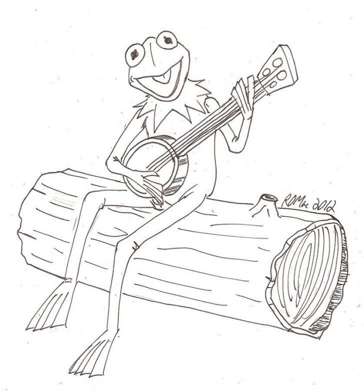
THE COMPLEXITY OF MATHEMATICAL STATEMENTS

MELISSA S. QUEEN



Advisors:
Amit Chakrabarti, *Dartmouth Computer Science Department*
Cris Calude, *University of Auckland Computer Science Department*

May 2013

ABSTRACT

In [4, 5, 7, 2], authors C. S. Calude, E. Calude, M. Dinneen and M. Burgin use algorithmic information theory to develop a method for measuring the complexity of mathematical statements. In the process, they define a statement's complexity as *the smallest program that completely describes that mathematical statement.* This paper discusses their proposed complexity measure and the results of that measure's application to real theorems and conjectures. We then briefly present a program that was developed to help researchers make these measurements efficiently and accurately. This practical program, called KERMIT (Kermit Encoded to Register Machine Instruction Translator), allows researchers to describe their algorithm in a higher level language than the minimal set of instructions required to make the final measurement.

After introducing KERMIT, we use it to measure the complexity of an important open problem: sensitivity versus block sensitivity. We design a program to describe this problem, and use KERMIT to reduce the program down into a standardized, rudimentary brand of assembly commonly used for making complexity measurements. We measure the size of this assembly-like program to determine the complexity of our problem, and compare this result to the measured complexities of other important problems.

CONTENTS

1 BACKGROUND

Algorithmic Information Theory (AIT) is the result of putting Shannon's information theory and Turing's computability theory into a cocktail shaker and shaking vigorously. The basic idea is to measure the complexity of an object by the size in bits of the smallest program for computing it.

— G. J. Chaitin [3]

Science can be described in many cases as the search for simple statements that describe complex, observed behaviors. This pursuit is bold and comes with no intrinsic guarantee of success. But science has, remarkably, been very successful. We can concisely describe the physical laws from which the motions of stars, the trajectory of a ball, or the rotational frequency of a spinning top can be calculated. The number π cannot be finitely represented by simply printing out its decimal representation, "3.14159...", but we have developed definitions of π that can be represented finitely. We have programs that—if allowed to run forever—can output every digit of π. Such a program holds all of the information associated with the number π, but is much more concise (finite, instead of infinite!) than a digit-by-digit specification.

Scientific theorems must be able to accurately describe observed phenomena. But to be useful, they must also be significantly less complex than these phenomena[1]. An infinite chart in which all (correct) trajectories for all possible projectiles are enumerated would indeed describe the observed physical motions, but is an unwieldy and excessively verbose description. We gain much more insight from the concise description of the relationship between force, mass and acceleration. Of course, the $F = ma$ equation is not the only concise description of motion; there are different, but equally effective, formulations in Lagrangian and Hamiltonian mechanics. There are thus many algorithms to describe the complicated observed phenomenon of motion, but in general we seek the most concise.

This idea of a compressed *algorithm* or *description* of a more complicated object is at the heart of Algorithmic Information Theory (AIT). In AIT, the *complexity* of an object is defined as the size of its most concise description.

1 This philosophy was identified by Leibniz in 1686 and paraphrased by Chaitin - if a physical law can be as complicated as the experimental data that it explains, then there is always a law, and the notion of "law" becomes meaningless. [10]

1.1 ALGORITHMIC INFORMATION THEORY

The idea that some description of an object could be used as a measure of that object's complexity was introduced independently by Solomonoff, Kolmogorov and Chaitin in the mid 1960s. The current formulation of a *descriptive complexity* can be informally stated as follows: The complexity of an object A is the size of the smallest program that can fully describe A. If the objects that we are considering are strings, then a string's complexity is the size of the smallest program that can output that string.

Other equivalent names for descriptive complexity include: Kolmogorov complexity, Kolmogorov-Chaitin complexity and algorithmic entropy.

Before we can talk about this program-size based complexity, we must define "program". A *program* is a string of bits that can be interpreted and run by some universal Turing machine. In AIT, we make the additional stipulation that this universal Turing machine is *prefix-free*. This means that any valid program cannot be the prefix to any other valid program. Put simply, we cannot just add bits to the end of a syntactically correct program to create another program. We call such prefix-free programs *self-delimiting*[2].

The following quote by Chaitin ([9], also quoted in [3]) provides a reasoning for this requirement:

> A key point that must be stipulated ... is that an input program must be self-delimited: it's total length (in bits) must be given within the program itself. (This seemingly minor point, which paralyzed the field for nearly a decade, is what entailed the redefinition of algorithmic randomness.) Real programming languages are self-delimiting, because they provide constructs for beginning and ending a program. [. . .] In essence the beginning and ending constructs for programs and subprograms function respectively like left and right parentheses in mathematical expressions.

Since we have now introduced the idea of a program-based complexity measure, it is worth stepping back and comparing this measure to other, more common, program complexities:

- Program-size Complexity - the number of bits needed to represent the program.
- Space Complexity - the space used by the program during execution
- Time Complexity - the time (number of steps) required by the program to complete the computation

2 As an example, notice that we can make any string x self-delimiting by transforming it into the string $0^{|x|}1x$. A Turing machine can read the string $0^{|x|}1x$ bit by bit and know exactly when it has reached the end - it does not require a special "end of string" symbol.

The last two measures are common and well-explored. The first measure, program-size complexity, is distinct in that it is a static measure. It does not concern itself with what happens when the program actually begins executing. It isn't characterized by by a growth rate or asymptotic behavior (in contrast to time and space complexities) but rather is a scalar measurement. Program-size complexity is therefore meaningful only in comparison to *other programs*. The insight comes from seeing how the complexity changes from description to description (program to program), rather than between different inputs to the same program.

For the remainder of this paper, "complexity" will refer exclusively to program-size complexity.

AIT has lead to many interesting advances in complexity theory - including new definitions of random strings, incompleteness, and the development of Chaitin's number Ω. It has been related to concepts in statistical physics and serves as a kind of bridge between the physics and information science realms.

1.2 THE ALGORITHMIC COMPLEXITY OF MATHEMATICAL STATEMENTS

When program-size complexity is applied to strings, we specify for a particular string, s, a program, P, such that when P is executed it produces s. We can say that P describes s.

How can we create a descriptive program to assess the complexity of a mathematical theorem or conjecture? One possible method is to view mathematical statements as describing the search for a counterexample [4, 5, 2, 7]. A program can easily encode this search. If the conjecture is false, a counterexample will eventually be found. But if a conjecture is true, the search will run on forever. If we could somehow determine ahead of time if the search will run forever, we would be able to prove the conjecture is true. Unfortunately, this equates to solving the halting problem, which is known to be undecidable. But all is not lost, since we are not actually trying to *solve* the mathematical conjectures, but rather to measure their complexity.

Another way to view this encoding is as a *reduction* of the mathematical statement to the halting problem. It is quite remarkable that, as we will show, there are a huge variety of mathematical statements for which this reduction is possible.

