BARRIERS IN COMPLEXITY THEORY

ARTHUR VALE

Abstract. Several proof methods have been successfully employed over the years to prove several important results in Complexity Theory, but no one has been able to settle the question **P** =?**NP**. This article surveys three of the main proof methods used in Complexity Theory, and then shows that they cannot be used to prove **P** =?**NP**, thus exposing technical barriers to the advance of Complexity Theory. The first of those is diagonalization, which is shown to prove that the Halting Problem is not computable. Then, the concept of relativization is used to prove that diagonalization alone cannot prove **P** =?**NP**. Circuits are then introduced as a way of settling **P** =?**NP**. Then, the concept of a natural proof is introduced, and is proven to not be able to solve **P** =?**NP**. The third method, is algebrization, which is closely related to some results that were proven after the introduction of natural proofs, but that fail to relativize and naturalize. A result, **IP** = **PSPACE**, that does not relativize and is used together with non-naturalizing techniques to overcome the barriers of relativization and naturalization in proving lower bounds is exposed and then algebrization is introduced as a framework to understand why the methods used in **IP** = **PSPACE** can't settle **P** =?**NP**.

Contents

0. Complexity Classes

Familiarity with the definition of deterministic, nondeterministic and probabilistic Turing machines are assumed for all of the article, as well as the definition of

Date: 3rd August 2016.

decidable and undecidable languages and computable and uncomputable functions. Furthermore, basic probability is assumed from Section 3 and on, and basic linear algebra is assumed in Section 4.2.

A complexity class $\mathbf{C}$ is a collection of languages $L \subseteq \{0,1\}^*$. Most of the complexity classes discussed in this article are defined in the following.

Definition 0.1. Let $f : \mathbb{N} \to \mathbb{N}$ be some function. Then, $\mathbf{DTIME}(f(n))$ is the set of all languages $L \subseteq \{0,1\}^*$ such that L is decided by a deterministic Turing machine in $f(n)$ time. Similarly, $\mathbf{NTIME}(f(n))$ is the set of all languages $L \subseteq \{0,1\}^*$ such that L is decided by a nondeterministic Turing machine in $f(n)$ time.

DTIME and **NTIME** are the basic complexity classes used to define the following standard complexity classes.

Definition 0.2.

$$\mathbf{P} = \cup_{c\geq 1}\mathbf{DTIME}(n^c)$$
$$\mathbf{QP} = \cup_{c\geq 1}\mathbf{DTIME}(2^{(\log n)^c})$$
$$\mathbf{SUBEXP} = \cap_{0<\varepsilon<1}\mathbf{DTIME}(2^{n^\varepsilon})$$
$$\mathbf{EXP} = \cup_{c\geq 1}\mathbf{DTIME}(2^{n^c})$$
$$\mathbf{NP} = \cup_{c\geq 1}\mathbf{NTIME}(n^c)$$

1. Diagonalization and Relativization

Diagonalization is a proof technique that was very successful in proving many results and was responsible for much of the work published in the early history of the theory of computing. Despite its successes, many important separation results, such as $\mathbf{P} \neq \mathbf{NP}$, are resilient to diagonalization.

What diagonalization actually entails is hard to make precise. But, in general, it involves plugging some encoding of a Turing machine in another Turing machine that then simulates it. Arora and Barak, in [AB09] define diagonalization as follows.

Definition 1.1. Diagonalization is any technique that only relies on the following two properties of a Turing Machine:

(1) the existence of short encodings for Turing machines;
(2) the existence Turing machines that can simulate other Turing machine given its encoding with small blow-up in its resource usage.

Of course this definition is rather imprecise in its use of "short" and "small". But the point is that it doesn't matter, as long as how short an encoding is or how small the blow-up is suffices to establish the desired result. The ability to simulate other Turing machines is addressed by the following.

Lemma 1.2. *There exists a universal Turing machine U such that, if M is an $f(n)$-time Turing machine, on input $(\llcorner M \lrcorner, x)$ the machine M runs in time $O(f(|x|) \log f(|x|))$ and outputs $M(x)$.*

A proof of Lemma 1.2 with quadratic overhead in the simulation is simple, using the natural constructions that simulate a bigger alphabet in a smaller alphabet and that simulate k tapes with a single tape. To achieve the desired logarithmic overhead, one needs to use a buffering technique that repositions the information on the single tape so as to reduce the movement the tape head needs to do. For a full proof read [HS66]. There are analogous, and very similar constructions for machines simulating all the types of Turing machines discussed in this paper.

Diagonalization can be used to prove several important results in Complexity Theory. In section 1.1, the undecidability of the HALT language is proven using diagonalization.

The problems resilient to diagonalization proofs became better understood with the development of the notion of relativization. One says that a mathematical statement about Turing machines relativizes, if for any language L, the statement still holds over tweaked Turing machines which have access to L. Relativization is discussed in Section 1.2

1.1. **HALT is undecidable.**

Probably the most famous result in computability theory, the fact that the HALT language is undecidable, can be proved using diagonalization. After this fact is established, a typical proof that some other language is undecidable might simply reduce HALT to the language in question. But first, the definition of the HALT language.

Definition 1.3.

$$\text{HALT} = \{(\llcorner M \lrcorner, \llcorner x \lrcorner) : x \in \{0,1\}^n \text{ for some } n \in \mathbb{N} \text{ and } M \text{ halts with input } x\}$$

With the definition, HALT can be proven to be undecidable.

Theorem 1.4. *The language* HALT*, as defined in Definition* 1.3*, is undecidable.*

Proof. For the sake of contradiction, assume HALT is decidable. Then, there exists a Turing machine M that decides HALT. An algorithm C will be defined below and then M will be used to implement C. Then, C will be used to arrive at a contradiction.

Algorithm 1 Algorithm C

```
$h = \mathbb{1}_{\text{HALT}}(x, x)$
if $h = 1$ then
    Do not halt
else
    return 1
end if
```

To implement C as a Turing machine M_C using M one does as follows. M_C first simulates M with input (x, x). Then, if the simulation outputs 1 it enters in a state q_1 such that, no matter what the tape heads of M_C read, the state has all the tape heads go right and transitions back to q_1. This way, if $h = 1$ then M_C does not halt, as desired. Now, if $h = 0$, the machine enters a state q_0 where it writes 1 to the output tape and then goes to the state q_{halt}.

Now, to arrive at a contradiction, consider what M_C does with input $\llcorner M_C \lrcorner$. It will decide whether or not $(\llcorner M_C \lrcorner, \llcorner M_C \lrcorner) \in \text{HALT}$. If $(\llcorner M_C \lrcorner, \llcorner M_C \lrcorner) \in \text{HALT}$, then M_C halts with input $\llcorner M_C \lrcorner$. On the other hand, since in this case $h = 1$, M_C enters a loop in state q_1 and does not halt, a contradiction. Similarly, if $(\llcorner M_C \lrcorner, \llcorner M_C \lrcorner) \notin \text{HALT}$, then M_C does not halt with input $\llcorner M_C \lrcorner$. On the other hand, in this case $h = 0$, and hence M_C enters state q_0 and outputs 1. In particular, it halts, another contradiction. In any case one arrives in a contradiction. This proves that HALT is undecidable. □

The result that HALT is undecidable shows both the power of diagonalization as well as how it fits Definition 1.1.

1.2. **Relativization.** An informal explanation of the concept of relatization was given in the introduction to this section. That notion will be now formalized in terms of oracle Turing machines, which are the tweaked Turing machines referenced then.

Definition 1.5. An oracle Turing machine is a Turing machine M with an extra tape, called an oracle tape, and three additional states q_{query}, q_{yes} and q_{no}. The machine also has oracle access to some predetermined language $O \subseteq \{0,1\}^*$. When M enters the state q_{query} with a string s written in its oracle tape, it transitions in a single step to q_{yes} if $s \in O$ and to q_{no} otherwise.

An oracle Turing machine might, contingent on which oracle they have access to, give nontrivial power in comparison to a simple Turing machine. Hence, it is not necessarily true that the typical complexity classes over oracle Turing machines are still the same as they were over simple Turing machines, nor that they hold the same relations between each other. This motivates the definition of oracle complexity classes.

Definition 1.6. Let $O \subseteq \{0,1\}^*$ be some fixed language. $\mathbf{P}^O$ denotes the complexity class of all languages that are decidable by a polynomial time deterministic oracle Turing machine with oracle access to O. $\mathbf{EXP}^O$ is the complexity class of all languages that are decidable by an exponential time deterministic oracle Turing machine with oracle access to O. Similarly, $\mathbf{NP}^O$ denotes the complexity class of all languages that are decidable by a polynomial time nondeterministic oracle Turing machine with oracle access to O.

A formal definition of relativization follows.

Definition 1.7. Let $\mathbf{C}$ and $\mathbf{D}$ be complexity classes. Then, the inclusion $\mathbf{C} \subseteq \mathbf{D}$ is said to relativize if for all oracles O, it holds that $\mathbf{C}^O \subseteq \mathbf{D}^O$. Similarly, the separation $\mathbf{C} \not\subseteq \mathbf{D}$ relativizes if for all oracles O, it holds that $\mathbf{C}^O \not\subseteq \mathbf{D}^O$.

This then relates to diagonalization for properties (1) and (2) of Definition 1.1 hold for oracle Turing machines. Therefore, all proofs based solely on diagonalization must relativize. In particular, the following theorem implies that diagonalization is not powerful enough to separate $\mathbf{P}$ and $\mathbf{NP}$.

