Electronic Colloquium on Computational Complexity, Report No. 56 (2017)

Towards a Unified Complexity Theory of Total Functions

Paul W. Goldberg* Christos H. Papadimitriou†

April 6, 2017

Abstract

The complexity class TFNP is the set of *total function* problems that belong to NP: every input has at least one output and outputs are easy to check for validity, but it may be hard to find an output. TFNP is not believed to have complete problems, but it encompasses several subclasses (PPAD, PPADS, PPA, PLS, PPP) that do, plus several other problems, such as FACTORING, which appear harder to classify. In this paper we introduce a new class, which we call PTFNP (for "provable" TFNP) which contains the five subclasses, has complete problems, and generally feels like a well-behaved close approximation of TFNP. PTFNP is based on a kind of consistency search problem that we call WRONG PROOF: Suppose that we have a consistent deductive system, and a concisely-represented proof having exponentially many steps, that leads to a contradiction. There must be an error somewhere in the proof, and the challenge is to find it. We show that this problem generalises various well-known TFNP problems, including the five classes mentioned above, making it a candidate for an overarching complexity characterisation of NP total functions. Finally, we note that all five complexity subclasses of TFNP mentioned above capture existence theorems (such as "every dag has a sink") that are true for finite structures but false for infinite ones. We point out that an application of Jacques Herbrand's 1930 theorem shows that this is no coincidence.

1 Introduction

The complexity class TFNP is the set of *total function* problems that belong to NP; that is, every input to such a nondeterministic function has at least one output, and outputs are easy to check for validity — but it may be hard to find an output. It is known from Megiddo [17] that problems in TFNP cannot be NP-complete unless NP is equal to co-NP. On the other hand, various TFNP problems, such as FACTORING and NASH are believed to be genuinely hard [20, 11, 9].

Presently, our understanding of the complexity of TFNP problems is a bit fragmented. Our only means for deriving any evidence of hardness for TFNP problems is by showing completeness in one of the five known subclasses of TFNP, corresponding to well-known elementary non-constructive existence proofs:

- PPP (embodying the pigeonhole principle);
- PPAD (embodying the principle "every directed graph with an unbalanced node must have another");
- PPADS (same as PPAD, except we are looking for an *oppositely unbalanced* node);
- PPA ("every graph with an odd-degree node must have another"), and
- PLS ("every dag has a sink").

Much is known about these classes. PPP is known to contain PPAD and PPADS, while essentially all other possible inclusions are known to be falsifiable by oracles, see for example [2]. They all have complete problems (actually, the most commonly used definition of,

*Oxford University, UK Paul.Goldberg@cs.ox.ac.uk

†University of California, Berkeley, USA christos@berkeley.edu

ISSN 1433-8092

for example, PPAD is "all NP search problems reducible to END OF THE LINE"), and most (PLS, PPAD, PPA) have many other natural complete problems besides the basic one.

Even the union of these classes does not provide a home for all natural TFNP problems. For example, FACTORING is only known to be reducible to PPP and PPA through randomized reductions [14]. The problem RAMSEY (e.g., "Given a Boolean circuit encoding the edges of a graph with 4^n nodes, find n nodes that are either a clique or an independent set") is not known to be in any one of the five classes, and the same obtains for a problem that could be called BERTRAND-CHEBYSHEV ("Given n, produce a prime between n and $2n$").

The status quo in TFNP, as described above, is a bit unsatisfactory. Many natural questions arise: Are there other important complexity subclasses of TFNP, corresponding to novel nonconstructive arguments? Can the three rogue problems above (along with a few others) be classified in a more satisfactory way?

More importantly, *is there a more holistic, unified approach to the complexity of TFNP problems?* For example, are there TFNP-complete problems? The answer here is strongly believed to be "no", as TFNP (the set of all polynomial-depth nondeterministic computations that have a witness, for every input) is very similar in spirit and detail to the classes UP (computations with at most one witness, for every input) and BPP (computations whose fraction of witnesses is bounded away from half, for every input), both known to have no complete problems under oracles [21, 12]. Indeed, Pudlák ([19], Section 6) presents a similar result specifically for TFNP. Hence, this route for a unified complexity view of total functions is not available.

In this paper we make a first step towards the development of a more unified complexity theory of TFNP problems. We define a new subclass of TFNP that includes all five known classes. This new class, which we call PTFNP (for "provable TFNP"), does have complete problems, and these problems are therefore natural generalisations of all known completeness results in TFNP.

In particular, we define a kind of *consistency search problem*, a notion that has recently been studied in the literature on Bounded Arithmetic [4]. Fix a consistent deductive system — in this paper we use a propositional proof system (the domain is the values TRUE/FALSE), in contrast with Bounded Arithmetic, in which variables range over the integers. Now consider a Boolean circuit which, when input an integer j, produces the jth line of an exponentially long purported proof in this system (the line itself is of polynomial length). Suppose further that this proof arrives at a contradiction (one of the lines is "false"). There surely must be a mistaken line in this proof; the challenge is to find it! We call this problem WRONG PROOF, and we define PTFNP as the set of all search problems reducible to it; it is obviously a subset of TFNP. We establish that PTFNP contains PPP (and by extension, PPAD and PPADS), and also PPA and PLS. The study of exponentially-long proofs that are presented concisely via a circuit was introduced by Krajíček [16].

Of course, any finite collection of problems — or classes with complete problems — can be generalised in a rather trivial way, by proposing a new problem or class that artificially incorporates the key features of the old ones. However, WRONG PROOF makes no explicit reference to the problems that are complete for the above complexity classes. Furthermore, it doesn't use very "powerful" logic; essentially we just use quantified boolean formulae with polynomially-many propositional variables, an exponential sequence of n-ary function symbols, and no predicates. The novel features that we exploit are the ability to use exponentially many steps, together with the exponential sequence of function symbols.

Does PTFNP contain FACTORING, RAMSEY, and BERTRAND-CHEBYSHEV? In the final section we discuss the question. Finally, notice that the heretofore "five subclasses" of TFNP correspond to five elementary non-constructive existence arguments in combinatorics, and all these five elementary arguments share one intriguing property: *They only hold for finite structures,* and are false in infinite ones. We show in Section 7 that this is no coincidence:

Herbrand's Theorem from 1930 [13, 5] tells us that any existential sentence in predicate calculus that is true for all models (finite and infinite) is equivalent to the disjunction of a finite number of quantifier-free formulas; it follows that the corresponding TFNP problem is necessarily in P.

Related Recent Work

Various connections have been made between the complexity of TFNP problems and formal proofs, a research direction that seems timely and productive. In a recent paper [3], Arnold Beckmann and Sam Buss, working within the tradition of bounded arithmetic [4], prove certain results that appear to be closely related to the present ones. They define a problem closely related to our WRONG PROOF, and in fact in two versions, one corresponding to Frege systems, and another to extended Frege. Then they show these to be complete for the classes of total function problems in NP whose totality is provable within the bounded arithmetic systems U_2^1 and V_2^1, respectively. The present paper differs from this (and other work in Bounded Arithmetic in general) in that we reduce TFNP problems to a propositional proof system. Also, our focus is on TFNP problem-instances represented using circuits, rather than via oracles.

There are some well-known reducibilities amongst PPAD-like complexity classes, for example that PPAD reduces to PPADS, which reduces to PPP. Buss and Johnson [8] connect these results with derivability relationships (in a proof system) amongst the combinatorial principles that guarantee that they represent total search problems; so for example, the principle underlying PPAD can be derived from the one underlying PPADS, and generally, any such derivability result would tell us that the deriving corresponding complexity classes generalises the other. Our focus here, in contrast, is on formal proofs that correspond with individual *instances* of TFNP problems (finding an error in the proof allows us to find a solution for the corresponding problem-instance).

Pudlák [19] shows how every TFNP problem reduces to a *Herbrand consistency search problem*: any TFNP problem X is characterised by an associated formula Φ whose Herbrand extension is guaranteed to be satisfiable, but the challenge of finding a satisfying assignment is equivalent to X. This correspondence is somewhat reminiscent of Fagin's theorem. The focus of [19] is not on syntactic guarantees that we have a total search problem: it would be hard to check whether a given Φ corresponds to a TFNP problem. By contrast, our definition of WRONG PROOF is intended as a highly-general TFNP problem for which there is a syntactic guarantee that any instance has a solution.

In contrast with most TFNP-related work within bounded arithmetic, we focus on the "white box" concise circuit model of the functions that define the problems characterising the complexity classes of interest. In some respects this makes a significant difference: for example, a recent paper of Komargodski et al. [15] shows that any such TFNP problem has a *query complexity* proportional to the description-size of a problem instance.

Background on propositional proofs and the pigeonhole principle

In 1979, Cook and Reckhow [10] initiated the study of the proofs of propositional tautologies, with regard to the question of how long do such proofs need to be. Abstractly, a *proof system* for a language (here, the set of tautologies) is a scheme for producing efficiently-checkable certificates for words in that language. As noted in [10], a *polynomially bounded* proof system for tautologies is only possible if NP is equal to co-NP. They obtain results that various proof systems can efficiently simulate each other; these results allow us to conclude that one such system is polynomially bounded if and only if another such system is.

[10] introduce *Frege* and *extended Frege* systems: roughly, in a Frege system a proof consists of a sequence of lines containing propositional formulae that are either generated

by some axiom scheme (and are known to hold for that reason) or are derivable by modus ponens from two formulae in previous lines of the proof. In an extended Frege system, we also allow lines that introduce a new propositional variable and set it to equal a propositional formula ϕ over pre-existing variables. The new variable can then be plugged in to a larger formula as a shorthand for ϕ, and if this process is iterated, it may result in an exponential saving in space. It remains a central open problem in proof complexity whether extended Frege proofs can in general be simulated by Frege proofs, with only a polynomial blowup in size of the proof.

In studying this question, various candidate classes of formulae have been considered, the most widely-studied being ones that express the *pigeonhole principle*, as introduced in [10]. The "$n+1$ into n" version of this, denoted PHP_n^{n+1}, states that a function from $n+1$ input values to n output values must map two different inputs to the same output. That is, $f : [n+1] \to [n]$ must have a *collision*: two inputs that f maps to the same output[1]. f can be described by a propositional formula ψ (whose variables indicate which numbers map to which according to f, specifically, variable P_{ij} is TRUE if and only if i is mapped to j) stating "each number in the domain maps to some number in the codomain, and any pair map to different values." By the pigeonhole principle, ψ is unsatisfiable, so its negation ϕ is a tautology (and ϕ has size polynomial in n). [10] gave polynomially-bounded extended Frege proofs of these expressions. Buss [6] subsequently gave polynomially-bounded Frege proofs of these, and in [7] quasi-polynomial size Frege proofs that are a reformulation of the extended Frege proofs of [10]. See [7] for a discussion of other candidate classes of formulae and progress that has been made on them.

Papadimitriou [18] introduced the PIGEONHOLE CIRCUIT problem, in which a pigeonhole function on an *exponential-sized* domain is concisely presented via a boolean circuit C. ψ as constructed above would be exponentially large in C, but a "dual" statement that two inputs to C map to the same output can still be expressed as a concise propositional formula ϕ. By construction, ϕ is satisfiable, and a short proof of this fact consists of a satisfying assignment, but in general such a satisfying assignment appears to be hard to find, and this search characterises the complexity class PPP. In seeking to better understand the challenge, we find a new point of contact between the pigeonhole principle and proof complexity. The difference here is we have a propositional formula that is known to be satisfiable; we want to exhibit a proof of this; but the naive approach of just exhibiting a satisfying assignment is believed to be hard, so instead we fall back on a long and "opaque" proof of satisfiability.