Currently, using a reduction to the halting problem is the only way that descriptive programs are built.[3] While other constructions might be possible, they are not actively sought because there are a number

3 It has been suggested that mathematical statements could be reduced to the totality problem. The possibility and effectiveness of such a reduction is an open problem.

of reasons why a reduction to the halting problem seems appropriate and satisfying. First, as stated above, it has been used to characterize a large number of mathematical statements - including all Π_1 statements - regardless of the statements' particular details or encompassing fields of study. Second, it seems to intuitively and completely describe the problem; regardless of a statement's open/solved status, the program will somehow encode the predicate, and will need to perform a search over all possible solutions. Last, it ties the *resolution* of the mathematical problem—the truth of the statement—to a well-defined feature of the program itself: whether or not it halts.

To formally specify descriptive programs, we adopt the following notation:

$P(n)$	A predicate on n
π	The Π_1 statement: $\forall\ n\ P(n)$
U	A universal prefix-free Turing machine
Ψ_π^U	A program defined for U that describes the statement π

We use the notation $\neg P(n)$ *to indicate* $P(n) = \texttt{false}$.

For any statement π we can construct the program Ψ_π^U:

```
**for all** $n$ **do**
  **if** $\neg P(n)$ **then** HALT;
**end for**
```

Clearly, π is true if and only if Ψ_π^U never halts. For some statements we know the halting behavior of Ψ_π^U. We know that for $\pi = \mathsf{FLT} =$ Fermat's Last Theorem, Ψ_{FLT}^U will never halt, because FLT has been proven true. For open problems, the halting behavior is not yet known.

Since we have now defined a way to build a descriptive program for a mathematical statement π, we can measure its program-size complexity. Of the many programs Ψ_π^U, we seek that one with lowest complexity:

$$C_U(\pi) = \min\{\ |\ \Psi_\pi^U\ |\ \}$$

The complexity $C_U(\pi)$ is uncomputable [3]; there is no general way to prove that we have the smallest program Ψ_π^U. But we can still work with an upper bound: Let $\mathfrak{C}_U(\pi)$ be the tightest known upper bound of $C_U(\pi)$. That is, $\mathfrak{C}_U(\pi)$ is the size of the smallest *known* program Ψ_π^U.

PROBLEM NAME	COMPLEXITY
Legendre's Conjecture	422
Fermat's Last Theorem	729
Goldbach's Conjecture	756
Dyson's Conjecture	1064
Euler's Integer Partition Theorem	2396
Riemann Hypothesis	2741
The 4-Color Theorem	3289

Table 1: Current complexity measures of some mathematical conjectures and theorems

The complexity measure $\mathfrak{C}_U$ is dependent on a fixed choice of U. However, changing U to a different U′ will adjust the complexity by only up to a fixed amount[4]:

$$| C_U(\pi) - C_{U'}(\pi) | \leqslant c$$

So when two complexity measures differ by a large amount for a Turing machine U, it is unlikely–and after a certain threshold, impossible–that their ordering will change for a different universal Turing machine. Of course, our concept of what an acceptable "large difference" is depends on our (evolving) knowledge of the complexity measure. Currently, experimental results indicate that for minimal machines $c \leqslant 2^{10}$[6]. This is large compared to some known complexities - but still leaves many pairs of problems confidently ordered.

In practice, to obtain the complexity $\mathfrak{C}_U(\pi)$, we must develop a program Ψ_π^U and evaluate its program-size complexity in some fixed universal formalism U. We can then improve our choice of Ψ_π^U to obtain a better measure of $\mathfrak{C}_U(\pi)$[5].

This complexity measure was introduced in 2006 [7] and has since been applied to a number of both open and closed mathematical conjectures. A summary of current standings is presented in Table 1.

4 This is an even better situation than that of time complexity – changing universal Turing machines can adjust the time complexity of a program by a polynomial, rather than fixed, amount.

5 This process of improving the best known program is not guaranteed to be insightful, but in practice it often becomes convincing that certain programs cannot be simplified much further.

1.3 FROM STANDARD COMPUTATION TO INDUCTIVE COMPUTATION

It is easy to determine the complexity $\mathfrak{C}_U$ of any conjecture for which we know a Π_1 statement – but we would also like to be able to determine the complexities of some other conjectures, specifically those for which we only know Π_2 statements.

If we apply the same "search for a counterexample" to a Π_2 statement, $\pi_2 = \forall n \; \exists m \; P(n, m)$ we get the program $\hat{\Psi}^U_{\pi_2}$:

```
for all  $n$  do
  for all  $m$  do
    if  $P(n, m)$  then break;
  end for
end for
```

Unfortunately, $\hat{\Psi}^U_{\pi_2}$ will never halt. If π_2 is true, then the program will iterate over all n and run forever. If π_2 is false, the program will find an n for which no satisfying m exists, and end up iterating over all m for forever. The program $\hat{\Psi}^U_{\pi_2}$ cannot be said to describe π_2 in any meaningful way.

To be able to characterize mathematical statements for which we know Π_2 formulations, we introduce a more powerful model of computation: the inductive Turing machine (ITM). This solution was introduced in 2011 [2]; whether or not a descriptive program for Π_2 statements exists for standard Turing machines is an open problem.

Under the inductive model, a program is allowed to run forever but still be considered to give an answer if after a finite number of steps the *output stabilizes* (does not ever change again). Inductive Turing computation is more powerful than standard Turing computation.

In the following examples, let Z be the output register of the program. For a standard Turing machine, the result of the program is the value of Z when the program halts. For an inductive Turing machine, the result of the program is the value of Z when the program halts, *or* the value of Z if after a certain time Z stabilizes. In the latter case, the program is not required to halt.

Consider the following program:

```
$Z = 1$;
Halt;
```

In both the standard and the inductive realms, this program will always halt, and will always give the result 1.

The next program contains an infinite loop. On a standard Turing machine, this program would never halt and thus would never give a result. But on an inductive Turing machine, we notice that after the first iteration of the loop, Z becomes set to 1 and never changes.

The output has *stabilized*. Thus, if we were to run this program on an inductive Turing machine it would return the result 1.

```
for all $n$ do
    $Z = 1$;
end for
Halt;
```

In this last example, we again have an infinite loop. But this time the value of Z oscillates between 0 and 1, never stabilizing. Running this program on an inductive Turing machine *will not* give a result.

```
for all $n$ do
    $Z = 0$;
    $Z = 1$;
end for
Halt;
```

Inductive computation is a theoretical construct and makes no assumptions or suggestions about what is physically possible. Can we really allow ourselves access to this powerful model of computation, even though it might be unrealizable in silico? Yes – the physical reality of computation is not the primary concern of this method. Even standard Turing machines are not physically realized, since they assume access to infinite memory! We simply require a formalism in which we can build *meaningful* and *descriptive* algorithms.

Before we use this model of computation to describe Π_2 statements, let us apply it to the Π_1 statements that we've already considered. Instead of defining the program Ψ_π^U for U—a regular Turing machine—let it be defined for an inductive Turing machine (ITM). To describe an ITM we will write pseudocode as before, but now we use register Z to indicate the special output register. We say that $\Psi_\pi^U = i$ if Ψ_π^U stabilizes to the value i. Consider, then, the following program:

```
$Z = 1$;
for all $n$ do
    if $\neg P(n)$ then $Z = 0$;
end for
Halt;
```

Instead of reducing π to the Halting problem, we have reduced it to whether or not the inductive program $\Psi_\pi^U = 0$ or $\Psi_\pi^U = 1$.

