Proof Triangles: Toward a Formal Theory of Mathematical Understanding

Brendan R. Meeder
Advisor: Manuel Blum
Graduate Advisor: Ryan Williams

May 2007

Contents

1 Introduction **1**
1.1 Previous Work 2

2 Technical Introduction **3**
2.0.1 Algorithmic Runtime 3
2.1 Asymptotic Analysis 3
2.2 Complexity Classes 4
2.3 Boolean Formulas 5

3 Proof Triangles **7**
3.1 Motivation for Proof Triangles 7
3.2 What are Proof Triangles? 8
3.3 Why Proof Triangles? 9

4 Algorithmic Understanding: Synthesizing New Proofs **11**
4.1 Introduction to algorithmic understanding 11
4.2 Formal definitions 11
4.3 Understanding in k-SAT formulas 12
4.3.1 Preliminaries for the system $\mathcal{S}_{kSAT}$ 12
4.3.2 A trivial algorithm with $\mathcal{O}(1)$ understanding 13
4.3.3 An algorithm with $\mathcal{O}(\log n)$ understanding 13
4.4 Limitations in understanding k-SAT formulas 14

5 Conclusions and Future Work **16**

Bibliography **18**

Acknowledgments

Although this experience has been far shorter than the usual PhD journey, I still want to thank two people who have been especially influential. First of all, I am grateful to Professor Manuel Blum advising me on this senior thesis. A lucid presentation of his "proof triangles" concept last spring inspired me to think more deeply about the way in which I do mathematics. Besides providing provocative ideas to chew on, he is always willing to talk about all sorts of problems not directly related to proof triangles. I have deeply enjoyed working on this problem and look forward to continued discussions with him during my PhD career.

Ryan Williams, my graduate student advisor, has been an especially influential and effective mentor. While Manuel and I enjoyed talking about the almost philosophical issues of problem solving, Ryan would help me work through the technical details which eventually needed to be carried out. Despite being a busy graduate student, Ryan made himself available for frequent discussions thoughout this time. Between having Ryan as a TA and as an advisor, he has proven to be a great role model; I hope to assist and inspire students just as effectively as he has influenced me during my academic career.

Abstract

Humans have developed great proficiency at creatively tackling mathematical problems. We seek to develop a computational model which encompasses many features of the human problem solving process. As a basic primitive for this model we have developed an object that we call a "proof triangle." A proof triangle for a theorem can be thought of as a triangle that has mathematical statements written on it. On its base appears the formal systems proof, and at the apex of the triangle are very short hints. The rest of the triangle is filled with more detailed hints leading to a derivation of the informal 'human-readable' proof, from which a formal proof can be derived. These objects encompass the knowledge required to guide an algorithm or human trying to solve the problem at hand, but not all information in a triangle may be required to find a solution.

We also develop formal definitions for concepts such as analogies and the extent to which an algorithm understands a proof P of a theorem T. Our approach is one rooted in complexity theory. We believe such methods will be useful in studying the broader problem of "understanding understanding." and provide a perspective which complements AI research in this area.

Chapter 1

Introduction

Mathematics can be thought of as one of the oldest and most influential intellectual activities. Over thousands of years, mathematics has served many roles; it is a tool to describe and predict the behavior of the heavenly bodies, a means of computing commercial transactions, and a way to formalize many problems that humans can pose. With the invention of calculating devices such as the abacus or Babbage's calculation engine, mathematics served as a tool of increasing usefulness. Once devices were built which could do thousands of additions in the time it took a human to do one, people were able to work out calculations never thought possible. The world truly changed because of these devices.

Logicians such as Frege, Russel, and Hilbert made great strides to formalize mathematics in a precise symbolic language. Today, we have symbolic calculi such as propositional calculus, first-order logic, and the lambda-calculus that unambiguously express mathematical statements. The Hilbert program was a push to put all of mathematics on a formal logical footing. Godel's incompleteness theorems showed that we cannot meet all of Hilbert's ideals, but we can still make an effort to formalize as much of our high-level theories as possible. Mathematicians of the Bourbaki school of thought did just this; they developed from axiomatic set theory the theory of analysis, topology, and algebra.

