2006

Limits of Diagonalization and the Polynomial Hierarchy

Kyle Barkmeier '06
Illinois Wesleyan University

Limits of Diagonalization and the Polynomial Hierarchy

Research Honors Thesis
Kyle Barkmeier
Department of Computer Science
Illinois Wesleyan University

Thanks to my honors committee:
Hans-Jörg Tiede, Project Advisor
Leonard Clapp
Lawrence Stout
Paul J. Kapitza

1 Introduction

Determining the computational complexity of problems is a large area of study. It seeks to seprate these problems into ones with "efficient" solutions, and those with "inefficient" solutions. Of course, the strata is much more fine-grain than this. Of special interest are two classes of problems: P and NP. These have been of much interest to complexity theorists for quite some time, because both contain many instances of important real-world problems, and finding efficient solutions for those in NP would be beneficial for computing applications. Yet with all this attention, there are still important unanswered questions about the two classes. It is known that $P \subseteq NP$, however it is still unknown whether $P = NP$ or if $P \subset NP$. Before we discuss why this problem is so crucial to complexity theory, an overview of P, NP, and coNP is necessary.

The class P is a model of the notion of "efficiently solvable", and thus contains all languages (problems) that are decidable in deterministic polynomial time. This means that any language in P has a deterministic Turing Machine (algorithm) that will either accept or reject any input in n^k steps, where n is the length of the input string, and k is a constant. The class NP contains all languages that are decidable in nondeterministic polynomial time. A nondeterministic Turing Machine is one that is allowed to "guess" the correct path of computation, and seems to be able to reach an accept or reject state faster than if it was forced to run deterministically. It is unknown whether NP is closed under complementation because of this nondeterminism. It is quite easy to show a class of deterministically-solvable languages (such as P) is closed under complementation: we simply reverse the accept and reject states. This method is not viable for a nondeterministic machine, since switching the accept and reject states will result in machine that computes a completely different language. Thus the class coNP is defined as containing the complement of every language in NP.

In the rest of this paper we will present structural definitions of P and NP as well as present example languages from each. These structural definitions will give insight into the arrangement of the polynomial hierarchy, which is discussed in section 3. A diagonalization proof is presented in section 4, and an explanation of the general usage of diagonalization follows. In section 5, universal languages are defined and an important result from Kozen is given. In the final section, the limits of diagonalization as they pertain to P and NP are outlined, as well as the same limits for relativized classes.

2 P and NP

Another way to illustrate the differences between these complexity classes is with quantified logic statements. A language in P can be defined by a predicate $P(x, y)$, where P relates x to y (we would merely code this pair as a string to evaluate it). For example, suppose y is a propositional logic formula and x is a truth value assignment to the variables in this formula. The relation $P(x, y)$ will either hold or fail for the truth assignment with relation to y and can be checked in polynomial time. All languages can be defined by a predicate such as the language $\{\langle x, y\rangle \mid x$ is a truth assignment that satisfies the propositional logic sentence $y\}$ which would be in P. The languages in NP contain an existential quantifier in front of them, thus: $\exists x P(x, y)$, thus the previous language would become a problem of satisfiability where we ask whether x satisfies y. This illustrates the distinction between solvablility and verifiability. For languages that can be solved in polynomial time, the x is "given", but for verifiable languages, the predicate merely states that there exists some x, so it must be found by the algorithm. With this definition, the languages in coNP defined with a universal quantifier: $\forall x P(x, y)$ where P can be checked in polynomial time. Now our satisfiable sentence of propositional logic asks whether the sentence is a tautology; if all x's satisfy y. For further insight into P and NP, consider the following problems from propositional logic.

The problem 2SAT is defined as having a set of variables U and a collection C of clauses over the set U, where each clause in C is in conjunctive normal form with exactly two variables. The question is whether there is a satisfying truth assignment for C. What this is describing is a sentence of propositional logic that contains a finite number of variables (such as x, y, z, etc.) all arranged into conjunctive normal form. A formula is in conjunctive normal form when it is a series of conjucts, each of which is a disjunct. For example, the sentence $(x \vee y \vee z) \wedge (\neg y \vee w \vee \neg z) \wedge (x \vee y)$ is in CNF. For 2SAT, each conjuct (a clause; $(x \vee y)$ in the previous example) must have only two literals. This problem is in P, as it has a deterministic polynomial time solution as given by the following metatheorem:

Proof First, use Conditional-Disjunction to transform all the disjuncts into conditional statements. With this information, a directed graph is drawn such that an arrow extends from one node to the next only if there is a conditional between the variables represented by those nodes. The formula is unsatisfiable if there are two paths that lead to contradictory variables form the same source node (for example, a path from y to x and another path from y to $\neg x$). For a satisfiable formula, all connected paths give satisfying truth assignments. This solution can be computed in polynomial time, as none of the steps take more than n^k time where n is the length of the input (the formula). □

This method will produce a satifying truth assignment for the given 2SAT formula without any input other than the formula itself. Now consider the the problem 3SAT. It is defined in the same way as 2SAT, except that each clause has three variables. It happens that 3SAT is in NP, meaning that unlike 2SAT, we do not know if it has a deterministic polynomial time solution for it, but we can easily construct a a deterministic polynomial time *verifier*, as follows:

Proof On an input containing the formula and a possible solution, check to see if there exists two contradictory truth assignments (x and $\neg x$, for example) in the possible

solution. If so, reject. Otherwise, check each clause to make sure at least on variable is true in each clause. If so, accept. Otherwise, reject. Each step can be computer in polynomial time or less, thus placing this language in NP. □

This algorithm merely verifies that a solution is in fact a satisfiable assignment. To find a solution to 3SAT, we could add a step to the beginning of this algorithm that nondeterministically "guesses" a satisfying solution, then go through and check that it is correct. Thus, 3SAT is in NP. In fact, 3SAT is NP-complete, which will be defined shortly, but first we must examine polynomial-time reducibility.

Definition 1 ***Polynomial-time reducibility.*** *A language L is polynomial-time reducible to a language R if and only if a polynomial time computable function* $f : \Sigma^* \rightarrow \Sigma^*$ *exists such that:*

$\forall x,\ x \in L \leftrightarrow f(x) \in R.$

What polynomial-time reduction embodies is the ability to efficiently convert strings recognized by one language into strings recognized by a second language. This grants the ability to compute answers for one language using a solution to another language. With this, a definition of completeness can be presented.

Definition 2 ***Completeness.*** *A language L is complete for complexity class* $\mathcal{A}$ *if and only if:*

1. $L \in \mathcal{A}$
2. *For all languages* $R \in \mathcal{A}$, $R \leq_p L$.

When a language is said to be complete for a class, it means that the language is as "hard" as every other language in that class, as given by part 2 of the definition. If finding a solution to every problem in a complexity class can be reduced to solving one single problem, it is obviously a very difficult one.

Now that we have become more familiarized with the nature of P and NP, let us examine a larger construct of important problems known as the polynomial hierarchy.

3 The Polynomial Hierarchy

The polynomial hierarchy in computational complexity refers to the structure of classes whose languages' P relations contain growing iterations of quantifiers. The hierarchy progresses upwards in a fashion similar to the description of P, NP, and coNP above. That is, P comprises the first level, and NP and coNP are the second level, as they are defined as an exists and a forall, respectively. The third level is comprised of one class that contains exactly one alternation of quantifiers, as in $\exists x \forall y P(x, y, z)$. Notice that $\exists x \exists y \exists z \forall s P(x, y, z, s)$ happens to be of the same complexity, since it is only the *alternation* of the quantifiers that matters. The fourth level would contain two classes each with two quantifier alternations, the fifth level one class with three alternations, and so on.

There are three main ways of defining the polynomial hierarchy, one of which was briefly touched on above. For this project, we will define the hierarchy in terms of oracle machines. An oracle TM is a regular TM augmented with an extra tape, called

the oracle tape (or query tape). The query tape corresponds to a "black box" type of computation for a certain language (called the oracle, or oracle language) that will tell the oracle TM in one step whether a given string written on the query tape is in the oracle language or not. For example, the class P^A would be the class of languages with polynomial time oracle Turing Machines where the oracle is A. This appears to give us greater versatility and computational "power", as we may be able to include certain languages in an "easier" class by solving them with respect to the language A (and thus add only a small overhead of time). With this definition, we can now define the polynomial time hierarchy a second way [1]:

Each class of the polynomial hierarchy is defined using either Σ_i, Π_i, and Δ_i. The hierarchy is defined in the following way:

$$\begin{aligned} \Sigma_0 = \Pi_0 &= \Delta_0 = P \\ \text{For all } i \geq 0 & \\ \Delta_{i+1} &= P^{\Sigma_i} \\ \Sigma_{i+1} &= NP^{\Sigma_i} \\ \Pi_{i+1} &= co - \Sigma_{i+1} \end{aligned}$$

In other words, Δ_{i+1} is the same as the class P with an oracle tape for the class of languages Σ_i (which is the previous level of NP languages). Σ_{i+1} is the same as the class NP with an oracle tape for the class of languages Σ_i, and Π_{i+1} is the complement of the current level of NP languages. When the "level" is referred to, it means the size of i in the inductive process, thus the level of $i = 3$ would be 3, and the previous level would be 2. As an example, $\Sigma_3 = NP^{NP^{NP^P}}$, though the final P can be omitted, since P is a subset of NP.