Now we show the true strength of the inductive model, by developing a program that describes any Π_2 statement. The trick is to functionally compose two ITM: to allow one ITM to query another ITM. In this way the outer loop over n can query an ITM as to whether there exists an m such that $P(n, m)$. If we carefully design the inner

ITM, it will *always* give a result. We call the resulting machine an ITM of second order.

To describe an ITM of second order, we let register Y be the reserved output register from the inner ITM. The outer ITM can query this register to determine the output of the inner ITM.

For the Π_2 statement π_2, create the program Ψ^U_π:

```
$Z = 1$;
for all $n$ do
    // Start of second ITM
    $Y = 0$;
    for all $m$ do
        if $P(n, m)$ then $Y = 1$;
    end for
    // End of second ITM
    if $Y = 0$ then $Z = 0$;
end for
Halt;
```

An alternate way to describe the same algorithm, which clearly shows the composition of two ITMs:

```
$Z = 1$;
for all $n$ do
    if $F(n) = 0$ then $Z = 0$;
end for
Halt;

Where $F(n)$ describes the ITM:
$Z = 0$;
for all $m$ do
    if $P(n, m)$ then $Z = 1$;
end for
Halt;
```

We can now formally describe the inductive complexity measures of first and second order:

$$C^{ind,1}_U(\pi) = \min\{\, |\, \hat{\Psi}^U_{\pi_1} \,|\, \}$$

$$C^{ind,2}_U(\pi) = \min\{\, |\, \hat{\Psi}^U_{\pi_2} \,|\, \}$$

Again, we must work with upper bounds on these complexity measures. Let $\mathfrak{C}^{ind,1}_U(\pi)$ and $\mathfrak{C}^{ind,2}_U(\pi)$ be the tightest known upper bounds on the complexity measure of statement π in the ITMs of first and second order, respectively.

1.4 A REGISTER MACHINE LANGUAGE

The programming language we use is introduced fully in [5, 7], but we provide an overview of the instructions here for convenience. We use a register machine language that allows the program access to an arbitrary number of registers, each of which can hold an unbounded integer. The registers can be thought of as the variables of the program. With just the few available instructions (described below) we can build complicated routines that manipulate these variables in many different and meaningful ways.

In the descriptions below, R2 and R3 may refer to the name of a register, or a literal value. R1 always refers to a register.

=R1,R2,R3

If the content of R1 and R2 are equal, then the execution continues at the R3rd instruction of the program. If the contents of registers R1 and R2 are not equal, then execution continues with the next instruction in sequence.

&R1,R2

The content of register R1 is replaced by R2.

+R1,R2

The content of register R1 is replaced by the sum of the contents of R1 and R2.

!R1

One bit from the input tape is read into the register R1, so the content of R1 becomes either 0 or 1. Any attempt to read past the last data-bit results in a run-time error. None of the programs presented here have any input, so this instruction is never used.

%

This is the last instruction for each register machine program before the input data. It halts the execution in two possible states: either successfully halts or it halts with an under-read error.

A register machine program is a finite list of these instructions. The prefix-free binary encoding of these instructions is discussed in detail in [4, 5], and in Appendix A. After encoding, the result is a program that can be interpreted by a prefix-free universal Turing machine (or

PROBLEM NAME	COMPLEXITY
Legendre's Conjecture	422
Fermat's Last Theorem	729
Goldbach's Conjecture	756
Dyson's Conjecture	1064
Euler's Integer Partition Theorem	2396
Riemann Hypothesis	2741
The 4-Color Theorem	3289
The Collatz Conjecture	516
The Twin Prime Conjecture	649
The P versus NP Problem	6495
Goodstein's Theorem	7909

Table 2: Current complexity standing of some mathematical conjectures and theorems

inductive Turing machine), meeting our formal specification of a descriptive program. In essence, the program Ψ_{π}^{U} becomes a string of bits that I could hand to you, along with instructions for interpreting them, and confidently claim "these ones and zeroes describe the mathematical statement π."

1.5 CURRENT MEASURED COMPLEXITIES

Table 2 shows a summary of the currently measured mathematical conjectures and theorems. They are divided into two groups by the horizontal line: The top group have been measured with an ITM of first order, and the bottom group have been measured with an ITM of second order.

2 DEVELOPMENT OF THE KERMIT PROGRAM

A note on semantics: KERMIT (Kermit Encoding to Register Machine Instruction Translator) refers to both the name of the translation program, and the programming language that the program recognizes.

2.1 INTRODUCTION

Creating register machine programs by hand is a very tedious and error-prone process. Many times the program will make use of rudimentary operations such as conditional if-else statements, while-loops and calls to subroutines. In the absence of an automated process, the programmer is forced to write many lines that are merely repetitions of a simple pattern.

KERMIT (Kermit Encoding to Register Machine Instruction Translator) was created to make the development of register machine programs faster, simpler and more accurate. Instead of hand-crafting a program from the five basic instructions offered by our register machine language, a user can build a program out of the many higher-level directives available in the KERMIT language. The translator will take such a program and parse it down into a concise register machine program. KERMIT is distinct from other compilers in that the goal is a concise–rather than fast–program.

Our goal is that this translation service will allow researchers to more conveniently and quickly make measurements of mathematical statements.

2.2 OVERVIEW OF THE KERMIT LANGUAGE

The KERMIT program supports three main types of instructions: conditionals, loops, and subroutine calls. Users may define their own subroutines, and KERMIT provides automatic access to a basic library with arithmetic subroutines MULT (multiplication), DIV (integer division), POW (exponentiation), CMP (comparison), SUBT (subtraction) and DIV2 (division by 2). A basic array library is also included, using Dinneen's encoding [11]. This includes subroutines: SIZE (get the size of an array), APPEND (append a new element), ELM (read the value of a particular element), and RPL (replace a particular element).

The programming style of KERMIT is similar to that of C, but KERMIT offers a much more primitive set of options. The language doesn't abstract away too much of underlying register machine costs;

there are very few single lines in KERMIT that translate to more than 4 or 5 register machine instructions. The fear is that we are much more accustomed to programming for *time* complexity, rather than program-size complexity. If we make KERMIT too similar to regular programming environments, we may lose the insights we need to make a program extremely concise.

2.3 OVERVIEW OF THE PROGRAM DESIGN

The translation process is broken into three main steps, outlined below.

1. *Labeling of the input file.* The program opens the input file and reads it in line by line. For each line, it checks to see if that line matches any of the valid formats for a KERMIT instruction. When a match is made, a node is created and added into a doubly linked list (DLL) of instructions.

   When labeling succeeds, we know that each line of the input file matches one of the expected formats. But that is not a guarantee that the syntax is completely valid.

2. *Linking of the instructions.* The final syntax checking is done in the linking stage. Linking connects related statements together. For example, every statement with an opening brace is connected to the corresponding closing brace. The breaks and continues within a loop are connected to that loop node. Return statements within a subroutine are connected to that subroutine node.

   These connections are made because in the final conversion at least one of the nodes in each connection needs to access information stored in the other node. For example, a break statement needs to know where the end of the loop is located.

3. *Conversion to register machine instructions.* After the DLL of instruction nodes has been linked, we can iterate through the list and convert each KERMIT instruction into the corresponding register machine instructions. Some instruction nodes will translate to a single register machine instruction. Others will create multiple consecutive instructions, and some may create instructions in different locations in the final program (Ex: Loops will have instructions at both the start and end of the loop).

2.4 RESULTS

The KERMIT program is working successfully, and in Chapter 3 we use it to help measure the complexity of an important open problem.