A general question we have only partly answered is, what sort of logic is needed to express such a proof? We have managed to keep it first-order, but we require (exponentially many) lines that define the behaviour of function symbols; thus we have something that corresponds with extended Frege, rather than just Frege proofs, and moreover, these extension rules introduce new functions rather than just new propositional variables. As we discuss in the penultimate section, it would be of interest to see if these proofs could be done just defining exponentially many additional propositional variables.

Organisation of this paper

Section 2 gives details of our deductive system and the problem WRONG PROOF. Section 3 shows how to prove unsatisfiability of certain existential expressions, in such a way that any error in the proof allows a satisfying assignment to be readily reconstructed. Sections 4,5,6 reduce PPP, PPA, and PLS problem-instances to proofs that corresponding existential expressions are satisfiable. (The expressions are the ones we can also "prove" unsatisfiable.) In Section 7 we apply Herbrand's theorem to show that only "finitely valid" combinatorial principles may give rise to hard total search problems. We conclude in Section 8.

[1] We use the standard notation that for a positive integer x, $[x]$ denotes the set $\{1, 2, \ldots, x\}$.

2 Deductive systems and the Wrong Proof problem

A deductive system (or proof system) is a mechanism for generating expressions in some well-defined (formal) language. The expressions should come with a semantics, defining which ones are true and which false. A basic property of a system is *consistency*, that it should not be able to generate two expressions that contradict each other. Consistency is ensured if the rules of the system are valid, in the sense that we cannot deduce any false expressions from true ones. For the deductive system in this paper, the language (and corresponding semantics) of interest is simple and straightforward, and it's not hard to check that it's consistent. The Wrong Proof problem of Definition 2 formalises the computational challenge of receiving a proof of two expressions that contradict each other, and searching for an erroneous step in the proof (guaranteed to exist by the contradiction that we are shown).

The set of expressions that can be produced by a deductive system are called the *theorems* of the system. The system is usually given in terms of a set of *axioms* and *inference rules* that allow theorems to be derived from other theorems. A proof consists of a sequence of numbered *lines*. A line contains a well-formed formula that either holds due to some axiom, or is inferable from the contents of previous lines. A typical line contains one of the following kinds of expression:

$$\ell, \ell' \vdash A, \quad \text{or} \quad \ell \vdash A, \quad \text{or} \quad \vdash A,$$

where A is a well-formed formula inferred at the current line, and ℓ, ℓ' are the numbers of earlier lines (ℓ, ℓ' are thus strictly smaller than the current line number). The expression "$\ell, \ell' \vdash A$" means that the current line claims that A is inferable from the formulae located at lines ℓ and ℓ' (using one of the given inference rules). "$\ell \vdash A$" means that A is inferable from the formula located at line ℓ. "$\vdash A$" means that A holds ipso facto (due to an axiom, e.g. rule (1) lets us write $\vdash (A \vee \neg A)$, for any well-formed formula A).

Our system makes use of a kind of *extension axiom* line, written as $f(x) \leftrightarrow \phi(x)$, where f is a new function symbol whose value on input x is defined by ϕ. f should not occur within ϕ, or in any previous line. So, these lines allow us to define new boolean functions that may appear in later lines.[2]

Definition 1 *With respect to some given consistent deductive system, a* circuit-generated proof *consists of a directed boolean circuit C having n input nodes. C has a corresponding formal proof having 2^n lines. The output of C on input $\ell \in [2^n]$ contains the theorem that has been deduced at line ℓ, together with the numbers of any earlier line(s) from which ℓ's theorem has been deduced.*

In constructing circuit-generated proofs, it's often convenient to identify various exponentially-long sequences of line numbers without assigning numerical values to those line numbers. We can accommodate a collection of such sequences in a circuit-generated proof of size $O(n)$, possibly padded out with unused lines whose theorems consist of the constant TRUE.

Definition 2 *Let S be a consistent deductive system having the property that any line ℓ of a proof that uses S can be checked for correctness in time polynomial in the length of ℓ. An instance of* Wrong Proof *consists of a circuit-generated proof Π_C represented by boolean circuit C.*

[2]This facility to define the behaviour of new functions is a rather novel feature of our system, and gives rise to the question of whether we should be able to make do with standard extended Frege axioms. An extended Frege system is a propositional proof system that allows us to use extension axiom lines of the form $x^{(new)} \leftrightarrow \phi$, where $x^{(new)}$ is a variable symbol that has not occurred previously in the proof, and ϕ is a formula that gives the value of $x^{(new)}$ in terms of pre-existing variables. So, we are allowing ourselves to define new functions on vectors of boolean variables, as opposed to just individual variables. In Section 3.1 we explain why it's useful to have these extension-axiom lines that define new functions. In Section A.8 we discuss obstacles to limiting ourselves to standard extended Frege proofs.

Π_C contains two given lines (say, lines 2^n and $2^n - 1$) that contradict each other: One of them contains as its theorem some expression A and other contains expression $\neg A$. The challenge is to identify some line number ℓ whose corresponding theorem is not derivable in the way stated by $C(\ell)$. Since S is consistent and we have observed a contradiction, such a line must exist.

Wrong Proof is in TFNP: any incorrect line of an instance of Wrong Proof can readily be verified to be incorrect. We have so far defined Wrong Proof rather abstractly, with respect to an unspecified deductive system. In this paper we focus on a specific —and conceptually simple— deductive system that we describe in detail in the rest of this section.

2.1 The formulae and theorems of a proof system; some notation

We work with expressions of quantified propositional logic (variables take values TRUE/FALSE), augmented with a sequence of n-ary function symbols. We also use, for convenience, symbols such as x and y to denote vectors of n propositional variables, and expressions like $x < y$ to denote relationships between x and y, regarding these vectors as representing numbers in $[2^n]$. $x^{(0)}, x^{(1)}, x^{(2)}$ denote respectively the n-vectors (FALSE, ..., FALSE), (FALSE, ..., FALSE, TRUE), (FALSE, ..., FALSE, TRUE, FALSE), or the numbers $2^n, 1, 2$. Since the all-zeroes vector $x^{(0)}$ corresponds to 2^n, this means that $x^{(0)} \geq x$ for any other vector x (this convention tends to reduce clutter in our expressions).

In this paper, the two contradictory statements in an instance of Wrong Proof take the form $\exists x, x'(\phi(x, x'))$ and $\neg\exists x, x'(\phi(x, x'))$, asserting that some $2n$-variable formula ϕ is (respectively, isn't) satisfiable. We continue with more detail the expressions used in our proofs.

For complexity parameter n, the vocabulary we use contains a polynomial-size collection of variable symbols, together with an *exponential-size* collection of n-ary function symbols; these are denoted by f_i where $i \in [2^n]$. In our proofs, f_{2^n} is defined in terms of an instance of some TFNP problem, and (for each $i \in [2^n]$) f_{i-1} is defined in terms of f_i via an extension-axiom line. There are no predicates. The expressions we use are first-order, in that they may have quantification over the variable symbols, but not the functions.

While we work with expressions whose variables represent vectors of propositional variables, note that such expressions represent polynomially-larger expressions whose variables are simple propositional variables. Variable x represents $(x_1, \ldots, x_n)$ where the x_i are propositional variables, and expressions involving x can be converted to basic propositional formulae in the individual x_i without an excessive blowup in the size of the formula. This extra syntax makes our expressions more concise and readable. For example, given non-zero vectors x, x', the expression $x < x'$ represents the following propositional formula involving the variables x_i and x'_i (treating x_1 and x'_1 as the most significant bits):

$$\neg x_1 \wedge x'_1 \vee (x_1 = x'_1 \wedge (\neg x_2 \wedge x'_2 \vee (x_2 = x'_2 \wedge (\neg x_3 \wedge x'_3 \vee \cdots (\neg x_n \wedge x'_n))) \cdots))$$

Another notational convenience that we use is expressions such as $\forall x < y(\phi(x, y))$, meaning $\forall x, y(x < y \to \phi(x, y))$, or if y is a vector of propositional constants, it would mean $\forall x(x < y \to \phi(x, y))$. Similarly, $\exists x \neq y\phi(x, x')$ means $\exists x, x'(x \neq x' \wedge \phi(x, x'))$.

2.2 Axioms and inference rules

We use the following kinds of rules:

- Axioms (written as $\vdash A$) let us write down certain expressions that can be seen to evaluate to TRUE based on some easily-checkable property, for example A is of the form $B \vee \neg B$.

- Inference rules, written as $A, B \vdash C$ for example, say that given expressions A and B, we can write the expression C.
- Equivalences, written as $A \equiv B$, say that two expressions are logically equivalent. An equivalence represents a *rule of replacement* in that it may be applied to sub-expressions of any expression that appears in a line of a proof. For example, using the equivalence $A \wedge B \equiv B \wedge A$ we could take a line ℓ containing the expression $\textsc{true} \vee (x_i \wedge y_i)$ and write a new line containing $\ell \vdash \textsc{true} \vee (y_i \wedge x_i)$.
- "Extension axiom" lines define new n-ary functions, and are written as $f(x) \leftrightarrow \phi(x)$, where f is a new symbol that has not appeared previously in the proof, and ϕ specifies how f behaves on input (n-vector) x. So, this kind of line can be taken to mean $\forall x(f(x) \triangleq \phi(x))$.

Some of the rules we list below are redundant in the sense that they could be simulated using the others. We have generally have tried to limit ourselves to rules that are not too novel and ad-hoc, that are clearly consistent, and which, crucially, allow that any individual line of a proof can be checked for correctness in time polynomial in n. Section 2.3 contains rules that we prove can be simulated by the ones in Section 2.2; usage of these additional rules allows some of the formal proofs to be presented more cleanly. We have not however tried to minimise the collection of rules in Section 2.2; some of the rules in the section can be simulated using the others.

Our extension axiom lines are somewhat novel. In a standard *extended Frege* system, a line of the proof may consist of an *extension axiom*, introducing a new propositional variable and setting its value to equal some expression in terms of pre-existing values. Here we use extended Frege-style lines that define new *functions* —see rule (13)— via expressions that define their behaviour in terms of pre-existing functions.

In the following, A, B, C represent arbitrary well-formed formulae and x, y are length-n vectors of propositional variables, where x (say) may also be thought of as ranging over integers in the range $[2^n]$, as noted in Section 2.1. The equivalences we allow ourselves to use include standard rules of replacement, such as commutativity, associativity, and distributivity of propositional connectives, removal of double negation, and de Morgan's rules. We also use $A \equiv A \vee A \equiv A \wedge A \equiv A \wedge \textsc{true} \equiv A \vee \textsc{false}$, also $A \to B \equiv \neg B \vee A$, and the identity $A \to (B \to C) \equiv (A \wedge B) \to C$. We also allow a step of a proof to rename a bound variable throughout the subexpression where it occurs. These equivalences may be applied to any expression arising in a derivation, also they may be applied (in a simple step) to any well-formed subexpression of a larger expression arising in a derivation. So, a line of a proof of the form $\ell \ \vdash A$ can be used to state that A is derived from the expression A' at line ℓ via applying one of these basic manipulations to A', or some subexpression of A'. It is easy to see that any such step may be checked for correctness in polynomial time, and there is no need for a line to specify which rule is being used.