This entire hierarchy is contained inside of PSPACE, which allows for an interesting result from Sipser [2]. But first, another language must be outlined.

Quantifier Boolean Formulas defines the problem of solving a boolean expression with quantified variables. An instance of the problem would be $F = (Q_1 x_1)(Q_2 x_2)\ldots(Q_n x_n)E$, where E a boolean expression containing the variables $x_1, x_2, \ldots, x_n$, and each Q_i is either "$\forall$" or "$\exists$". QBF is a PSPACE-complete language.

Now we can proceed with the proof [2].

Theorem 1 *There is an oracle A such that $P^A = NP^A$.*

Proof Let our oracle be QBF. Thus, $NP^{QBF} \subseteq$ NPSPACE since any nondeterministic oracle machine can be reduced to a nondeterministic polynomial space machine that directs its queries to QBF directly instead of using a QBF oracle. This is possible because QBF is PSPACE-complete and therefore as "hard" as every language in PSPACE (which includes the polynomial hierarchy). By Savitch's theorem, NPSPACE $\subseteq$ PSPACE. Since QBF is PSPACE-complete, PSPACE $\subseteq P^{QBF}$ (any language in PSPACE can be reduced to queries to QBF, which we can solve in constant time via the oracle). Thus, $P^{QBF} = NP^{QBF}$ □

With this information, we are prepared to discuss another important idea: diagonalization and its limits.

4 Diagonalization

Diagonalization is a proof technique used to separate sets, which was first introduced by Cantor in 1874, its name coming from the fact that Cantor's proof process could be drawn as a table and the reverse (in terms of true/false) of the diagonal elements would comprise the separating language. Its most common form is what we will call "constructive separation" and in complexity theory involves the defining of a specific language (usually through recursion) and from there proving that this language is in one complexity class, but not the other. To illustrate this method, we will present a proof that there exists an oracle language B such that $\mathrm{P}^B \neq \mathrm{NP}^B$ [2].

Theorem 2 *There exists an oracle B such that* $P^B \neq NP^B$.

Proof Let L be a language. $L = \{w \mid \exists x \in B$ such that $\mid x \mid = \mid w \mid\}$ that is, L is the language containing all strings where a string of equal length exists in the oracle B. L is in NP^B since we can easily create a machine that nondeterministically finds all strings the same length as those in the oracle. To prove that $\mathrm{L} \notin \mathrm{P}^B$, we will consider the list of all polynomial time oracle TMs $M_1, M_2, \ldots$ where each M_i runs in time n^i. B will be constructed in stages, each of which decides only a finite number of strings.

Stage i Choose an n such that $2^n > n^i$ (meaning the total number of strings of length n is greater than the running time of M_i^B). We will run M_i^B on 1^n. At this point, there is a finite list of strings that have been determined to be in B or out of B. When M_i queries the oracle with a string, if that string's status has been determined, we will respond consistently. However, if that string's inclusion or exclusion has not been determined yet, we will respond with NO and declare it to be out of B. When we run M_i on 1^n, it will accept if it finds a string of length n that is in B. However, M_i does not have enough time to query all strings of length n, since we have chosen n such that $2^n > n^i$, so when M_i halts (and has not found a string of length n in B), it will have to make a decision whether or not to accept or reject 1^n. This is where diagonalization comes in. We will make sure any decision M_i makes is wrong. This is done by expanding the set of strings in B. If M_i accepts 1^n, we declare every string of length n that M_i did not query to be out of B, and therefore 1^n cannot be in L. If M_i rejects 1^n, we find an unqueried string of length n and declare it to be in B, thus $1^n \in L$. We then continue with stage $i + 1$.

Once all stages have finished, we declare all strings left undetermined to be out of B. At each stage of the construction of B, every polynomial time oracle turing machine has failed to decide L with the oracle B. Thus, $L \notin \mathrm{P}^B$. Therefore, there exists an oracle B such that $\mathrm{P}^B \neq \mathrm{NP}^B$. □

Our method of diagonalization in this proof is a constructive method, since we are constructing B so that no matter what any of the M's do, they cannot accept it. This is the basic template of constrictive separation by diagonalization: define a language using a set's own properties so that this language cnanot possibly be contained in the set. This idea will become clearer with the definition of universal languages that follows.