3 SENSITIVITY AND BLOCK SENSITIVITY OF BOOLEAN FUNCTIONS

Sensitivity is one the simplest, and block sensitivity one of the most useful, invariants of boolean functions.

— Claire Kenyon and Samuel Kutin [13]

3.1 INTRODUCTION

A boolean function of n variables is a function $f : \{0,1\}^n \to \{0,1\}$. Sensitivity and block sensitivity are complexity measures of boolean functions. In the definitions below we will use the notation introduced by Nisan [14]: Let w be a boolean string of length n , and let S be any subset of indices, $S \subseteq (1 \dots n)$. Let w^S mean the string w, with exactly the bits at indices S flipped. We say that function f with input w is *sensitive* to indices S if $f(w) \neq f(w^S)$.

We begin by defining the sensitivity and block sensitivity of a function with regards to a particular input. The overall sensitivity and block sensitivity are then the maximum values of these individual sensitivities over all inputs.

DEFINITION 1: The sensitivity of function f with input w, denoted $s(f, w)$ is the number of indices for which f is sensitive. That is, the number of indices i such that $f(w) \neq f(w^{\{i\}})$. The *sensitivity of function* f, denoted $s(f)$, is the maximum sensitivity over all possible inputs:

$$s(f) = \max_w (s(f, w)) \qquad (1)$$

DEFINITION 2: For block sensitivity, we consider the behavior of the boolean function when a subset of indices is flipped, rather than just a single index. For a function f with input w, we define the block sensitivity as the maximum number of disjoint subsets for which f is sensitive. Let $B_i \subseteq \{1 \dots n\}$ denote a non-empty subset of indices. Consider the OR function over 2 bits:

$$\mathtt{OR}(00) = 0; \quad \mathtt{OR}(01) = 1; \quad \mathtt{OR}(10) = 1; \quad \mathtt{OR}(11) = 1;$$

The possible B_i are the subsets $\{0\}$, $\{1\}$, and $\{0, 1\}$. We say that $f(w)$ is sensitive to a particular B_i if $f(w) \neq f(w^{B_i})$. In our example, $\mathtt{OR}(11)$ is sensitive only to subset $\{0, 1\}$, while $\mathtt{OR}(00)$ is sensitive to all three subsets. We seek the maximum number of *disjoint* subsets, so we find that:

$$\mathrm{bs}(\mathtt{OR}, (00)) = 2;$$

$$\mathrm{bs}(\mathtt{OR}, (01)) = 1;$$

$$\mathrm{bs}(\mathtt{OR}, (10)) = 1;$$

$$\mathrm{bs}(\mathtt{OR}, (11)) = 1;$$

In general, then, the block sensitivity of f with input w, $\mathrm{bs}(f, w)$, is the maximum number of disjoint subsets $B_1 \dots B_k$ for which $f(w) \neq f(w^{B_i})$.

From this definition of block sensitivity for a particular input, we build the overall *block sensitivity of function* f by taking the maximum over all possible inputs:

$$\mathrm{bs}(f) = \max_{w}(\mathrm{bs}(f, w)) \qquad (2)$$

The overall block sensitivity in our example is then $\mathrm{bs}(\mathtt{OR}) = 2$.

Block sensitivity was defined by Nisan in 1989 [14] and shown to be polynomially related to decision tree complexities. Nisan was further able to show that the block sensitivity of a boolean function characterizes the time needed to compute that function on a CREW (Concurrent-Read, Exclusive-Write) PRAM (Parallel Random Access Machine) with infinite processors[1].

Block sensitivity has since been shown to be entwined with such diverse areas of complexity as: boolean circuit reliability [12], the degree of the (unique) real multilinear polynomial which characterizes a boolean function [14], and even quantum oracle complexity[1].

While block sensitivity has been shown to be polynomially related to decision tree depth and certificate complexity, it has been difficult to develop a tight bound on the relationship between block sensitivity and sensitivity. We can conclude that $\mathrm{bs}(f) \geqslant s(f)$, simply by noticing that any sensitivity measure can be transformed into a block sensitivity measure by choosing blocks of size 1. Rubinstein showed in 1992 that there is an infinite class of boolean functions for which the sensitivity and block sensitivity are quadratically related: $\mathrm{bs}(f) = 2s(f)^2$ [15]. In 2010, Virza improved this result to show that there is an infinite set of boolean functions for which $\mathrm{bs}(f) = \frac{1}{2}(s(f)^2 + s(f))$ [16]. The best general upper bound that we have, however, is exponential: $\mathrm{bs}(f) \leqslant \frac{e}{\sqrt{2\pi}} e^{s(f)} \sqrt{s(f)}$, shown in [13].

Whether or not sensitivity and block sensitivity are polynomially related is an open problem of considerable interest.

1 Specifically, the time to compute is a logarithm of the block sensitivity, up to a bounded factor.

3.2 OUR ALGORITHM

The conjecture "Sensitivity and Block Sensitivity are polynomially related" can be formally stated:

$$\exists(C, J) \textit{ such that } \forall f,\ \mathrm{bs}(f) \leqslant s(f)^J + C \tag{3}$$

Where f is a boolean function and (C, J) are a pair of integers that define a polynomial. Restating the conjecture as its negation gives us a Π_2 statement, for which we can write a descriptive program. The conjecture is thus:

Note that since we are searching for any *polynomial bound, not necessarily the tightest one, we only need to consider polynomials of the form $x^J + C$.*

$$\forall(C, J)\ \exists f, \textit{ such that } \mathrm{bs}(f) \geqslant s(f)^J + C \tag{4}$$

From this equation we can construct the following program, using an inductive Turing machine of the second order:

```
$Z = 0$;
for all (C, J) do
  // START OF SECOND ITM
  $Y = 0$;
  for all boolean functions $f$ do
    if $(\mathrm{bs}(f) > s(f)^J + C)$ then $Y = 1$;
  end for
  // END OF SECOND ITM
  if $Y == 0$ then $Z = 1$;
end for
```

Registers Z and Y are the output registers for the outer and inner inductive Turing machines, respectively.

To implement this program, we must enumerate all pairs (C, J). We chose to do this by enumerating all integers, interpreting each integer as an array (using the array encoding developed by Dinneen [11] and implemented in the KERMIT program). If the array has exactly two elements, we interpret these as (C, J). Otherwise, we simply loop to the next integer. If we were concerned with the running time of this program, out methods would be incredibly inefficient – most arrays do not have only 2 elements! But we are concerned only with brevity, and found this to be a concise way to describe the loop.

To perform the enumeration over all boolean functions, we use a similar trick. We again loop over all integers, and interpret each as an array. If the array has 2^k elements for some integer k, then we treat this array as a truth table for a boolean function of k variables. In this way, we can enumerate all boolean functions for any number of input variables.

To test the sensitivity of a function F with a particular input w, we generated a $\mathtt{flip}(w, i, j)$ subroutine that flips the bits of w from index i through j. The KERMIT encoded subroutine is included below:

```
// Flips the bit at index i. Alters w directly. Let |w| = k.
flip(w, i, k) {
  // We will be reading bits out of w and into copy
  copy = 1;
  counter = 0;
  while (counter != k) {
    // Recall the DIV2 divides w by two,
    // and saves the remainder into bit
    bit = DIV2(w);
    if (counter == i) {
      // Flip the bit at index i
      if (bit == 0) {
        copy += 1;
      }
    }
    else {
      copy += bit;
    }
    copy = copy + copy;
    counter++;
  }
  // The result in copy is reversed, we
  // must iterate through again to correct
  w = 0;
  while (copy != 1) {
    bit = DIV2(copy);
    w += bit;
    w = w + w;
  }
}
```

Listing 1: Flip subroutine

To read the bits of a string w we use `DIV2`, which alters the string directly. We read the bits and save them back into another register `copy`, flipping bit `i` when we come to it. Saving the bits back into `copy` reverses their order, so before we finish we must iterate through the bits of `copy` and write them back into `w`. The result is `w` with the bit at index `i` flipped.