Computers are currently limited to manipulating mathematics in a very rigid symbolic system. Their view of the world is that of Bourbaki; from a set of axioms one must formally derive logical consequences one step at a time. Although there is merit to ensuring that mathematical proofs are undoubtedly correct, it is difficult for many mathematicians to work as such a low level. Often abstractions or conceptualizations are utilized when doing math. While discussing properties of continuous functions, it is often helpful (but not always correct) to think about curves which can be drawn without lifting the pencil. Computers only have the formal delta-epsilon definition to work with, while mathematicians have these mental images of continuous functions.

How is it that we could make a computer understand mathematics at a higher level? By thinking about the process by which we solve problems, we attempt to

understand how our approach brings together numerous ideas and methods. As instructors and students, the notion of 'hint' is frequently encountered. What is good advice for one student might not be useful for another. How can we quantify 'usefulness' of a hint? A more basic problem is defining the concept of a hint. Should we just take a hint to be bits of nondeterminism in a computational process? Should a hint reference a similar problem or 'strategy'? We venture into relatively uncharted territory by asking these questions. However, we feel that by approaching these questions in a rigorous manner, we can begin to answer questions about the mathematical problem solving process. If these ideas can be expressed formally, it may be possible to implement a system which does mathematics in a spirit more akin to human problem solving rather than raw symbol manipulation.

1.1 Previous Work

This project is mainly focused on a blend of topics not usually studied. Few investigations have taken place in work similar to proof triangles, and we outline this work here. Alan Bundy [2,3] has worked with high-level constructs in theorem proving. In [3], Bundy discusses techniques for planning proofs. Proof planning is restricted mainly to the domain of inductive reasoning, where an automated theorem prover must decide what to use as the inductive hypothesis and how to prove the inductive step. Proof patching sees when a strategy fails and offers alternative proof routes for recovering from these failures. Bundy and his students have created technique called rippling that is used in proving inductive statements. Rippling controls the application of term rewrites to ensure that rewrite termination occurs. Overall, Bundy's work has been focused primarily on developing automated theorem proving techniques for the problem of proving inductive proofs.

Beame et al. in [1] show lower bounds on the runtimes of Davis-Putnam resolution theorem provers. Namely, they show that random formulas will require exponentially long resolution proofs with high probability. The significance of this work to us is that in a certain model of theorems and proofs, there are proofs that require exponential space to write down. It's also interesting to see the techniques they use to prove lowerbounds for the runtime of the Davis-Putnam algorithm.

Chapter 2

Technical Introduction

This section develops the mathematical background needed to read this paper. We assume that the reader is familiar with the notion of algorithm. We briefly review the ideas of algorithmic runtime and asymptotic analysis. The required complexity classes and reductions between problems are reviewed. We also discuss SAT, conjuctive normal form, and their relation to NP-completeness.

2.0.1 Algorithmic Runtime

An *algorithm* is a process or procedure defined by a definite, fixed set of instructions. Given a particular computational model, we can measure the amount of 'time' it takes the algorithm to run. For each computational model, we have some notion of atomic or primitive operations. In the Turing Machine model, the number of times the machine reads a symbol can be a measurement of time it takes the algorithm to run. On a 'real' computer like an x86 box, we can measure running time in clock cycles, or we can abstract the notion of clock cycles and measure the number of machine instructions that execute while running the program.

Definition 1. *Let $T : \mathbb{N} \to \mathbb{N}$. We say that an algorithm $\mathcal{A}$ runs in time $T(n)$ if for all inputs x of length n, the number of steps required for $\mathcal{A}$ to run on x is less than or equal to $T(n)$. We denote the set of algorithms which run in time $T(n)$ by $DTIME(T(n))$*

2.1 Asymptotic Analysis

Most readers will be familiar with the notation of $\mathcal{O}$, Ω, and Θ. As a refresher:

Definition 2. *Let $f, g : \mathbb{N} \to \mathbb{N}$ be given functions. We say $f(n) = \mathcal{O}(g(n))$ if there exists a real number $c > 0$ and natural number N_0 such that*

$$\forall n[n \geq N_0 \;\Rightarrow\; f(n) \leq c \cdot g(n)]$$

(in words, whenever n is at least N_0, $f(n)$ is bounded above by $c \cdot g(n)$.)

We say $f(n) = \Omega(g(n))$ if there exists a real number $c > 0$ and natural number N_0 such that

$$\forall n[n \geq N_0 \Rightarrow f(n) \geq c \cdot g(n)].$$

That is, g bounds f from below.

Finally, $f(n) = \Theta(g(n)$ iff $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$.