Theorem 1.8. $\mathbf{NP} \subseteq \mathbf{P}$ *does not relativize.*

Proof. From Definition 1.7, it is enough to find an oracle O such that $\mathbf{NP}^O \not\subseteq \mathbf{P}^O$. O will be constructed inductively on some arbitrary ordering of polynomial time machines. Since there are only countably many polynomial-time oracle Turing machines, we can assume that each natural number represents a polynomial-time Turing machine. Let M_i denote the polynomial-time oracle Turing machine represented by i. Furthermore, let p_i be the polynomial such that $p_i(|x|)$ is the running time of M_i on input x. Then, let O_i denote the current construction of O at stage i. For the initial step $O_0 = \emptyset$ and $n_0 = 0$. Then, at stage i choose $n > n_i$ so that $p_i(n) < 2^n$. Then, run M_i on input $x_i = 0^n$. If $M_i(x_i) = 1$, then $O_i = O_{i-1}$. Otherwise, since $p_i(n) < 2^n$, M_i didn't query all 2^n binary strings of length n, and hence, there is some least (using the canonical ordering) binary string $l \in \{0,1\}^n$ that was not queried by M_i while computing $M_i(x_i)$. Then, add l to O_i, set $n_{i+1} = 2^n$ and start state $i+1$. Then $O = \cup_{i=0}^{\infty} O_i$.

Since at the end of stage i either n_{i+1} is set to 2^n, or no string of length n is added to O_i, clearly after adding one string $|x|$ such that $n \leq |x| < 2^n$ is added, no other string of length smaller or equal to 2^n is added to O. In particular, it follows that the computation done by M_i is the same having oracle access to O_i or O.

Now, define the following language:

$$L_O = \{x : \exists y \in O \text{ such that } |x| = |y|\}$$

Then for every i, by construction, M_i rejects x_i if and only if there is some string of length $|x_i| \in O$. But this is the same as $x_i \in L_O$. Hence, no M_i decides L_O for every input, and hence $L_O \notin \mathbf{P}^O$.

On the other hand, $L_O \in \mathbf{NP}^O$ for one can define a nondeterministic Turing machine with oracle access to O that on input x just nondeterministically guesses every possible string of length $|x|$, queries it in its oracle tape and outputs whatever the reply to the query is. Clearly, M decides L_O and does it in linear time. Hence, L_O is indeed in $\mathbf{NP}^O$, thus establishing $\mathbf{P}^O \neq \mathbf{NP}^O$. □

The proof of Theorem 1.8 will also be a key result to prove that $\mathbf{NP} \subseteq \mathbf{P}$ does not algebrize in the sense of Section 4.

1.3. PSPACE Oracles.

Proving that $\mathbf{NP} \not\subseteq \mathbf{P}$ does not relativize isn't very hard, and there are methods that yield proofs that do not rely on the theory of space bounded computation, while still being very short proofs. Despite that, it will be of relevance for proving that $\mathbf{P} =?\mathbf{NP}$ does not algebrize in Section 4 to consider **PSPACE** oracles, which requires some development of the theory of space bounded computation. It is first necessary to define **PSPACE**.

Definition 1.9. Let $S : \mathbb{N} \to \mathbb{N}$ be some function. Then, $\mathbf{SPACE}(S(n))$ is the set of all languages $L \subseteq \{0,1\}^*$ decidable by a deterministic Turing machine M such that, for all $n \in \mathbb{N}$, the number of tape cells visited by M's heads on input $x \in \{0,1\}^n$ is $O(S(n))$.

Similarly, $\mathbf{NSPACE}(S(n))$ is the set of all languages $L \subseteq \{0,1\}^*$ decidable by a nondeterministic Turing machine M such that, for all $n \in \mathbb{N}$, the number of tape cells visited by M's heads on any sequence of nondeterministic choices that accepts an input $x \in \{0,1\}^n$ is $O(S(n))$.

With **SPACE** and **NSPACE** defined, the definition of **PSPACE** and **NPSPACE** follows.

Definition 1.10.

$$\mathbf{PSPACE} = \cup_{c>0} \mathbf{SPACE}(n^c)$$
$$\mathbf{NPSPACE} = \cup_{c>0} \mathbf{NSPACE}(n^c)$$

Space-constructible functions are relevant to formalizing some results.

Definition 1.11. A function $S : \mathbb{N} \to \mathbb{N}$ is space-constructible if there exists a deterministic Turing machine that on input n computes $S(n)$ in $O(S(n))$ space.

A deterministic machine M that runs in polynomial space can run an exponential number of time steps, since there are $2^{p(n)}$ possible strings that can be written in M's tapes for some polynomial p. Similarly, a nondeterministic polynomial-time Turing machine M' has in some sense an exponential advantage to a deterministic polynomial-time Turing machine, for, if it runs in $p(n)$-time for some polynomial p, there are $2^{p(n)}$ possible nondeterministic choices, and for a fixed choice, M' does a deterministic computation. Hence, it might be expected that a nondeterministic Turing machine with some space bound does not have much more power than a deterministic Turing machine with the same space bound. Indeed, Theorem 1.15, proven in 1970 by Savitch in [Sav70], formalizes this heuristic. Its proof relies on a technique commonly used to prove space-related results, that of a configuration graph.

Definition 1.12. Let M be some Turing machine. A configuration C of M is a string specifying the position of M's tape heads, its state and the content of all nonblank tapes at some particular time step. The configuration graph $G_{M,x}$ of M on input $x \in \{0,1\}^*$ is the directed graph where each node is associated with some configuration of M running on input x, and two configurations C and C' are connected if C can transition to C' in one step.

Remark 1.13. Notice that from Definition 1.12 it follows that M accepts x if and only if there exists a path from C_{start} to C_{accept} in $G_{M,x}$, where C_{start} is the configuration in which M is in its start state and C_{accept} is some node where M enters the accept state.

Now, a technical lemma regarding configuration graphs.

Lemma 1.14. *Let $S : \mathbb{N} \to \mathbb{N}$ be some function, M be some $S(n)$-space Turing machine with k tapes and $G_{M,x}$ its configuration graph on input $x \in \{0,1\}^n$. Then any node of $G_{M,x}$ has size at most $O(S(n))$, and $G_{M,x}$ has at most $2^{O(S(n))}$ nodes.*

Proof. The information necessary to encode a node of $G_{M,x}$ is the contents of all of M's nonblank tapes (each of the k tapes with space bounded by $S(n)$), the current state of M (which takes at most $|Q|$ bits, where Q is the set of possible states of M) and M's tape head positions (which only takes at most k tape cells to encode, by putting a special symbol on the string with the content of M's tapes that marks where the head is). Hence, indeed, encoding a node takes at most $O(S(n))$ space. Now, there are 2^t possible strings of size t, and therefore, there are at most $2^{O(S(n))}$ configurations. $\square$

Now, the proof of Savitch's Theorem can be presented.

Theorem 1.15 (Savitch's Theorem)**.** *Let $S : \mathbb{N} \to \mathbb{N}$ be some space-constructible function such that $S(n) \geq \log n$ for all $n \in \mathbb{N}$. Then:*

$$\mathbf{NSPACE}(S(n)) \subseteq \mathbf{SPACE}(S(n)^2)$$

Proof. Let $L \in \mathbf{NSPACE}(S(n))$. Then, there exists an $S(n)$-space nondeterministic Turing machine M that decides L. Now, on input $x \in \{0,1\}$, by Lemma 1.14, the configuration graph $G_{M,x}$ has at most $2^{c'S(n)}$ nodes. Then, notice that there exists a path of length at most 2^i between C and C'' if and only if there exists a node C' such that there exists a path of length at most 2^{i-1} from C to C' and a path of length at most 2^{i-1} from C' to C''. This fact can be used recursively to check deterministically if there exists a path from C_{start} to C_{accept} in $G_{M,x}$.

For that, let M' be a deterministic Turing machine with 3 tapes. On input $x \in \{0,1\}^n$, M' divides its first tape in $c'S(n)$ blocks of length $cS(n)$ (this is possible because $S(n)$ is space-constructible). Now, if M' needs to find a path of length at most 2^i between C and C'' and it has $m+1$ blank blocks, it stores C in the first blank block of the first tape and C'' in the second blank block, leaving $m-1$ blocks blank. Now, M' has $cS(n)-1$ blocks to find a $C' \in G_{M,x}$ such that there exists a path of length at most 2^{i-1} from C to C', and a path of length at most 2^{i-1} from C' to C''. The machine M' uses the second tape to record if there exists such a path between two nodes. For that, let the nodes be ordered in some arbitrary ordering. The machine M' registers 1 in the jth tape cell if there exists a path from the node currently under consideration to node j, and 0 otherwise. At the end of each recursive call, M' erases the whole second tape. Now, when M' is testing some C' to check if it has paths of length 2^{i-1} from C to C' and from C' to C'', it writes the label of C' in the first blank block. If there is such a C', M' proceeds the recursion, otherwise it erases the block where C' is specified and writes a new candidate node.

