Gottfried Wilhelm Leibniz Universität Hannover
Institut für Theoretische Informatik

Independence Results in Complexity Theory

Masterarbeit

Jakob Lennart Wege
Matrikelnr. 10019635

Hannover, den 26. April 2024

Erstprüfer:	Prof. Dr. Heribert Vollmer
Zweitprüfer:	PD Dr. Arne Meier
Betreuer:	Prof. Dr. Heribert Vollmer

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 26. April 2024

Jakob Lennart Wege

Contents

1 Introduction

The $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ problem is one of the most well-known open problems in theoretical computer science, if not all of mathematics. Due to the enormous practical and philosophical implications of the question "Can every problem whose solutions can be easily verified also be easily solved?", a lot of research has been done in an attempt to prove equality or inequality of these two fundamental complexity classes since the question was first formally posed in 1971 [Coo71]. There is even a $1 million prize offered by the Clay Mathematics Institute for a solution [Cla], established in 2000 along with prizes for six other then-unsolved problems. However, one possibility often gets ignored when discussing the problem and the attempts to solve it: What if we cannot prove *either one* of $\mathsf{P} = \mathsf{NP}$ and $\mathsf{P} \neq \mathsf{NP}$? This is known as *independence* from a logical theory. When David Hilbert posed his famous list of unsolved problems in 1900, which was a major inspiration for the Clay Institute's Millenium Prize, one of the problems on his list was the continuum hypothesis, which was shown to be independent of our typical axiom system for set theory in 1963 [Coh63].

While there is of course no known independence proof for $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$, research into this possibility has been done. Here, we aim to present several of the results of that research, focusing on two main topics.

After giving some definitions and specifying common notation in Chapter 2, Chapter 3 discusses *relativized* variations of the $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ problem, i.e. versions using oracle machines. We present the well-known result from [BGS75] that $\mathsf{P}^B = \mathsf{NP}^B$ and $\mathsf{P}^C \neq \mathsf{NP}^C$ both hold for different decidable oracles. Building on this, a result from [HH76] then shows the existence of a machine M such that $\mathsf{P}^{L(M)} = \mathsf{NP}^{L(M)}$ is independent of typical theories. In addition to this *representation-dependent* result, we then present a framework from [Har85] that shows *representation-independent* independence results, i.e. the existence of oracles A such that $\mathsf{P}^{L(M)} = \mathsf{NP}^{L(M)}$ is independent of a theory for *any* machine M that decides A. These results demonstrate that an independence result is not out of the question for the unrelativized version.

In Chapter 4, we then investigate possible consequences of a hypothetical independence result for the unrelativized version. Specifically, [BH91] show that an independence result for $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ would imply certain runtime bounds for NP-complete problems such as SAT and prevent the existence of a certain type of one-way functions.

A previous survey of results in this area has been given by Aaronson in [Aar03]. However,

that survey focuses mostly on presenting a broad collection of results, with almost no proofs. Here, we instead aim to give an overview of important results on the topic that includes their proofs and the various proof techniques that have been employed in this area.

2 Definitions

First, we define a set of consistent notations to present all results in a common framework and give some standard definitions and results from computability theory and logic that are not specific to the presented topics.

2.1 Turing machines

Definition 2.1. *We choose some standard Gödelization (encoding) scheme for deterministic Turing machines. For any $i \in \mathbb{N}$, M_i is the deterministic Turing machine encoded by i, or some fixed Turing machine $\hat{M}$ if i is not a valid encoding in our scheme. The encoding is chosen such that a machine can efficiently simulate M_i given i. Analogously, we choose a scheme such that N_i is the nondeterministic Turing machine encoded by i.*

Nondeterministic machines will generally only be used in contexts where a runtime bound is given, since they are equivalently powerful in terms of accepting languages with no complexity bounds and they do not cleanly define functions.

Definition 2.2. *$M_i(x)$ denotes the computation of M_i on the input x, e.g. "$M_i(x)$ halts" means that M_i halts on the input x.*

Definition 2.3. *A Turing machine* accepts *an input iff it halts and outputs 1 when given that input. The language* recognized *by a Turing machine M_i is the set of all inputs that the machine accepts and is denoted by $L(M_i)$.*

Both of the previous definitions apply analogously to nondeterministic machines.

Definition 2.4. *Unless specified otherwise, the term* function *is used to refer to partial functions. This means that functions may be undefined for some inputs.*

Definition 2.5. *For any $i \in \mathbb{N}$, φ_i is the function computed by M_i, i.e. $\varphi_i(x) = y$ iff $M_i(x)$ halts and outputs y, and $\varphi_i(x)$ is undefined iff $M_i(x)$ does not halt.*

Definition 2.6. *For any $i \in \mathbb{N}$, alphabet Σ, and language $A \subseteq \Sigma^*$, M_i^A is the oracle Turing machine encoded by i with the oracle A. This uses an adjusted encoding scheme in which the oracle question and answer states are also specified. We define nondeterministic oracle Turing machines N_i^A accordingly.*

Note that because the index i is being interpreted differently for oracle machines, the machine M_i^A may behave completely differently from M_i.

Definition 2.7. *A Turing machine M_i is* total *iff φ_i is total, i.e. the machine halts for all inputs.*

Definition 2.8. *If M_i is total, we say that it* decides *the language $L(M_i)$.*

Every Turing machine *recognizes* a language, but only total Turing machines *decide* a language.
In some situations, we want to restrict Turing machines to only those that decide the languages of a specific complexity class. To ensure we can still index those Turing machines using natural numbers, we use a *recursive representation* (see [Har85]):

Definition 2.9. *A* recursive representation *of a family C of languages is a total computable function f such that the members of C are exactly those languages that are recognized by some Turing machine $M_{f(n)}$, where all of those machines are total.*

For a simple time- or space-bounded complexity class, we can accomplish this by having the function f modify the machine M_i by adding a counter to it that tracks the time or space used by the original computation, and immediately halting and rejecting the input if a certain bound is exceeded. We illustrate this for the polynomial time classes.
The system for polynomial time Turing machines defined in [BGS75] can be implemented by defining a function P such that the machine $M_{P(i)}$ has an artificial runtime bound of $p_i(n) \coloneqq i + n^i$, where n is the input length. We refer to this system and analogous systems for other complexity classes as *standard enumerations* of those classes.
For simplicity's sake, we use $N_{P(i)}$ to denote a nondeterministic Turing machine transformed using the same process, even though the actual function P would have to be different due to the different interpretations of i in the different encodings. The same applies to the oracle machine versions.
Note that for a given problem in e.g. P decided by a given polynomial-time machine M_i, that machine might require more than $p_i(n)$ steps and thus $M_{P(i)}$ will halt too early and decide the wrong language. However, for every such Turing machine, there are infinitely many machines with the same behavior as M_i, which can be constructed e.g. by adding unreachable states with arbitrary transitions to the encoding. Eventually, this will produce an encoding j such that M_j behaves the same as M_i, but with j sufficiently large such that $p_j(n)$ bounds its runtime. $M_{P(j)}$ is then a machine from the standard enumeration that decides the desired language. Therefore, for every problem in P or NP respectively, there will be a machine in this numbering that has enough available runtime to solve it. The same principle applies to standard enumerations of other complexity classes.

2.2 Logic

2.2.1 General terms

All results presented here use first-order predicate logic as their basis. This is because Gödel's completeness theorem [BBJ07, Ch. 14] shows the existence of a complete and sound formal calculus, i.e. a formal proof system in which precisely all true implications in first-order logic can be proven. This allows Turing machines to verify whether a given formal proof is correct, and it means that any statement that semantically follows from a set of formulas has a formal proof from those formulas. The specific choice of proof calculus does not matter for any results shown here, so we can assume e.g. the Gentzen calculus of natural deduction.

