_n R(x_1, x_2, \ldots x_n)\} \\ =& \{x | \forall x_1 \exists x_2 \ldots Q_n x_n \neg R(x_1, x_2, \ldots, x_n)\}\end{aligned}$$

and the relation $\neg R$ is decidable whenever R is, so $A \in \Sigma_n^0 \Rightarrow \overline{A} \in \Pi_n^0$. The proof of the other direction is similar. □

A special case of this definition is $\Sigma_0^0 = \Pi_0^0$ that is just the set of decidable languages, and any decidable language except $\emptyset$ and Σ^* is both Σ_0^0-complete and Π_0^0-complete. A more interesting example of a Π_k^0-complete language is given, without proof of completeness, in [16]:

Example 3. The language

$$C = \{\langle M, N\rangle | \forall y : \text{both } M(y) \text{ and } N(y) \text{ halts} \Rightarrow M(y) = N(y)\}$$

of pairs of consistent Turing machines is Π_1^0-complete.

[1] The arithmetical hierarchy is the unbounded counterpart of the polynomial hierarchy as defined in footnote 4 in Chapter 3.

Proof. To show that C is in Π_1^0, we have to put it on the form

$$C = \{x | \forall x_1 : R(x_1, x)\}.$$

We want x to be $\langle M, N\rangle$, so we define R to be false whenever x is not on this form. If we just take x_1 to be the same as y, and define $R(x_1, \langle M, N\rangle)$ to be "both $M(x_1)$ and $N(x_1)$ halts $\Rightarrow M(x_1) = N(x_1)$", then R is not decidable. Instead we let x_1 be a input together with a time limit, $x_1 = \langle y, t\rangle$, and define $R(x_1, x)$ to be true if x is on the correct form, but x_1 is not, and define

$$R(\langle y, t\rangle, \langle M, N\rangle) = \text{"both } M(y) \text{ and } N(y) \text{ halts within } t \text{ steps} \Rightarrow M(y) = N(y)\text{"}$$

this is decidable and $C = \{x | \forall x_1 R(x_1, x)\}$.

To show that C is complete in Π_1^0 let $A \in \Pi_1^0$ and let R_A be a decidable relation such that $A = \{x | \forall x_1 R(x_1, x)\}$. We can now construct a Turing machine that on input x returns $\langle (x_1 \mapsto R(x_1, x)), 1\rangle$, where $(x_1 \mapsto R(x_1, x))$ denote a Turing machine that on input x_1 returns $R(x_1, x)$ and 1 denotes the Turing machine that accepts on any input. We see that $x \in A \Leftrightarrow \langle (x_1 \mapsto R(x_1, x)), 1\rangle \in C$, and this works for any $A \in \Pi_1^0$, so C is Π_1^0-complete. □

We will need the following example of a Π_2^0-complete language later.

Lemma 45. *The language*

$$ALL = \{\langle M\rangle | L(M) = \Sigma^*\}$$

is Π_2^0-complete.

Proof. We see that

$$ALL = \{\langle M\rangle | \forall x \in \Sigma^* \exists t \in \mathbb{N} : M(x) \text{ accepts in time } t\},$$

so clearly $ALL \in \Pi_2^0$.

To show that A is complete in Π_2^0 let $A \in \Pi_2^0$ and let R_A be a decidable relation such that $A = \{x | \forall x_1 \exists x_2 R_A(x_1, x_2, x)\}$. We can now construct a Turing machine M_A such that $x \in A \Leftrightarrow M_A(x) \in ALL$. Here $M_A(x)$ will be the description of a Turing machine, and we will denote this Turing machine by $M_{A,x}$. Now $M_{A,x}$ should be the machine that on input x_1 tests $R(x_1, \epsilon, x)$, $R(x_1, 0, x)$, $R(x_1, 1, x)$, $R(x_1, 00, x)$, ... until it finds a x_2 such that $R(x_1, x_2, x)$ is true. If it finds such a x_2 it accepts. Clearly $x \in A \Leftrightarrow M_A(x) \in ALL$. □

The following lemma essentially shows that there exists Π_k^0-complete and Σ_k^0-complete languages for all $k \in \mathbb{N}$.