The sensitivity-calculating routine is presented below. Since this subroutine is called only once from within the main program, in the final version this code is executed directly in main, rather than via a subroutine call. For clarity, it is presented separately here.

```
// F is a truth table, stored as an array of size maxw, where maxw
     = 2^k
s(F, maxw, k) {
  S = 0;
    w = 0;
    maxw++;
    while ( w != maxw ) {
        // Count the sensitive indices for this particular w
        count = 0;
        index = 0;
        while ( index != k ){
            wcopy = w;
            flip(w,index);
            // Check if it's sensitive
            match = 0;
            new = F[w];
            if (new != 0) {
                match++;
            }
            w = wcopy;
            orig = F[w];
            if (orig != 0) {
                match++;
            }
            // When match == 1, the index is sensitive
            if (match != 1) {
                count++;
            }
            index++;
        }
        // Take the maximum
        if (count > S) {
            S = count;
        }
    }
    return S;
}
```

Listing 2: Sensitivity subroutine

The sensitivity subroutine is given a function F in the form of an array of size $maxw$ representing a truth table. The function F takes k variables ($maxw = 2^k$). Only one of k or $maxw$ is necessary, but they are both passed to the function so that they only need to be calculated once in the main program.

The subroutine loops over all possible inputs w to the function, values 0 through $maxw - 1$ (hence the name $maxw$). For each input, the subroutine counts the number of indices for which the function is sensitive. It does this by iteratively invoking the `flip` subroutine for every index $0 \ldots k - 1$, and comparing the result of the function on the original and flipped inputs. Taking the maximum of all these counts, we acquire the sensitivity of function F.

The block sensitivity subroutine is more complicated, requiring three nested loops. The outer loop goes over all inputs to the function F, the same as for the sensitivity calculation. But then, instead of looping over each index, we must loop over all possible sets of disjoint subsets of indices. We use Observation 1 to make a shortcut in our enumeration of subsets: We need only enumerate sets of continuous blocks. This is easily done by interpreting the alternating blocks of 1s and 0s in a binary number as defining subsets of indices. For example, the binary number 111101100 defines subsets $\{0,1\},\{2,3\},\{4\},\{5,6,7,8\}$.

DEFINITION 1: Let $bs'(f)$ be the block sensitivity of f, with the restriction that all disjoint subsets must be contiguous blocks of indices.

OBSERVATION 1: For every f such that $bs(f) \neq bs'(f)$, $\exists$ at least 1 other function f' such that:

1. $bs(f') = bs(f)$
2. $s(f') = s(f)$
3. $bs(f') = bs'(f')$

Proof: Take any function $f : \{0,1\}^n \rightarrow \{0,1\}$ for which $bs(f) \neq bs'(f)$, and consider the input $\hat{w}$ for which it had maximum block sensitivity. Now consider applying a "reordering" function $g : \{0,1\}^n \rightarrow \{0,1\}^n$ that arranges the input $\hat{w}$ such that the sensitive, disjoint subsets are in contiguous blocks. Now construct another boolean function f' such that:

$$f'(g(w)) = f(w)\ \forall\ w$$

Clearly, $f'(g(w^S)) \neq f'(g(w)) \iff f(w^S) \neq f(w)$. Thus, the sensitivity and block sensitivity of the two functions must be equivalent: $bs(f) = bs(f')$ and $s(f) = s(f')$. This satisfies requirements 1 and 2, and by our choice of g we also satisfy 3.
■

```
bs(F, maxw) {
  BS = 0;
    w = 0;
    maxw++;
    while ( w != maxw ) {
        count = 0;
        // Instead of just looping over i, we need to generate
            partitions
        p = 0;
        while ( p != maxw ) {
            // Partitions are represented as continuous blocks of
                1s or 0s
            // Read off the next partition, and for each index
                flip that bit.
            subcount = 0;
            pcopy = p;
            flag = DIV2(pcopy);
            index = 0;
            // And for each partition, check the sensitivity of each subset
            while (pcopy != 0) {
                wcopy = w;
                do {
                    flip(w, index);
                    index++;
                    bit = DIV2(pcopy);
                }
                while (bit == flag);
                // We have flipped an entire subset, now check if it is sensitive
                match = 0;
                new = F[w];
                if (new != 0) {
                    match++;
                }
                w = wcopy;
                orig = F[w];
                if (orig != 0) {
                    match++;
                }
                // When match == 1, the subset is sensitive
                if (match == 1) {
                    subcount++;
                }
                flag = bit;
            }
            // Take the maximum over all possible partitions
            if (subcount > count) {
                count = subcount;
            }
            p++;
        }
        // Take the maximum over all possible inputs
        if (count > BS) {
            BS = count;
        }
        w++;
    }
  return BS;
}
```

Listing 3: Block sensitivity subroutine

The main program consists of two nested inductive Turing machines, described in the following code as nested loops. The outer loop enumerates polynomials, while the inner loop attempts to break this polynomial bound by finding a function F such that $\mathrm{bs}(F) \geqslant s(F)^J + C$.

The full program, assembled in the format in which it was passed to the KERMIT program, is included in Appendix B.

```
Z = 0;
CJ = 0;
while ( TRUE ) {
  CJ++;
  // See if JC is a valid pair
  numelm = SIZE(CJ);
  if (numelm != 2) {
    // Try another (C,J) candidate
    break;
  }
  else {
    C = CJ[0];
    J = CJ[1];
  }
  // START OF SECOND INDUCTIVE TURING MACHINE
  Y = 0;
  F = 0;
  while ( TRUE ) {
    F++;
    numelm = SIZE(F);
    // Check to see if |F| is a power of 2 by seeing if there
    // is exactly one 1 in the binary representation of F
    cp = numelm;
    k = 0;
    bit = DIV2(cp);
    while (bit != 1) {
      k++;
      bit = DIV2(cp);
    }
    if (cp != 0) {
      // F is not a valid truth table
      break;
    }
    S = s(F, numelm, k);
    BS = bs(F, numelm);
    pol = POW(S,J);
    pol = pol + C;
    if (BS > pol) {
      // This polynomial is not a bound — try another polynomial
      Y = 1;
      break;
    }
  }
  // END OF SECOND INDUCTIVE TURING MACHINE

  // Check the output of the inner inductive turing machine
  if (Y == 0) {
    // Y = 0 indicates that all functions obey the polynomial
        relationship,
    // so we have proven the theorem
    Z = 1;
    HALT;
  }
}
```

Listing 4: Main program

3.3 RESULTS

We ran KERMIT on the full program (included in appendix B), and made the following adjustments to the output before continuing:

- We put the POW subroutine directly into the main program, since it was only called once. The translator printed a warning alerting us to this fact.
- In subroutine flip we adjusted the use of return register retval, and removed lines flip7, flip17, and flip20.
- On lines L8 through L20 we have an if-statement that contains only a break. When an if-else statement is translated, it normally creates a branch at the end of the if-statement body that jumps over the body of the else-statement. But since the the body of the if-statement is an unconditional branch, the second branch on line L10 is unnecessary, and we removed it.
- The second call to ELM (line L16) unnecessarily re-wrote the array CJ into input register a. Register a was already set to CJ by the previous subroutine call. We removed this line. Similarly, we remove L63 and L118.
- In lines L28 through L42 we are determining if the size of F is a power of 2. The KERMIT program uses a register called bit, but in the register machine code it became clear that it was more concise to use register c, the default output register of the DIV2 subroutine. This removed lines L33 and L39.
- When a subroutine call occurs directly before an unconditional branch, we can remove the unconditional branch by setting the subroutine return location (ret) to be the location that the unconditional branch would regularly go to. With this reasoning we were able to remove lines L40 and L169.
- The given input string to subroutine flip is always the string in register w and the index in register index. We were able to remove these input assignment lines: L51, L52, L100 and L101.