These three relationships form the basis of most asymptotic analysis. Sometimes, however, it is useful to talk about growth rates in which we don't allow the reflexivity that $f(n) = \mathcal{O}(f(n))$ and $f(n) = \Omega(f(n))$. To describe these situations, there is the notation $f(n) = o(g(n))$ and $f(n) = \omega(g(n))$.

Definition 3. *A function $f(n) = o(g(n))$ if $f(n)/g(n) \to 0$ as $n \to \infty$. This is the same as $f(n) = \mathcal{O}(g(n))$ but $f(n) \neq \Theta(g(n))$. In this case, g bounds f from above but grows faster than any constant multiple of f. ω is defined in a similar way in which $f(n) = \omega(g(n))$ if and only if $f(n)/g(n) \to \infty$ as $n \to \infty$.*

2.2 Complexity Classes

Complexity classes describe the time or space complexity of a certain problem. For example, we know that bubble sort requires $\mathcal{O}(n^2)$ steps to sort a list of n integers. This is what's called *polynomial* growth since the runtime complexity grows like n^k for some natural number k. On the other hand, writing out all permutations of an n element list takes $n!$ print statements. It is easy to show that $n! = \Omega(n^k)$ for all natural number k, and furthermore that $n! = \Omega(c^n)$ for all real values $c > 1$.

Definition 4. *A language L of strings is decidable if there exists an algorithm $\mathcal{A}$ which returns true / false values such that $\mathcal{A}(x)$ always terminates and $\mathcal{A}(x) = T \iff x \in L$. A language is said to be undecidable if no such algorithm exists.*

There are many examples of decidable and undecidable languages. Turing's famous example of an undecidable language is the set of programs P that halt when given themselves as input. Some examples of decidable languages include the set of boolean formulas that have satisfying assignments and the set of undirected graphs with Eulerian tours. The difference between these two problems is the efficiency with which we can solve them. No known efficient algorithms exists to test whether a boolean formula is satisfiable, while one can quickly check whether a graph has an Eulerian cycle.

Definition 5. *PTIME is the set of languages that can be recognized in polynomial time by a deterministic Turing machine. Formally,*

$$PTIME = \bigcup_{k=0}^{\infty} DTIME(n^k)$$

Languages that are recognizable in polynomial time are said to be the tractible languages. Although a function such as n^1000 is polynomial and is initially much larger than $2^{n/1000}$, the exponential eventually dominates the polynomial.

The class of NP is characterized by languages that have short, polynomial length certificates of membership. This definition avoids introducing the idea of a nondeterministic machine. A central question in complexity theory is whether all languages in NP have a PTIME algorithm.

2.3 Boolean Formulas

The set of Boolean formulas φ is constructed recursively as follows:

1. A propositional variable x is a Boolean formula.
2. If φ is a Boolean formula, then $\neg\varphi$ is a Boolean formula.
3. If φ and ψ are Boolean formulas, then $(\varphi \vee \psi)$ and $(\varphi \wedge \psi)$ are Boolean formulas.

An *assignment* A to the variables in a formula φ is a map from the set of propositional variables in the formula to the set $\{T, F\}$. The symbols $\neg, \vee, \wedge$ are given their usual interpretations as NOT, OR, and AND, respectively. We denote the value of a formula under the assignment A as $\mathcal{V}_A(\varphi)$.

Definition 6. *A formula is said to be* satisfiable *if there is some assignment of its variables such that it evaluates to true. A formula φ is a* tautology *if every assignment to its variables makes φ true.*

The set of satisfiable formulas is of particular interest in computer science. Many problems can be phrased in terms of propositional logic such that the corresponding formula is satisfiable iff the original problem is satisfiable. The technique of casting a problem into a boolean formula is used to show that problems are NP-complete. Sometimes we are interested in formulas of a particular format. There are several so-called normal forms for Boolean formulas. Of particular interest is conjunctive normal form, as all SAT formulas φ can be put into a CNF formula ψ which is only polynomially larger than ϕ.

Definition 7. *A formula φ is a k-conjunctive normal form (k-CNF) formula with n clauses if*

$$\varphi = \bigwedge_{i=1}^{n} (c_{i,1} \vee c_{i,2} \vee \ldots \vee c_{i,k})$$

where $c_{i,j}$ is either a propositional variable or the negation of a propositional variable.