For any well-formed expression A, and any vector x of propositional variables, we may use any of the following lines in our proofs:

$$\vdash (A \to A), \quad \vdash (A \vee \neg A), \quad \vdash \textsc{true} \tag{1}$$

Modus ponens (rule (2)) states that if lines ℓ and ℓ' contain theorems A and $A \to B$ respectively, a subsequent line containing the expression "$\ell, \ell' \vdash B$" is a valid line.

$$A,\ A \to B\ \vdash\ B \tag{2}$$

"Conjunction introduction" (rule (3)) states that if lines ℓ and ℓ' contain theorems A and B respectively, a subsequent line containing the expression "$\ell, \ell' \vdash A \wedge B$" is valid.

$$A,\ B\ \vdash\ A \wedge B \tag{3}$$

A "case analysis" rule (4) (a form of disjunction elimination) means that if lines ℓ and ℓ' contain theorems $B \to A$ and $\neg B \to A$, then a subsequent line containing "$\ell, \ell' \vdash A$" is valid.

$$B \to A,\ \neg B \to A \ \vdash\ A \tag{4}$$

The disjunction introduction rule (5) means that if line ℓ contains theorem A, then a subsequent line containing $\ell \vdash A \vee B$ is valid.

$$A \ \vdash\ (A \vee B) \tag{5}$$

Antecedent strengthening:

$$(A \to C) \ \vdash\ (A \wedge B \to C) \tag{6}$$

Basic equivalences for quantified variables: let x_i be an individual propositional variable; let $A(\text{TRUE})$ and $A(\text{FALSE})$ be obtained by plugging in the constants TRUE and FALSE respectively in place of x_i, in $A(x_i)$. Then we have:

$$\begin{aligned} \exists x_i(A(x_i)) &\equiv A(\text{TRUE}) \vee A(\text{FALSE}) \\ \forall x_i(A(x_i)) &\equiv A(\text{TRUE}) \wedge A(\text{FALSE}) \end{aligned} \tag{7}$$

Distributive rules for quantifiers (recall x is a vector of variables):

$$\begin{aligned} \exists x(A(x)) \vee \exists x(B(x)) &\equiv \exists x(A(x) \vee B(x)) \\ \forall x(A(x)) \wedge \forall x(B(x)) &\equiv \forall x(A(x) \wedge B(x)) \end{aligned} \tag{8}$$

(In the context of circuit-generated proofs, the distributive rules (8) can be derived from the previous rules. Recall that x denotes the n-vector $(x_1, \ldots, x_n)$. Starting from the expression $\forall x(A(x) \wedge B(x))$, we go via intermediate expressions of the form $\forall(x_1, \ldots, x_j)(\forall(x_{j+1}, \ldots, x_n)A(x) \wedge \forall(x_{j+1}, \ldots, n)B(x))$ to end up with $\forall x(A(x)) \wedge \forall x(B(x))$, while keeping all intermediate expressions to be of polynomial length.)

Bringing quantifier to front: suppose A contains no variables in x, then if $\circ$ is any boolean connective, we have

$$\begin{aligned} A \circ \exists x(B) &\equiv \exists x(A \circ B) \\ A \circ \forall x(B) &\equiv \forall x(A \circ B) \end{aligned} \tag{9}$$

Universal instantiation: let $A(t)$ be the expression obtained by plugging in term t in place of variable symbol x (t is any term, i.e. a propositional variable or constant, or a function symbol applied to other terms.)

$$\forall x(A(x)) \ \vdash\ A(t) \tag{10}$$

Universal generalization: if x and y are n-vectors of propositional variables, and x is a vector of free variables, we have

$$A(x) \ \vdash\ \forall y A(y) \tag{11}$$

Existential generalization: if $A(x)$ is obtained by plugging in variable(s) x in place of term(s) t, we have

$$A(t) \ \vdash\ \exists x(A(x)) \tag{12}$$

Extended Frege-style definitions of functions:

We use extension axioms written as:

$$f(x) \leftrightarrow \phi(x) \tag{13}$$

where ϕ is an expression that defines the value of $f(x)$. ϕ may include functions defined earlier, but not f. f is a new function symbol, x is a vector of variable symbols, and $\phi(x)$ is a formula that specifies the value taken by f on any input x. This rule can be understood as saying $\forall x(f(x) \triangleq \phi(x))$.

2.3 Further rules derivable from the ones of Section 2.2

It's useful to note the following further rules for writing down lines of a proof, which can be simulated by the ones of Section 2.2. We can assume we have the "hypothetical syllogism" rule, $A \to B,\ B \to C \vdash A \to C$ (we can simulate this using the rules of Section 2.2: a combination of modus ponens and case analysis). We can also assume we have an "axiom" saying that expressions of the following form can be written down for free: $\forall x(A(x)) \to A(t)$, where t is a n-vector of terms that is plugged in for (n-vector) x in A. (We can write down $\forall x(A(x)) \to \forall x(A(x))$, equivalently $\forall x(A(x)) \to \forall y(A(y))$, where y is another n-vector of propositional variables, equivalently $\forall x, y(A(x) \to A(y))$, then by universal instantiation, $\forall x(A(x) \to A(t))$, which is equivalent to $\forall x(A(x)) \to A(t)$.) In a similar way, we can write down expressions of the form $A(t) \to \exists x(A(x))$.

We also use the equivalences (derivable from (7) and de Morgan's rules):

$$\begin{aligned} \neg\exists x(A) &\equiv \forall x(\neg A) \\ \neg\forall x(A) &\equiv \exists x(\neg A) \end{aligned} \tag{14}$$

In constructing circuit-generated proofs, it's convenient to allow the following kind of line. Suppose ϕ is a propositional formula over a vector x of n terms, consisting of variables, or functions applied to variables. Let $i \in [2^n]$ be a satisfying assignment of ϕ, so i is a vector of n constants TRUE/FALSE. We may use the rule

$$\vdash\ x = i \to \phi(x) \tag{15}$$

Rule (15) can be simulated using previous rules, as follows. Using the axiom $A \to A$, we can write a line containing $\vdash\ (x = i \to \phi(x)) \to (x = i \to \phi(x))$. We then apply a sequence of basic manipulations to the first occurrence of $(x = i \to \phi(x))$, simplifying it to the constant TRUE: provided that i really satisfies ϕ, this should be achievable. (These manipulations just do the job of plugging into ϕ the n propositional constants in vector i, and simplifying. We can ensure that intermediate expressions are of polynomial size, by pushing any occurrences of $\neg$ to the bottom of the parse tree of ϕ; write the expression as $(x_1 = i_1 \to (x_2 = i_2 \to \ldots x_n = i_n \to \phi(x))\ldots)$, and repeatedly use equivalences $A \to B \circ C \equiv (A \to B) \circ (A \to C)$, for $\circ \in \{\wedge, \vee\}$.) This leaves us with TRUE $\to (x = i \to \phi(x))$, which is equivalent to $x = i \to \phi(x)$.

We also make use of equivalence (16), which can be simulated in a straightforward way using the previous rules. Letting x be an n-vector of propositional variables and i an n-vector of propositional constants, and ϕ a quantifier-free boolean formula, we have

$$x = i \to \phi(x) \quad \equiv \quad \phi(i) \tag{16}$$

3 Preliminaries to the reductions to Wrong Proof

In this section we establish results that are useful subsequently, and we discuss certain features that our reductions all have in common with each other.

An instance of Wrong Proof is supposed to consist of proofs of two contradictory statements, and in our reductions, these statements take the form $\exists(x, x')\phi(x, x')$ and $\neg\exists(x, x')\phi(x, x')$, for n-vectors x, x' of propositional variables. ϕ depends on the specific instance of a TFNP problem that we reduce from.

Any problem in TFNP is reducible to the search for a satisfying assignment to a propositional formula ϕ, where ϕ obeys some syntactic constraint that guarantees that it does, in fact, have a satisfying assignment.[3] In reducing to WRONG PROOF, we "prove" the contradictory statements $\exists(x,x')\phi(x,x')$ and $\neg\exists(x,x')\phi(x,x')$ where x,x' are vectors of n propositional variables. In fact, the ϕ that we use is not purely propositional; it includes a function symbol that's constructed (using our extension-axiom rule) to encode a TFNP problem-instance, in a way described in Section 3.2.

The proofs of these contradictory statements consist of sequences of applications of the rules of Sections 2.2, 2.3, and they are instances of WRONG PROOF, i.e. long proofs presented via a circuit. The error occurs in the "proof" of $\neg\exists(x,x')\phi(x,x')$. Of course, it's trivial to exhibit a faulty proof of the unsatisfiability of ϕ, but we require something more, namely that any error should let us efficiently reconstruct a satisfying assignment of ϕ. Lemma 1 shows how to construct such a proof. The three expressions in the statement of Lemma 1 correspond to the existence principles underlying PPP, PPA, and PLS (recall that PPAD and PPADS are special cases of PPP).

The proofs of $\exists(x,x')\phi(x,x')$ are done separately for each TFNP problem of interest, in Sections 4, 5, 6. Section 3.1 introduces the general approach taken in Sections 4, 5, 6 to construct those proofs. Section 3.2 presents Lemma 1 that shows how to make a suitable proof of $\neg\exists(x,x')\phi(x,x')$. Section 3.3 proves some technical results that show how circuit-generated proofs of certain expressions can be constructed.

3.1 Overview of the reductions presented in Sections 4, 5, 6

In Sections 4, 5, 6, we consider computational problems PIGEONHOLE CIRCUIT, LONELY, and ITER, which are complete for PPAD, PPA, and PLS respectively. We reduce each of these problems to WRONG PROOF.

Any instance of the problems PIGEONHOLE CIRCUIT, LONELY, and ITER is defined in terms of a boolean circuit C. Section 3.2 begins with a general method to define a function f using the rules of our deductive system, so that f is the function computed by C. We derive from C an existential formula $\Phi = \exists(x,x')\phi(x,x')$ in terms of f stating (correctly) that there is a solution associated with the instance of the problem. We have noted that Section 3.2 shows how to "prove" $\neg\Phi$. Sections 4, 5, 6 show how to construct contrasting (and correct!) circuit-generated proofs of Φ. The approach to proving that Φ is satisfiable, is based on a syntactic feature that assures us that it is, indeed, satisfiable. These syntactic features are different for the three problems under consideration (which is why we have three different complexity classes), so we need three distinct reductions.

At this point we are ready to explain our usage of extension axioms (rules of type (13)) to define long sequences of new n-ary boolean functions. In the context of PIGEONHOLE CIRCUIT, any instance I has an associated function $f_I : [2^n] \to [2^n - 1]$, and the search is for two inputs to f_I that map to the same output. Call such a pair of inputs a "collision" for f_I. We reduce the search for a collision for f_I to the search for a collision for a new function $f'_I : [2^n - 1] \to [2^n - 2]$. f'_I is defined in terms of f_I using an extension-axiom line. We reduce this in turn to the search for a collision for a new function $f''_I : [2^n - 2] \to [2^n - 3]$, and so on. With an exponential sequence of similar reductions (that can all be efficiently generated via a circuit), we eventually reduce to the search for a collision of a function from $\{1,2\}$ to $\{1\}$, whose existence has a simple (formal) proof. LONELY and ITER have similar sequences of functions.