Lemma 46. *Fix $n \geq 1$. There is a language $E_\Sigma \in \Sigma_n^0$ such that for any $A \in \Sigma_n^0$ there is a M_A such that*

$$x \in A \Leftrightarrow \langle M_A, x\rangle \in E_\Sigma.$$

Similarly there is a language $E_\Pi \in \Pi_n^0$ such that for any $A \in \Pi_n^0$ there is a M_A such that

$$x \in A \Leftrightarrow \langle M_A, x\rangle \in E_\Pi.$$

Proof. For odd n we define E_Σ as

$$E_\Sigma = \{x | \exists x_1 \forall x_2 \ldots \exists x_n R_{E_\Sigma}(x_1, \ldots, x_n, x)\}$$

where $R_{E_\Sigma}(x_1, \ldots x_n, x)$ is defined as "x is on the form $\langle M, x'\rangle$ where M is a Turing machine and x' is a string, x_n is on the form $\langle x_n', t\rangle$ where $x_n' \in \Sigma^*$ and $t \in \mathbb{N}$ and $M(\langle x_1, \ldots x_{n-1}, x_n', x'\rangle)$ accepts within time t". This relation is decidable so $E_\Sigma \in \Sigma_n^0$. For any $A \in \Sigma_n^0$ we can write

$$A = \{x | \exists x_1 \forall x_2 \ldots \exists x_n R(x_1, x_2, \ldots, x_n, x)\}$$

where R is a relation decided by some Turing machine M_A. Now

$$x \in A \Leftrightarrow \langle M_A, x\rangle \in E_\Sigma.$$

We now define $E_\Pi = \overline{E_\Sigma}$, and by Proposition 44 $E_\Pi \in \Pi_n$. For any $A \in \Pi_n^0$ we have $\overline{A} \in \Sigma_n^0$ and

$$x \in A \Leftrightarrow x \notin \overline{A} \Leftrightarrow \langle M_{\overline{A}}, x\rangle \notin E_\Sigma \Leftrightarrow M_{\overline{A}}, x\rangle \in E_\Pi$$

as we wanted.

For even n we define E_Π as we defined E_Σ before. The reason that the above construction of E_{Σ_n} works for odd n, is that the innermost quantifier in

$$\{x|\exists x_1 \forall x_2 \dots Q_n x_n R(x_1, \dots, x_n, x)\}$$

has to be a $\exists$, and this is also the case for the innermost quantifier in

$$\{x|\forall x_1 \exists x_2 \dots Q_n x_n R(x_1, \dots, x_n, x)\}$$

when n is even. We then use negation to define E_Σ. □

From the definition, it is clear that when $k < n$ we have the inclusions:

$$\Sigma_k^0 \subseteq \Sigma_n^0, \quad \Sigma_k^0 \subseteq \Pi_n^0, \quad \Pi_k^0 \subseteq \Sigma_n^0, \quad \Pi_k^0 \subseteq \Pi_n^0$$

If a language A is on the form $\{x|Q_1 x_1 \dots Q_k x_k R(x_1, \dots x_k, x)\}$ we can simply add quantifiers over variables that do not affect R. The following theorem implies that these inclusions are strict [18, Theorem IV.1.13].

Theorem 47. *For $n \geq 1$ we have:*

1. $\Sigma_n^0 \setminus \Pi_n^0 \neq \emptyset$
2. $\Pi_n^0 \setminus \Sigma_n^0 \neq \emptyset$

Proof. Fix $n \geq 1$. Let E_Σ be the relation given by Lemma 46, and define $P = \{x|\langle x, x\rangle \in E_\Sigma\}$. Now $P \in \Sigma_n^0$. Assume for contradiction that $P \in \Pi_n^0$. Then $\overline{P}$ would be in Σ_n^0, so there would be some $M_{\overline{P}}$ such that

$$\langle M_{\overline{P}}, x\rangle \in E_\Sigma \Leftrightarrow x \in \overline{P} \Leftrightarrow \langle x, x\rangle \notin E_\Sigma.$$

But this gives a contradiction for $x = M_{\overline{P}}$. Thus $P \in \Sigma_n^0 \setminus \Pi_n^0$ and this implies $\overline{P} \in \Pi_n^0 \setminus \Sigma_n^0$. □

Corollary 48. *If A is Π_n^0-complete, then $A \notin \Pi_k^0$ for $k < n$ and $A \notin \Sigma_k^0$ for $k \leq n$. Similarly, if B is Σ_n^0-complete, then $B \notin \Sigma_k^0$ for $k < n$ and $B \notin \Pi_k^0$ for $k \leq n$.*

Proof. We will only prove that if A is Π_n^0-complete then $A \notin \Sigma_k^0$ for $k \leq n$, the other proofs are similar.