Definition 2.10. *A* sentence *is a logical formula with no free variables.*

Definition 2.11. *If A and B are logical formulas or sets of logical formulas, then $A \vdash B$ means that B can be proven from A.*

Definition 2.12. *A logical* theory *T is a set of sentences that is closed under deduction, i.e. for any sentence A, $T \vdash A$ implies $A \in T$. The members of T are also known as its* theorems *and are said to be* provable *in T.*

Essentially, a theory is a set of statements that we consider to be provably true. For a theory to be interesting to us, we generally require it to have a number of additional properties.

Definition 2.13. *A theory T is* axiomatizable *iff it is the closure under deduction of a decidable set of* axioms.

Axiomatizable theories are useful in a computational context because they are recursively enumerable [Rog87, p. 95], which means they are recognized by some Turing machine.

Definition 2.14. *A theory T is* consistent *iff there is no sentence A such that $A \in T$ and $\neg A \in T$.*

Going forward, we will only study theories that can be seen as making statements about the natural numbers. Any other objects that we want to discuss, e.g. Turing machines or strings, will be encoded as natural numbers using some kind of Gödelization/encoding scheme.

Definition 2.15. *A theory T that can be interpreted as having the natural numbers as its domain of discourse is* sound *iff all of its theorems are true about the standard natural numbers.*

We refer to statements that are true about the standard natural numbers simply as *true*. It is crucial to keep in mind the distinction between this truth and *provability*, which depends on the specific theory we are studying. From Gödel's first incompleteness theorem [BBJ07, Ch. 17], we know that for any axiomatizable, consistent, sound theory, there will be true statements that are not provable in that theory. In fact, statements of that kind will be our main focus:

Definition 2.16. *A sentence A is* independent *of a theory T iff neither A nor $\neg A$ is provable in T.*

A specific sound and axiomatizable theory that we will often discuss is *Peano arithmetic*. We give a version of the formulation from [Men97], where S denotes the successor function:

Definition 2.17. Peano arithmetic *or $\mathcal{PA}$ is the theory generated by the following axioms:*

(1) $\forall a \forall b(a = b \leftrightarrow S(a) = S(b))$

(2) $\forall a(S(a) \neq 0)$

(3) $\forall a(a + 0 = a)$

(4) $\forall a \forall b(a + S(b) = S(a + b))$

(5) $\forall a(a \cdot 0 = 0)$

(6) $\forall a \forall b(a \cdot S(b) = (a \cdot b) + a)$

for every predicate P in the language of arithmetic:

(7P) $(P(0) \wedge \forall a(P(a) \rightarrow P(S(a)))) \rightarrow \forall a P(a)$

Note that (7P) is not a single axiom, but an axiom schema providing an axiom for every possible predicate. This is allowed for an axiomatizable theory because the set of axioms, while infinite, is decidable.
$\mathcal{PA}$ is useful because it is powerful enough to express statements about the behavior of Turing machines. Although there are smaller theories that also allow this, for consistency's sake, we will mostly study $\mathcal{PA}$ and larger theories in the upcoming chapters.

2.2.2 Classes of formulas

For some results, we need to classify sentences based on the quantifiers that they use. This system can be found in standard works such as [Rog87, p. 303]. We use this classification both for logical formulas and for metalogical statements where necessary.

Definition 2.18. *We use* $(\forall a < b)F$ *and* $(\exists a < b)F$ *as abbreviations for* $\forall a(a < b \to F)$ *and* $\exists a(a < b \wedge F)$*. These are known as* bounded quantifiers*.*

Definition 2.19. *A formula containing only bounded quantifiers is known as a* Δ_0 *formula.*

Bounded quantifiers are important because their application does not change the decidability of a relation. For example, if the set $\{(a, b) \mid R(a, b)\}$ is decidable for some binary relation R, the set $\{b \mid (\forall a < b)R(a, b)\}$ is also decidable. This is because the decision algorithm can simply be extended to check all finitely many possible values for the variables being quantified by bounded quantifiers.
Prefixes of unbounded quantifiers are characterized by the number of groups of alternating quantifiers, along with whether the first quantifier is $\forall$ or $\exists$.

Definition 2.20. *A* Σ_n *formula is a formula of the form* $\exists x_1 \forall x_2 \exists x_3 \ldots Q x_n F$*, where* Q *is the appropriate quantifier to complete the alternating pattern and* F *is a* Δ_0 *formula. A* Π_n *formula is defined by the analogous construction beginning with a* $\forall$ *quantifier.*

Similarly, a condition is a Σ_n condition if it can be expressed as $\exists x_1 \forall x_2 \exists x_3 \ldots Q x_n P$, where P is a condition that can be decided by a Turing machine. In this case, the quantifiers are not logical symbols, but just a notation for the structure of a statement. Again, Π_n is used analogously.

2.2.3 The arithmetical hierarchy

Using these quantifier prefixes, we can define a hierarchy of sets of natural numbers known as the *arithmetical hierarchy*, as given in [Rog87, pp. 301–305].

Definition 2.21. Σ_n *is the class of all sets of natural numbers for which membership can be expressed as a* Σ_n *condition, or equivalently all sets which can be arithmetically defined by a* Σ_n *formula. The* Π_n *classes are defined analogously.*

We use the following standard results about the hierarchy:

Lemma 2.22. *The following hold for all* n*:*

- $\Sigma_n \subsetneq \Sigma_{n+1}$
- $\Sigma_n \subsetneq \Pi_{n+1}$
- $\Sigma_n \neq \Pi_n$
- Σ_n *is closed under* $\leq_m$ *reduction.*

All of these also hold when swapping Σ and Π.
An important consequence of these results is that because the classes Σ_n and Π_n are distinct and closed under reduction, any Σ_n-complete sets cannot be members of Π_n and vice versa.

3 Relativization

Since $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ is a difficult question, we can turn to variations of the problem in the hope that their solution can help us answer the original question. Specifically, we look at relativized versions of the question. This means that instead of studying standard Turing machines, we study oracle Turing machines relative to some oracle A. Then, P^A is the class of problems that can be solved by a deterministic oracle Turing machine with oracle A in polynomial time, and NP^A is the nondeterministic equivalent. We can now study $\mathsf{P}^A \stackrel{?}{=} \mathsf{NP}^A$ for different oracles A. It is clear that if $A \in \mathsf{P}$, then $\mathsf{P}^A = \mathsf{P}$ and $\mathsf{NP}^A = \mathsf{NP}$. Therefore an answer for such an oracle (or an answer for a wider class of oracles that includes such oracles) would give us an answer to the original question. Thus we want to study which answers to $\mathsf{P}^A \stackrel{?}{=} \mathsf{NP}^A$ are even possible.

3.1 Representation-dependent results

The first important results in this area are presented in [BGS75]. There, the authors show that there are both decidable oracles B such that $\mathsf{P}^B = \mathsf{NP}^B$ and decidable oracles C such that $\mathsf{P}^C \neq \mathsf{NP}^C$. A summary of those proofs follows.