Functions defined using rules of type (13) have the codomain $\{\text{TRUE}, \text{FALSE}\}$. f_I can of course be defined in terms of n n-ary functions that map to individual bits of the output of

[3] To see this, note that for any problem $X \in$ TFNP, any instance I of size n has a solution S_I of size $poly(n)$; solutions are checkable with a poly-time algorithm $\mathcal{A}$ that takes candidate solutions as input and outputs "yes" iff $\mathcal{A}$ received a valid solution. $\mathcal{A}$ can be converted to a circuit and thence to a propositional formula that is satisfied by inputs representing any valid solution S_I of instance I along with extra propositional variables for gates of the circuit.

f_I, as can each of the exponential sequence of functions that is derived from it.

We have aimed to make the presentation as consistent as possible for the three reductions to Wrong Proof. The following presentational aspects are shared by the reductions. We let C denote a typical instance of a TFNP problem, since the problem-instances we consider are represented as (boolean) circuits. Π_C denotes the corresponding instance of Wrong Proof. We describe Π_C in terms of the lines of Π_C, as opposed to the circuit that generates it: for the exponential sequences of lines that we define, we assume it's easy to check that they can be compactly represented using a circuit. f denotes the function computed by C; f is constructed using extension-axioms as described at the start of the next subsection. We set a new function f_{2^n} equal to f. The reductions use sequences of well-formed expressions that appear in the instances of Wrong Proof, that we denote A_i, C_i and F_i, for $i \in [2^n]$. F_i is an extension-axiom line that defines new function f_{i-1} in terms of f_i. A_i asserts implicitly (or non-constructively) that an instance of a problem corresponding to function f_i has a guaranteed solution (due to a syntactic property of f_i). C_i is an existential expression that asserts that same thing explicitly. We end up proving C_{2^n} that states the existence of a solution, and C_{2^n} is equivalent to Φ. This contradicts the expression $\neg\Phi$ that is "proved" using Lemma 1.

We work through the formal steps for the first reduction (from Pigeonhole Circuit) in some detail (mainly in the appendices), and do rather less detail on the formal steps for the reductions from Lonely and Iter.

3.2 Construction of functions from circuits, and a method for locating the errors in instances of Wrong Proof

Given a boolean circuit C with n input nodes, our deductive system can define a function f that computes C as follows. Each gate g of C has an associated n-ary function f_g mapping the inputs to C to the value taken at g. We can construct f using a sequence of extension-axiom rules (of type (13)), in which if, say, gate g is the AND of gates g' and g'', then we add the rule $f_g(x) \leftrightarrow f_{g'}(x) \wedge f_{g''}(x)$. If g is the j-th input gate, then f_g is defined by $f_g(x) \leftrightarrow x_j$, where x_j is the j-th component of n-vector x.

Lemma 1 *Suppose f is defined according to the above construction. Consider the expressions*[4]

- $\exists(x,x')((x \neq x' \wedge f(x) = f(x')) \vee f(x) = x^{(0)})$,
- $\exists(x,x')(f(x^{(1)}) \neq x^{(1)} \vee (x \neq x^{(1)} \wedge f(x) = x) \vee (x' = f(x) \wedge x \neq f(x'))$,
- $\exists(x,x')(f(x^{(1)}) = x^{(1)} \vee f(x) < x \vee (x' = f(x) \wedge f(x') = f(x))$.

We can efficiently construct circuit-generated proofs of the negations of these expressions in such a way that any error in the proof allows us to efficiently construct (x,x') satisfying the expression.

The expressions in the statement of Lemma 1 are the principles underlying PPP, PPA, and PLS, used in Theorems 1, 2, 3. They are all satisfiable, so their negations are all false.

Proof. The negation of any of the above expressions takes the form $\forall(x,x')(\phi(x,x'))$, where ϕ performs some test on values of x, x', $f(x)$, and $f(x')$. For example, the negation of the first of these expressions is

$$\forall(x,x')\neg\Big((x \neq x' \wedge f(x) = f(x')) \vee f(x) = x^{(0)}\Big). \tag{17}$$

We show how to construct a circuit-generated proof of (17) such that any error will identify a pair of n-vectors x, x' whose existence is claimed by the first of the three existential

[4]Recall that $x^{(0)}$ and $x^{(1)}$ denote the all-zeroes bit-vector, and the bit vector corresponding to number 1.

statements. The following approach applies also to the negations of the other two existential expressions in the statement of this lemma.

Let M be the matrix of (17), i.e. the subexpression $\neg((x \neq x' \wedge f(x) = f(x')) \vee f(x) = x^{(0)})$. We continue by giving a method for proving the following stronger expression, from which (17) is derivable:

$$\forall(x,x')(C_1 \wedge \ldots \wedge C_m \wedge M) \tag{18}$$

where C_i are clauses that construct the values of $f(x), f(x')$ by working through the values taken at the gates of the circuit; the C_i are of the form $f_g(x) = f_{g'}(x) \circ f_{g''}(x)$ (for $\circ \in \{\wedge, \vee\}$), or $f_g(x) = \neg f_{g'}(x)$, or $f_g(x) = x_j$ (in the case that g is the j-th input gate). M is a boolean combination of expressions of the form $f_g(x) = f_{g'}(x')$ or $f_g(x) \neq f_{g'}(x')$, for output gates g, g', or of the form $f_g(x) = \text{TRUE}/\text{FALSE}$.

Let $\phi'(x,x') = C_1 \wedge \ldots \wedge C_m \wedge M$, and for $i \in [2^{2n}]$ let Φ'_i be the formula $\forall(xx' \leq i)\phi'(x,x')$, where xx' represents the $2n$-digit number $2^n(x-1)+x'$. It can be formally proved that (18) is equivalent to $\Phi'_{2^{2n}}$; we omit the details. For each $i \in [2^{2n}]$, some line ℓ_i of the proof contains Φ'_i. We show below how to prove expressions of the form $(xx' = i) \rightarrow \phi'(x,x')$, which we then use to derive Φ'_i from Φ'_{i-1} in conjunction with $(xx' = i) \rightarrow \phi'(x,x')$. In particular we can derive $\Phi'_{i-1} \wedge ((xx' = i) \rightarrow \phi'(x,x'))$, equivalently $\forall xx'((xx' < i \rightarrow \phi'(x,x')) \wedge (xx' = i \rightarrow \phi'(x,x')))$, equivalently $\forall xx'((xx' < i \vee xx' = i) \rightarrow \phi'(x,x'))$, equivalently (details in Section A.7), $\forall xx'(xx' \leq i \rightarrow \phi'(x,x'))$, which is the same as Φ'_i.

How to formally prove $(xx' = i) \rightarrow \phi'(x,x')$:

For each gate g of C, in the order in which the functions f_g are defined, we can prove a line saying

$$(xx' = i) \rightarrow f_g(x) = j_g(x)$$

where $j_g(x) \in \{\text{TRUE}, \text{FALSE}\}$ is the appropriate propositional constant. This is done by using the extension-axiom line that defines f_g, with gate g's inputs. (If say g takes inputs from g' and g'', we use previous lines containing expressions $(xx' = i) \rightarrow f_{g'}(x) = j_{g'}(x)$, $(xx' = i) \rightarrow f_{g''}(x) = j_{g''}(x)$.)

Letting $g(1), \ldots, g(m)$ be the sequence of gates, listed in the order in which their functions $f_{g(1)}, \ldots, f_{g(m)}$ are defined, we have

$$(xx' = i) \rightarrow \bigwedge_{r \in [m]} (f_{g(r)}(x) = j_{g(r)}(x), f_{g(r)}(x') = j_{g(r)}(x'))$$

It then suffices to prove

$$\Big((xx' = i) \wedge \bigwedge_{r \in [m]} (f_{g(r)}(x) = j_{g(r)}(x), f_{g(r)}(x') = j_{g(r)}(x'))\Big) \rightarrow M$$

which is a line of type (15), and can be proved by the procedure of plugging in the constants $i, j_{g(r)}(x), j_{g(r)}(x')$ in place of the terms $x, x', f_{g(r)}(x), f_{g(r)}(x')$ in the way described below Equation (15). An error in the proof will correspond to this expression evaluating to FALSE, and getting treated as TRUE.

To conclude, note that we can construct a small circuit that on input $i \in [2^{2n}]$, outputs the above proof of $(xx' = i) \rightarrow \phi'(x,x')$. The circuit can be extended to a concise proof of (17). □

3.3 Technical lemmas

The following results are useful for showing how to construct certain aspects of circuit-generated proofs, but can be skipped at a first reading. Lemma 2 and Corollary 1 are conceptually similar to Lemma 1: Corollary 1 applies to arbitrary formulae $\Phi = \exists x\phi(x)$ where ϕ is entirely propositional, having no function symbols (by contrast, Lemma 1 applies to a special class of ϕ's that contain function symbols). Lemma 3 is a more sophisticated version of Lemma 2; it is used in the proofs in the appendix, along with Lemma 4.

Lemma 2 *Let $\phi(x)$ be a propositional formula over n-vector x. We can construct in time polynomial in the size of ϕ, a circuit C that generates a proof Π of $\forall x\phi(x)$ such that*

- *if ϕ is a tautology, then Π is a valid proof, using the rules of Section 2.2,*
- *if ϕ is not a tautology, any error in Π allows us to construct some $\hat{x}$ for which $\neg\phi(\hat{x})$ holds.*

Proof. Let Φ_i be the formula $\forall x \leq i(\phi(x))$. It can be proved formally that Φ_{2^n} is equivalent to $\forall x\phi(x)$; we omit the details.

For each $i \in [2^n]$, Π contains a line ℓ_i containing Φ_i, which may be formally derived from Φ_{i-1} (itself located at a known line $\ell_{i-1} < \ell_i$) together with a line stating that i satisfies ϕ, which we give more detail on as follows.

Using rule (15) we can write a line containing the expression

$$(x = i) \rightarrow \phi(x).$$

(If i does not satisfy ϕ, this line is incorrect, and the error allows us to recover the value i that does not satisfy ϕ.)

By universal generalisation (rule (11)) we can deduce $\forall x(x = i \rightarrow \phi(x))$.

Applying conjunction introduction (rule (3)), we can deduce from this and Φ_{i-1} (recall that $\Phi_{i-1} = \forall x(x \leq i-1 \rightarrow \phi(x))$):

$$\forall x(x \leq i-1 \rightarrow \phi(x)) \wedge \forall x(x = i \rightarrow \phi(x)).$$

Using rule (8) we get $\forall x((x \leq i-1 \rightarrow \phi(x)) \wedge ((x = i) \rightarrow \phi(x)))$; using the equivalence $A \rightarrow B \equiv \neg A \vee B$, and distribution of disjunction over conjunction we get $\forall x((x \leq i-1 \vee x = i) \rightarrow \phi(x))$. Finally $x \leq i-1 \vee x = i$ can be manipulated further to get $x \leq i$ (see Section A.7), from which we get Φ_i. □

Corollary 1 *Let Φ be a formula of the form $\exists x\phi(x)$, where x is a vector of propositional variables that constitute the free variables of propositional formula ϕ. We can efficiently construct a circuit-generated proof Π of $\neg\Phi$, such that if ϕ is unsatisfiable (and thus $\neg\Phi$ holds), then Π has no errors, and if ϕ is satisfiable then Π has at least one error, and given any error in Π we can efficiently recover a satisfying assignment of ϕ.*

Corollary 1 follows by noting that $\neg\Phi$ is equivalent (by (14)) to $\forall x(\neg\phi(x))$. We then apply Lemma 2 to $\neg\phi(x)$. Corollary 1 is a general construction of a circuit-generated proof that a propositional formula ϕ is unsatisfiable: the proof is correct if indeed ϕ is unsatisfiable, and from any error we can easily recover a satisfying assignment. The reader might briefly wonder whether a similarly general circuit-generated proof should be constructible that ϕ is satisfiable. The answer is no (unless NP=co-NP): such a result would provide unsatisfiable formulae with concise certificates (of unsatisfiability). In Sections 4, 5, 6 we give separate proofs of satisfiability that exploit structural properties of formulae corresponding to the syntactic TFNP complexity classes of interest there.