5 Universal Languages and Kozen's Result

Universal languages are a useful tool for dealing with classes of languages. A universal language is simply a language which contains all the languages of a specific class. Think of them as a large 0,1-valued (binary) matrix, each row corresponding to a language, and each individual cell in the row corresponding to a single string that is in that language. Thus, the universal language U is the matrix $[a_{ij}]$, where the $i$the row corresponds to a language U_i, and the jth entry is 1 or 0 corresponding to the jth string's inclusion or exclusion from the language. Each universal language has a linear-time computable pairing function that allows us to "view" each row, as well as individual cells. This function is denoted as $U(\langle i, j\rangle)$ where i is the row and j is the desired string.

There exists two different types of universal languages. The first is known as a strong universal language (or strict universal language). A strong universal language is one where every row corresponds to a language in one specific class. To illustrate, suppose U is a strong universal language for a class $\mathcal{A}$ (denoted $U \multimap\!\bullet\, \mathcal{A}$). This means that $\mathcal{A} = \{U(\langle i, *\rangle) \mid i \in \mathbb{N}\}$. The other kind of universal language is known as a weak universal language (denoted $U \multimap \mathcal{A}$), meaning that the rows corresponding to a class $\mathcal{A}$ are still in U, but U contains more than just those rows. So $U \multimap \mathcal{A}$ if and only if $\mathcal{A} \subseteq \{U(\langle i, *\rangle) \mid i \in \mathbb{N}\}$.

Definition 3 *The diagonal of a universal language U is denoted as $diag_U$ and is defined as $diag_U(x) := 1 - U_x(x)$. Meaning that the $diag_U(x)$ is the opposite of the value in the cell located at the xth row and xth.*

Definition 4 *A set of languages $\mathcal{A}$ differs under finite variations if and only if $(\forall B \in \mathcal{A})(A =^* B \in \mathcal{A} \rightarrow A \in \mathcal{A})$ where $A =^* B$ means that A and B differ on a finite number of positions [4].*

This leads us to an important theorem from Kozen [3]:

Theorem 3 ***(Kozen)*** *If a class $\mathcal{A}$ is closed under finite variations and there is a universal language V such that $V \multimap\!\bullet\, \mathcal{A}$, then for every $L \notin \mathcal{A}$, there is a universal language U computable in V and L such that $U \multimap\!\bullet\, \mathcal{A}$ and $L = diag_U$.*

What this says is that any proof of a language L's exclusion from a class $\mathcal{A}$ can be rewritten as a diagonalization proof, since any language not in a class that is closed under finite variations is the diagonal of some universal language for that class.

This proof, when combined with theorem 1, presents a problem. Suppose that there is a proof of P $\neq$ NP. Kozen's result tells us that this proof can be made into a diagonalization proof of P $\neq$ NP. Since diagonalization proofs generally hold for relativized cases, it must be that $P^A \neq NP^A$, but theorem 1 states that $P^A = NP^A$! So how do we resolve this apparent contradiction?

6 Final Results

6.1 The Limits of Diagonalization for P and NP

Our first step to resolving the contradictions between theorem 1 and theorem 3 is to prove that Kozen's result does in fact hold for P, since. To do this, we need to show that P is closed under finite variations.

Theorem 4 *P is closed under finite variations*

Proof Assume that a language $L \in \mathrm{P}$, and x is a string. There is machine that can decide the language containing just the string x. Furthermore, this machine runs in linear time. Since $L \in \mathrm{P}$, there must be a machine that decides L in polynomial time. Thus, we can augment L's machine with the machine that decides $\{x\}$ and only add a linear time overhead. Therefore, there is a polynomial time DTM that decides $L \cup \{x\}$, and thus $L \cup \{x\} \in \mathrm{P}$.

This same proof technique works to prove that $L - \{x\} \in \mathrm{P}$. Since $\{x\}$ has a linear time deterministic machine that decides it, we simply replace the accept state of that machine with a reject state, and then augment L's machine with this new machine, computing $L - \{x\}$ in polynomial time. Therefore, $L - \{x\} \in \mathrm{P}$.

Thus, P is closed under finite variations. □

Since P is closed under finite variations and all classes have at least one universal language, Kozen's theorem holds for P. This means that any proof of a language's exclusion from P can be recast as a diagonalization proof. How do we reconcile this with our results from theorems 1 and 2?

The simple answer is this: strong diagonalization does not relativize. Consider these theorems from [Nash, Impagliazzo, & Remmel 2003].