Theorem 3.1. *There is a decidable set B such that $\mathsf{P}^B = \mathsf{NP}^B$.*

Proof. Let B be any PSPACE-complete set (such as QBFSAT). Then by definition, $B \in \mathsf{PSPACE}$ and $\mathsf{PSPACE} \subseteq \mathsf{P}^B$. Also, $\mathsf{NP}^B \subseteq \mathsf{NPSPACE}$ by oracle replacement, i.e. by simulating the PSPACE machine for the oracle. Since $\mathsf{NPSPACE} = \mathsf{PSPACE}$ by Savitch's theorem [Sav70], we have $\mathsf{NP}^B \subseteq \mathsf{NPSPACE} = \mathsf{PSPACE} \subseteq \mathsf{P}^B$. The opposite inclusion $\mathsf{P}^B \subseteq \mathsf{NP}^B$ holds trivially, proving equality. □

The following proof relies on the function P that limits the runtime of Turing machines as described earlier.

Theorem 3.2. *There is a decidable set C such that $\mathsf{P}^C \neq \mathsf{NP}^C$.*

Proof. For any set X, let $W(X) \coloneqq \{x \mid \exists y \in X : |y| = |x|\}$, that is, the set of all words x that have the same length as some word y in X. $W(X) \in \mathsf{NP}^X$ via an oracle machine $N^X_{P(i)}$ that nondeterministically guesses a string of the same length as its input, queries the oracle X, and accepts iff the oracle accepts. Note that if $W(X)$ contains a string x,

it must also contain all other strings of length $|x|$.
We construct a set C in stages such that $W(C) \notin \mathsf{P}^C$. Let $C(i)$ denote the (finite) set of strings placed in C before stage i, $C(0) = \emptyset$, and $n_0 = 0$.
In each stage i, we do the following: Choose $n > n_i$ sufficiently large that $p_i(n) < 2^n$ (with $p_i(n) \coloneqq i + n^i$ as defined earlier). This is always possible because 2^n grows asymptotically faster than any polynomial. Simulate $M_{P(i)}^{C(i)}$ on the input $x_i \coloneqq 0^n$. If the machine accepts, add nothing to C in this stage, i.e. $C(i+1) \coloneqq C(i)$. Otherwise, add the first string in lexicographical order of length n that was not used as an oracle query in that computation to C. There are 2^n binary strings of length n, but the number of oracle queries is bounded by the runtime of $p_i(n)$, which is strictly less than 2^n by choice of n. Therefore, such a string will always exist. Finally, let $n_{i+1} = 2^n$. This ensures that every additional string added in later stages will be longer than 2^n.
Since every string added to C in later stages is longer than 2^n and thus longer than any of the oracle queries, whose length is bounded by the runtime of $p_i(n)$, and since the only string that can be added during the stage itself is one that was not queried by the oracle computation, the simulation of $M_{P(i)}^{C(i)}$ behaves exactly as if C had been used as the oracle instead of $C(i)$, so the computation is indistinguishable from that of the machine $M_{P(i)}^{C}$. Thus, each stage i ensures that the machine $M_{P(i)}^{C}$ does not recognize $W(C)$ as follows:
If stage i adds a string of length n to C, then $M_{P(i)}^{C}$ must have rejected the input 0^n. But since C contains a string of length n, we know that $0^n \in W(C)$ by definition. Therefore, $M_{P(i)}^{C}$ rejected a member of $W(C)$ and thus does not recognize $W(C)$.
If stage i does not add a string to C, then $M_{P(i)}^{C}$ must have accepted the input 0^n for that stage's choice of n. But since no other stage can add a string of length n to C, C cannot contain a string of that length and thus $0^n \notin W(C)$. Therefore, $M_{P(i)}^{C}$ accepted a string that is not in $W(C)$ and thus does not recognize $W(C)$.
Since every language in P^C is recognized by some machine $M_{P(i)}^{C}$, we therefore know that $W(C) \notin \mathsf{P}^C$. But since $W(C) \in \mathsf{NP}^C$, we have shown that $\mathsf{P}^C \neq \mathsf{NP}^C$, and C is decidable via the above construction. □

These results tell us that we cannot resolve $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ by showing a general result for all oracles. However, if we could at least show that for any oracle A, one of $\mathsf{P} = \mathsf{NP}$ or $\mathsf{P} \neq \mathsf{NP}$ were provable, we could rule out an independence result. This possibility was first eliminated in [HH76]. Here, the authors show, building on the results in [BGS75], that decidable oracles A can be constructed for which $\mathsf{P}^A = \mathsf{NP}^A$ is independent of a given theory. This result is presented below.

Definition 3.3. *Let T be an axiomatizable, consistent, sound theory with $\mathcal{PA} \subseteq T$.*

Theorem 3.4. *For every such theory T, we can effectively construct a Turing machine M such that the statement $\mathsf{P}^{L(M)} = \mathsf{NP}^{L(M)}$ is independent of T.*

Proof. Let B and C be decidable sets such that $\mathsf{P}^B = \mathsf{NP}^B$ and $\mathsf{P}^C \neq \mathsf{NP}^C$ as in Theorems 3.1 and 3.2. Define a function $p : \mathbb{N}^2 \to \{0, 1\}$ as follows:

$$p(x, j) = \begin{cases} 1, & \text{if } x \in C \text{ and there exists a proof for } \mathsf{P}^{L(M_j)} = \mathsf{NP}^{L(M_j)} \text{ among the first} \\ & x \text{ proofs in } T, \text{ or if } x \in B \text{ and there exists a proof for } \mathsf{P}^{L(M_j)} \neq \mathsf{NP}^{L(M_j)} \\ & \text{among the first } x \text{ proofs in } T \\ 0, & \text{otherwise.} \end{cases}$$

Here, "the first x proofs in T" means the first x valid proofs in quasi-lexicographical order.

Since T is axiomatizable, a Turing machine can enumerate its valid proofs and check if they prove a given statement. Since B and C are also decidable, p is computable. According to the S_n^m theorem [Soa87, p. 16], we can thus construct a computable function σ for which the following holds for all j:

$$\varphi_{\sigma(j)}(x) = p(x, j)$$

According to Kleene's recursion theorem [Soa87, p. 36], σ has a "fixed point" i_0 with the following property:

$$\varphi_{i_0}(x) = \varphi_{\sigma(i_0)}(x)$$

Combining the two equations above results in:

$$\varphi_{i_0}(x) = p(x, i_0)$$

Intuitively, i_0 is the encoding of a Turing machine that, given an input x, searches the first x proofs in T for a proof that P is or is not equal to NP relative to *its own language* as an oracle. It also checks for membership in B or C as appropriate and chooses its output accordingly.

Since T is consistent, there cannot be both a proof in T that $\mathsf{P}^{L(M_{i_0})} = \mathsf{NP}^{L(M_{i_0})}$ and a proof in T that $\mathsf{P}^{L(M_{i_0})} \neq \mathsf{NP}^{L(M_{i_0})}$. Therefore, if there is a proof in T that $\mathsf{P}^{L(M_{i_0})} = \mathsf{NP}^{L(M_{i_0})}$, then for all x greater than or equal to that proof's index in the enumeration, M_{i_0} will accept iff $x \in C$. Therefore, $L(M_{i_0})$ differs at most finitely from C. If two oracles A_1 and A_2 differ finitely, then $\mathsf{P}^{A_1} = \mathsf{P}^{A_2}$ and $\mathsf{NP}^{A_1} = \mathsf{NP}^{A_2}$, since the differences can be hardcoded into each machine and handled in constant time before each oracle query. Therefore, since $\mathsf{P}^C \neq \mathsf{NP}^C$, $\mathsf{P}^{L(M_{i_0})} \neq \mathsf{NP}^{L(M_{i_0})}$ must also be true.