Lemma 3 *Suppose we have a circuit that takes as input $i \in [2^n]$, and outputs a proof of $x = i \rightarrow (\phi(x) \rightarrow \psi(x))$, where x is a vector of n propositional variables. Then we can efficiently construct a circuit-generated proof of $\forall x\phi(x) \rightarrow \forall x\psi(x)$.*

Proof. Let Π_i be the proof of $x = i \to (\phi(x) \to \psi(x))$, constructed by the circuit. We show how to construct a proof Π of $\forall x \phi(x) \to \forall x \psi(x)$. Π contains, for each $i \in [2^n]$, a copy of Π_i, containing at some line ℓ_i the expression $x = i \to (\phi(x) \to \psi(x))$.

Via a sequence of elementary manipulations we can derive from ℓ_i the following line ℓ'_i (ℓ_i, ℓ'_i are easily computable from i; $\ell'_i > \ell_i > \ell'_{i-1}$) containing the expression:

$$(x = i \to \phi(x)) \to (x = i \to \psi(x)).$$

Let $\Phi = \forall x \phi(x)$ and $\Psi = \forall x \psi(x)$, thus Π should end with a line containing $\Phi \to \Psi$.

Let $\Phi_i = \forall x \leq i(\phi(x))$ and $\Psi_i = \forall x \leq i(\psi(x))$.

Π contains a straightforward proof of $\Phi_1 \to \Psi_1$ (at a line with number ℓ''_1) and for each $i > 1$, $i \in [2^n]$, a line with number $\ell''_i > \ell''_{i-1}$ containing $\Phi_i \to \Psi_i$, whose proof uses ℓ''_{i-1} and ℓ'_i.

We derive $\Phi_i \to \Psi_i$, starting from ℓ''_{i-1} containing $\Phi_{i-1} \to \Psi_{i-1}$ and ℓ'_i, for all $i \in [2^n]$, $i \geq 2$. These derivations can then be chained together to obtain a circuit-generated proof of $\Phi_{2^n} \to \Psi_{2^n}$, which will be seen to be equivalent to $\forall x \phi(x) \to \forall x \psi(x)$.

Using conjunction introduction on lines ℓ''_{i-1} and ℓ'_i, we have

$$(\Phi_{i-1} \to \Psi_{i-1}) \wedge \Big((x = i \to \phi(x)) \to (x = i \to \psi(x))\Big).$$

From this we can derive

$$(\Phi_{i-1} \wedge (x = i \to \phi(x))) \to (\Psi_{i-1} \wedge (x = i \to \psi(x))).$$

Writing Φ_{i-1} and Ψ_{i-1} in full, we have

$$\Big(\forall x(x < i \to \phi(x)) \wedge (x = i \to \phi(x))\Big) \to \Big(\forall x(x < i \to \psi(x)) \wedge (x = i \to \psi(x))\Big).$$

Next, we want to replace the subexpression $(x = i \to \phi(x))$ with $\forall x(x = i \to \phi(x))$, and similarly the expression $(x = i \to \psi(x))$ with $\forall x(x = i \to \psi(x))$. To do this, we show a chain of logical equivalences, applied to the first of these subexpressions. (It is tempting to apply universal generalisation (rule 11) to these subexpressions, but (11) can only be applied to an entire expression, not a subexpression.) The expression $\forall x(x = i \to \phi(x))$, written out in full, is

$$\forall (x_1, \ldots, x_n)\Big((x_1 = i_1 \wedge \ldots \wedge x_n = i_n) \to \phi(x)\Big).$$

Applying the basic equivalence of quantified variable x_n, rule (7), we have

$$\forall (x_1, \ldots, x_{n-1})((x_1 = i_1 \wedge \ldots \wedge x_{n-1} = i_{n-1} \wedge \text{TRUE} = i_n) \to \phi(x; x_n = \text{TRUE})) \wedge$$

$$(x_1 = i_1 \wedge \ldots \wedge x_{n-1} = i_{n-1} \wedge \text{FALSE} = i_n) \to \phi(x; x_n = \text{FALSE}))$$

If, say, $i_n = \text{FALSE}$, then the subexpression $(x_1 = i_1 \wedge \ldots \wedge x_{n-1} = i_{n-1} \wedge \text{TRUE} = i_n) \to \phi(x; x_n = \text{TRUE}))$ evaluates to TRUE, and so can be eliminated, leaving just the other subexpression. This step uses a sequence of equivaences, so can be applied to the subexpressions. Continuing in this way, we get a suitable sequence of equivalences. We have

$$\Big(\forall x(x < i \to \phi(x)) \wedge \forall x(x = i \to \phi(x))\Big) \to \Big(\forall x(x < i \to \psi(x)) \wedge \forall x(x = i \to \psi(x))\Big).$$

Applying the distributive rule for the universal quantifier (8), we have

$$\forall x\Big((x < i \to \phi(x)) \wedge (x = i \to \phi(x))\Big) \to \forall x\Big((x < i \to \psi(x)) \wedge (x = i \to \psi(x))\Big).$$

which can be converted using elementary manipulations (see Section A.7), to

$$\forall x(x \leq i \to \phi(x)) \to \forall x(x \leq i \to \psi(x))$$

as required for $\Phi_i \rightarrow \Psi_i$.

We end with $\Phi_{2^n} \rightarrow \Psi_{2^n}$, i.e. $\forall x(x \leq 2^n \rightarrow \phi(x)) \rightarrow \forall x(x \leq 2^n \rightarrow \psi(x))$. The tautologous subexpression $x \leq 2^n$ can be replaced by TRUE via further basic manipulations, then after using the equivalence $(\text{TRUE} \rightarrow A) \equiv A$, we end up with $\forall x\phi(x) \rightarrow \forall x\psi(x)$. □

We also use the following extension of Lemma 3.

Lemma 4 *Suppose we have a circuit that takes as input $i \in [2^n]$ and proves $\phi(i) \wedge \psi(i) \rightarrow \xi(i)$. Then we can use it to make a circuit-generated proof of a statement of the form $\forall x\phi(x) \wedge \exists y\psi(y) \rightarrow \exists z\xi(z)$.*

Proof. We want to prove $\forall x\phi(x) \wedge \exists y\psi(y) \rightarrow \exists z\xi(z)$, equivalently $\forall x(\exists y(\phi(x) \wedge \psi(y))) \rightarrow \exists z\xi(z)$. At the beginning of Section 2.3, we noted that it's possible to prove theorems of the form $\forall x(A(x)) \rightarrow A(t)$ where t is a vector of terms that is plugged in for x in A, so we can prove the theorem

$$\forall x(\exists y(\phi(x) \wedge \psi(y))) \rightarrow \exists y(\psi(y) \wedge \phi(y)).$$

Then it's sufficient to prove

$$\exists y(\psi(y) \wedge \phi(y)) \rightarrow \exists z\xi(z).$$

Equivalently,

$$\forall z(\neg\xi(z)) \rightarrow \forall z(\neg(\psi(z) \wedge \phi(z))).$$

which can be done with a concise circuit-generated proof, using Lemma 3 and our assumption that we have a circuit that can prove, for any z, $\psi(z) \wedge \phi(z) \rightarrow \xi(z)$, which is equivalent to $\neg\xi(z) \rightarrow \neg(\psi(z) \wedge \phi(z))$. □

4 Reduction from PPP to Wrong Proof

In this section we establish the following result:

Theorem 1 *Any problem that belongs to the complexity class* PPP *(which includes* PPAD *and* PPADS*) is reducible to* Wrong Proof *(with respect to the deductive system of Sections 2.1, 2.2).*

The complexity class PPP is defined as the set of all problems reducible to the problem Pigeonhole Circuit, which is informally described as follows: suppose we are given a boolean circuit having n bits of input and output. Suppose that no input maps to the all-ones output. By the pigeonhole principle, there must be a *collision*, a pair of input vectors that map to the same output. The problem is to find a collision. Notice that this problem is in NP, since a collision is easy to verify, but finding one seems hard. We use the following definition of Pigeonhole Circuit.

Definition 3 *An instance of* Pigeonhole Circuit *consists of a circuit C having n input bits and n output bits. A solution consists of either a n-bit string that C maps to the all-zeroes string, or two n-bit strings that C maps to the same output string.*

Proof. (of Theorem 1) We reduce from Pigeonhole Circuit to Wrong Proof. Given an instance C of Pigeonhole Circuit we need to construct (in time polynomial in the size of C) a circuit-generated proof Π_C (an exponentially-long, concisely-represented formal proof containing a known contradiction) whose error(s) allow us to find solution(s) to C.

Recall that n-bit strings correspond with numbers in $[2^n]$ (2^n being the all-zeroes string). We include in Π_C a function $f : [2^n] \rightarrow [2^n]$, which we construct using our deductive system according to the first paragraph of Section 3.2. The (2^n into $2^n - 1$) pigeonhole principle assures us that

$$\exists(x,x')\Big((x \neq x' \wedge f(x) = f(x')) \vee f(x) = 2^n\Big) \tag{19}$$

Lemma 1 of Section 3.2 tells us how to generate a purported proof that (19) does not hold; the proof will be incorrect, but from error(s) in that proof we can efficiently recover satisfying assignments of $(x \neq x' \wedge f(x) = f(x')) \vee f(x) = 2^n$, which in turn identify solutions to the original Pigeonhole Circuit problem C.

So the challenge is to write down a (correct) circuit-generated proof of (19). Let T_C denote the formula of (19) (the "target" formula to be proved, for given C). The proof of (19) has a known line containing T_C, whose formal proof begins as follows.

Let $S_C = \exists x(f(x) = 2^n)$. Then using the case analysis rule (4), T_C is inferable from $S_C \to T_C$ and $\neg S_C \to T_C$. $S_C \to T_C$ is straightforward; note that it is of the form:

$$\exists x A(x) \to \exists x, y(A(x) \vee B(x,y))$$

In Section A.1 we show how to prove this using the system of Section 2.2.

So, the main challenge that remains is to generate a proof of

$$\neg S_C \to T_C. \tag{20}$$

We give the constructions of the formulae A_i, C_i, and F_i discussed in Section 3. Thus, A_i asserts a property of some instance i that implies, non-constructively, the existence of a solution. C_i is the explicit existential statement of a solution's existence. F_i is an extension axiom of the form (13), defining the construction of function f_{i-1} in terms of f_i.

For $i \in [2^n]$, $i \geq 2$, let A_i be the sentence

$$\forall x\Big(x \leq i \ \to \ f_i(x) \leq i-1\Big). \tag{21}$$

A_i states that $f_i([i]) \subseteq [i-1]$ (which implies, non-constructively, that f_i has a collision in the range $[i-1]$).

For $i \in [2^n]$, $i \geq 2$, let C_i be the sentence

$$\exists x \neq x'\Big(x \leq i \ \wedge \ x' \leq i \ \wedge \ f_i(x) = f_i(x') \ \wedge \ f_i(x) \leq i-1\Big). \tag{22}$$

C_i states *explicitly* that f_i has a collision in the range $[i-1]$, with the two colliding inputs in the range $[i]$. The pigeonhole principle tells us that C_i should follow from A_i, and we will achieve this (i.e derive C_i from A_i) using exponentially many steps of a circuit-generated proof.