Some Problems about Boolean Formulas

Definition 8. *Define the set SAT to be the collection of satisfiable formulas.*

$$SAT = \{\varphi \mid \exists A \ \mathcal{V}_A(\varphi) = T\}$$

Theorem 1. *Cook '71, Levin '72 Deciding membership in SAT is NP-complete.*

This is the famous result that started our interest in NP-completeness. Karp showed in 1972 that a plethora of problems are NP-complete, two of which are 3-SAT and k-SAT for $k > 3$.

Chapter 3

Proof Triangles

3.1 Motivation for Proof Triangles

Many mathematicians work at different levels of abstraction when solving a problem. At times, it is necessary to perform steps of a proof at a very low level as to ensure that every last detail of the proof works out. In other situations, it's beneficial to remove oneself from the details and work with higher level constructs. This may involve creating mental representations or objects, converting a problem from one form into another, or just finding new representations for the problem. Two tactics used by mathematicians are *abstraction* and *adaptation*.

We use abstraction in one sense to mean the process of taking a problem formalizing it. Stripping away unnecessary details is part of this process. There's a cute problem about sorting stacks of pancakes so that they are arranged largest on bottom and smallest on top, using only a spatula. A key step in solving this problem is to abstract away the pancakes and their exact sizes, and just think about the relative size of the pancakes. Thus, we can take the stack of n pancakes and assign to each a number from 1 to n. From this point, we can formalize the flipping operation and give bounds on the problem.

Another use of abstraction is to mean taking a problem from one domain and transferring it into another. Descartes revolutionized mathematics by creating a connection between geometrical figures and algebraic equations. Taking a problem and transforming it across representations is critical in solving hard problems. Many major breakthroughs in mathematics and computer science happen when someone frames a problem in a way no one had seriously thought about previously.

Adaptation is the ability to take a proof which one knows and utilize it in generating a new proof. Suppose that you knew the standard proof that $\sqrt{2}$ is irrational. One could use this proof as a framework to show that $\sqrt{3}$ is irrational. The arguments differ slightly in how the end conclusion in reached, but the basic idea is the same in both proofs. We can also think of adaptation

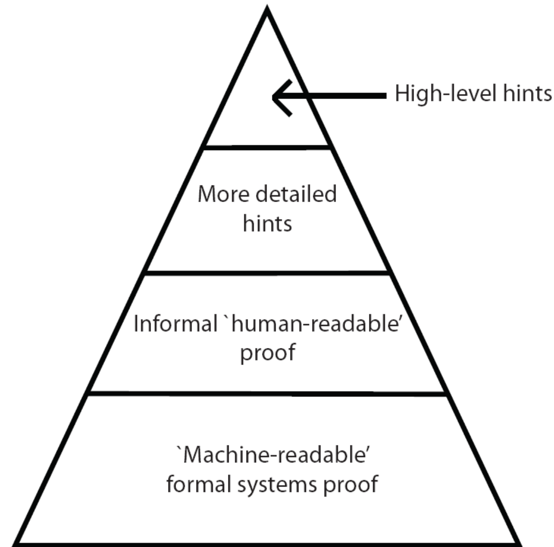
as being able to utilize the results of other theorems to prove some particular theorem. Arguments in mathematical analysis often have this flavor; given that a function has some particular properties, we can apply a theorem whose conclusion is the result we want.

Adaptation and abstraction don't always work. Suppose we consider a theorem G to be a *generalization* of theorem T if it is obtained by universally or existentially generalizing a variable in T. For instance, there is the theorem "There are no solutions (x, y, z) to the equation $x^n + y^n = z^n$, when $n = 3$." The generalized version of this is "For all $n \geq 3$, there are no solutions (x, y, z) to the equation $x^n + y^n = z^n$." This is Fermat's famous 'last' theorem. As far as we know, no proof for Fermat's Last Theorem (FLT) is similar in structure to a proof for FLT when $n = 3$. We see that two problems, strikingly similar in formulation, have vastly different solutions. Obviously, if we have a proof for a theorem of the form $(\forall n \in \mathbb{N})P(n)$, then we have a similar proof for the theorem $P(k)$ for some particular value k. Namely, write down the proof of the theorem $(\forall n \in \mathbb{N})P(n)$ and instantiate the universal quantifier to the particular value of k to conclude $P(k)$. Proofs translations in the opposite direction sometimes work; the key concepts in the proof for a particular instance of a theorem are the same ideas used in the general setting. However, there are no promises that such a transfer of proofs will work.