Analogously, if there is a proof in T that $\mathsf{P}^{L(M_{i_0})} \neq \mathsf{NP}^{L(M_{i_0})}$, then $L(M_{i_0})$ differs at

most finitely from B and therefore $\mathsf{P}^{L(M_{i_0})} = \mathsf{NP}^{L(M_{i_0})}$ must be true.
Since T is sound, neither of those cases is possible. Therefore, there is no proof in T for either $\mathsf{P}^{L(M_{i_0})} = \mathsf{NP}^{L(M_{i_0})}$ or $\mathsf{P}^{L(M_{i_0})} \neq \mathsf{NP}^{L(M_{i_0})}$, that is, $\mathsf{P}^{L(M_{i_0})} = \mathsf{NP}^{L(M_{i_0})}$ is independent of T. □

Now, we claimed above that there are decidable *oracles* A for which $\mathsf{P}^A = \mathsf{NP}^A$ is independent of F. However, that is not quite what the result above shows. The result shows specifically that there is a *Turing machine* M_{i_0} such that the statement $\mathsf{P}^{L(M_{i_0})} = \mathsf{NP}^{L(M_{i_0})}$ is independent of F. However, the conclusion we come to at the end of the proof above shows that p will actually always return 0, since the existence of a proof that would make it output 1 would imply the inconsistency of T. Therefore, our oracle $A = L(M_{i_0})$ must be the empty set. Since we know that $\mathsf{P}^{\emptyset} = \mathsf{NP}^{\emptyset}$ is equivalent to $\mathsf{P} = \mathsf{NP}$, this suggests at a glance that the unrelativized version must be independent of T. But crucially, according to Gödel's second incompleteness theorem [BBJ07, Ch. 18], T cannot prove its own consistency, so it cannot *prove* that A is the empty set.
This means that the independence of $\mathsf{P}^{L(M_{i_0})} = \mathsf{NP}^{L(M_{i_0})}$ from T is not a property of the language used as an oracle, but of the Turing machine that we use to define said language. Such an independence result is said to be *representation-dependent*, since it depends on the representation of the language by a machine.

3.2 Representation-independent results

Thus, the natural question arises whether there are any languages for which $\mathsf{P}^A = \mathsf{NP}^A$ is independent no matter which representation of A we choose. This turns out to also be possible. In [Har85], the author shows that there are languages relative to which $\mathsf{P} = \mathsf{NP}$ is independent of a theory for any recursive representation of the problem. This is accomplished via a relatively generic framework using the arithmetical hierarchy which we will present in the following section.

3.2.1 Recursive representations and the Naming Lemma

Lemma 3.5 (Naming Lemma)**.** *A family C of decidable languages has a recursive representation iff there exists an axiomatizable, consistent, sound theory T such that for every $L \in C$, there is a total Turing machine M that decides that language and for which it can be proven in T that M is total and that $L(M) \in C$.*

Proof. If C has a recursive representation f, let T be Peano arithmetic with the additional axiom "$\forall n(L(M_{f(n)}) \in C$ and $M_{f(n)}$ is total)". Because f is computable, we can construct this axiom by expressing the Turing machine that computes f using logical formulas. T can prove the required statements by construction.

Conversely, let T be a theory that fulfills the conditions above. We can recursively enumerate all i for which "$L(M_i) \in C$ and M_i is total" is provable in T by recursively enumerating all valid proofs and checking whether they prove a statement of the desired form, which gives us a recursive representation for C. □

Definition 3.6. *Let R be a decidable set whose members are encodings of total Turing machines. Then an* R-representation *of the family of languages C is a recursive representation of C whose image is a subset of R.*

The main proof technique used in [Har85] to show representation-independent independence results relies on the arithmetical hierarchy. First, we prove that a certain set D is Π_2-complete. Then, we show that the existence of a recursive or R-representation for some set C would show membership of D in Σ_2. Since Π_2-complete sets cannot be in Σ_2, we can conclude that no such representation can exist.
We present the version of this result that is used for relativized versions of $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ here. For this proof, we require a standard enumeration $M_{E(0)}, M_{E(1)}, \ldots$ of Turing machines using $\mathcal{O}(2^{2^n})$ space and a standard enumeration $M_{P(0)}, M_{P(1)}, \ldots$ of polynomial time deterministic Turing machines. Now, we define a set D that we show to be Π_2-complete, which will allow us to apply the proof technique described above.

Lemma 3.7. $D = \{i \mid \mathsf{P}^{L(M_{E(i)})} \neq \mathsf{NP}^{L(M_{E(i)})}\}$ *is Π_2-complete.*

Intuitively, D is the set of all encodings of $\mathcal{O}(2^{2^n})$ space Turing machines whose accepted language L is an oracle for which $\mathsf{P}^L \neq \mathsf{NP}^L$.

Proof. First, we need to prove membership in Π_2.
For any oracle A, let N^A_{com} be a nondeterministic polynomial time Turing machine that decides the NP^A-complete language

$$U^A_{\mathsf{NP}} = \{i\#x\#\#^{|x|^k \cdot |i|} \mid N_i \text{ accepts } x \text{ in time } |x|^k\},$$

based on the universal complete languages defined in [Har89]. Using this, D can be defined as follows:

$$D = \left\{ i \;\middle|\; \forall j \exists x \left(M^{L(M_{E(i)})}_{P(j)}(x) \text{ accepts} \Leftrightarrow N^{L(M_{E(i)})}_{\text{com}}(x) \text{ rejects} \right) \right\}.$$

Intuitively, the reason that this is an equivalent formulation of D is that $\mathsf{P}^{L(M_{E(i)})} \neq \mathsf{NP}^{L(M_{E(i)})}$ (the defining condition of D) holds iff no deterministic polynomial time Turing machine with oracle $L\left(M_{E(i)}\right)$ decides an $\mathsf{NP}^{L(M_{E(i)})}$-complete language (analogous to the fact that $\mathsf{P} \neq \mathsf{NP}$ iff no polynomial time Turing machine decides an NP-complete language). This holds iff for every such Turing machine $M^{L(M_{E(i)})}_{P(j)}$ defined

by an index j, there is an input x for which the output differs from the machine for the complete language. Since the runtimes of the machines in the predicate are bounded, the predicate is decidable. Therefore, this formulation proves that $D \in \Pi_2$.
Secondly, we need to prove Π_2-hardness.
The idea of the proof presented in [Har85] is to reduce the known Π_2-hard problem $\mathsf{INF} = \{i \mid L(M_i) \text{ is infinite}\}$ to D by constructing a machine whose accepted language differs only finitely from oracles with $\mathsf{P}^B = \mathsf{NP}^B$ and $\mathsf{P}^C \neq \mathsf{NP}^C$ respectively. However, while this idea appears sensible, the stage construction used in the paper to prove it appears ill-defined. Therefore, no detailed hardness proof is presented here. □

Using this language, we can now prove the representation-independent independence result:

Theorem 3.8. *There exists an oracle $A \in \mathsf{SPACE}(2^{2^n})$ such that $\mathsf{P}^A \neq \mathsf{NP}^A$, but for no $M_{E(i)}$ with $L\left(M_{E(i)}\right) = A$, it can be proven in T that $\mathsf{P}^{L\left(M_{E(i)}\right)} \neq \mathsf{NP}^{L\left(M_{E(i)}\right)}$.*