We include a sequence of extension-axiom lines —of type (13)— as follows. For $i \in [2^n]$, $i \geq 2$, line $\ell(F_i)$ contains expression F_i defining function f_{i-1} in terms of f_i (see Figure 1). We also use a special line ℓ_F —also an extension-axiom line of type (13)— that sets f_{2^n} equal to f: formally, ℓ_F contains the expression $F := f_{2^n}(x) \leftrightarrow f(x)$. Section A.2 shows how to prove A_{2^n} based on F together with $\neg S_C$. For $i \in [2^n]$, $i \geq 2$, F_i defines f_{i-1} as follows.

$$f_{i-1}(x) \leftrightarrow \begin{cases} i-2 & \text{if } \ x < i \wedge f_i(i) = i-1 \wedge f_i(x) = i-1 \\ f_i(i) & \text{if } \ x < i \wedge f_i(i) < i-1 \wedge f_i(x) = i-1 \\ f_i(x) & \text{otherwise. } \ \text{(i.e. } x \geq i \vee f_i(x) < i-1) \end{cases} \tag{23}$$

F_i states that f_{i-1} is derived from f_i as follows. f_{i-1} and f_i are intended to satisfy A_{i-1} and A_i respectively, and suppose we know that f_i satisfies A_i and want to construct f_{i-1} from

f_i in such a way that f_{i-1} satisfies A_{i-1}. (23) ensures that for $x \in [i-1]$, $f_{i-1}(x) \in [i-2]$. If $x \in [i-1]$ is mapped by f_i to $i-1$, it is redirected to $i-2$ if $f_i(i) = i-1$, and if $f_i(i)$ is less than $i-1$, it is redirected to $f_i(i)$. *The construction is designed to allow us to reconstruct a collision for f_i based on an explicit statement of a collision for f_{i-1}.* For that, it does not work to just take inputs that f_i maps to $i-1$, and let f_{i-1} send them to $i-2$; the more complicated rule of (23) seems necessary. The construction is related to the one of [10], that also sets $f_{i-1}(x)$ to $f_i(i)$ whenever $f_i(x) = i-1$, but we have a different treatment of the case that $f_i(i) = i-1$, which allows us to recurse all the way down to $i = 2$.

We define a sequence of lines of Π_C as follows. For all $i \in [2^n]$, $i \geq 3$, we include lines $\ell(A_i)$ (each line number $\ell(A_i)$ and its contents are efficiently computable from i), such that $\ell(A_i)$ contains the expression:

$$\ell'(A_i), \ell''(A_i) \vdash (F_i \wedge A_i) \rightarrow A_{i-1}; \tag{24}$$

$\ell(A_i)$ states that if function f_{i-1} is constructed from f_i according to formula F_i, and f_i satisfies A_i, then f_{i-1} satisfies A_{i-1}. $\ell'(A_i)$ and $\ell''(A_i)$ contribute to a formal proof of the expression of $\ell(A_i)$; all these lines are distinct. In Section A.3 we show how to do this, hence proving $f_i([i]) \subseteq [i-1]$ for all $i \geq 2$, by backwards induction starting at $f = f_{2^n}$. Given all these lines of type (24), together with a line containing $\neg S_C \rightarrow A_{2^n}$ (proved in Section A.2), and the lines containing F_i, we can infer a sequence of lines containing $\neg S_C \rightarrow A_i$, for all $i \geq 2$.

Π_C contains a special line $\ell(C_2)$, saying that if we have A_2, then C_2 can be proved. C_2 is the "obvious" statement that f_2, which maps both $x^{(1)}$ and $x^{(2)}$ to $x^{(1)}$, has a collision. Line $\ell(C_2)$ is of the form

$$\ell'(C_2) \vdash A_2 \rightarrow C_2; \tag{25}$$

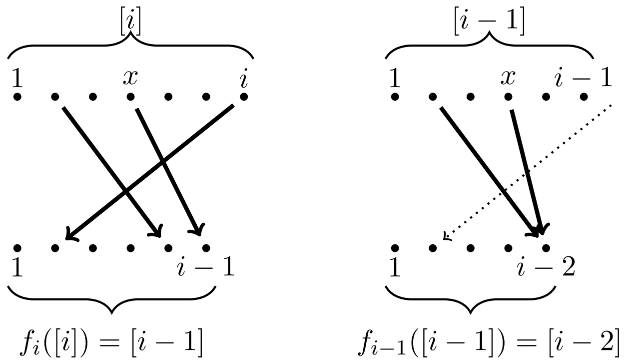
for some other special line $\ell'(C_2)$ used in a single self-contained proof of (25). $\ell(C_2)$ states that C_2 can be deduced without any further assumptions about f_2. By construction, f_2 maps both $x^{(1)}$ and $x^{(2)}$ to $x^{(1)}$, so we know where to look for a collision! In Section A.5 we show how to formally prove that f_2 has this "obvious" collision.

For $i \in [2^n]$, $i \geq 3$, we include lines $\ell(C_i)$, (again, these line numbers and the lines themselves are efficiently computable from i), where $\ell(C_i)$ contains the expression

$$\ell'(C_i), \ell''(C_i) \vdash (A_i \wedge F_i \wedge C_{i-1}) \rightarrow C_i; \tag{26}$$

$\ell(C_i)$ states that if C_{i-1} can be established, then given F_i and A_i we can deduce C_i (a collision for function f_i) where $\ell'(C_i)$ and $\ell''(C_i)$ are some further lines used in the proof of (26). In Section A.4 we give some more detail on how to construct a formal proof of (26) using our deductive system.

Putting it all together, we noted earlier that we have a sequence of lines containing $\neg S_C \rightarrow A_i$, for $i \in [2^n], i \geq 2$. We also know that C_2 follows from A_2 (25). We may use these, along with the lines $\ell(F_i)$ that give us F_i, and the lines $\ell(C_i)$ (i.e. of the form (26)) to deduce (by repeated applications of modus ponens and conjunction introduction) $\neg S_C \rightarrow C_{2^n}$; using ℓ_F we get (20) as desired. This completes the construction of a formal proof according to the strategy outlined at the end of Section 3. $\square$



"naive" choice of $f_{i-1}(x)$ for x such that $f_i(x) = i-1$, is to set $f_{i-1}(x)$ to be some fixed value in $[i-2]$ (here, $i-2$). We construct f_{i-1} as shown in examples below.

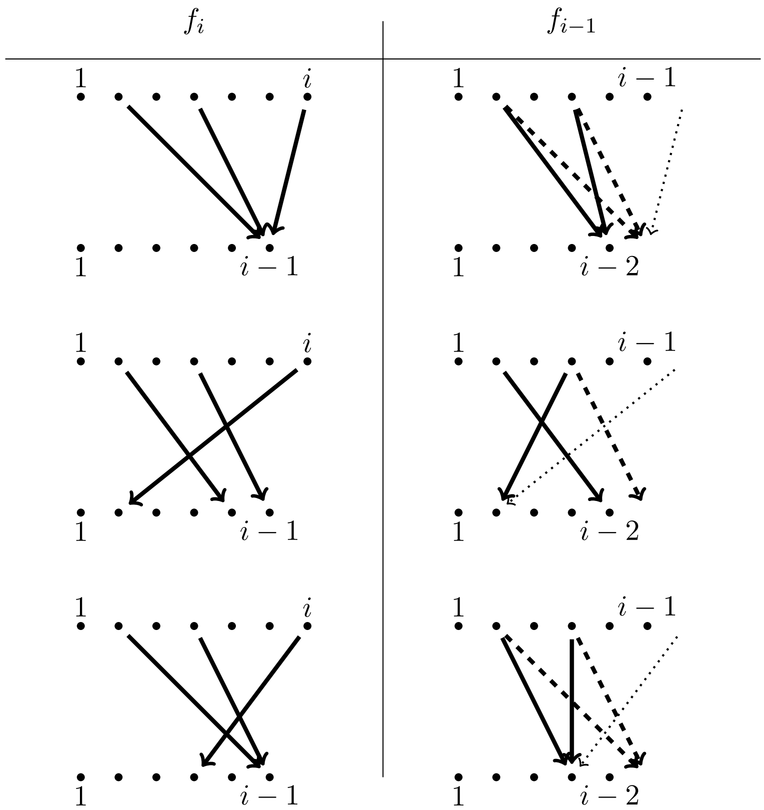


Figure 1: Construction of f_{i-1} from f_i (re proof of Theorem 1), such that from $f_i(x) < i$ for all $x \leq i$, we have $f_{i-1}(x) < i-1$ for all $x \leq i-1$. Dotted lines represent evaluations of f_{i-1} on i, and we are just interested in f_{i-1} on the domain $[i-1]$. Dashed lines are ones that have been "redirected" in construction of f_{i-1}.
The naive approach of setting f_i to some value less than i, may create collisions for f_{i-1} for which we can't reconstruct a collision for f_i based on a collision we found for f_{i-1}.

5 Reduction from PPA to Wrong Proof

In this section we establish the following theorem:

Theorem 2 *Any problem that belongs to the complexity class* PPA *is reducible to* Wrong Proof.

To prove Theorem 2, we make use of the following PPA-complete problem, due to Buss and Johnson [8]. Suppose we represent an undirected graph on the set $[2^n]$ via a function $f : [2^n] \to [2^n]$ such that an edge $\{x, x'\}$ is present iff $f(x) = x'$ and $f(x') = x$. Suppose that some given $\bar{x} \in [2^n]$ satisfies $f(\bar{x}) = \bar{x}$ (so, $\bar{x}$ is a "lonely" vertex that is unattached to any other). Since $[2^n]$ has an even number of elements, there must exist another unattached vertex. The following formula captures this *parity principle* that if we have a finite set with an even number of elements, some of which are paired off with each other, and we are shown an element that is not paired off, then there should exist another element that is not paired off.

$$f(\bar{x}) = \bar{x} \to \exists x\Big(x \neq f(f(x)) \vee (x \neq \bar{x} \wedge x = f(x)\Big) \tag{27}$$

In the following definition, we let the bit string corresponding to the number 1, which we denote $x^{(1)}$, be the special element of $[2^n]$ —having the role of $\bar{x}$— that is mapped to itself.

Definition 4 *The problem* Lonely *is defined as follows. Given a function $f : [2^n] \to [2^n]$ presented as a boolean circuit C having n inputs and n outputs, find $x \neq x^{(1)}$ such that either (a) $f(x^{(1)}) \neq x^{(1)}$, or (b) $f(x) = x$, or (c) $x \neq f(f(x))$.*

It can be shown that this problem is PPA-complete by reduction from Leaf (the original PPA-complete problem of [18]); using Lonely simplifies the reduction to Wrong Proof.

Proof. (of Theorem 2) We reduce from Lonely to Wrong Proof. Let C be the circuit in an instance of Lonely. We may assume C is syntactically constrained so that its function f satisfies $f(x^{(1)}) = x^{(1)}$ and for all x, $f(f(x)) = x$: the problem of finding $x \neq x^{(1)}$ with $f(x) = x$ remains PPA-complete.[5]

Given C, the circuit representing an instance of Lonely, we construct a Wrong Proof instance Π_C as follows. We start by including in Π_C a construction of the function f computed by C as described at the start of Section 3.2.