Now, M' applies this process recursively to find a path of length at most $2^{c'S(n)}$ from C_{start} to C_{accept}. In this case, $m = c'S(n)$, and hence, the recursion can be done $c'S(n)$ times within the preset blocks as described above. This way, at the last step of the recursion M' just needs to check if there exists $I_2 \in G_{M,x}$ such that there exists a path of length $2^{c'S(n)-c'S(n)} = 1$ between nodes I_1 and I_2 and then between nodes I_2 and I_3. This clearly can be done deterministically, since each single step computation of M has only 2 possible ways of transitioning and each is done deterministically by M, so that M' can simulate both in the third tape using only $S(n)$ space. Hence, indeed M' decides the same language as M, and does so in space $(cS(n)) \cdot (c'S(n)) + 2S(n) = O(S(n)^2)$ as desired. □

Corollary 1.16.

$$\mathbf{PSPACE} = \mathbf{NPSPACE}$$

Savitch's Theorem, and Corollary 1.16, are the main tools to find a **PSPACE** oracle O such that $\mathbf{P}^O = \mathbf{NP}^O$. There are in fact many such oracles O. In particular, the following property on an oracle O, which is a key concept in Complexity Theory, is sufficient to have $\mathbf{P}^O = \mathbf{NP}^O$.

Definition 1.17. Let $L, L' \subseteq \{0,1\}^*$ be languages. L' is polynomial-time reducible to L if there exists a polynomial-time deterministic Turing machine M such that $M(x) \in L$ if and only if $x \in L'$.

L is **PSPACE**-hard if every language $L' \in \mathbf{PSPACE}$ is polynomial-time reducible to L.

L is **PSPACE**-complete if $L \in \mathbf{PSPACE}$ and L is **PSPACE**-hard.

An example of **PSPACE**-complete language follows. It will be of particular importance in Section 4, because it is used to prove the inclusion $\mathbf{PSPACE} \subseteq \mathbf{IP}$.

Definition 1.18. A quantified Boolean formula (also referred as QBF) is a predicate of the form $Q_1x_1Q_2x_2\ldots Q_nx_n\varphi(x_1,x_2,\ldots,x_n)$ where for all $i \in [n]$, Q_i is either $\forall$ or $\exists$, $x_i \in \{0,1\}$, and φ is an unquantified Boolean formula.

The True Quantified Boolean Formulae language, TQBF, is the language of all QBFs that are true.

TQBF is **PSPACE**-complete, and the proof makes use of the concept of configuration graph introduced in Definition 1.12. The proof is due to Stockmeyer and Meyer in [SM73] and is very similar to that of Savitch's Theorem and also uses techniques from the proof of the Cook-Levin Theorem from [Coo71] and [Lev73], similar to those used in Theorem 2.8, so that it will be ommited.

Theorem 1.19. TQBF *is* **PSPACE**-*complete.*

It follows as an easy consequence of Savitch's Theorem that any **PSPACE**-complete language O yields an oracle such that $\mathbf{P}^O = \mathbf{NP}^O$. The following simple proof is due to Baker, Gill and Solovay in [BGS75].

Theorem 1.20. $\mathbf{P} \not\subseteq \mathbf{NP}$ *does not relativize.*

Proof. By definition 1.7 it is enough to find an oracle O such that $\mathbf{NP}^O \subseteq \mathbf{P}^O$. So let O be some **PSPACE**-complete language. The inclusion $\mathbf{P}^O \subseteq \mathbf{NP}^O$ is trivial. Furthermore, it is clear by the definition of a **PSPACE**-complete language that $\mathbf{PSPACE} \subseteq \mathbf{P}^O$ (a Turing machine proving this inclusion would just reduce the **PSPACE** language in question using the fact that O is **PSPACE**-complete, and then query O) and $\mathbf{NP}^O \subseteq \mathbf{NPSPACE}$ (a polynomial-space nondeterministic Turing machine can solve any oracle query from a Turing machine deciding a language in $\mathbf{NP}^O$ by directly simulating a Turing machine deciding O). Then, Corollary 1.16 gives that, $\mathbf{NP}^O \subseteq \mathbf{NPSPACE} = \mathbf{PSPACE} \subseteq \mathbf{P}^O$, and hence $\mathbf{P}^O = \mathbf{NP}^O$. $\square$

Relativization is a valuable technique for research in Complexity Theory, for it may expose that certain approaches do not work to settle some conjecture. For an interesting survey on the role of relativization in Complexity Theory check [For94]. The proof of Theorem 1.20 will also be a key result to prove that $\mathbf{NP} \not\subseteq \mathbf{P}$ does not algebrize in the sense of Section 4.

2. Circuit Lower Bounds

Relativization exposed a need to find non-relativizing techniques for proving separation results. One insight that was key for later success is that diagonalization does not say anything about the inner workings of a Turing machine, instead treating it as a black box that receives an input and spits some function applied to the input. There was hope that by actually going through a hands-on approach that directly analyzed this inner workings one could prove these separation results. But actually using this hands-on approach directly on Turing machines is hard, for Turing machines, despite the simplicity of their definition, work in complicated ways. Hence, by looking to other models of computations, which might be easier to apply this hands-on approach, one could try to establish some result that implies the desired separation result. Circuits are a simple model of computation that allows that. In fact, it turns out that proving circuit lower bounds for functions frequently implies some separation result. In particular, this section exposes how proving lower bounds could separate **P** from **NP**.

The circuits considered here are very simple structures.

Definition 2.1. An n-input, single-output Boolean circuit C is a directed labeled acyclic graph with n sources, labeled as variable nodes, and a single sink node with only one incoming edge, labeled as an output node. All other nodes are called gates and are labeled as either $\wedge$, $\vee$ or $\neg$. Those labeled with $\wedge$ or $\vee$ have fan-in 2 and those labeled with $\neg$ have fan-in 1. The size $|C|$ of C is the number of nodes in C. With input $x \in \{0,1\}^n$ the output of C under x, $C(x)$, is the result of applying the logic operators in the label of each node to the bits corresponding to its incoming edges. That is, define recursively the function $out(v)$, that gives the value of a node v, as the value of the node itself if v is a variable gate, the value of the single node connected to v if it is an output node, and the value of applying the logical operator in the label of v to the bits corresponding to the values of the nodes in its in-neighborhood if it is a gate. $C(x)$ is then equal to the value $out(v_{\text{out}})$ of the output node v_{out}.

Circuits as defined above are the basic structures of this section. In order to define $\mathbf{P}/poly$, the definition of a family of circuits is necessary.

Definition 2.2. Let $S : \mathbb{N} \to \mathbb{N}$ be some function. An $S(n)$-size circuit family is a collection of Boolean circuits $\mathcal{C} = \{C_n\}_{n \in \mathbb{N}}$, where, for every $n \in \mathbb{N}$, C_n is an n-input, single output Boolean circuit of size $|C_n| \leq S(n)$.

Now, circuity families can be used to define the complexity classes **SIZE**, which play a similar role to **DTIME** in this section.

Definition 2.3. Let $S : \mathbb{N} \to \mathbb{N}$ be some function. $\mathbf{SIZE}(S(n))$ is the set of all languages $L \subseteq \{0,1\}^*$ such that there exists an $O(S(n))$-size circuit family such that for every $n \in \mathbb{N}$ it holds that for every $x \in \{0,1\}^n$, $x \in L$ if and only if $C_n(x) = 1$.

Now, in a similar vein to the definition of **P**, the definition of $\mathbf{P}/poly$ follows.

Definition 2.4.
$$\mathbf{P}/poly = \cup_{c \geq 0} \mathbf{SIZE}(n^c)$$

Like **P** the class $\mathbf{P}/poly$ corresponds to problems that are in some sense tractable. However, the two classes are not the same.

Lemma 2.5.
$$\mathbf{P} \neq \mathbf{P}/poly$$

Proof. To prove this it is enough to prove that there exists some language $L \subseteq \{0,1\}^*$ such that $L \in \mathbf{P}/poly$ but $L \notin \mathbf{P}$. The following constructs such a language.

First, note that if U is a unary language, i.e. $U = \{1^n : n \in S \subseteq \mathbb{N}\}$, then U is in $\mathbf{P}/poly$. For this, construct the following circuit family $\{C_n\}$. If $n \in S$ then let C_n be the conjunction of all n variable nodes (to make this with fan-in 2 one just builds a tree with $n-1$ $\wedge$ gates $\{g_i\}$ such that g_i is the conjunction of the $i+1$ variable with g_{i-1}), else let C_n be the empty circuit (i.e., the circuit that always outputs 0). Then, clearly, $\{C_n\}$ decides U. On the other hand, for every $n \in \mathbb{N}$, $|C_n| \leq n$ and therefore $U \in \mathbf{P}/poly$.

Now, define the following language, called unary halt.
$$\text{UHALT} = \{1^n : \llcorner n \lrcorner = (\llcorner M \lrcorner, \llcorner x \lrcorner) \text{ where } (\llcorner M \lrcorner, \llcorner x \lrcorner) \in \text{HALT}\}$$

Since UHALT is unary, it follows that UHALT $\in \mathbf{P}/poly$. On the other hand, UHALT is undecidable for Turing machines. For the proof, let's reduce HALT to UHALT through a Turing machine M' with oracle access to UHALT defined as follows. On input (M, x), M' converts (M, x) to the corresponding natural number n and then queries 1^n in the query tape, then, M' outputs 1 if $1^n \in$ UHALT and 0 otherwise. Clearly, $M'(M, x) = 1$ if and only if $(M, x) \in$ HALT and therefore, indeed HALT reduces to UHALT. Now, if UHALT is decidable then so is HALT, a contradiction, hence UHALT is undecidable. In particular, UHALT $\notin \mathbf{P}$. □

This suggests that $\mathbf{P}/poly$ may be strictly more powerful than $\mathbf{P}$. Indeed, the inclusion will now be proved along with a corollary that gives a way of separating $\mathbf{P}$ from $\mathbf{NP}$. To prove it, it is necessary to first give the following definition.