Proof. Assume that for each $A \in \mathsf{SPACE}(2^{2^n})$ with $\mathsf{P}^A \neq \mathsf{NP}^A$, there is a machine $M_{E(i)}$ with $L\left(M_{E(i)}\right) = A$ for which T can prove $\mathsf{P}^{L\left(M_{E(i)}\right)} \neq \mathsf{NP}^{L\left(M_{E(i)}\right)}$. Then by the Naming Lemma, per definition of D, the language family $\{L \mid \exists i \in D : L = L\left(M_{E(i)}\right)\}$ has a recursive representation f. But given that recursive representation, D can be formulated as

$$D = \left\{i \,\middle|\, \exists j \forall x \left(M_{E(i)}(x) = M_{f(j)}(x)\right)\right\}$$

with a decidable predicate. Intuitively, this expresses D as the set of all i for which a machine $M_{f(j)}$ exists in the recursive representation which agrees with $M_{E(i)}$ for all inputs. The structure of this formulation implies $D \in \Sigma_2$. Since D is Π_2-complete, this is a contradiction. Therefore, there must be an A for which the assumption does not hold. □

3.2.2 Chunky sets

In [KOR87], the authors construct a more general framework for proving representation-independent independence results and use it to prove, among others, many of the theorems from [Har85] presented above. Unfortunately, a lot of non-trivial results in this paper are presented without proof and the core terminology is not well explained. Therefore, we will only give a very brief overview here:
The core idea is the definition of a class of sets of languages called *chunky sets*, which contain sets with arbitrarily fast growth in a certain sense. For certain pairs of complexity classes R and S with closure properties relating to chunky sets, there are then sets L_0 in $R \setminus S$ for which the infinity of $L(M)$ for any machine M that decides L_0 is independent of a given theory. This methodology can be used, with varying choices of R and S, to prove most of the results from [Har85].

4 Consequences of an independence result for $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$

While independence from a mathematical theory is a fairly theoretical concept, $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ is a question whose answer has significant practical consequences for the runtime of actual algorithms. Therefore, the question arises what the practical consequences of an independence result would be. In [BH91], the authors show that if $\mathsf{P} = \mathsf{NP}$ is independent of Peano arithmetic or a larger theory, then NP is in some sense very close to P, that is, problems in NP have deterministic runtime bounds that are very close to polynomial.

4.1 The Wainer hierarchy

The proof relies on the Wainer hierarchy, a family of fast-growing functions introduced in [Wai70]. The version used in [BH91] uses the following basic definition:

$$F_1(n) = 2n$$
$$F_{k+1}(n) = F_k^{(n)}(n)$$

where $F^{(n)}$ is the function F iterated n times. The hierarchy also allows infinite ordinals as indices. We give a brief primer on ordinals here.
The finite ordinals are the natural numbers $0, 1, 2, \ldots$, interpreted as the position of items in some linearly ordered set. For example, 3 can denote the 3rd item in a set. However, for some infinite linearly ordered sets, these ordinals may not suffice. For example, if we define the set $\{0, 1, 2, \ldots, x\}$, where $x \notin \mathbb{N}$ is some new element, and we extend the definition of the order relation such that $n < x$ for all $n \in \mathbb{N}$, no finite ordinal can denote the position of the element x in this set. Instead, we define a *limit ordinal* ω, which is the limit of the sequence of finite ordinals $0, 1, 2, \ldots$. We can then say that x is the ωth member of the set. Every limit ordinal α has such a defining sequence, known as its *fundamental sequence* $\{\alpha\}(n)$. The formal details of ordinals are beyond the scope of this summary and can be found e.g. in [Jec03, Ch. 2], but we can use these concepts to define the functions in the Wainer hierarchy with limit ordinal

indices as follows, using the fundamental sequence of the ordinal:

$$F_\alpha(n) = F_{\{\alpha\}(n)}(n)$$

For example, F_ω is a version of the Ackermann function.
We define some terms relating to functions:

Definition 4.1. *A function f* dominates *a function g iff $f(n) > g(n)$ for all sufficiently large n.*

For every $\alpha < \beta$, F_β dominates F_α.

Definition 4.2. *ε_0 is the first ordinal α satisfying $\omega^\alpha = \alpha$. (This is the limit ordinal of the sequence $\omega, \omega^\omega, \omega^{\omega^\omega}, \ldots$.)*

From here on out, we take "the Wainer hierarchy" to be those functions F_α for which $\alpha < \varepsilon_0$. If any of those functions dominate a function f, f is said to be dominated by the Wainer hierarchy.

Definition 4.3. *A total computable function f is* provably recursive *in a theory $T \supseteq \mathcal{PA}$ iff there is a Turing machine M that computes f for which it can be proven in T that M is total.*

Note that the definition of provably recursive functions does not require the theory to be able to prove that the algorithm computes the desired function. This restricts the power of some of the future results in subtle ways.
[Wai70] proves the following lemmas:

Lemma 4.4. *Every function F_α in the Wainer hierarchy is provably recursive in $\mathcal{PA}$.*

Lemma 4.5. *If a total computable function $f : \mathbb{N} \to \mathbb{N}$ is provably recursive in $\mathcal{PA}$, it is dominated by the Wainer hierarchy.*

Therefore, the functions in the Wainer hierarchy are useful for measuring the growth rate of provably recursive functions in $\mathcal{PA}$.

Definition 4.6. *A function $f : \mathbb{N} \to \mathbb{N}$ is a* standard complexity function *iff it is time-constructible and it is provable in $\mathcal{PA}$ that f is total, monotonic, and unbounded.*

Note that time-constructibility (or indeed computability) are not required in the definition given in [BH91]. However, these properties appear to be necessary for the proofs using this definition, which is why we have added them here.
Standard complexity functions will often be used as space or time bounds to define complexity classes. The vast majority of functions commonly used for this purpose (e.g.

n^k, $\log n$, 2^n) are standard complexity functions, as are some more niche ones like the Ackermann function and its inverse.
A number of results will require "inverting" functions that are not bijective. Specifically, these will often be functions representing growth rates. Therefore, we define a more general notion of inverse functions that works for those functions as follows:

Definition 4.7. *For any total, monotonic, and unbounded function f, let $f^{-1}(n) \coloneqq \max\{i \mid f(i) \leq n\}$.*

Note that this definition does not apply only to standard complexity functions, but also to functions that have the same function properties, but not provably so.
The following lemma is presented in [BH91] without proof:

Lemma 4.8. *If f and h are standard complexity functions and g is a monotonic function that is not dominated by the Wainer hierarchy, there are infinitely many $n \in \mathbb{N}$ satisfying $\forall m((n < m < h(n)) \to (g^{-1}(m) < f(m) < g(m)))$.*

Intuitively, h is used to define the size of progressively larger intervals, and the lemma states that there are infinitely many of those intervals within which the function f is sandwiched between g and its inverse.

4.2 Strong proof systems

We present a series of results from [BH91]. We again let T denote an axiomatizable, consistent, sound theory that includes Peano arithmetic. As a working tool, we introduce a stronger theory T_1 for every such T as follows:

Definition 4.9. *T_1 is the theory generated by the set*
$T \cup \{\varphi \mid \varphi$ is a Π_1 formula that is true (in the standard model of arithmetic)$\}$.