To convert the register machine program into its binary representation, we used the *Register machine syntax checker and translator (Version 0.0.3.1)* by Aniruddh Gandhi, University of Auckland. Before making the above adjustments, the program was measured at 4916 bits. After making the optimizing adjustments, it measured at 4594.

3.3.1 *Discussion*

Previous complexity measurements of Π_2 statements have seen an order of magnitude gap between "simple" statements and more complex ones (see Table 2). Our measurement of the Sensitivity Versus Block Sensitivity statement creates a much-needed bridge between these extremes. It also provides a measure for an open problem in computational complexity, while many of the other measured problems come from number theory.

3.3.2 *Assessment of KERMIT*

Making adjustments by hand to the translated register machine program improved the complexity measure by only 6.5%, indicating that the KERMIT program did a good job at concisely translating the Sensitivity Versus Block Sensitivity program. Furthermore, the adjustments that we did make were well-defined and generalizable, indicating that they could be incorporated into the KERMIT program in the future.

It should be noted that KERMIT does not assess or attempt to improve the conciseness of the chosen algorithm. For instance, we could have chosen to represent the boolean functions in many different ways, each of which would have changed our enumeration and many other parts of the main program. These choices still need to be made by the designer. We do hope, though, that because KERMIT makes the compiling of a high-level algorithm to register machine code so much faster, investigators in the future will be able to explore many different algorithms and choose the tightest possible bound.

4 CONCLUSIONS AND FUTURE WORK

4.1 CONCLUSIONS

We have introduced a new translator, KERMIT, that greatly assists the development of descriptive programs for mathematical conjectures. KERMIT was applied to an open problem of considerable interest - the Sensitivity Versus Block Sensitivity of Boolean functions. Using a combination of the KERMIT program and some minor hand-tuning, we created a concise register machine program of size 4594 bits. This puts the sensitivity versus block sensitivity problem between the Twin Prime Conjecture (low complexity) and the P versus NP Problem (high complexity).

4.2 A BRIEF EXPLORATION OF THE P VERSUS BPP PROBLEM

The complexity of the P versus NP problem has been measured [8], and it could be insightful to compare this to other open problems in computational complexity theory. We consequently decided to investigate the P versus BPP problem. The P (polynomial) class is the same as in P versus NP, and is therefore the class of problems solvable in polynomial time on a standard Turing machine. BPP (bounded-error probabilistic polynomial time) characterizes a class of problems that can be solved in polynomial time on a probabilistic Turing machine, with error less than $\frac{1}{3}$. A probabilistic Turing machine is allowed access to a random bit generator (a fair coin). A major open question is: Does P = BPP?

We approached this problem in the same way that we approached P versus NP problem, but quickly ran into trouble. For P versus NP, we had searched for a program P and a polynomial bound (C, J) such that P can solve every instance of Subset Sum in polynomial time. Because Subset Sum is an NP-complete problem, finding such a program means that we have shown P to be equal to NP. Otherwise, $P \subset NP$. But there is no known BPP-complete problem! So we can't simply search for a polynomial time solution to a *particular* problem; we need to search for a polynomial time solution to *all possible problems in BPP*. To determine if we have found a polynomial time solution, we need to test it on every instance of the particular BPP problem. Clearly, this can be formulated only as a Σ_3 or Π_3 statement, and will require three nested loops and an inductive Turing machine of third order. No problem of the third order has been measured, P v. BPP would be the first of that class.

(C, J) *again characterizes the polynomial* $x^J + C$.

We did not continue to investigate P v. BPP after realizing these difficulties. But a descriptive program could be built, and an inductive complexity of the third order measured. It is interesting to note that we were forced into Π_3 by the lack of a BPP-complete problem, but such a problem might exist. If at some point in the future such a problem is discovered, P v. BPP could be described with a Π_2 statement and would be comparable to P v. NP. So while our complexity measure attempts to view mathematical conjectures without regard to their open/closed status, it can still be dependent on our current knowledge or understanding of the problem. In this case, it appears that the complexity of P versus BPP is closely tied to the discovery (or proof of non-existence) of a BPP-complete problem.

4.2.1 *Open Problems*

This research area is filled with fascinating open problems. A small sampling:

- What is the complexity of other major open problems, ex: the Poincare Conjecture?
- Is this complexity measure an effective "difficulty" ranking?
- Can we prove *lower* bounds on this complexity measure?
- Goodstein's Theorem is given a very high complexity measure. Though it has been proven in second order arithmetics, it is unprovable in Peano Arithmetic. Is this a "source" of its complexity? How does it compare to other statements independent of Peano Arithmetic?
- Is there a relationship between the complexity of solved problems, and their proof complexity?

APPENDIX

A THE REGISTER MACHINE INSTRUCTION PREFIX-FREE ENCODING

Each instruction has its own binary op-code. Register names are encoded as the string $\text{code}_1 = 0^{|x|}1x,\ x \in \{0,1\}^*$ and literals are encoded $\text{code}_2 = 1^{|x|}0x,\ x \in \{0,1\}^*$. Some instructions can take registers or literals, but this encoding gives an unambiguous distinction between the two options. The encodings are summarized below:

- `& R1,R2` is coded in two different ways depending on R2:

  $$01\text{code}_1(\text{R1})\text{code}_i(\text{R2}),$$

  where $i = 1$ if R2 is a register and $i = 2$ if R2 is an integer.

- `+ R1,R2` is coded in two different ways depending on R2:

  $$111\text{code}_1(\text{R1})\text{code}_i(\text{R2}),$$

  where $i = 1$ if R2 is a register and $i = 2$ if R2 is an integer.

- `= R1,R2,R3` is coded in four different ways depending on the data types of R2 and R3:

  $$00\text{code}_1(\text{R1})\text{code}_i(\text{R2})\text{code}_j(\text{R3}),$$

  where $i = 1$ if R2 is a register and $i = 2$ if R2 is an integer, $j = 1$ if R3 is a register and $j = 2$ if R3 is an integer.

- `!R1` is coded by

  $$110\text{code}_1(\text{R1}).$$

- `%` is coded by

  $$100.$$

As a concrete example, the subtraction routine from the basic library is given below. It uses registers `a, b, d, e` and `ret`. It computes `a` − `b`, puts the answer in `d` then returns to the line number stored in `ret`. It assumes that `a` ⩾ `b`.