3.2 What are Proof Triangles?

A proof triangle for a theorem encompasses information that is useful in discovering a proof. Visually, the width of the triangle at a particular level corresponds to the amount of information at that level. At the apex we have short, high-level hints. Beneath these hints are more detailed hints elaborating on specific points in the proof. Below all of the hints we find an informal proof that a mathematician would write. At the base of the triangle is a symbolic, formal system proof whose correctness is easily verifiable. This is the level at which machines currently do mathematics. Our ultimate goal is to formalize all levels of the proof triangle by defining relationships between the higher levels with the formal base of the triangle.

A Proof Triangle:



Most of the content of a proof triangle is currently thought of in high-level terms. We haven't been able to formalize the contents of the triangle, except in a somewhat naive manner. One possible formulation for a proof triangle is as follows: Suppose that we have an encoded version of the formal proof that is n bits long. We will construct a triangle that has n layers to it, in which layer i has the first i bits of the proof. The short prefixes of the proofs are general hints; it's likely that many n bit proofs start with some particular prefix. Furthermore, the amount of space required to write down the triangle is small. This view of a proof triangle isn't very satisfying because it doesn't formalize the high-level human readable proof.

3.3 Why Proof Triangles?

In our experience, a key part in the problem solving process is getting off the ground, so to speak. A student without any idea of how to start can usually make significant progress if 'nudged' toward the right first step. Someone who has solved a good number of problems knows that a choice made at the beginning of the proof can dramatically simplify or complicate the theorem proving process. We believe that instructional, high-level hints are a required part of guiding a student or algorithm toward a proof. In 'the middle of things,' detailed hints about how to work through particular matters of the proof serve a similar purpose to the high level hints.

One appealing aspect of proof triangles is that they can be viewed as self-

contained sets of information that could be used in a formal computational model. One proof triangle referencing another makes sense in this context. Such a reference could exist if the proof method for one theorem is utilized in a proof for another theorem. For example, a proof triangle for the theorem "$\sqrt{3}$ is irrational" could reference the "$\sqrt{2}$ is irrational" triangle.

Lastly, we have found that proof triangles are a very nice representation of information a student would find useful in proving a theorem. As creators of proof triangles, we must critically examine what properties or processes are utilized in proving a theorem.

When computers solve mathematical problems they manipulate equations or statements in a formal language, one in which our mental abstractions have no place. We have seen from experience that the output of an automatically generated proof can be quite unreadable. If there were a way of relating the output of theorem proving systems to the constructs mathematicians are used to working with, we could greatly enhance the usefulness of theorem provers in the day-to-day activities of a mathematician.

Chapter 4

Algorithmic Understanding: Synthesizing New Proofs

4.1 Introduction to algorithmic understanding

As instructors and students, we have all experienced judging one's understanding of a topic. For instance, a proof of a theorem might be presented in lecture. On the exam, students might prove a variant of that theorem. Often a proof for the test question will be similar in structure to the proof taught in class. A student with reasonable understanding will be able to figure out how to modify the proof so that certain assumptions or conditions are changed. If the student cannot solve the test question, then often it is believed that the student "didn't understand" the original proof. However, great ingenuity is sometimes required to make steps in a proof that wouldn't be apparent to every student. Typically, these jumps in reasoning result from the synthesis of multiple 'proof techniques' and relationships between problems. Once someone has seen a 'trick' they are able to apply it in a variety of contexts when it might not be obvious to do so.

4.2 Formal definitions

Let us fix a formal system $\mathcal{S}$ over symbols Σ. The system consists of a set of strings $w \in \Sigma^*$ which are the well-formed formulas (wffs) of the system. Let $\mathbf{T}_\mathcal{S}$ be the set of theorems in the system $\mathcal{S}$ and let $\mathbf{P}_\mathcal{S}$ be a set of proofs such that every theorem in $\mathbf{T}_\mathcal{S}$ has some proof. The system $\mathcal{S}$ could be that of propositional calculus, first-order logic, or type theory to name a few.

Definition 9. *Let $\Delta : \mathbf{T} \times \mathbf{T} \to \mathbb{R}$ be a metric over the space of theorems. Furthermore, let $\delta : \mathbf{P} \times \mathbf{P} \to \mathbb{R}$ be a metric over the space of proofs.*

Naturally, in all of the discussion which follows our concept of understanding will be greatly dependent on our choice of 'distance' between theorems and between proofs. For some formal systems, there are very natural ideas of distance.