Equation (28) is analogous to equation (19) in Theorem 1: it's a formula involving a function f that's derived easily from C, being plugged in to a combinatorial principle (here, the PPA principle) stating that some solution exists. Recall that Lemma 1 explained how to construct a "proof" of the negation of (28), in such a way that any error in the proof lets us reconstruct (x, x') that satisfy it. Π_C contains that proof.

$$\exists(x, x')(f(x^{(1)}) \neq x^{(1)} \vee (x \neq x^{(1)} \wedge f(x) = x) \vee (x' = f(x) \wedge x \neq f(x')). \tag{28}$$

Using our assumption that C has been syntactically constrained as described above, Π_C also proves (29), from which (28) (unnegated) is derivable:[6]

$$\exists(x, x')(x \neq x^{(1)} \wedge f(x) = x) \tag{29}$$

We introduce a sequence of functions f_i, for *even* numbers i in the range $2 \leq i \leq 2^n$, where $f_{2^n} = f$, constructed as follows. f_i represents an instance of Lonely on the domain $[i]$. f_{i-2} is derived from f_i as follows (see Figure 2 for an illustration):

[5] We can take an unrestricted circuit C and modify it (without excessive increase in size) such that these conditions are met, and a solution for the modified circuit (call it C') yields a solution to C. C' should map $x^{(1)}$ to itself, and for any $x \neq x^{(1)}$, it should check whether $f(f(x)) \neq x$, if so, map x to itself, rather than to $f(x)$.

[6] An alternative approach would be to note that the technique of Lemma 1 applies directly to (29).

1. If f_i maps i and $i-1$ to i and $i-1$, then set $f_{i-2}(x) = f_i(x)$ for all x.
2. If $f_i(i) = i$ and $f_i(i-1) = y < i-1$ then set $f_{i-2}(y) = y$; for other elements x of $[i-2]$, $f_{i-2}(x) = f_i(x)$.
3. If $f_i(i-1) = i-1$ and $f_i(i) = y < i-1$ then set $f_{i-2}(y) = y$; for other elements x of $[i-2]$, $f_{i-2}(x) = f_i(x)$.
4. If $f_i(i) = y < i-1$ and $f_i(i-1) = y' < i-1, y' \neq y$, then set $f_{i-2}(y) = y'$ and $f_{i-2}(y') = y$ and for $x \neq y, y'$, set $f_{i-2}(x) = f_i(x)$.

We do not have to consider a case where $f_i(i) = f_i(i-1)$: it does not arise due to our assumption that $f(f(x)) = x$ for all x.

For even numbers $i \in [2^n]$, let A_i be the sentence

$$(f_i(x^{(1)}) = x^{(1)}) \wedge \forall x, x' \leq i\Big(f_i(x) \leq i \wedge (f_i(x) = x' \to f_i(x') = x)\Big) \tag{30}$$

A_i states that f_i is a valid instance of Lonely on domain $[i]$. Analogously with (21), this is an implicit, or non-constructive statement that f_i has a fixpoint in $\{2, \ldots, i\}$.

Similar to (23), for even numbers $i < 2^n$ we include an extension-axiom line (of type (13)) with line number $\ell(F_i)$ containing F_i given as in (31). F_i defines f_{i-2} in terms of f_i. As in Theorem 1 we also use a special line ℓ_F —also an extension-axiom line of type (13)— that sets f_{2^n} equal to f: formally, ℓ_F contains the expression $F := f_{2^n}(x) \leftrightarrow f(x)$.

$$f_{i-2}(x) \leftrightarrow \begin{cases} f_i(i) & \text{if } f_i(i-1) = i-1 \wedge f_i(i) < i-1 \wedge x = f_i(i) \\ f_i(i-1) & \text{if } f_i(i) = i \wedge f_i(i-1) < i-1 \wedge x = f_i(i-1) \\ f_i(i) & \text{if } f_i(i) < i-1 \wedge f_i(i-1) < i-1 \wedge x = f_i(i-1) \\ f_i(i-1) & \text{if } f_i(i) < i-1 \wedge f_i(i-1) < i-1 \wedge x = f_i(i) \\ f_i(x) & \text{otherwise. (i.e. } x \neq f_i(i), f_i(i-1).) \end{cases} \tag{31}$$

Similar to (22), for even numbers $i \in [2^n]$, let C_i be the sentence

$$\exists x \leq i\Big(x \neq x^{(1)} \wedge f_i(x) = x\Big) \tag{32}$$

C_i states that the Lonely instance associated with f_i restricted to domain $[i]$ has a solution. It remains for us to construct a circuit-generated formal proof of C_{2^n}.

We proceed in a similar way as previously, omitting details of the application of the inference rules. f is constructed so as to satisfy A_{2^n}, and using our proof system, it can be shown that

1. A_{2^n} holds. A_{2^n} is a universally quantified sentence that has a circuit-generated proof using the technique of Lemma 1, that checks all possible values of the quantified variables in A_{2^n}. By our assumption that C has been modified so that $f(x^{(1)}) = x^{(1)}$ and $f(f(x)) = x$ for all x, this proof will be correct.
2. for $4 \leq i \leq 2^n$, A_{i-2} is derivable from A_i and F_i, i.e. $A_i \wedge F_i \to A_{i-2}$.
3. C_2 follows from A_2, i.e. $A_2 \to C_2$,
4. for $4 \leq i \leq 2^n$, C_i is derivable from F_i and C_{i-2}, i.e. $F_i \wedge C_{i-2} \to C_i$. (We don't seem to need A_i here, in contrast with Theorem 1, where we proved $A_i \wedge F_i \wedge C_{i-1} \to C_i$.)

Finally, (29) is the same as C_{2^n}.

We omit the details of item (2).

To prove item (3), note that A_2 is the expression

$$(f_2(x^{(1)}) = x^{(1)}) \wedge \forall x, x' \leq x^{(2)}(f_2(x) \leq x^{(2)} \wedge (f_2(x) = x' \to f_2(x') = x))$$

C_2 is the expression

$$\exists x \leq x^{(2)}(x \neq x^{(1)} \wedge f_2(x) = x)$$

A_2 is equivalent to a version where the quantifier appears at the front; then as noted in Section 2.3, we can prove the following theorem, saying that A_2 implies a version where $x^{(2)}$ and $x^{(1)}$ have been plugged in for x and x':

$$A_2 \to (f_2(x^{(1)}) = x^{(1)}) \wedge f_2(x^{(2)}) \leq x^{(2)} \wedge (f_2(x^{(2)}) = x^{(1)} \to f_2(x^{(1)}) = x^{(2)})$$

Letting R denote the right-hand side of this, we can separately prove $R \to x^{(2)} \neq x^{(1)} \wedge f_2(x^{(2)}) = x^{(2)}$. Using the rules in Section 2.3, we can write $x^{(2)} \neq x^{(1)} \wedge f_2(x^{(2)}) = x^{(2)} \to C_2$. Finally, we can deduce $A_2 \to C_2$ by a sequence of applications of the hypothetical syllogism rule.

The proof of item (4) above proceeds by case analysis (4) on values of $f_i(i)$ and $f_i(i-1)$. We give some details on two of the cases. The expression $F_i \wedge C_{i-2} \to C_i$ that we aim to prove, can be written as (renaming bound variables):

$$F_i \wedge \exists y \leq i-2(y \neq x^{(1)} \wedge f_{i-2}(y) = y) \to \exists z \leq i(z \neq x^{(1)} \wedge f_i(z) = z)$$

In the case that $f_i(i-1) = i$ and $f_i(i) = i-1$, or indeed where $f_i(i-1) = i-1$ and $f_i(i) = i$, F_i simplifies to $\forall x(f_{i-2}(x) = f_i(x))$, and so it suffices to prove

$$\forall x(f_{i-2}(x) = f_i(x)) \wedge \exists y \leq i-2(y \neq x^{(1)} \wedge f_{i-2}(y) = y) \to \exists z \leq i(z \neq x^{(1)} \wedge f_i(z) = z),$$

which can be proved using Lemma 4.

Consider the case that $f_i(i) = i$ and $f_i(i-1) = y < i-1$. From this and F_i it follows that

$$\forall x \leq i \begin{cases} f_i(x) \neq f_i(i-1) \to f_{i-2}(x) = f_i(x) \\ f_{i-2}(i) = f_i(i) \\ f_i(x) = f_i(i-1) \to f_{i-2}(x) = f_i(x) \end{cases} \tag{33}$$

We want to prove

$$(33) \wedge \exists y \leq i-2(y \neq x^{(1)} \wedge f_{i-2}(y) = y) \to \exists z \leq i(z \neq x^{(1)} \wedge f_i(z) = z)$$

and the right-hand side follows by putting $z = i$.

In the case that $f_i(i) = y < i-1$, $f_i(y) = i$, $f_i(i-1) = y' < i-1$, $f_i(y') = i-1$, the element of $[i-2]$ that is said to exist by C_{i-2} is the one that we use to satisfy the matrix of C_i. □

6 Reduction from PLS to Wrong Proof

In this section we establish the following theorem:

Theorem 3 *Any problem that belongs to the complexity class* PLS *is reducible to* Wrong Proof*.*

To prove this, we make use of the following PLS-complete problem, due to Buss and Johnson [8]. Equation (34) captures the *iteration principle*, that if $f : \{0, \ldots, N\} \to \{0, \ldots, N\}$ maps 0 to a positive number, and any number i to a number at least as large as i, then there exists x such that $f(x) > x$ and $f(f(x)) = f(x)$. Notice that such an x can be found by

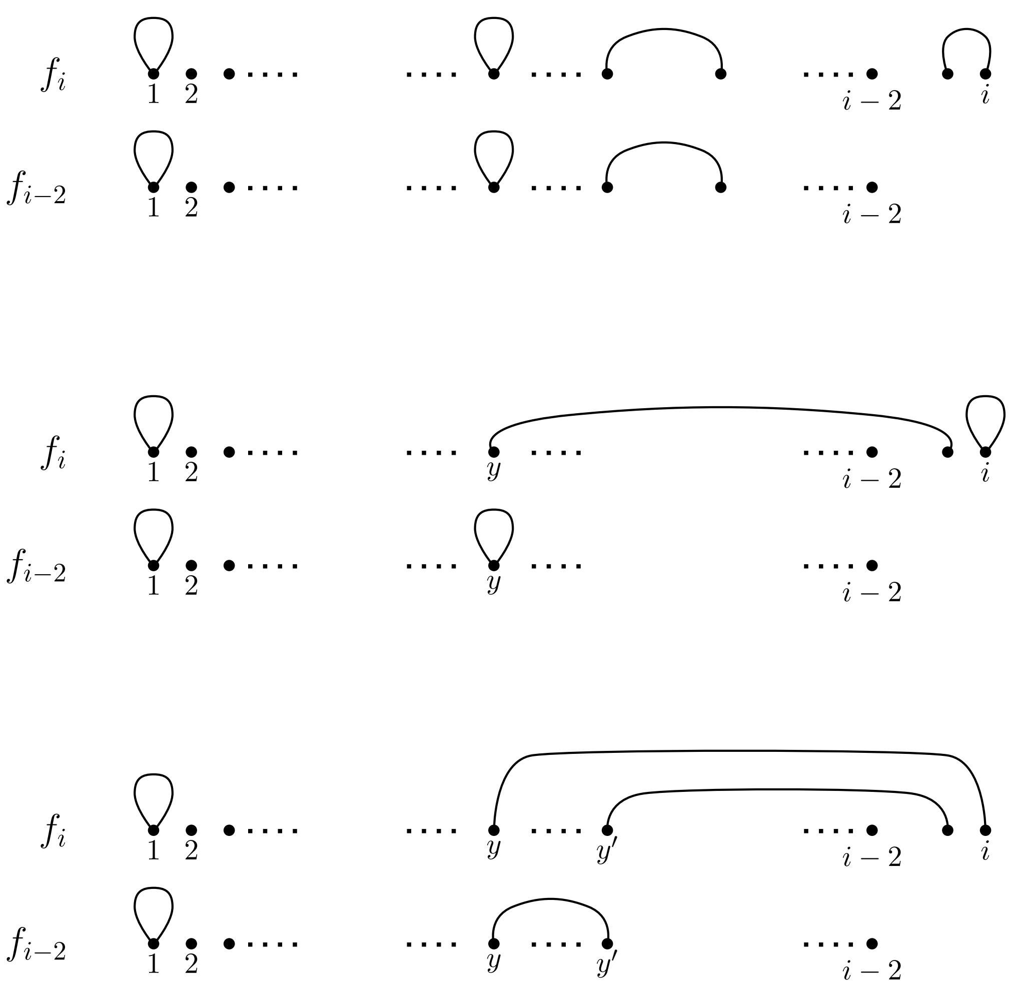


Figure 2: Construction of f_{i-2} from f_i (re proof of Theorem 2).