Assume for contradiction that A is Π_n^0-complete and $A \in \Sigma_k^0$. Since $\Sigma_k^0 \subseteq \Sigma_n^0$ this implies that $A \in \Sigma_n^0$. Let $L \in \Pi_n^0$. By assumption, A is Π_n^0-complete, so there exists a Turing machine M_L such that $x \in L \Leftrightarrow M_L(x) \in A$. Furthermore $A \in \Sigma_n^0$, so we can write $A = \{x|\exists x_1 \forall x_2 \dots Q_n x_n R(x_1, x_2, \dots, x_n, x)\}$ where R is a decidable relation. Now $L = \{x|\exists x_1 \forall x_2 \dots Q_n x_n R(x_1, x_2, \dots, x_n, M(x))\}$, so $L \in \Sigma_n^0$. But we can do this for any $L \in \Pi_n^0$, so $\Pi_n^0 \subseteq \Sigma_n^0$ contradicting the previous theorem. □

Bibliography

[1] T. Baker, J. Gill, and R. Solovay. Relativizations of the $\mathcal{P}$ =?$\mathcal{NP}$ Question. *SIAM J. Comput.* Vol. 4 No. 4, 1975.

[2] C. Bennett, J. Gill. Relative to a Random Oracle A, $P^A \neq NP^A \neq co - NP^A$ with Probability 1, *SIAM Journal on Computing* Vol. 10, pp. 96–113, 1981.

[3] C. Chang *et. al.*. The Random Oracle Hypothesis is False. *Journal of Computer and System Sciences*, Vol. 49, pp. 24-39, August 1997.

[4] P. Cohen. The Independence of the Continuum Hypothesis. *Proc. Nat. Acad. Sci. U. S. A.*, Vol. 50, pp. 1143-1148. 1963.

[5] P. Cohen. The Independence of the Continuum Hypothesis II. *Proc. Nat. Acad. Sci. U. S. A.*, Vol. 51, pp. 105-110. 1964.

[6] S. Cook. The complexity of theorem-proving procedures. *Proceedings of the third annual ACM symposium on Theory of computing (STOC '71)* pp. 151-158, 1971.

[7] D. Du and K. Ko. *Theory of Computational Complexity.* Wiley-Interscience. 2000.

[8] L. Fortnow. Diagonalization. *Bulletin of the European Association for Theoretical Computer Science*, Computational Complexity Column. Vol. 71, pp. 102-112. 2000.

[9] M. Furst, J. Saxe and M. Sipser. Parity, Circuits, and the Polynomial-Time Hierarchy, *Theory of Computing Systems* Vol. 17 No. 1. 1984.

[10] W. Gasarch. The P =?NP poll. *SIGACT News* Volume 33 Issue 2, pp 34–47. 2002.

[11] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I. *Monatshefte für Mathematik und Physik* Vol. 38 pp. 173-98, 1931.

[12] J. Hartmanis. Independence Results about Context-Free Languages and Lower Bounds. *Information Processing Letters*, Vol. 20, No. 5, pp. 241-248, 1985.

[13] J. Hartmanis and J. Hopcroft. *Independence results in computer science.* SIGACT News Vol. 8, No. 4, pp. 13-24, 1976.

[14] F. Hennie and R. Stearns. Two-Tape Simulation of Multitape Turing Machines, *J. ACM* Vol. 13 No. 4, 1966.

[15] D. E. Knuth. *The Art of Computer Programming*, Vol. 2. Addison-Wesley. 1997.

[16] S. Kurtz, M. O'Donnell and J. Royer. How to Prove Representation-Independent Independence Results. *Inf. Process. Lett.* Vol. 24 No. 1, pp. 5-10, 1987.

[17] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*, Second Edition. Springer, 1997.

[18] P. Odifreddi. *Classical Recursion Theory.* Studies in logic and the foundation of mathematics, Vol. 125, 1992.

[19] C. Papadimitriou. *Computational complexity.* Addison-Wesley, 1994.

[20] T. Rado. On Non-Computable Functions. *Bell System Technical Journal*, Vol. 41, No. 3, pp. 877–884, 1962.

[21] A. Razborov and S. Rudich. Natural Proofs, *Journal of Computing and Systems Sciences*, Vol. 55, No. 1, pp. 24-35, 1997.

[22] K. Regan. The Topology of Provability in Complexity Theory. *Journal of Computer and System Sciences*, Vol. 36, No. 3, pp. 384-432, 1988.

[23] B. Rosser. Extensions of Some Theorems of Gödel and Church, *The Journal of Symbolic Logic*, Vol. 1, No. 3, pp. 87-91, 1936

[24] M. Sipser. *Introduction to the Theory of Computation*. Course Technology, Cengage learning 2006.

[25] A. Turing. *On computable numbers, with an application to the Entscheidungsproblem*. Proceedings of the London Mathematical Society, Vol. 2, No. 42, pp. 230–265, 1936.