In others, it might be more difficult to view the distance between two theorems as simply a syntactic measure. Whenever we present particular systems and claims of understanding in these systems, we will be clear as to what distance measures we are using.

Definition 10. *Extent to which an algorithm understands in a system $\mathcal{S}$*

Fix a particular theorem $T \in \mathbf{T}_{\mathcal{S}}$ and a proof P of T. Define n to be the length of the theorem T plus the length of a minimal length proof for T. Let $f : \mathbb{N} \to \mathbb{N}$ be a given monotone increasing function, and let the functions Δ and δ be specified for the system $\mathcal{S}$. We say $\mathcal{A}$ has understanding $f(n)$ of the pair (T, P) if for every $T' \in \mathbf{T}_{\mathcal{S}}$ such that

1. $\Delta(T, T') \leq f(n)$
2. *There is a proof P' of T' where $\delta(P, P') \leq f(n)$*

$\mathcal{A}$ can find a proof of T' in time poly(n).

In the case that $\mathcal{A}$ has understanding $f(n)$ of every pair (T, P), then we simply say $\mathcal{A}$ has understanding $f(n)$.

What is this definition saying? For a theorem T and a proof P, we're able to prove theorems which are $f(n)$ close to T provided that proofs $f(n)$ close to P exist for these new theorems. We include the length of the minimal in the parameter for f for two reasons. The first is that in some systems a proof for a theorem might require exponential space to write down. Consider a proof that a TQBF formula is valid. If we have k alternations of quantifiers, we require space exponential in k to write down a proof of validity. On the other hand, proofs might only be polynomial length in the statement of the theorem.

4.3 Understanding in k-SAT formulas

4.3.1 Preliminaries for the system $\mathcal{S}_{kSAT}$

The first formal system which we will consider working with is the system $\mathcal{S}_{kSAT}$.

Definition 11. *The system $\mathcal{S}_{kSAT}$ consists of the well-formed formulas of propositional calculus in k-CNF form. The theorems $\mathbf{T}_{\mathcal{S}}$ of $\mathcal{S}$ are satisfiable k-CNF formulas. A proof for a theorem (formula) φ is any satisfying assignment of the variables in φ. Given a formula φ, the length of a proof is the number of variables in φ and the length of φ is the number of disjunction clauses*

This system arises naturally in computer science as many different problems can be expressed as k-SAT formulas. Determining whether a k-CNF formula is satisfiable is NP-complete for $k \geq 3$. We are interested in our ability to solve the k-SAT problem when we have an instance of k-SAT (a formula and a satisfying assignment) and we're looking at similar formulas.

Define the distance between two satisfying assignments A and B to be the number of assigned variables which differ plus the number of variables which

aren't common to both A and B. For example, suppose A is the assignment $\{x_1 = T, x_2 = F, x_3 = T\}$ and B is the assignment $\{x_1 = F, x_3 = T, x_4 = F\}$. Then $\delta(A, B) = 3$ as there are two variables which are not common to both assignments and one variable which differs between the two assignments. To measure the distance between two k-CNF formulas φ and ψ, we count the number of clauses in which they differ, plus the number of clauses not common to both. If we think of formulas as sets of clauses, the distance between them is the cardinality of the symmetric difference: $\Delta(\varphi, \psi) = |(\varphi \setminus \psi) \cup (\psi \setminus \varphi)|$.

4.3.2 A trivial algorithm with $\mathcal{O}(1)$ understanding

Given a formula φ with satisfying assignment A, how do we quickly find a satisfying assignment to a formula ψ , given that there's a satisfying assignment to ψ that doesn't differ at more than $f(n) = k$ variables in A? We can simply go through all of the different subsets of size less $k+1$ of variables to flip and test these modified assignments of A on the formula ψ. We are promised that one of them will be a satisfying assignment to ψ. For a fixed value k, the number of possible assignments for ψ is

$$\sum_{i=0}^{k} \binom{n}{k} = \mathcal{O}(n^0) + \mathcal{O}(n^1) + \ldots + \mathcal{O}(n^k) = \mathcal{O}(n^k).$$

Thus, there are only a polynomial number of possible changes we can make to A, so simply enumerating then and testing each to see if it is a satisfying assignment is a polynomial time procedure.