Theorem 5 *There is a computable oracle A such that $\exists U(U \multimap\bullet P^A)$ and $U \in NP^A$.*

Theorem 6 *There are computable languages A and C such that $\exists U(U \multimap\bullet P^A)$ and $U \in NP^A$, and yet $(NP^A)^C = (P^A)^C$. Therefore, $\forall U(U \multimap\circ (P^A)^C)$, $U \notin (NP^A)^C$.*

Proof Using the same A as in theorem 5, get that $\exists U(U \multimap\bullet \mathrm{P}^A$ and $U \in \mathrm{NP}^A)$. If we let C be a PSPACE^A-complete language, we can obtain the following containments:

$$(\mathrm{P}^A)^C \subseteq (\mathrm{NP}^A)^C \quad (1)$$

$$(\mathrm{NP}^A)^C \subseteq (\mathrm{PSPACE}^A)^C \quad (2)$$

$$(\mathrm{PSPACE}^A)^C = \mathrm{P}^C \quad (3)$$

$$\mathrm{P}^C \subseteq (\mathrm{P}^A)^C \quad (4)$$

Containments 1 and 2 hold by the definition of the polynomial hierarchy. $(\mathrm{PSPACE}^A)^C = \mathrm{P}^C$ since C is a PSPACE^A-complete language, and therefore any language in PSPACE^A can be reduced to queries to C. Containment 4 holds by the definition of oracles. Thus, $(\mathrm{NP}^A)^C = (\mathrm{P}^A)^C$, and therefore $\forall U(U \multimap\circ (\mathrm{P}^A)^C)$, $U \notin (\mathrm{NP}^A)^C$, since any class in the polynomial hierarchy can contain a weak universal language for itself [4]. □

This theorem shows that strong diagonalization does not relativize. Thus, we can conclude that the existence of oracles proving P and NP separate and equal is, in fact, reconcilible with Kozen's result. The reason for this is that Kozen's theorem uses a form of diagonalization that does not relativize, so contradictory proofs involving relativized cases have no bearing.

6.2 The Limits of Diagonalization for P^{NP} and NP^{NP}

Now we will show that diagonalization's inability to separate P and NP holds for the level of the polynomial hierarchy directly above it, specifically $i = 1$. This will give us $\Delta_2 = P^{NP}$ and $\Sigma_2 = NP^{NP}$.

Theorem 7 *There exists and oracle A such that* $P^{NP^A} = NP^{NP^A}$.

Proof We will proceed as in theorem 1. Using QBF as our oracle, we get:

$$NP^{NP^{QBF}} \subseteq NPSPACE \subseteq PSPACE \subseteq P^{NP^{QBF}}$$

All of the containments hold as in theorem 1. Thus, $P^{NP^{QBF}} = NP^{NP^{QBF}}$, and as such there exists and oracle A such that $P^{NP^A} = NP^{NP^A}$. □

Theorem 8 P^{NP} *is closed under finite variations.*

Proof We will proceed as in theorem 4. Let $L \in P^{NP}$ and x be a string. There is a linear time machine that decides $\{x\}$, and a machine that decides L. By augmenting L's machine with the machine for $\{x\}$, we will obtain a machine that recognizes $L \cup \{x\}$ with a negligible linear time overhead. Thus, $L \cup \{x\} \in P^{NP}$.

Since there is a linear time machine for $\{x\}$, there is a machine that will reject just x by replacing the accept state of the previous machine with a reejct state. If we augment L's machine with the machine for $\{\overline{x}\}$, we will obtain a machine that accepts $L \cup \{x\}$ with only a negligible linear time overhead. Thus, $L \cup \{x\} \in P^{NP}$).

Therefore P^{NP} is closed under finite variations. □

As previously stated, every class of languages has a universal language, thus P^{NP} has a universal language. Since P^{NP} is also closed under finite variations, theorem 3 holds for P^{NP}.

7 Conclusion

While the P versus NP question is still unresolved, we have refined our focus on the subject. Even though there is strong evidence against diagonalization's ability to separate P and NP, we have shown that it this does not apply to our notion of strong diagonalization. Furthermore, strong diagonalization is the only way to separate these P and NP, as Kozen demonstrated. Finally, this limit on separational proof techniques extends to the second level of the polynomial hierarchy, as we have shown.

References

[1] Rogers, H Jr. *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York, 1967.

[2] Sipser, M. *Introduction to the Theory of Computation*, p. 320 PWS Publishing, Boston, 1997.

[3] Kozen, D. "Indexings of Subrecursive Classes." *Theoretical Computer Science*, 11:277-301, 1980.

[4] Nash, A., Impagliazzo, R., Remmel, J. "Universal Languages and the Power of Diagonalization." *18th Annual IEEE Conference on Computational Complexity (CCC'03)*, p. 337, 2003.