Definition 2.6. An oblivious Turing machine with k tapes is a Turing machine M such that there exists a function $h : \mathbb{N}^2 \to \mathbb{N}^k$ such that for every $x \in \{0,1\}^*$, $h(|x|, t)$ is equal to the k-tuple where the ith coordinate gives the position of the head of the ith tape of M at step t of the computation M does with input x.

Note that h only depends on the size of x. Intuitively, an oblivious Turing machine does the same tape head movements for any given input of fixed size. This way, one has a log of the movements of the machine before it even runs, for one just needs to run it on one input of fixed length to know how it moves for every input of that length. For an oblivious Turing machine, it is as if the position of the machine's tape heads were independent from the input. The following result is of relevance to proving the inclusion of $\mathbf{P}$ in $\mathbf{P}/poly$.

Lemma 2.7. *Let $T : \mathbb{N} \to \mathbb{N}$ be some function. For every $T(n)$-time Turing machine M there exists an oblivious Turing machine M' running in $O(T(n)\log T(n))$ that decides the same language decided by M.*

It is fairly simple to prove Lemma 2.7 weakened to yield a quadratic overhead. The logarithmic overhead follows from using the same buffering technique from Lemma 1.2. In meager terms, the way one proves this lemma is by having the tape heads of M' swipe all k tapes from the first position up to the $O(T(n))$ position and then swipe back, so that the position of the tape heads is always determined. Whenever M' is in a cell where M does some computation, M' does so and saves the position that the tape heads of M would be in. Now, let's proceed to prove the inclusion of $\mathbf{P}$ in $\mathbf{P}/poly$.

Theorem 2.8.
$$\mathbf{P} \subsetneq \mathbf{P}/poly$$

Proof. Let $L \in \mathbf{P}$. Since $L \in \mathbf{P}$, it follows that there exists a polynomial time Turing machine that decides L. Then we can assume that M is an oblivious Turing machine, because otherwise Lemma 2.7 gives such a machine still running in polynomial time. Hence, say M runs in time $p(n)$ for some polynomial $p : \mathbb{N} \to \mathbb{N}$. Then, since L was chosen arbitrarily among all languages in $\mathbf{P}$, it is enough to show that there exists a $O(p(n))$-size circuit family $\{C_n\}$ such that $C_{|x|}(x) = M(x)$ for every $x \in \{0,1\}^*$.

This will be done through the concept of a transcript of a Turing machine execution. The transcript of M on input x is a sequence of snapshots $\{s_i\}_{i \in [p(|x|)]}$, in which, for each i, s_i represents the machine's current state, and the symbol read by all tape heads. Clearly, this sequence completely determines the execution of M. Furthermore let $b(t)$ be the k-tuple $(h_1, \dots, h_k)$ where h_j is the last time step when M's jth tape head was at the same position as it is at step t (and let $b_j(t)$ be the jth coordinate of $b(t)$). Then, since M is deterministic s_i is completely determined by s_{i-1}, $s_{b_1(i)}$, …, $s_{b_k(i)}$ and x. For, if in two different time steps $i < i'$ all of these quantities are the same the machine has entered a loop, because it will just repeat every snapshot that led from z_i to $z_{i'}$ due to its deterministic behavior and the fact that these provide everything that the transition functions need to determine the next state. In particular, there exists a function F such that for every

i, $s_i = F(s_{i-1}, s_{b_1(i)}, \ldots, s_{b_k(i)}, x)$. It is well known that every Boolean function can be written as a predicate involving only the operators $\wedge$, $\vee$ and $\neg$. Hence, every Boolean function can be written as a circuit as defined in Definition 2.1. In particular, since a snapshot has some constant length K in the size of the input and there are only 2^K strings of size K, it follows that for every i there is a circuit C'_i of constant size that computes s_i from $s_{i-1}, s_{b_1(i)}, \ldots, s_{b_k(i)}$ and x. Hence, if one connects each of these circuits, one gets a circuit that computes $s_{p(|x|)}$ on input x. Now, one can build a constant sized circuit that outputs 1 if $s_{p(|x|)}$ is in the accept state and 0 otherwise. Composing all of these circuits yields a circuit of size $O(p(n))$ that decides M. In particular $L \in \mathbf{P}/poly$.

The strict inclusion follows from Lemma 2.5. □

The following is direct from Theorem 2.8.

Corollary 2.9. *If* $\mathbf{NP} \not\subseteq \mathbf{P}/poly$ *then* $\mathbf{P} \neq \mathbf{NP}$.

This is enough to show that proving a superpolynomial lower bound for the circuit size of **NP** would separate **P** from **NP**. Unfortunately, as will be exposed in Section 3, no "natural proof" can achieve such a separation.

3. Natural Proofs

Despite the hope in circuit lower bounds as a method of proving separation results, and the early results proving some lower bounds on particular complexity classes, little progress was made towards the big separation results. This hope was all but destroyed in a 1999 article by Razborov and Rudich where they define the concept of a natural proof. With natural proofs defined, they show that all circuit lower bound results by the time of the publication of the article were natural proofs. Finally they prove that under a widely believed cryptography conjecture, natural proofs can't prove super-polynomial lower bounds, what will henceforth be called the self-defeatism of natural proofs.

3.1. Cryptography.

In order to prove the self-defeatism of natural proofs, some concepts from cryptography need to be defined and some related results presented. Cryptography relies heavily on the existence of an one-way function, for such a function is necessary for many secure cryptographic systems. The definition of a one-way function follows.

Definition 3.1. Let $h : \mathbb{N} \to \mathbb{N}$ be some function. A function $f : \{0,1\}^* \to \{0,1\}^*$ is a h-hard one-way function if it is polynomial-time computable and, for every $h(n)$-time probabilistic Turing machine E and for every $n \in \mathbb{N}$

$$\Pr_{x \in_R \{0,1\}^n \ \wedge \ y = f(x)} [E(y) = x' \text{ such that } f(x') = y] < \frac{1}{n^{\omega(1)}}$$

Functions $\epsilon : \mathbb{N} \to [0,1]$ such that $\varepsilon(n) = n^{-\omega(1)}$ are called negligible. Intuitively, an h-hard one-way function is a function that can be computed efficiently, but that no h-time algorithm can invert in more than a negligible fraction of all possible inputs. It is an open problem whether or not a polynomially-hard one-way function exists as well as whether or not subexponentially-hard one way functions exist. On the other hand, the following result indicates how hard it is to prove they exist.

Theorem 3.2. *If there exists a polynomially-hard one-way function then* $\mathbf{P} \neq \mathbf{NP}$.

Proof. Consider the contrapositive. That is, assume $\mathbf{P} = \mathbf{NP}$ and then let's prove no polynomially-hard one-way function exists. For that, let $f : \{0,1\}^* \to \{0,1\}^*$ be a function that can be computed in polynomial time. Since f is polynomially-time computable for every x, there exists a polynomial p such that $|f(x)| \leq p(|x|)$. It then follows that there exists a polynomial q such that $|x| \leq |q(f(x))|$. Hence, consider the following nondeterministic Turing machine M. On input y, M computes $|q(y)|$ and then nondeterministically guesses $f(x')$ for every x' such that $|x'| \leq |q(y)|$. It then outputs x' if $f(x') = y$ and outputs the blank output otherwise. Since f is polynomial time computable, and q is a polynomial it follows that M runs in polynomial time. Furthermore, since it makes guesses for every x' such that $|x'| \leq |q(y)|$ and if there exists x such that $f(x) = y$ then $|x| \leq |q(y)|$, M outputs a nonblank tape if and only if $f(M(y)) = y$. Inverting f is thus an $\mathbf{NP}$ problem. Now, if $\mathbf{P} = \mathbf{NP}$ then, there exists a machine M' running in polynomial time such that for all y, it holds that $M'(y) = M(y)$. Therefore, M' inverts f for every input. In particular

$$\Pr_{x \in_R \{0,1\}^n \wedge y = f(x)}[M'(y) = x' \text{ such that } f(x') = y] = 1$$

and therefore no polynomially computable function f is polynomially-hard. □

It is widely believed that polynomially-hard one-way functions exist, and there are many candidate functions such as the RSA function and Integer Factorization, among others. Of particular relevance for natural proofs are subexponentially-hard one-way functions, which are also believed to exist but no function has been proven to be so; known candidates include functions based on the discrete logarithm.

The relevance of one-way functions is clear by their definition. If somehow a one-way function is used to encode a message, than decoding it would involve inverting the one-way function, which, in case the one-way function is polynomially-hard, is only possible to be done efficiently to a negligible fraction of the set of encoded messages. Most cryptography systems though, do not use directly a one-way function, but rather use a pseudorandom generator or a pseudorandom family of functions, both defined below.

Definition 3.3. Let $h : \mathbb{N} \to \mathbb{N}$ be some function. Furthermore, let $\ell : \mathbb{N} \to \mathbb{N}$ be a function such that $\ell(n) > n$ for every n. A polynomial-time computable function $G : \{0,1\}^* \to \{0,1\}^*$ is an h-hard pseudorandom generator of stretch $\ell(n)$ if

1. $|G(x)| = \ell(|x|)$ for every $x \in \{0,1\}^*$
2. For every $h(n)$-time probabilistic Turing machine E there exists a negligible function ϵ such that

$$\left|\Pr[E(G(U_n)) = 1] - \Pr[E(U_{\ell(n)})) = 1]\right| < \epsilon(n)$$

   where U_n is the uniform random distribution on binary strings of length n.