4.3.3 An algorithm with $\mathcal{O}(\log n)$ understanding

When $f(n)$ isn't a constant function, the above method doesn't work. In particular if we have $f(n) = \mathcal{O}(\log n)$, asymptotically there are $n^{\log n}$ $\log n$ sized subsets of an n element set. Therefore, enumerating all possible changes to the assignment A that we already know will not work. We can be clever and still do this in polynomial time, however. Given a formula ψ, use the assignment A and see which clauses are not satisfied. By the distance restriction between φ and ψ, there are no more than $\log n$ clauses unsatisfied. For at most $f(n)$ steps, pick an unsatisfied clause and set one of the terms to be true, modifying the assignment. Intuitively it's clear that some assignment gotten in this manner must be a satisfying assignment. We can do a simple proof by induction on the number of clauses to rigorously show that this is in fact the case. The process by which we create new assignments to test can be represented as tree of depth $f(n)$ and branching factor k. Thus, there are $k^{f(n)} = k^{\mathcal{O}(\log n)} \leq k^{c \log n}$ for large enough n. $k^{\log n}$ is polynomial, so there are only a polynomial number of potential assignments to check. Therefore, this method will find a satisfying assignment to ψ in polynomial time.

4.4 Limitations in understanding k-SAT formulas

We have demonstrated an algorithm $\mathcal{A}$ which can demonstrate $f(n) = \mathcal{O}(\log n)$ understanding. The algorithm didn't exhibit clever sophistication; in the case that the number of changed variables is bounded by $\mathcal{O}(\log n)$, we can perform a necessary enumeration of potential solutions in polynomial time. It's natural to wonder if this is the best we can do? Can we find algorithms with $\mathcal{O}(\log^k n)$ understanding?

Theorem 2. *An algorithm with $f(n) = \mathcal{O}(n^\epsilon)$ understanding for k-SAT ($k > 2$) exists if and only if $P = NP$*

Proof. It is trivial to see that if P = NP, then an algorithm with $f(n) = \mathcal{O}(n^\epsilon)$ understanding exists. If P = NP, then the general k-SAT problem can be solved in polynomial time. Given a formula φ with n clauses and a satisfiable formula ψ such that $\Delta(\varphi, \psi) \leq f(n)$, a satisfying assignment to any satisfiable formula within polynomial distance of φ can be found in time polynomial in n.

We will now show that given an algorithm $\mathcal{A}$ with $f(n) = \mathcal{O}(n^\epsilon)$ understanding for 3-SAT, P = NP. Let ψ be an arbitrary 3-CNF formula with n clauses and no more than n variables. Create a formula φ with m clauses over the variables $x_1, x_2, \ldots, x_m, y_1, y_2$ distinct from those variables in ψ where

$$\varphi = \bigwedge_{i=1}^{m} (x_i \vee y_1 \vee y_2)$$

Clearly, φ is satisfiable by the assignment A which sets all of the variables to true. Consider the formula $\varphi' = \varphi \wedge \psi$. If we make m large enough such that $n \leq f(m)$, then $\Delta(\varphi', \varphi) \leq f(m)$. Furthermore, any satisfying assignment B to φ' can be made such that the only variable assignments which differ between A and B are the new variables introduced in B, hence $\delta(A, B) \leq f(m)$. Notice that $m = \mathcal{O}(n^{1/\epsilon})$ suffices for the inequalities to hold, so the formula φ' will only be polynomially larger than ψ. Using the algorithm $\mathcal{A}$ we find a satisfying assignment to φ', from which we can extract a satisfying assignment to ψ. Since m is polynomial in n and $\mathcal{A}$ runs in time polynomial in m, we have constructed an algorithm which solves 3-SAT in polytime. Thus, P = NP as we've demonstrated a polytime algorithm for an NP-complete problem. □

Reflecting on this proof, we see that having a polynomial amount of understanding is too powerful. By creating a trivial formula φ such that the conjunction of φ and ψ isn't too far away φ, we're able to extract a satisfying assignment to ψ. The main reason as to why polynomial understanding is too much is because we can create these 'dummy' formulas which are only polynomially larger than the formulas to which we'd like to find satisfying assignments. Since it's unlikely that we will ever have polynomial understanding, is it possible to get polylog understanding? The next theorem posits this possibility.