Since T is sound and we only add true statements to it, T_1 is also sound. However, it can be easily seen that T_1 is not axiomatizable:
Let $K \coloneqq \{i \mid M_i(i)$ halts$\}$ be the halting problem and $\bar{K}$ its complement. We can reduce $\bar{K}$ to T_1 by expressing the condition "$M_i(i)$ does not halt" as a Π_1 formula of the form "$\forall n(M_i(i)$ does not halt after n steps)". Since K is undecidable and recursively enumerable, $\bar{K}$ is not recursively enumerable, and therefore T_1 cannot be recursively enumerable either. But since all axiomatizable theories are recursively enumerable, T_1 cannot be axiomatizable.
Usually, non-axiomatizable theories are not particularly useful. However, T_1 is still a useful theory for the following reason:

Lemma 4.10. *A function is provably recursive in T_1 iff it is provably recursive in T.*

Proof. The proof for this lemma is given in [FLO83] as follows:
Since $T \subseteq T_1$, any provably recursive function in T is trivially provably recursive in T_1. To prove the converse, let f be provably recursive in T_1. Then there exists a Turing machine M that computes f, and there is some proof P in T_1 of the statement "M is total". Therefore, "M is total" can be proven in T from a true Π_1 sentence $\forall z F$ which combines all of the "additional axioms" of T_1 (the ones that are not provable in T) used in the proof P. (Since all of those "axioms" are Π_1 sentences, their conjunction is equivalent to a single Π_1 sentence.) Therefore, the statement "$(\forall z F) \rightarrow (M \text{ is total})$" is provable in T. This can be rephrased as "M is total or $\exists z \neg F$". We define a machine M' that simultaneously simulates M on its input and checks the truth of $\neg F(z)$ for all natural numbers z in order (F includes only bounded quantifiers, so this is decidable for each z). If the simulation of M halts, M' immediately halts and returns the result of the simulation. If a z is found for which $\neg F(z)$ is true, M' immediately halts with some arbitrary output. Now, "M' is total" is provable in T by construction. Additionally, since we know that $\forall z F$ is true, M' will always halt because of the simulation of M returning a result, not because of a counterexample for F being found. Therefore, M' computes the same function as M, namely f. Thus, f is provably recursive in T. (Note again that the definition of "provably recursive" does not require a proof in T that M' computes f.) □

No proof is given in [BH91] for the next lemma, and it is attributed to "the folklore of proof theory".

Lemma 4.11. *For a relational structure M, let $\Pi_1(M)$ denote the set of Π_1 sentences that are true in M. If the independence of a statement φ from a theory T can be demonstrated by starting with a model M of T in which φ holds and constructing a submodel in which $\neg\varphi$ holds, then φ is also independent of $T \cup \Pi_1(M)$.*

The authors of [BH91] claim that any imaginable technique for proving independence from sufficiently strong theories meets the assumptions of this lemma. Under those circumstances, they give the following corollary:

Theorem 4.12. *Any independence result from an axiomatizable, consistent, sound theory T can be extended to an independence result from T_1.*

4.3 Approximating a language by a complexity class

To achieve the goal of proving that an independence result for $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ implies that NP is close to P, [BH91] defines a measure of distance between a language L and a complexity class C called the *approximation rate*. We will show that $L \notin C$ is not

provable iff the approximation rate of L by C is an extremely fast-growing function, i.e. L's complexity is very well approximated by C.

Definition 4.13. *Let $L \subseteq \Sigma^*$ be a language, C be a complexity class, and f be a recursive enumeration of C. The* approximation rate of L by C *is the function*

$$R_L^C(i) = \max_{j \leq i}\{\min\{|x| \mid x \in L \Leftrightarrow M_{f(j)} \text{ rejects } x\}\}$$

Intuitively, $R_L^C(i)$ determines, for each individual machine $M_{f(j)}$ from $M_{f(0)}, \ldots, M_{f(i)}$, the shortest word that witnesses that $M_{f(j)}$ does not recognize L, and returns the length of the longest of these minimal witnesses. For example, if $R_L^C(20) = 10$, then for every machine $M_{f(0)}, \ldots, M_{f(20)}$, there is a word of length at most 10 that is either contained in L, but rejected by the machine, or vice versa. Therefore, if R_L^C grows very quickly, the enumeration of Turing machines quickly produces Turing machines that act like they recognize L until the words exceed a high length threshold. In particular, if $L \in C$, R_L^C is undefined for large inputs because there is a machine $M_{f(i)}$ that recognizes L, so there is no minimum word length that witnesses the difference. This proves the following lemma:

Lemma 4.14. *For every language L and complexity class C, R_L^C is a total function iff $L \notin C$.*

Note also that for any decidable language L, R_L^C is computable via an algorithm that simulates each machine $M_{f(0)}, \ldots, M_{f(i)}$ on all possible inputs x in quasi-lexicographical order until it fails to correctly decide an input's membership in L, which it can verify via a decision algorithm for L.
What interests us here is the approximation rate of SAT by P. If this is a fast-growing function, then SAT, an NP-complete problem, is "close" to P. We pick a standard enumeration f of deterministic polynomial time Turing machines such that $M_{f(i)}$ can be simulated with only linear overhead and runs for at most $n^{\log i}$ steps on every input.

Lemma 4.15. *Let $R \coloneqq R_{\mathsf{SAT}}^{\mathsf{P}}$ be the approximation rate of SAT by P. If the inverse function R^{-1} is bounded by some time-constructible function g, then $\mathsf{SAT} \in \mathsf{DTIME}(n^{1+\log g(n+1)} \cdot g(n+1))$.*

Proof. We claim that the following deterministic algorithm decides SAT:

input: x
for $i = 0$ to $g(|x| + 1)$ **do**
Use a simulation of $M_{f(i)}$ to find a satisfying assignment for x. If a satisfying assignment is found, accept x.
end for
Reject x.

First, note that even though the machines $M_{f(i)}$ are treated as acceptors for a language, it is possible to use a machine that decides SAT to find a satisfying assignment by running the machine a linear number of times due to the self-reducibility of SAT [Rot05, p. 107].
Since the algorithm checks whether the produced assignment is valid, it will never incorrectly accept an input, so only members of SAT are accepted. Since R is the approximation rate of SAT by P, $R(i)$ gives the minimum length of x for which no machine $M_{f(0)}, \ldots, M_{f(i)}$ will correctly decide the membership in SAT of all words up to that length. Conversely, this means that for $i \coloneqq R^{-1}(|x| + 1)$, there is a machine $M_{f(0)}, \ldots, M_{f(i)}$ which correctly decides the membership in SAT of all words up to length $|x|$. If $x \in \mathsf{SAT}$, that machine will allow finding a satisfying assignment, since the self-reduction does not require deciding any longer words. Since R^{-1} is bounded by g, the loop will find that machine at some point, find the satisfying assignment, and accept x. Therefore, every member of SAT is accepted and the algorithm decides SAT.
For an input of length n, the loop runs at most $g(n + 1) + 1$ times. In each iteration, we simulate a machine $M_{f(i)}$ with $i \leq g(n + 1)$, which takes at most $n^{\log g(n+1)}$ steps, and we simulate it at most n times to find the assignment, so each loop iteration takes $n \cdot n^{\log g(n+1)} = n^{1+\log g(n+1)}$ steps. Computing $g(n+1)$ is possible in time $g(n+1)$ because g is time-constructible. This gives a total runtime in $\mathcal{O}(n^{1+\log g(n+1)} \cdot g(n + 1))$. □

4.4 Approximation and independence

We can now show the following relationship between approximation rates and provability as presented in [BH91]:

Theorem 4.16. $\mathsf{P} \neq \mathsf{NP}$ *is provable in the theory* T_1 *iff* $R^{\mathsf{P}}_{\mathsf{SAT}}$ *is dominated by the Wainer hierarchy.*

Proof. " $\Longleftarrow$ ": Let F_α be a function from the Wainer hierarchy that dominates $R^{\mathsf{P}}_{\mathsf{SAT}}$. The definition of the approximation rate means the following formula is true:
$\forall i \exists x, |x| \leq R^{\mathsf{P}}_{\mathsf{SAT}}(i) : (x \in \mathsf{SAT} \leftrightarrow M_{f(i)}$ rejects $x)$. Since F_α dominates $R^{\mathsf{P}}_{\mathsf{SAT}}$, this implies the following formula is also true: $\forall i \exists x, |x| < F_\alpha(i) : (x \in \mathsf{SAT} \leftrightarrow M_{f(i)}$ rejects $x)$.