Property (1) above is the stretch property, which means that G is able to stretch true random strings of length n to strings of length $\ell(n)$. Property (2), the security property, guarantees that those stretched random strings are random enough. The existence of pseudorandom generators is also an open problem, and in fact a pseudorandom generator exists if and only if one-way functions exist. Only the reverse direction will be discussed here, but no proof will be provided. Now, pseudorandom families of functions are defined similarly to pseudorandom generators.

Definition 3.4. Let $h : \mathbb{N} \to \mathbb{N}$ be some function. A family of functions $\{f_k\}_{k\in\{0,1\}^*}$ where $f_k : \{0,1\}^{|k|} \to \{0,1\}^{|k|}$ is an h-hard pseudorandom family of functions if

(1) For every $k \in \{0,1\}^*$, f_k is polynomial-time computable.
(2) For every h-time probabilistic oracle Turing machine E there is a negligible function ϵ such that for every n

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

where $\mathcal{F}_n$ is the set of all functions from $\{0,1\}^n \to \{0,1\}^n$

Similarly to the definition of pseudorandom generators, property (2) guarantees its security. Now, it is also true that pseudorandom generators exist if and only if pseudorandom families of functions exist. The following sequence of lemmas prove that from a one-way function one can construct an pseudorandom family of functions. The first such result was proved by Håstad, Impagliazzo, Levin and Luby in [HILL99]

Lemma 3.5. *If there exists a polynomially-hard one-way function then there exists a polynomially-hard pseudorandom generator of arbitrary stretch.*

They prove so through a series of reductions, with which one can verify that starting with a subexponentially-hard one-way function one constructs a subexponentially-hard pseudorandom generator. With a pseudorandom generator, one can then construct a pseudorandom family of functions. This was proven in the article [GGM86] by Goldreich, Goldwasser and Micali.

Lemma 3.6. *If there exists a polynomially-hard pseudorandom generator of stretch $\ell(n) = 2n$ then there exists a polynomially-hard pseudorandom family of functions.*

In fact, the construction in the proof of Lemma 3.6 given in [GGM86], without change, also gives that if one starts with a subexponentially-hard pseudorandom generator of stretch $\ell(n) = 2n$ one obtains a subexponentially-hard pseudorandom family of functions.

3.2. **Natural Proofs are Self-Defeating.**

With the fact that the existence of subexponentially-hard one-way functions imply the existence of subexponentially-hard pseudorandom families of functions, we can proceed to define natural proofs and prove that natural proofs are self-defeating. The reason for calling this result the self-defeatism of natural proofs is because it will be proven that with a natural proof proving hardness one is able to destroy the hardness of pseudorandom families of functions. Here, Arora's and Barak's construction of natural proofs from [AB09] will be followed, which is slightly different from the original construction by Razborov and Rudich from [RR97]. But, even though it loses a little in generality, it does gain a lot in easiness of exposition. The definition of a natural proof follows.

Definition 3.7. Let $\mathcal{P}$ be a predicate on Boolean functions. $\mathcal{P}$ is called a natural combinatorial property if it satisfies the following.

(1) Constructiveness: There exists a $2^{O(n)}$-time Turing machine that on input the truth table of some Boolean function $g : \{0,1\}^n \to \{0,1\}$ outputs $\mathcal{P}(g)$.
(2) Largeness: $\Pr_{g\in_R B_n}[\mathcal{P}(g) = 1] \geq \frac{1}{n}$, where B_n is the set of all Boolean functions from $\{0,1\}^n$ to $\{0,1\}$.

Furthermore, let $h : \mathbb{N} \to \mathbb{N}$ be some function. A predicate $\mathcal{P}$ is $h(n)$-useful for proving a lower bound on some family of Boolean functions $\{f_k\}_{k\in\mathbb{N}}$ if for every k, $\mathcal{P}(f_k) = 1$ but for every Boolean function $g \in \mathbf{SIZE}(h(n))$ it holds that $\mathcal{P}(g) = 0$.

The reasoning behind the definition is that a "natural" proof of a lower bound involves finding some combinatorial property that implies hardness, and holds for the target function but not for functions of the size one wants to bound. Such a proof clearly needs to satisfy $h(n)$-usefulness for the desired lower bound h. Now, the interest in constructiveness is, according to Razborov and Rudich in [RR97], justified by empirical claims. That is, "the vast majority of Boolean functions [...] that one encounters in combinatorics are at worst exponential-time decidable". Largeness is somewhat weird on first sight, but it turns out to be natural, and Razborov and Rudich prove in [RR97] how any complexity measure bounding one Boolean function must bound a large fraction of all Boolean functions. Therefore, a property that lower bounds one function should also lower bound a non-negligible fraction of all Boolean functions. A natural proof is harder to define. Informally, it is a proof that relies on some natural combinatorial property that is $h(n)$-useful to prove an h bound on the size of some family of Boolean functions. An example of natural proof can be seen in the proof of $\mathbf{AC}^0 \neq \mathbf{NC}^1$ originally proven in [FSS81]. The main result of this section, namely the fact that natural proofs are self-defeating, follows.

Theorem 3.8. *Suppose a subexponentially-hard one-way function exists. Then there exists a polynomial p such that no natural combinatorial property $\mathcal{P}$ is $p(n)$-useful.*

Proof. Consider the contrapositive. That is, assume $\mathcal{P}$ is a natural combinatorial property. Then, for the sake of contradiction, assume there exists a subexponentially-hard one-way function. Then Lemma 3.5 constructs from this one-way function a subexponentially-hard pseudorandom generator, and then Lemma 3.6 constructs from this pseudroandom generator a subexponentially-hard pseudorandom family of functions $F = \{f_k\}_{k\in\mathbb{N}}$ where, for $k \in \{0,1\}^m$, $f_k : \{0,1\}^m \to \{0,1\}$.

The predicate $\mathcal{P}$ will be used to break the security of F. For that, since F is subexponentially-hard it is 2^{m^ε}-secure for every ε (in the sense that no probabilistic Turing machine running in time $O(2^{m^\varepsilon})$ can distinguish oracle access to a function from F from oracle access to a random Boolean function). Hence, define M, which will be the oracle Turing machine with oracle access to some function $g : \{0,1\}^m \to \{0,1\}$ breaking the security of F, as follows. Machine M sets $n = m^{\varepsilon/2}$ and then constructs the truth table of $h : \{0,1\}^n \to \{0,1\}$ defined as $h(x) = g(x0^{m-n})$. The padding guarantees that the truth table has size $2^{O(n)}$ and hence, due to M having oracle access to g, can be constructed in $2^{O(n)}$ time. Then, M computes $\mathcal{P}(h)$ and outputs $\mathcal{P}(h)$. Since $\mathcal{P}$ is constructive, it also can be computed in $2^{O(n)}$, and therefore M runs in $2^{O(n)}$-time.

Now, consider the possibilities for g. If g is some random function then so is h. Due to the largeness of $\mathcal{P}$, M outputs 1 with probability $1/n$. On the other hand, if $g = f_k$ for some $f_k \in F$, $g(x)$ can be computed in polynomial time for every x. Therefore, $g \in \mathbf{P}/poly$, and hence, there exists a polynomial q over m such that g can be computed in $q(m)$-time. In particular, there exists a polynomial p over n such that $q(m) = p(n)$. Hence, $g \in \mathbf{SIZE}(p(n))$ and in particular $h \in \mathbf{SIZE}(p(n))$. Therefore, if $\mathcal{P}$ is $p(n)$-useful then $\mathcal{P}(h) = 0$. Therefore, if $\mathcal{P}$ were $p(n)$ useful, then

M is able to distinguish a function from F from a random Boolean function with probability at least $1/n$, which is non-negligible, and does so in $2^{O(n)} = 2^{O(m^{\varepsilon/2})}$ time, thus breaking the subexponential-security of F.

Hence, if subexponential one-way functions exist, then subexponentially-hard pseudorandom family of functions exist, and therefore no natural combinatorial property may be useful for every polynomial. □

In particular, this proves that natural proofs can't prove superpolynomial lower bounds, and hence can't separate **P** from **NP**. Natural proofs then give another barrier to proving separation results in Complexity Theory. In particular, similar to relativization, one can inquire if some proof naturalizes (meaning that it can be tweaked so as to fit this framework). In case it does, the researcher then knows that this technique can't be used to prove superpolynomial bounds. Furthermore, notice that diagonalization fails to naturalize, because it doesn't satisfy largeness, since it focuses on how a single function works over several machines. Clearly a machine such as that implementing Algorithm 1 in Theorem 1.4 is too specific for the properties it has using diagonalization to extend to a non-negligible ammount of random functions.

4. Algebrization

After Razborov and Rudich introduced the concept of natural proofs, and proved that they are self-defeating, some lower bounds were proven that failed to relativize and failed to naturalize. This was done by using both non-relativizing and non-naturalizing techniques. One such non-relativizing technique was the ability to "arithmetize" a Boolean function φ, that is, find a polynomial $p : \mathbb{F}_2 \to \mathbb{F}_2$ such that $p = \varphi$, and then extend it to bigger fields, so that the extended polynomial $\tilde{p}$ would hold more information than p or φ. This again created some hope that these arithmetization methods, together with non-naturalizing techniques such as diagonalization, could be able to prove some of the main separation results in Complexity Theory. Yet again, these methods turn out to not be enough, despite being able to hoodwink both naturalization and relativization. This was proven by Aaronson and Wigderson in [AW08], where they introduce a generalized version of relativization, called "algebrization", prove that all of these new lower bounds algebrize (in similar sense to saying that they relativize) and then prove that **P** = ?**NP** does not algebrize. This section then explores this third barrier. First a result that neither relativizes nor naturalizes is proved, that **IP** = **PSPACE**, which despite being proven before Razborov's and Rudich's article about natural proofs, is the result that allowed Santhanam in [San07] to overcome both the barriers imposed by relativization and naturalization. Then, algebrization is introduced, **IP** = **PSPACE** is proven to algebrize and **P** =?**NP** is proven to not algebrize.