Theorem 3. *If there exists an algorithm $\mathcal{A}$ for 3-SAT with understanding $f(n) = \omega(\log n)$, then there exists a subexponential time algorithm to solve 3-SAT.*

Proof. The proof is similar to the above construction. Again, we are given a formula ψ with n clauses and no more than kn variables for which we would like to find a satisfying assignment. We create the dummy formula φ with m clauses and $m + 2$ variables:

$$\varphi = \bigwedge_{i=1}^{m} (x_i \vee y_1 \vee y_2)$$

In this case, we want to choose m large enough such that $n \leq f(m)$. This way the distance between the dummy formula φ and the formula ψ is less than $f(m)$, as well as the distance between the satisfying assignments for these two formulas. If $n \leq f(m)$, then $m \geq f^{-1}(n)$. Since we have $f(n) = \omega(\log n)$, we have that $f^{-1}(n) = c^{o(n)}$. Thus, if we make a dummy formula φ which has size $m = \Theta(c^{o(n)})$, we can use our understanding of φ to find a satisfying assignment to ψ. The running time for this process is going to be $\left(c^{(o(n))}\right)^k = (c^k)^{o(n)} = c'^{o(n)}$. We've now shown how to use an algorithm with $\omega(\log n)$ understanding to construct a subexponential time algorithm to solve k-SAT. □

Chapter 5

Conclusions and Future Work

We have started to explore the possibility of 'understanding understanding' through a complexity theoretic approach. Complexity theory has had great success in describing and understanding different phenomenon in computer science, physics, and mathematics. We believe that we can use similar approaches and create new techniques to tackle the problem of mathematical understanding. We have a working notion of proof triangle that needs to still be formalized. After iterating through several possible formulations for the representation of content in the triangle it's still unclear which is the right way to go. The most simple definition has been given, but this formalization lacks defies some of our intuitive desires for what a proof triangle should be. Most of the difficulty in formalizing our intuition has been trying to make rigorous notions such as hints and strategies. A result of us thinking about these different formulations is that we have worked on problems such as algorithmic understanding and problem structure.

Our analysis of algorithmic understanding has yielding interesting results. Given one possible formalization of understanding, we have shown that there is a threshold for understanding satisfiable k-CNF formulas. Unless the complexity theory world is a strange place, there is a sharp limit to understanding with amount $\mathcal{O}(\log n)$. It is possible that we might be able to achieve something like $\mathcal{O}(\log^k n)$ understanding but that would require, or result in, significant improvement in current SAT solving techniques.

In the future, we would like to make progress in determining what is a good formalization for the content of proof triangles. We believe in the usefulness of proof triangles as a concept, but in order to implement them on a computer we need to formalize a system that 'computes' with triangles. Defining what a hint is would also mark a significant step toward formalizing our understanding of mathematics. Ideally, a hint isn't quite as powerful as nondeterminism because sometimes a hint doesn't get you all the way to the solution. We plan to

approach this problem by looking at randomized algorithms for solving SAT, and see if we can work the notion of hint into this framework. The proof theoretic work has shown that there are sharp bounds on the efficiency of SAT solving algorithms unless P = NP. If these questions are resolved and we have formalized proof triangles, then the next step would be to implement these ideas in a theorem proving system. This would hopefully demonstrate the computers can solve mathematical problems in a more human-like manner.

Bibliography

[1] P. Beame, R. Karp, T. Pitassi, M. Saks, "The Efficiency of Resolution and Davis-Putnam Procedures," *SIAM Journal on Computing* 31(4), 2002, pp. 1048–1075.

[2] A. Bundy, "The Automation of Proof by Mathematical Induction" in *Handbook of Automated Reasoning*, vol. 1, A. Robinson and A. Voronkov, Ed., Elsevier Science, 2001, pp. 845–911.

[3] A. Bundy, "Planning and Patching Proof," *Proc. of Artificial Intelligence and Symbolic Computation* (2004), pp. 26–37.

[4] S. Cook, "The complexity of theorem-proving procedures," *ACM Symposium on Theory of Computation* (1971), pp. 151–158.

[5] M. Sipser, *Introduction to the Theory of Computation, 2nd ed.*, MIT Press, 2005.

[6] A. Wigderson, "P, NP and Mathematics - a computational complexity perspective," *Proc. of the ICM 06 (Madrid)*, Vol. I, pp. 665-712.