Label	Instruction	Comments	Binary representation
SUBT1	`& d, 0`		`01 00111 100`
SUBT2	`& e, d`		`01 00111 100`
SUBT3	`+ e, b`		`100 011 00100`
SUBT4	`= e, a, ret`	`// d+b=a`	`101 011 010 00101`
SUBT5	`+ d, 1`		`100 00111 101`
SUBT6	`= a, a, SUBT2`	`// loop`	`101 010 010 11010`

The register names, `a` = 010, `b` = 00100, `ret` = 00101, `d` = 00111, and `e` = 011, were chosen to minimize the overall number of bits used. In total, this routine is represented by the 70-bit string:

$$0100111100010011110010001100100101 0$$

$$11010001011000011110110101001011010$$

B THE COMPLETE SENSITIVITY VERSUS BLOCK SENSITIVITY KERMIT PROGRAM

```
Z = 0;
CJ = 0;
while ( TRUE ) {
  CJ++;
  // See if JC is a valid pair
  numelm = SIZE(CJ);
  if (numelm != 2) {
    // Try another (C,J) candidate
    break;
  }
  else {
    C = CJ[0];
    J = CJ[1];
  }
  // START OF SECOND INDUCTIVE TURING MACHINE
  Y = 0;
  F = 0;
  while ( TRUE ) {
    F++;
    numelm = SIZE(F);
    // Check to see if |F| is a power of 2 by seeing if there
    // is exactly one 1 in the binary representation of F
    cp = numelm;
    k = 0;
    bit = DIV2(cp);
    while (bit != 1) {
      k++;
      bit = DIV2(cp);
    }
    if (cp != 0) {
      // F is not a valid truth table
      break;
    }
    // Calculate the sensitivity
    S = 0;
    maxi = k + 1;
    w = 0;
    while ( w != numelm ) {
      // Count the sensitive indices for this particular w
      count = 0;
      index = 0;
      while ( index != maxi ){
        wcopy = w;
        flip(w,index);
        orig = F[wcopy];
        new = F[w];
        if (orig != new) {
          count++;
        }
        w = wcopy;
        index++;
      }
      // Take the maximum over the counts for all inputs
      if (count > S) {
        S = count;
      }
      w++;
    }

    // Calculate the block sensitivity
    BS = 0;
    w = 0;
    while ( w != numelm ) {
      count = 0;
      // Instead of just looping over i, we need to generate
          partitions
      p = 0;
      while ( p != numelm ) {
        // Partitions are represented as continuous blocks of 1s
            or 0s
        // Read off the next partition, and for each index flip
            that bit.
        subcount = 0;
        pcopy = p;
        flag = DIV2(pcopy);
        index = 0;
        // And for each partition, check the sensitivity of each
            subset
        while (pcopy != 0) {
          // make a working copy
          neww = w;
          do {
            flip(neww, index);
            index++;
            bit = DIV2(pcopy);
          }
          while (bit == flag);
          // We have flipped an entire subset, now check if it is
              sensitive
          orig = F[w];
          new = F[neww];
          if (orig != new) {
            subcount++;
          }
          flag = bit;
        }
        // Take the maximum over all possible partitions
        if (subcount > count) {
          count = subcount;
        }
        p++;
      }
      // Take the maximum over all possible inputs
      if (count > BS) {
        BS = count;
      }
      w++;
    }

    pol = POW(S,J);
    pol = pol + C;
    if (BS > pol) {
      // This polynomial is not a bound — try another polynomial
      Y = 1;
      break;
    }
  }
  // END OF SECOND INDUCTIVE TURING MACHINE

  // Check the output of the inner inductive turing machine
  if (Y = 0) {
    // Y = 0 indicates that all functions obey the polynomial
        relationship,
    // so we have proven the theorem
    Z = 1;
    HALT;
  }
}
```