4.1. IP = PSPACE.

PSPACE was introduced in section 1.3, hence it is only necessary to introduce **IP** to understand the statement **IP** = **PSPACE**. **IP** is the class of interactive proofs, which are protocols where a prover interacts with a verifier with the goal of proving to the verifier that something is true. The prover has unlimited computational power, while the verifier uses a polynomial-time probabilistic Turing machine. This kind of interaction has a similar structure to cryptographic protocols, internet protocols and program checking. To define **IP** it is first necessary to define what is meant by an interaction.

Definition 4.1. Let $f, g : \{0,1\}^* \to \{0,1\}^*$ be functions. A k-round interaction $\langle f, g\rangle$ of f and g on input $x \in \{0,1\}^*$, for some $k > 0$, is the k-tuple $(r_1, \dots, r_k)$ where

$$\begin{aligned}
&r_1 = f(x)\\
&r_2 = g(x, r_1)\\
&r_3 = f(x, r_1, r_2)\\
&\vdots\\
&r_{2i} = g(x, r_1, \dots, r_{2i-1}) \text{ where } 2i \leq k\\
&r_{2i+1} = f(x, r_1, \dots, r_{2i}) \text{ where } 2i+1 \leq k.
\end{aligned}$$

The output $O_f\langle f, g\rangle(x)$ of f on input x is defined as

$$O_f\langle f, g\rangle(x) = f(x, r_1, \dots, r_k)$$

where it is assumed that $f(x, r_1, \dots, r_k) \in \{0,1\}$

This allows **IP** to be defined.

Definition 4.2. For $k \geq 1$, $\mathbf{IP}[k]$ is the complexity class of all languages L such that there is a k-round interaction between a probabilistic polynomial-time Turing machine V (the veriefier) and a function $P : \{0,1\}^* \to \{0,1\}^*$ such that

(1) $x \in L \Rightarrow \exists P$ such that $\Pr[O_V\langle V, P\rangle(x) = 1] \geq \frac{2}{3}$
(2) $x \notin L \Rightarrow \forall P$ it holds that $\Pr[O_V\langle V, P\rangle(x) = 1] \leq \frac{1}{3}$

Property (1) is called Completeness while property (2) is called Soundness. Furthermore,

$$\mathbf{IP} = \cup_{c\geq 1}\mathbf{IP}[n^c]$$

The reason for choosing probabilistic Turing machines is because the analogous deterministic version of **IP** is equal to **NP**. This is because if V is deterministic, each round can be expressed as a sequence of nondeterministic choices. For a proof, see [AB09]. A surprising fact about interactive proofs, is that despite P having no bound on its power (it can even compute uncomputable functions), **IP** has bounded power due to the limitations on V, as seen in Lemma 4.3.

Lemma 4.3.

$$\mathbf{IP} \subseteq \mathbf{PSPACE}$$

Proof. Let $L \in \mathbf{IP}$. Then, there is an n^c-round interaction between some verifier V and some prover P satisfying Definition 4.2. For the proof, a Turing machine M will be defined working in polynomial space that computes the maximum probability that an n^c interaction between any prover will end with the verifier accepting. For that, let T be a tree such that the ith level corresponds to the ith round of interaction and each node in the ith level corresponds to a possible message to be sent at that round. Then, M recursively labels T, so that for a node v in level i, its label is the maximum probability that V accepts at round i. For that, M labels the leaves of T with 1 or 0 by directly simulating V. Then, if i is a level where V would send a message, the label of a node in the ith level is the average over the values of its children weighed over the probability that V would send each message. If i is a level where P would send a message, the label of a node in the ith level is the maximum of all children. M then outputs 1 if the root of T is labeled with a value bigger or equal to 2/3 and 0 otherwise. Checking that this recursion indeed makes every label the maximum probability that V accepts clearly follows by induction. Furthermore, since $L \in \mathbf{IP}$, T has polynomial depth and since V runs in polynomial time each node has at most $2^{n^{c'}}$ for some constant c'. Hence, all these operations can be carried over in polynomial space. Hence M runs in polynomial space and decides L, proving the lemma. □

The other direction, namely $\mathbf{PSPACE} \subseteq \mathbf{IP}$ is quite surprising, since it completely characterizes $\mathbf{IP}$ as a well-known complexity class that is seemingly unrelated. Furthermore, its proof introduces techniques not seen in any of the previous results.

Theorem 4.4.

$$\mathbf{IP} = \mathbf{PSPACE}$$

Proof. The direction $\mathbf{IP} \subseteq \mathbf{PSPACE}$ follows from Lemma 4.3.

For the direction $\mathbf{IP} \supseteq \mathbf{PSPACE}$, note that since TQBF is $\mathbf{PSPACE}$-complete, by Theorem 1.19 and $\mathbf{IP}$ is clearly closed under polynomial-time reductions (both the prover and the verifier can perform them) it suffices to prove that TQBF $\in \mathbf{IP}$. For that, a technique called arithmetization will be used. It consists of turning QBFs into low-degree polynomials.

For that, let $\psi = Q_1 x_1 \dots Q_n x_n \varphi(x_1, \dots, x_n)$ be QBF as defined in Definition 1.18. Then, one can recursively make a polynomial from QBF using the following transformations.

(1) Substitute every instance of x_i with z_i, a variable over $\mathbb{Z}$.
(2) Substitute $\neg x_i$ with $(1 - z_i)$.
(3) Substitute $x_i \wedge x_j$ with $z_i \cdot z_j$.
(4) Substitute $x_i \vee x_j$ with $z_i + z_j$
(5) Substitute $\forall x_i$ with $\prod_{z_i \in \{0,1\}}$
(6) Substitute $\exists x_i$ with $\sum_{z_i \in \{0,1\}}$

Call the polynomial obtained from ψ through these substitutions the arimethization of ψ. It is not hard to check that a QBF ψ, as defined in Definition 1.18, is true if and only if the arimethization of ψ is nonzero (The proof is a simple induction over first φ and then over the quantifiers of ψ). But this polynomial has degree exponential in n, which does not allow it to be processed by the polynomial-time verifier. Hence, it is necessary to reduce its degree somehow.

Define the following operator over polynomials:

$$\begin{aligned} L_i p(z_1, \dots, z_n) = {} & z_i \cdot p(z_1, \dots, z_{i-1}, 1, z_{i+1}, \dots, z_n) \\ & + (1 - z_i) p(z_1, \dots, z_{i-1}, 0, z_{i+1}, \dots, z_n) \end{aligned}$$

In particular, $L_i p$ is linear over z_i. Clearly, this linearization will only agree with the original polynomial p when $z_i \in \{0, 1\}$. Similarly, define the operators

$$\forall x_i p(z_1, \dots, z_n) = p(z_1, \dots, z_{i-1}, 1, z_{i+1}, \dots, z_n) \cdot p(z_1, \dots, z_{i-1}, 0, z_{i+1}, \dots, z_n)$$

$$\exists x_i p(z_1, \dots, z_n) = p(z_1, \dots, z_{i-1}, 1, z_{i+1}, \dots, z_n) + p(z_1, \dots, z_{i-1}, 0, z_{i+1}, \dots, z_n)$$

Then, if $\psi = Q_1 x_1 \dots Q_n x_n \varphi(x_1, \dots, x_n)$ is a QBF, and p_φ is the arimethization of φ then one can define the following arimethization of ψ,

$$P(z_1, \dots, z_n) = Q_1 x_1 L_1 Q_2 x_2 L_1 L_2 \dots L_{n-1} Q_n L_1 L_2 \dots L_n p_\varphi(z_1, \dots, z_n)$$

where the quantifiers Q_i are seen as the operators defined above. It is not hard to check that applying any of the operators L_i, $\forall x_i$ and $\exists x_i$ makes a polynomial that agrees with the one obtained using substitutions 1-6 on $\{0,1\}$, so P is indeed an arimethization of ψ. In particular, ψ is true if and only if P is nonzero. Furthermore, since after applying any of the operators $Q_i x_i$ in the expression of P, $Q_i x_i$ is being applied to a multilinear polynomial, it follows that the size of P is $O(1+2+\ldots+n) = O(n^2)$. Furthermore, the degree of P is bounded by $3m$ where m is the number of clauses of φ (this is because p_φ clearly has degree at most $3m$, and any intermediate polynomial after applying $L_1 \ldots L_n$ has degree at most 2).