Because F_α is provably recursive by Lemma 4.4, the existential quantifier is bounded. Thus, this is a true Π_1 formula, which makes it provable in T_1. Since the formula implies that no machine $M_{f(i)}$ can recognize SAT correctly on all inputs, this shows that $\mathsf{P} \neq \mathsf{NP}$ is also provable in T_1.
" $\Longrightarrow$ ": Let $\mathsf{P} \neq \mathsf{NP}$ be provable in the theory T_1. Since T_1 is sound, this means $\mathsf{P} \neq \mathsf{NP}$ must be true. Therefore $\mathsf{SAT} \notin \mathsf{P}$ is true and provable and by Lemma 4.14, $R^{\mathsf{P}}_{\mathsf{SAT}}$ must be total. Suppose there is no F_α in the Wainer hierarchy that dominates $R^{\mathsf{P}}_{\mathsf{SAT}}$. Since $R^{\mathsf{P}}_{\mathsf{SAT}}$ is total and computable, Lemma 4.5 then shows that $R^{\mathsf{P}}_{\mathsf{SAT}}$ is not provably recursive. But if $\mathsf{P} \neq \mathsf{NP}$ (and thus $\mathsf{SAT} \notin \mathsf{P}$) can be proved in T_1, then the naive algorithm for $R^{\mathsf{P}}_{\mathsf{SAT}}$ is provably total, which would make $R^{\mathsf{P}}_{\mathsf{SAT}}$ provably recursive. This contradiction shows that the dominating function F_α must exist. □

Combining this with Theorem 4.12 gives the following corollary:

Corollary 4.17. *If the independence of* $\mathsf{P} \neq \mathsf{NP}$ *from* T *is provable by any "imaginable technique", then the search problem for* SAT *(finding a satisfying assignment for a known member of* SAT*) is in* $\mathsf{DTIME}(n^{f^{-1}(n+1)})$*, where* f *is a function not dominated by the Wainer hierarchy.*

Proof. By Theorem 4.12, the independence result from T can be extended to an independence result from T_1. Therefore, Theorem 4.16 shows that $R^{\mathsf{P}}_{\mathsf{SAT}}$ is not dominated by the Wainer hierarchy. We set $f \coloneqq R^{\mathsf{P}}_{\mathsf{SAT}}$. Now we can use the algorithm from the proof of Lemma 4.15 to find a satisfying assignment for x. Since $x \in \mathsf{SAT}$, the loop will terminate prematurely and we do not need to compute the bound $f^{-1}(|x|+1)$. This gives us a deterministic algorithm for the search problem with runtime $f^{-1}(n+1) \cdot n^{1+\log f^{-1}(n+1)} \leq n^{f^{-1}(n+1)}$. □

Since the Wainer hierarchy is a family that contains very fast-growing functions, f must grow extremely quickly, and f^{-1} must therefore grow extremely slowly, i.e. be very close to constant. More precisely, for any n, f^{-1} is constant on the interval from $f(n)$ to $f(n+1)$. These constant intervals grow in size very quickly. By Lemma 4.8, $n^{f^{-1}(n+1)}$ is infinitely often bounded from above throughout very long intervals by e.g. $n^{\alpha(n+1)}$, where α is the inverse Ackermann function, an extremely slow growing standard complexity function. This means that the runtime for this algorithm is, in a sense, very close to polynomial.

4.5 One-way functions

One common real-world application of the $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ question is the existence of one-way functions, which are crucial to cryptography. $\mathsf{P} = \mathsf{NP}$ would imply the nonexistence of one-way functions and therefore the impossibility of various cryptographic primitives,

although the converse is not necessarily true. However, [BH91] show that the nonexistence of a certain type of one-way functions is already implied if $\mathcal{PA}_1$ cannot prove $\mathsf{P} \neq \mathsf{NP}$, even if it cannot prove $\mathsf{P} = \mathsf{NP}$ either. This is another expression of the idea that an independence result for $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ implies that P and NP are "almost" equal.
First, we define our notion of one-way functions.

Definition 4.18. *Let f be a function computable in polynomial time.*
Let g be a standard complexity function. f is a uniform g-one-way function *iff for every deterministic Turing machine M with runtime in $\mathcal{O}(n^{g(n)})$, there is a minimum word length N such that for all $n > N$, there is a word $x \in \{0,1\}^n$ of length n for which $f(M(x)) \neq x$.*
f is a uniform one-way function *iff there exists a standard complexity function g such that f is a uniform g-one-way function.*

Intuitively, for sufficiently high word lengths, any algorithm attempting to invert f will produce the wrong result for at least one word of a given length. In one sense, this is weaker than the typical definition of a one-way function, which requires that the probability of inverting a random input successfully is negligible. On the other hand, this definition is stronger in the sense that it demands this property not only of polynomial time algorithms, but of any algorithms with runtime in $\mathcal{O}(n^{g(n)})$, which is a super-polynomial runtime bound.

Theorem 4.19. *If $\mathcal{PA}_1$ cannot prove $\mathsf{P} \neq \mathsf{NP}$, uniform one-way functions do not exist.*

Proof. Let f be a function that is computable in polynomial time and g be a standard complexity function. Let $R \coloneqq R^{\mathsf{P}}_{\mathsf{SAT}}$. By Lemma 4.15, there is a deterministic algorithm for SAT with runtime in $\mathcal{O}(n^{1+\log R^{-1}(n+1)} \cdot R^{-1}(n+1))$. The problem of inverting the polynomial time function f is clearly in FNP and thus reducible to SAT. This shows the existence of a deterministic machine M that inverts f in time $\mathcal{O}(n^{k\cdot(1+\log R^{-1}(n^k))} \cdot R^{-1}(n^k))$ for some k (substituting polynomial terms for the linear ones to account for the runtime of the reduction function).
Since g is a standard complexity function, so is the function h defined by $h(n) = 2^{\frac{1}{2k}g(\log n)}$, as well as the function h' given by $h'(n) = h(n^k)$. By Theorem 4.16, the nonprovability of $\mathsf{P} \neq \mathsf{NP}$ in $\mathcal{PA}_1$ implies that R is not dominated by any function in the Wainer hierarchy, and therefore neither is its composition with n^k. Thus, according to Lemma 4.8, there exist infinitely many n with $R^{-1}(n^k) < h'(n) = 2^{\frac{1}{2k}g(\log n^k)}$. Taking the logarithm on both sides and rearranging gives $2k \log(R^{-1}(n^k)) < g(\log n^k) < g(n)$ for infinitely many n.