Listing 5: Complete program

C THE REGISTER MACHINE INSTRUCTION TRANSLATION

```
L0              = a, a, L1                      // Jump to the start of MAIN
flip1           & copy, 1
flip2           = w, 0, flip12                  // While loop
flip3           & a, w                          // Input to subroutine
flip4           & flipret, ret                  // Store the return location for the curre
flip5           & ret, flip7                    // Save the return location
flip6           = a, a, DIV21                   // Goto function
flip8           & bit, c                        // Save the result
flip9           + copy, bit

flip10          + copy, copy
flip11          = a, a, flip2                   // Jump to start of the loop
//  Now go through again, and flip the ith bit we come to
flip12          & counter, 0
flip13          & w, 0
flip14          = copy, 1, flip30               // While loop
flip15          + counter, 1
flip16          & a, copy                       // Input to subroutine
flip18          & ret, flip20                   // Save the return location
flip19          = a, a, DIV21                   // Goto function
flip21          & bit, c                        // Save the result
flip22          = counter, index, L25           // If statement
flip23          + w, bit
flip24          = a, a, L28                     //  Jump to end
//  Flip the bit
flip25          = bit, 0, L27                   // If statement
flip26          = a, a, L28                     // Jump to end
flip27          + w, 1
flip28          + w, w
flip29          = a, a, flip14                  // Jump to start of the loop
flip30          = a, a, flipret                 // Return

L1              & Z, 0
L2              & CJ, 0
L3              + CJ, 1
//  See if JC is a valid pair
L4              & a, CJ                         // Input to subroutine
L5              & ret, L7                       // Save the return location
L6              = a, a, SIZE1                   // Goto function
L7              & numelm, c                     // Save the result
```

```
L8              = numelm, 2, L11                // If statement
//  Try another (C,J) candidate
L9              = a, a, L170                    // Break
L11             & a, CJ                         // Input to subroutine
L12             & I, 0                          // Input to subroutine
L13             & ret, L15                      // Save the return location
L14             = a, a, ELM1                    // Goto function
L15             & C, d                          // Save the result
L17             & I, 1                          // Input to subroutine
L18             & ret, L20                      // Save the return location
L19             = a, a, ELM1                    // Goto function
L20             & J, d                          // Save the result
//  START OF SECOND INDUCTIVE TURING MACHINE
L21             & Y, 0
L22             & F, 0
L23             + F, 1
L24             & a, F                          // Input to subroutine
L25             & ret, L27                      // Save the return location
L26             = a, a, SIZE1                   // Goto function
L27             & numelm, c                     // Save the result
//  Check to see if |F| is a power of 2 by seeing if there
//  is exactly one 1 in the binary representation of F
L28             & cp, numelm
L29             & k, 0
L30             & a, cp                         // Input to subroutine
L31             & ret, L34                      // Save the return location
L32             = a, a, DIV21                   // Goto function
L34             = c, 1, L41                     // While loop
L35             + k, 1
L36             & a, cp                         // Input to subroutine
L37             & ret, L34                      // Save the return location
L38             = a, a, DIV21                   // Goto function
L41             = cp, 0, L43                    // If !=, jump to end
//  F is not a valid truth table
L42             = a, a, L165                    // Break
//  Calculate the sensitivity
L43             & S, 0
L44             & w, 0
L45             = w, numelm, L85                // While loop
//  Count the sensitive indices for this particular w
L46             & count, 0
L47             & index, 0
L48             + k, 1
L49             = index, k, L77                 // While loop
L50             & wcopy, w
L53             & ret, L55                      // Save the return location
```

```
L54             = a, a, flip1                   // Goto function
L55             + w, 1
//  Because indexing starts at 1, but input w=0 sometimes, we always +1
L56             & a, F                          // Input to subroutine
L57             & I, w                          // Input to subroutine
L58             & ret, L60                      // Save the return location
L59             = a, a, ELM1                    // Goto function
L60             & new, d                        // Save the result
L61             & w, wcopy
L62             + w, 1
L64             & I, w                          // Input to subroutine
L65             & ret, L67                      // Save the return location
L66             = a, a, ELM1                    // Goto function
L67             & orig, d                       // Save the result
L68             & match, 0
L69             = orig, 0, L71                  // If !=, jump to end
L70             + match, 1
L71             = new, 0, L73                   // If !=, jump to end
L72             + match, 1
L73             = match, 1, L75                 // If !=, jump to end
L74             + count, 1
L75             + index, 1
L76             = a, a, L49                     // Jump to start of the loop
//  Take the maximum over the counts for all inputs
L77             & a, count                      // Input to subroutine
L78             & b, S                          // Input to subroutine
L79             & ret, L81                      // Save the return location
L80             = a, a, CMP1                    // Goto function
L81             = d, 2, L83                     // If statement
L82             = a, a, L84                     // Jump to end
L83             & S, count
L84             = a, a, L45                     // Jump to start of the loop
//  Calculate the block sensitivity
L85             & BS, 0
L86             & w, 0
L87             = w, numelm, L150               // While loop
L88             & count, 0
//  Instead of just looping over i, we need to generate partitions
L89             & p, 0
L90             = p, numelm, L141               // While loop
//  Partitions are represented as continuous blocks of 1s or 0s
//  Read off the next partition, and for each index flip that bit.
L91             & subcount, 0
L92             & pcopy, p
L93             & a, pcopy                      // Input to subroutine
L94             & ret, L96                      // Save the return location
```

```
L95             = a, a, DIV21                   // Goto function
L96             & flag, c                       // Save the result
L97             & index, 0
//  And for each partition, check the sensitivity of each subset
L98             = pcopy, 0, L132                // While loop
//  make a working copy
L99             & wcopy, w
L102            & ret, L104                     // Save the return location
L103            = a, a, flip1                   // Goto function
L104            + index, 1
L105            & a, pcopy                      // Input to subroutine
L106            & ret, L108                     // Save the return location
L107            = a, a, DIV21                   // Goto function
L108            & bit, c                        // Save the result
L109            = bit, flag, L100               // Do loop: jump to start
//  We have flipped an entire subset, now check if it is sensitive
L110            + w, 1
//  Because indexing starts at 1, but input w=0 sometimes, we always +1
L111            & a, F                          // Input to subroutine
L112            & I, w                          // Input to subroutine
L113            & ret, L115                     // Save the return location
L114            = a, a, ELM1                    // Goto function
L115            & new, d                        // Save the result
L116            & w, wcopy
L117            + wcopy, 1
L119            & I, wcopy                      // Input to subroutine
L120            & ret, L122                     // Save the return location
L121            = a, a, ELM1                    // Goto function
L122            & orig, d                       // Save the result
L123            & match, 0
L124            = orig, 0, L126                 // If !=, jump to end
L125            + match, 1
L126            = new, 0, L128                  // If !=, jump to end
L127            + match, 1
L128            = match, 1, L130                // If !=, jump to end
L129            + subcount, 1
L130            & flag, bit
L131            = a, a, L98                     // Jump to start of the loop
//  Take the maximum over all possible partitions
L132            & a, subcount                   // Input to subroutine
L133            & b, count                      // Input to subroutine
L134            & ret, L136                     // Save the return location
L135            = a, a, CMP1                    // Goto function
L136            = d, 2, L138                    // If statement
L137            = a, a, L139                    // Jump to end
L138            & count, subcount
```

```
L139            + p, 1
L140            = a, a, L90                     // Jump to start of the loop
//  Take the maximum over all possible inputs
L141            & a, count                      // Input to subroutine
L142            & b, BS                         // Input to subroutine
L143            & ret, L145                     // Save the return location
L144            = a, a, CMP1                    // Goto function
L145            = d, 2, L147                    // If statement
L146            = a, a, L148                    // Jump to end
L147            & BS, count
L148            + w, 1
L149            = a, a, L87                     // Jump to start of the loop

POW1            & pol, 1
POW2            & e, 0
POW3            = e, J, L142 // Break
POW4            + e, 1
POW5            & f, 1
POW6            & dp, pol
POW7            = f, S, POW3  // pol = pol*S = (pol + pol + ... + pol)
POW8            + pol, dp
POW9            + f, 1
POW10           = a, a, POW7

L155            + pol, C
L156            & a, BS                         // Input to subroutine
L157            & b, pol                        // Input to subroutine
L158            & ret, L160                     // Save the return location
L159            = a, a, CMP1                    // Goto function
L160            = d, 2, L162                    // If statement
L161            = a, a, L164                    // Jump to end
//  This polynomial is not a bound - try another polynomial
L162            & Y, 1
L163            = a, a, L165                    // Break
L164            = a, a, L23                     // While (true) loop
//  END OF SECOND INDUCTIVE TURING MACHINE
//  Check the output of the inner inductive turing machine
L165            = Y, 0, L167                    // If statement
L166            = a, a, L3                      // Loop
//  Y = 0 indicates that all functions obey the polynomial relationship,
//  so we have proven the theorem
L167            & Z, 1
L168            = a, a, HALT                    // Halt the program (jump to end)
HALT            %                               // End of program - Halt
```

BIBLIOGRAPHY

[1] R. Beals and et. al. Quantum lower bounds by polynomials. *Proceedings of the 39th IEEE FOCS*, 1998.

[2] M. Burgin, C. S. Calude, and E. Calude. Inductive complexity measures for mathematical problems. *CDMTCS Research Report* 416, 2011.

[3] C. S. Calude. *Information and Randomness: An Algorithmic Perspective*. Springer, 2nd edition, 2002.

[4] C. S. Calude and E. Calude. Evaluating the complexity of mathematical problems. part 1. *Complex Systems*, 2009.

[5] C. S. Calude and E. Calude. Evaluating the complexity of mathematical problems. part 2. *Complex Systems*, 2010.

[6] C. S. Calude and E. Calude. Algorithmic complexity of mathematical problems: An overview of results and open problems. *CDMTCS Research Report* 410, 2012.

[7] C. S. Calude, E. Calude, and M. J. Dinneen. A new measure of the difficulty of problems. *Journal for Multiple-Valued Logic and Soft Computing*, 2006.

[8] C. S. Calude, E. Calude, and M. S. Queen. Inductive complexity of the p versus np problem. *Parallel Processing Letters*, 2013.

[9] G. J. Chaitin. *Algorithmic Information Theory*. Cambridge University Press, 1987.

[10] G. J. Chaitin. Lecture notes on algorithmic information theory from the 8th estonian winter school in computer science, ewscs'03, 2003.

[11] M. J. Dinneen. A program-size complexity measure for mathematical problems and conjectures. In M. J. Dinneen, B. Khoussainov, and A. Nies, editors, *Computation, Physics and Beyond*. Springer, 2012.

[12] P. Gacs and A. Gal. Lower bounds for the complexity of reliable boolean circuits with noisy gates, 1994.

[13] C. Kenyon and S. Kutin. Sensitivity, block sensitivity, and l-block sensitivity of boolean functions. *Information and Computation*, 2002.

[14] N. Nisan. Crew prams and decision trees. *Proc. of "21-st ACM Symposium on the Theory of Computing"*, 1989.

[15] D. Rubinstein. Sensitivity vs. block sensitivity of boolean functions. *Combinatorica*, 1995.

[16] Madars Virza. Sensitivity versus block sensitivity of boolean functions. *Information Processing Letter*, 2011.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography *"The Elements of Typographic Style"*. `classicthesis` is available for both LaTeX and LyX:

`http://code.google.com/p/classicthesis/`