The **IP** protocol is then defined inductively over the operators in the expression of P. In the protocol the prover has to convince a verifier that some polynomial q satisfies $q(b_1, \ldots, b_n) = c$ with probability 1 for any b_1, $\ldots$, b_n when it is true and with negligible probability when it is false. In each round, the verifier tries to drop one operator of the candidate for P. This is done as follows. In each round the prover sends a polynomial to the verifier, who then checks if it agrees with what the verifier expects, and sends a further check to the prover. At the start of the interaction, let $c_0 = 1$ and $p_0 = p$. Furthermore, the verifier chooses a prime $2^n < l < 2^{2n}$ (primality can be verified in polynomial-time using even a deterministic Turing machine as seen in [AKS04]), and henceforth all polynomials are considered as over $\mathbb{F}_l$. Since P can't evaluate to more than 2^n with $x_i \in \{0,1\}$, it follows that this doesn't impair generality but is necessary to make V be able to do the computations below. At step k, $P_k = Q_1 x_1 L_1 Q_2 \ldots O_{k+1} p_k$. There are three cases, and all cases start with the prover providing some polynomial $q_{k+1}(z_i)$ with degree d that it claims is $P_{k+1}(z_1, \ldots, z_i, b_{i+1}, \ldots, b_n)$. The cases are:

(1) $O_{k+1} = \exists x_i$. In this case, verifier checks if $q_{k+1}(0) + q_{k+1}(1) = c_k$, if not then the verifier rejects. Else, the verifier randomly chooses a number $b_i \in \mathbb{F}_p$, sets $c_{k+1} = q_{k+1}(b_i)$, and asks the prover to prove $c_{k+1} = P_{k+1}(z_1, \ldots, b_i, \ldots, b_n)$
(2) $O_{k+1} = \forall x_i$. In this case, verifier checks if $q_{k+1}(0) \cdot q_{k+1}(1) = c_k$, if not then the verifier rejects. Else, the verifier randomly chooses a number $b_i \in \mathbb{F}_p$, sets $c_{k+1} = q_{k+1}(b_i)$, and asks the prover to prove $c_{k+1} = P_{k+1}(z_1, \ldots, b_i, \ldots, b_n)$
(3) $O_{k+1} = L_i$. In this case, since b_i has already been chosen, verifier checks if $b_1 q_{k+1}(1) + (1 - b_1) q_{k+1}(0) = c_k$, if not then the verifier rejects. Else, the verifier randomly chooses a number $b_i \in \mathbb{F}_p$, sets $c_{k+1} = q_{k+1}(b_i)$, and asks the prover to prove $c_{k+1} = P_{k+1}(z_1, \ldots, b_j, \ldots, b_i, \ldots, b_n)$

It is clear by the choice of l that everything V needs to compute can indeed be computed in polynomial time. Now, it is only necessary to check completeness and soundness. Assume ψ is satisfiable. For completeness, notice that assuming the prover is being honest, in case 1, 2, 3, the verifier won't reject, and no matter what b_i is chosen, equality will hold. Hence, inductively, at each step in case ψ is satisfiable, the verifier does not reject at that round with probability 1. This implies that after all the $O(n^2)$ rounds of interaction, if ψ is satisfiable, V accepts with probability 1 and therefore completeness is satisfied. For soundness, assume ψ is not satisfiable. Then notice that if the prover is honest, then V always rejects. Now, assume the prover sends some polynomial q_{k+1} such that $q_{k+1} \neq P_{k+1}$. Since the degree of both q_{k+1} and P_{k+1} is d, it follows that $q_{k+1} - P_{k+1}$ has at most d roots, and hence there are at most d values b such that $q_{k+1}(b) = P_{k+1}(b)$. Therefore, the probability that the verifier does not reject is at most d/l. Then, in the first n rounds the probability that the verifier does not reject is bounded above by $3m/l$, and then at most $2/l$ for any of the following operators, so that the probability that the verifier accepts wrongfully is bounded by,

$$\frac{3mn}{2^n} + \sum_{i=1}^{n} \frac{2}{2^n} = \frac{3mn + n(n+1)}{2^n} = O\left(\frac{mn + n^2}{2^n}\right),$$

proving soundness. □

4.2. **Algebrization and P =?NP.** The proof of **IP** = **PSPACE** is quite notable, for it is known that **IP** = **PSPACE** does not relativize because there exists an oracle such that $\mathbf{coNP}^O \not\subseteq \mathbf{IP}^O$ as seen in [FS88]. After it was proven, it was used to prove some lower bounds that neither relativize or naturalize. In particular, Santhanam uses **IP** = **PSPACE** together with diagonalization to prove one such result in [San07]. These mixed methods can't actually settle many of the main separation conjectures as proved by Aaronson and Wigderson in [AW08]. This is done by introducing a more general version of relativization, that takes into account the possibility of arithmetizing Boolean functions. The definition is similar to that of relativization as seen in Section 1.2.

Definition 4.5. Let O be some oracle. Then, let $O_n = \{x \in O : \ x \in \{0,1\}^n\}$. Then, let $\mathbb{F}$ be some finite field. Then, an extension of O_n over the finite field $\mathbb{F}$ is $\tilde{O}_{n,\mathbb{F}} : \mathbb{F}^n \to \mathbb{F}$ such that $\tilde{O}_{n,\mathbb{F}}(x) = \mathbb{1}_{O_n}$ for every $x \in \{0,1\}^n$. Furthermore, the extension $\tilde{O}$ of an oracle O, is a collection of polynomials $\tilde{O}_{n,\mathbb{F}} : \mathbb{F}^n \to \mathbb{F}$ over all $m \in \mathbb{N}$ and all finite fields, where

(1) $\tilde{O}_{n,\mathbb{F}}$ is an extension of O_n for all n and all $\mathbb{F}$.
(2) The degree of $\tilde{O}_{n,\mathbb{F}}$ is uniformly bounded over all n and all $\mathbb{F}$.

This allows the definition of complexity classes with extensions of oracles.

Definition 4.6. Let **C** be some complexity class and O some oracle. Then $\mathbf{C}^{\tilde{O}}$ is the complexity class decidable by **C** machines that can query $\tilde{O}_{n,\mathbb{F}}$ for any finite field $\mathbb{F}$ and any positive integer n.

Algebrization, short for algebraic relativization, has a very similar definition to that of relativization, except for the fact that it is asymmetric.

Definition 4.7. Let **C** and **D** be complexity classes. Then, the inclusion $\mathbf{C} \subseteq \mathbf{D}$ is said to algebrize if for all oracles O and all oracle extensions $\tilde{O}$ of O, $\mathbf{C}^O \subseteq \mathbf{D}^{\tilde{O}}$. Similarly, the separation $\mathbf{C} \not\subseteq \mathbf{D}$ algebrizes if for all oracles O and all oracle extensions $\tilde{O}$ of O, $\mathbf{C}^O \subseteq \mathbf{D}^{\tilde{O}}$.

The justification given by Aaronson and Wigderson in [AW08] for the lack of symmetry in these definitions is that under the symmetrical but analogous definition of algebrization, it becomes strict enough so that they can't prove the algebrization of known results, while the main open separation results in Complexity Theory already algebrize under the asymmetrical version. And indeed, Aaronson and Wigderson do prove that **IP** = **PSPACE** algebrizes under this definition in [AW08]. It's proof will be omitted. Instead, the proof that **P** =?**NP** needs non-algebrizing techniques will be exposed. For it, some technical lemmas are necessary, and they are presented in the following.

Lemma 4.8. *Let $\mathbb{F}$ be a finite field and $F : \{0,1\}^n \to \mathbb{F}$ be some function. Then, F has a unique multilinear extension $\tilde{F}_{n,\mathbb{F}} : \mathbb{F}^n \to \mathbb{F}$*

Proof. Define $\delta_0(x) = 1 - x$ and $\delta_1(x) = x$. Then,

$$\tilde{F}_{n,\mathbb{F}}(x_1, \ldots, x_n) = \sum_{(a_1,\ldots,a_n)\in\{0,1\}^n} \prod_{i=1}^{n} F(a_1, \ldots, a_n)\delta_{a_i}(b_i).$$

It is clear by definition that $\tilde{F}_{n,\mathbb{F}}$ is indeed multilinear and extends F to $\mathbb{F}$. For uniqueness, suppose $Z : \mathbb{F}^n \to \mathbb{F}$ is multilinear and its restriction to $\{0,1\}^n$ is the zero constant function. It will be proven by induction that Z is in fact the zero function over $\mathbb{F}^n$. For that, let $\Delta(x_1, \ldots, x_n)$ be the number of variables x_i that are different from 0 and 1. Then, notice that if $\Delta(x_1, \ldots, x_n) = 0$, then $Z(x_1, \ldots, x_n) = 0$. Now suppose that if $\Delta(x_1, \ldots, x_n) = i$ then $F(x_1, \ldots, x_n) = 0$. Then, let $\Delta(x_1, \ldots, x_n) = i + 1$ and let $x_k \notin \{0,1\}$. Then, notice that by the induction hypothesis

$$F(x_1, \ldots, x_{k-1}, 0, x_{k+1}, \ldots, x_n) = F(x_1, \ldots, x_{k-1}, 1, x_{k+1}, \ldots, x_n) = 0.$$

But then, since F is linear over x_i it follows that $F(x_1, \ldots, x_n) = 0$. $\square$

The following is direct from the definition of $\tilde{F}_{n,\mathbb{F}}$ given in Lemma 4.8

Corollary 4.9. *If $\mathbf{S} = \{0, 1, \ldots, N-1\}$, $\tilde{F}_{n,\mathbb{F}} : \mathbf{S}^n \to \mathbb{F}$ is the multilinear extension of $F : \{0,1\}^n \to \mathbb{F}$, then, for all $x \in S^n$,*

$$|\tilde{F}(x)| \leq (2N)^n$$

In particular, for fixed N, if $F(x)$ is polynomially sized on input $x \in \{0,1\}^n$, then the multilinear extension of F over $\mathbf{S}^n$ is also polynomially sized.

This is enough to prove that $\mathbf{NP} \not\subseteq \mathbf{P}$ does not algebrize.