We rewrite the runtime bound for M as follows:

$$\begin{aligned} n^{k\cdot(1+\log R^{-1}(n^k))} \cdot R^{-1}(n^k) &= n^{k\cdot(1+\log R^{-1}(n^k))} \cdot n^{\frac{\log(R^{-1}(n^k))}{\log n}} \\ &= n^{k\cdot(1+\log R^{-1}(n^k))+\frac{\log(R^{-1}(n^k))}{\log n}} \end{aligned}$$

Let $e(n)$ be the exponent in the above expression. For sufficiently large n, we can bound the exponent from above:

$$\begin{aligned} e(n) &= k \cdot (1 + \log R^{-1}(n^k)) + \frac{\log(R^{-1}(n^k))}{\log n} \\ &= k + k \cdot \log R^{-1}(n^k) + \frac{\log(R^{-1}(n^k))}{\log n} \\ &< k + k \cdot \log R^{-1}(n^k) + \log(R^{-1}(n^k)) \\ &< k \cdot (\log R^{-1}(n^k) - \frac{\log R^{-1}(n^k)}{k}) + k \cdot \log R^{-1}(n^k) + \log(R^{-1}(n^k)) \\ &= k \cdot \log R^{-1}(n^k) - \log R^{-1}(n^k) + k \cdot \log R^{-1}(n^k) + \log R^{-1}(n^k) \\ &= 2k \log(R^{-1}(n^k)) \end{aligned}$$

As shown above, $2k \log(R^{-1}(n^k))$ is bounded above by $g(n)$ for infinitely many n. Therefore the runtime of M, the algorithm for inverting f, is bounded by $n^{g(n)}$ for infinitely many n. We construct a machine M' that simulates M for at most $n^{g(n)}$ steps, returning 0 if the simulation does not halt in that time. For infinitely many n, the simulation will halt and M' will succeed at inverting f. Since the definition of uniform one-way functions allows at most finitely many n for which an algorithm may succeed at inverting f for all inputs of length n, f is not a uniform g-one-way function. But since f and g were chosen arbitrarily, this means no function computable in polynomial time can be uniform g-one-way for any standard complexity function g and thus uniform one-way functions do not exist. □

Applying Theorem 4.12, this means that if $\mathsf{P} \neq \mathsf{NP}$ is independent of $\mathcal{PA}$, uniform one-way functions do not exist, potentially having a similar practical impact as if $\mathsf{P} = \mathsf{NP}$ were provable, although the nonstandard definition of one-way functions used for this result should be kept in mind here.

5 Conclusion

5.1 Other results

We focused on independence results related to $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$. However, there are also other similar results in complexity theory and formal language theory. In [Har85], various results are presented that show the existence of a language $L \in A \setminus B$ for some classes A and B such that "$L \notin B$" is independent of a theory for any representation of the language L. For example, such results exist for non-regular context-free languages and for languages in $\mathsf{SPACE}(n^2) \setminus \mathsf{SPACE}(n)$.
A runtime bound for SAT in case of an independence result for $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$, similar to Corollary 4.17, is also constructed in [KOR87].

5.2 Future work

Little progress has been made in this area in recent decades. Of the primary results presented in this paper, the most recent ones were published in 1991 [BH91].
In the area of relativization, the usefulness of relativized results has since been called into question due to the proof that the random oracle hypothesis is false [Cha+94]. Specifically, $\mathsf{IP} = \mathsf{PSPACE}$, but for almost all oracles A, $\mathsf{IP}^A \neq \mathsf{PSPACE}^A$. This means that even results concerning equality of complexity classes such as P^A and NP^A that hold for almost all oracles would not necessarily imply any results about the unrelativized versions.
An area that shows more potential for future exploration is the potential consequences of an independence result. There are many known consequences of $\mathsf{P} = \mathsf{NP}$, such as the nonexistence of many cryptographic primitives and the existence of efficient algorithms for many problems known to be in NP. Similarly, there are many results in various areas of complexity theory that are known to follow from $\mathsf{P} \neq \mathsf{NP}$. However, the consequences of an independence result are underexplored. It would be interesting to see, for example, an extension of Theorem 4.19 to a more standard definition of one-way functions, or an exploration of other independence results that might follow from one for $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ and their respective consequences.

Bibliography

[Aar03] Scott Aaronson. "Is P versus NP formally independent?" In: *Bulletin of the EATCS* (2003). URL: https://www.scottaaronson.com/papers/indep.pdf.

[BBJ07] George S. Boolos, John P. Burgess, and Richard C. Jeffrey. *Computability and Logic*. Cambridge University Press, 2007. ISBN: 978-0-521-87752-7. DOI: 10.1017/CBO9780511804076.

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

[BH91] Shai Ben-David and Shai Halevi. *On the independence of P versus NP*. Tech. rep. 699. Department of Computer Science, Technion, 1991. URL: https://api.semanticscholar.org/CorpusID:2979321.

[Cha+94] Richard Chang et al. "The random oracle hypothesis is false". In: *Journal of Computer and System Sciences* 49.1 (1994), pp. 24–39. DOI: 10.1016/S0022-0000(05)80084-4.

[Cla] Clay Mathematics Institute. *P vs NP - Clay Mathematics Institute*. URL: https://www.claymath.org/millennium/p-vs-np/ (visited on 2024-03-14).

[Coh63] Paul J. Cohen. "The independence of the continuum hypothesis". In: *Proceedings of the National Academy of Sciences of the United States of America* (1963). DOI: 10.1073/pnas.50.6.1143.

[Coo71] Stephen A. Cook. "The complexity of theorem-proving procedures". In: STOC '71. Association for Computing Machinery, 1971, pp. 151–158. ISBN: 9781450374644. DOI: 10.1145/800157.805047.

[FLO83] Steven Fortune, Daniel Leivant, and Michael J. O'Donnell. "The expressiveness of simple and second-order type structures". In: *J. ACM* 30 (1983), pp. 151–185. DOI: 10.1145/322358.322370.

[Har85] Juris Hartmanis. "Independence results about context-free languages and lower bounds". In: *Information Processing Letters* 20.5 (1985), pp. 241–248. DOI: 10.1016/0020-0190(85)90026-2.

[Har89] Juris Hartmanis. *Feasible Computations and Provable Complexity Properties*. Society for Industrial and Applied Mathematics, 1989. ISBN: 0-89871-027-8. DOI: 10.1137/1.9781611970395.

[HH76] Juris Hartmanis and John E. Hopcroft. "Independence results in computer science". In: *SIGACT News* 8 (1976), pp. 13–24. DOI: 10.1145/1008335.1008336.

[Jec03] Thomas Jech. *Set Theory*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. ISBN: 978-3-540-44761-0. DOI: 10.1007/3-540-44761-X_2.

[KOR87] Stuart A. Kurtz, Michael J. O'Donnell, and James S. Royer. "How to prove representation-independent independence results". In: *Information Processing Letters* 24.1 (1987), pp. 5–10. DOI: 10.1016/0020-0190(87)90191-8.

[Men97] Elliott Mendelson. *Introduction to Mathematical Logic*. Fourth Edition. Discrete Mathematics and Its Applications. Chapham & Hall/CRC, 1997. ISBN: 978-0-412-80830-2. DOI: 10.1007/978-1-4615-7288-6_1.

[Rog87] Hartley Rogers. *Theory of Recursive Functions and Effective Computability*. Cambridge, MA, USA: MIT Press, 1987. ISBN: 0-262-68052-1. DOI: 10.5555/28907.

[Rot05] Jörg Rothe. *Complexity Theory and Cryptology: An Introduction to Cryptocomplexity*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, p. 107. ISBN: 978-3-540-28520-5. DOI: 10.1007/3-540-28520-2_3.

[Sav70] Walter J. Savitch. "Relationships between nondeterministic and deterministic tape complexities". In: *Journal of Computer and System Sciences* 4.2 (1970), pp. 177–192. DOI: 10.1016/S0022-0000(70)80006-X.

[Soa87] Robert I. Soare. *Recursively Enumerable Sets and Degrees: A Study of Computable Functions and Computably Generated Sets*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987. ISBN: 3-540-15299-7. DOI: 10.5555/22895.

[Wai70] Stanley S. Wainer. "A classification of the ordinal recursive functions". In: *Archiv für mathematische Logik und Grundlagenforschung* 13 (1970), pp. 136–153. DOI: 10.1007/BF01973619.