Theorem 4.10. $\mathbf{NP} \not\subseteq \mathbf{P}$ *does not algebrize.*

Proof. By Definition 4.7 it is enough to prove that there exists an oracle O such that $\mathbf{NP}^{\tilde{O}} \subseteq \mathbf{P}^{O}$. So let O be some **PSPACE**-complete language and $\tilde{O}$ be its multilinear extension. Then, by Corollary 4.9 it follows that $\tilde{O}$ is also in **PSPACE**. In particular, since $\tilde{O}$ agrees with O for queries involving only 0 and 1, it follows that $\tilde{O}$ is **PSPACE**-complete. Hence, similar to Theorem 1.20,

$$\mathbf{NP}^{\tilde{O}} \subseteq \mathbf{NPSPACE} = \mathbf{PSPACE} \subseteq \mathbf{P}^{O}.$$

$\square$

As in the case of Theorem 1.8, the proof that $\mathbf{NP} \subseteq \mathbf{P}$ does not algebrize is more complicated then the proof that $\mathbf{NP} \not\subseteq \mathbf{P}$ does not algebrize. In fact, the proof that $\mathbf{NP} \subseteq \mathbf{P}$ does not algebrize follows closely the proof that $\mathbf{NP} \subseteq \mathbf{P}$ does not relativize, but relies on some technical lemmas addressing oracle extensions. They are presented in the following.

Lemma 4.11. *Let $\mathbb{F}$ be some field, a_1, …, a_m be points in $\mathbb{F}^n$. Then, there exists a multiquadratic polynomial extension $p' : \mathbb{F}^n \to \mathbb{F}$ such that $p'(y) = 1$ for at least $2^n - m$ points $y = (y_1, \ldots, y_n) \in \{0,1\}^n$ where $y \neq a_i$ for all $i \in [m]$ and $p'(z) = 0$ for all $z \neq y$.*

Proof. First, let

$$p(x_1, \dots, x_n) = \sum_{(y_1,\dots,y_n)\in\{0,1\}^n} p(y) \prod_{i:y_i=1} x_i \prod_{j:y_j=0} (1 - x_j).$$

Clearly p is multilinear. Now, notice that $\{m(a_i) = 0\}_{i\in[m]}$ is a set of m linear equations relating the 2^n coefficients $p(y)$. It then follows that there is a way of satisfying the m constraints and having at least $2^n - m$ points y such that $m(y) = 1$.

Now, consider the following multiquadratic extension p' of p.

$$p'(x_1, \dots, x_n) = p(x_1, \dots, x_n) \prod_{i:y_i=1} x_i \prod_{j:y_j=0} (1 - x_j)$$

It is clear that p' satisfies all properties needed for the lemma. □

Lemma 4.11 is actually a special case of a more general phenomenon, as shown in Lemma 4.12.

Lemma 4.12. *Let $f : \{0,1\}^n \to \{0,1\}$ be a Boolean function, $\mathcal{F}$ be a collection of fields, and $p_{\mathbb{F}} : \mathbb{F}^n \to \mathbb{F}$ be a multiquadractic polynomial extending f over $\mathbb{F}$ for every $\mathbb{F} \in \mathcal{F}$. Furthermore, let $A_{\mathbb{F}} \subseteq \mathbb{F}^n$ for $\mathbb{F} \in \mathcal{F}$ and $m = \sum_{\mathbb{F}\in\mathcal{F}} |A_{\mathbb{F}}|$. Then, there exists $B \subseteq \{0,1\}^n$ such that $|B| \le m$ and for all Boolean functions $f' : \{0,1\}^n \to \{0,1\}$ that agree with f on B there exists multiquadratic polynomials $p'_{\mathbb{F}} : \mathbb{F}^n \to \mathbb{F}$, over all $\mathbb{F} \in \mathcal{F}$, that satisfy: (i) $p'_{\mathbb{F}}$ extends f' and (ii) $p'_{\mathbb{F}}(a) = p_{\mathbb{F}}(a)$ for all $a \in A_{\mathbb{F}}$.*

Proof. Let a point y be called good if for every $\mathbb{F} \in \mathcal{F}$, there exists a multiquadratic polynomial $q_{\mathbb{F},t} : \mathbb{F}^n \to \mathbb{F}$ such that $q_{\mathbb{F},y}(y) = 1$ and $q_{\mathbb{F},y} = 0$ or every $y' \neq y$. By Lemma 4.11 it follows for each $\mathbb{F}$ at most $|A_{\mathbb{F}}|$ points are bad. This implies that at least $2^n - m$ points are good. So let $B \subseteq \{0,1\}^n$ be the set of all bad points and G be the set of good points, so that for all $\mathbb{F} \in \mathcal{F}$ the following $p'_{\mathbb{F}}$ satisfies the lemma.

$$p'_{\mathbb{F}}(x) = p_{\mathbb{F}}(x) + \sum_{y\in G} (f'(y) - f(y)) q_{\mathbb{F},y}(x)$$

□

This is all that is necessary to prove that $\mathbf{NP} \subseteq \mathbf{P}$ does not algebrize.

Theorem 4.13. $\mathbf{NP} \subseteq \mathbf{P}$ *does not algebrize.*

Proof. By Definition 4.7 it is enough to prove that there exists an oracle O such that $\mathbf{NP}^{\tilde{O}} \not\subseteq \mathbf{P}^{O}$ where $\tilde{O}$ is some algebraic extension of O. This will be done analogously to Theorem 1.8. That is, let $M_1, \dots, M_n$ be some enumeration of all polynomial time deterministic Turing machines. So, let L_O be as in Theorem 1.8, that is

$$L_O = \{x : \exists y \in O \text{ such that } |x| = |y|\}.$$

$\tilde{O}$ will be defined inductively. For that, at stage i let $n > n_{i-1}$ be such that $n^{\log n} < 2^n$ and let p_i be a polynomial bounding the running time of M_i. Then, let Q_i be all queries M_i makes to some $\tilde{O}_{n,\mathbb{F}}$ with input 0^n and $R_i = \cup_{j<i} Q_j$. Then, for $t \in T_i \cap Q_i$, let $\tilde{O}_{n,\mathbb{F}}(t)$ agree with its values on previous states. And if $t \in Q_i \setminus T_i$, let $\tilde{O}_{n,\mathbb{F}}(t) = 0$. Now, for all $n' \in Q_i \setminus T_i$ such that $n' \neq n$, let $\tilde{O}_{n',\mathbb{F}} = 0$. Then, if $M_i(0^n) = 1$, then let, for all $\mathbb{F}$, $\tilde{O}_{n,\mathbb{F}} = 0$ (and hence $n \notin L_O$). And if $M_i(0^n) = 0$, for all $\mathbb{F}$, let $A_{\mathbb{F}} = \{a \in \mathbb{F}^n : a \in Q_i\}$. Then, $\sum_{\mathbb{F}} |A_{\mathbb{F}}| \le n^{\log n}$. Then, by Lemma

4.12, there exists z such that there exists a multiquadratic extension $\tilde{O}_{n,\mathbb{F}}$ such that $\tilde{O}_{n,\mathbb{F}}(z) = 1$ and $\tilde{O}_{n,\mathbb{F}}(y) = 0$ for $y \neq z$ (this way, $z \in O$). Finally, by the same argument from Theorem 1.8 it follows that $\mathbf{NP}^{\tilde{O}} \not\subseteq \mathbf{P}^{O}$. $\square$

The following is then immediate.

Corollary 4.14. $\mathbf{P} =?\mathbf{NP}$ *does not algebrize.*

Acknowledgments. I would like to thank John Wilmes for his willingness to share his knowledge and for his support through the project, and Prof. Peter May and those involved in organizing the REU.

References

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity A Modern Approach.* Cambridge University Press, Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, Sao Paulo, Delhi, 2009.

[AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. Primes is in **P**. *Annals of Mathematics*, 160(2):781–793, 2004.

[AW08] S. Aaronson and A. Wigderson. Algebrization: A new barrier in complexity theory. *STOC*, pages 731–740, 2008.

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

[Coo71] S. A. Cook. The complexity of theorem proving procedures. *Proceedings of 3rd Annual ACM Symposium in Theory of Computing*, pages 151–158, 1971.

[For94] L. Fortnow. The role of relativization in complexity theory. *Bulletin of the EATCS*, 52:229–244, 1994.

[FS88] L. Fortnow and M. Sipser. Are there interactive protocols for **coNP** languages? *Information Processing Letters*, 28:249–251, 1988.

[FSS81] M. Furst, J. Saxe, and M. Sipser. Parity, circuits, and the polynomial time hierarchy. *Mathematical Sytems Theory*, 17:13–27, 1981.

[GGM86] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, 1986.

[HILL99] J. Hastad, R. Impagliazzo, L. A. Levin, and M. Luby. A pseudorandom generator from any one-way function. *SIAM Journal of Computing*, 28(4):1364–1396, 1999.

[HS66] F. C. Hennie and R. E. Stearns. Two-tape simulation of multitape turing machines. *Journal of the ACM*, 13(4):533–546, 1966.

[Lev73] L. A. Levin. Universal sequential search problems. *PINFTRANS: Problems of Information Transmission*, 9:24–35, 1973.

[RR97] A. A. Razborov and S. Rudich. Natural proofs. *Journal of Computer and System Sciences*, 55(1):24–35, 1997.

[San07] R. Santhanam. Circuit lower bounds for merlin-arthur classes. *STOC*, pages 275–283, 2007.

[Sav70] W. J. Savitch. Relationships between nondeterministic and deterministic tape. *Journal of Computer and System Sciences*, 4:177–192, 1970.

[SM73] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time. *STOC*, pages 1–9, 1973.