following the sequence $0, f(0), f(f(0)), \ldots$ and taking the number that occurs just before the fixpoint of f.

$$0 < f(0) \land \forall x(x \leq f(x)) \to \exists x(x < f(x) \land f(f(x)) = f(x)) \tag{34}$$

In our context we apply the principle to the numbers in $[2^n]$ as before, so our definition uses 1 as the smallest number rather than 0, and recall $x^{(1)}$ is the bit-string representing 1.

Definition 5 *The problem* ITER *is defined as follows. Given a function $f : [2^n] \to [2^n]$ presented as a boolean circuit having n inputs and n outputs, find x such that either (a) $f(x^{(1)}) = x^{(1)}$, or (b) $f(x) < x$, or (c) $x < f(x)$ and $f(f(x)) = f(x)$.*

Proof. (of Theorem 3) Given the circuit C that defines an instance of ITER, we construct an instance Π_C of WRONG PROOF as follows. As before, Π_C constructs the function f computed by C as described at the start of Section 3.2. ((35) corresponds to (19) in Theorem 1). C has corresponding formula (35) that's satisfiable by some pair (x, x') due to the iteration principle.

$$\exists(x, x')\Big(f(x^{(1)}) = x^{(1)} \lor f(x) < x \lor (x' = f(x) \land f(x') = f(x)\Big) \tag{35}$$

As in the two previous theorems, Π_C contains a proof of the negation of (35) constructed according to Lemma 1 of Section 3.2. It remains to devise a correct (circuit-generated) proof that (35) holds, which can be incorporated into Π_C.

We introduce functions $f_i : [i] \to [i]$ for $i \in [2^n]$, $i \geq 2$, and set $f_{2^n} = f$. f_{i-1} is derived from f_i according to (36); it can be seen that f_i is like f but with a ceiling of i imposed on the value it can take, i.e. $f_i(x) = \min\{i, f(x)\}$. Let F_i be the extension-axiom expression that defines f_{i-1} in terms of f_i, thus taking any number that maps to i, and mapping it to $i - 1$ instead:

$$f_{i-1}(x) \leftrightarrow \begin{cases} i-1 & \text{if } f_i(x) = i \\ f_i(x) & \text{otherwise. (i.e. } x > i \lor f_i(x) < i) \end{cases} \tag{36}$$

For $i \in [2^n]$, $i \geq 2$, let A_i be the sentence

$$(f_i(x^{(1)}) > x^{(1)}) \land \forall x \leq i(f_i(x) \geq x \land f_i(x) \leq i)$$

A_i states that f_i obeys the iteration principle for the domain and codomain $[i]$ (hence some number $x \leq i$ should be a fixpoint of f_i). As before, this statement of existence of such a fixpoint in implicit, not explicit.

For $i \in [2^n]$, $i \geq 2$, let C_i be the sentence

$$\exists x, x' \leq i\Big(f_i(x) > x \land x' = f_i(x) \land f_i(x') = f_i(x)\Big)$$

C_i states *explicitly* that f_i has a fixpoint in $[i]$.

We proceed in a similar way to Theorems 1, 2, omitting details of the sequence of steps of the formal proof. Using our deductive system, it can be shown that

1. for $i \in [2^n]$, $i \geq 2$, $(A_i \land F_i) \to A_{i-1}$,
2. $A_2 \to C_2$,
3. for $i \in [2^n]$, $i \geq 2$, $(A_i \land F_i \land C_{i-1}) \to C_i$.

To prove item (1), we use Lemma 3. $(A_i \land F_i)$ is equivalent to (using the distributive rule for the universal quantifier (8)):

$$\begin{aligned} \forall x \leq i \quad & (f_i(x^{(1)}) = x^{(1)} \land f_i(x) \geq x \land f_i(x) \leq i \land \\ & (f_i(x) < i \to f_{i-1}(x) = f_i(x)) \;\land\; (f_i(x) = i \to f_{i-1}(x) = i-1)) \end{aligned}$$

A_{i-1} is equivalent to $\forall x(f_{i-1}(x^{(1)}) > x^{(1)} \wedge x \leq i-1 \rightarrow (f_{i-1}(x) \geq x \wedge f_{i-1}(x) \leq i-1))$. For any value of x, the matrix of this is efficiently derivable from the matrix of the expression for $(A_i \wedge F_i)$, so Lemma 3 can be applied.

To prove item (2), the expression $A_2 \rightarrow C_2$, we have

$$C_2 = \exists x < x^{(2)}, x' \leq x^{(2)} \Big(f_1(x) > x \wedge x' = f_1(x) \wedge f_1(x') = f_1(x) \Big)$$

$$A_2 = f_2(x^{(1)}) > x^{(1)} \wedge \forall x \leq x^{(2)} (f_2(x) \geq x \wedge f_2(x) \leq x^{(2)})$$

The proof of $A_2 \rightarrow C_2$ is similar to the one for the corresponding expression in Theorem 2, and we omit the details.

For item (3) above, C_{i-1} identifies $x < i-1$ that f_{i-1} maps to a fixpoint x' of f_{i-1}. To identify a solution for f_i we proceed by case analysis, rule (4). If $x' < i-1$ then (based on the way F_i constructs f_{i-1} from f_i) we can deduce that x must be a solution of f_i (in that $f_i(x) = x'$ and $f_i(x') = x'$). If $x' = i-1$ then we proceed by case analysis according to whether $f_i(x) = i-1$ (in which case x is a solution of f_i), and the alternative is $f_i(x) = i$, in which case, since we know from A_i that $f_i(i) = i$, x continues to be a solution for f_i. $\square$

7 Finitary Existential Sentences and TFNP

To end on a different note, let us look back at the five classes: All five correspond to elementary combinatorial existence arguments (such as "every dag has a sink", recall the five bullets in the Introduction). Importantly, all five combinatorial existence arguments yielding complexity classes are *finitary:* They are true of finite structures and not true of all infinite structures. *Is this a coincidence?* Can there be an interesting complexity subclass of TFNP defined in terms of an existence argument that is not finitary, but is true of all structures, finite *and* infinite?

Seen as sentences in logic, these combinatorial arguments are statements of the form "for all finite structures (such as topologically sorted dags) there exists an element (a node) that satisfies a property (has no outgoing edges)." The corresponding logical expression is a sentence $\exists \bar{x} \Phi(\bar{x})$ in predicate logic, involving a set of existentially quantified variables $\bar{x}$ and an expression Φ with any number of other variables, as well as function symbols capturing structures such as undirected and directed graphs, pigeonhole functions, or total orders and potential functions. The "for all finite structures" quantification is implicit in the requirement that the sentence $\exists \bar{x} \Phi(\bar{x})$ be *valid on finite structures.*

And conversely, it is easy to see any such sentence yields a problem FIND WITNESS$_\Phi$ in TFNP (and consequently a complexity class, through reductions). FIND WITNESS$_\Phi$ is defined as follows: "Given a finite structure for Φ, where the finite universe can be assumed to be an initial segment of the nonnegative integers and the structures are presented implicitly through circuits computing the functions of Φ on elements of the universe encoded in binary, find a tuple $\hat{x}$ of integers that satisfy Φ."

We can now formulate the question in the section's opening paragraph in logic terms: All five sentences Φ corresponding to the five known complexity subclasses of TFNP are of course true in any finite model, but all of them happen to be false for some infinite models (for example, "every dag has a sink" fails for the totally ordered integers). Is this necessary? *Can there be an interesting subclass of TFNP based on a* valid sentence $\exists \bar{x} \Phi(\bar{x})$*, that is, one that is true of* all *models, finite or infinite?*

Employing an ancient theorem in Logic due to Jacques Herbrand [13] (1930) one can show that the answer is negative:

Theorem 4 *For any valid sentence in predicate logic of the form $\exists \bar{x} \Phi(\bar{x})$, the corresponding problem* Find Witness$_\Phi$ *can be solved in polynomial time.*

Sketch: Herbrand's theorem [13] states that any valid sentence

$$\exists x_1 \cdots \exists x_k \Phi(x_1, \ldots, x_k)$$

is equivalent to a finite disjunction of the form

$$\bigvee_{i=1}^{K} \Phi(t_{i1}, \ldots, t_{ik}),$$

where the t_{ij}'s are terms involving the function symbols and constants of Φ, for some fixed K depending on Φ. Solving Find Witness$_\Phi$ entails evaluating each of these K logical formulae of fixed size to identify the combination of terms, and thus ultimately elements of the universe computed in linear time (with respect to the length of the input) through the circuits of the input, that indeed satisfy Φ.

□

8 Discussion

We have defined PTFNP, a subclass of the total function problems with NP verification of witnesses, which we see as a "syntactic" (in the sense of having complete problems) approximation of TFNP. We showed that PTFNP contains the five known classes PPP, PPA, PPAD, PPADS, and PLS.

The question remains, is Factoring in PTFNP? It would seem that, in order to prove that it is, one needs a propositional proof of the correctness of [1], which seems very tricky. But there is an easier alternative: A propositional proof of any sort of concise primality certificates would suffice. Still, at present we see concrete obstacles to proceeding in this direction. Alternatively, what other more powerful class would include Factoring and the other "rogue problems"? How should the deductive system be strengthened? The work by Beckmann and Buss mentioned in the introduction may provide an answer. A related question is whether the system can be *weakened* while still generalising PPP and the related classes. In particular, whether we really need to generate a long sequence of functions via extended Frege-style lines (13). (It is tempting to try to define a function via its bit graph: instead of defining f, could we just introduce boolean variables $f(x)$ for each $x \in [2^n]$, and have separate (standard extended Frege) lines for each of them? But this doesn't work on its own; for more discussion see Section A.8.)

Acknowledgments: Many thanks to the "PPAD-like classes reading group" at the Simons Institute during the Fall 2015 program on Economics and Computation for many fascinating interactions, and to Sam Buss and Pavel Pudlák for an inspiring lunch conversation in November 2015. We also thank Arnold Beckmann and Sam Buss for helpful comments on earlier versions of this paper. Thanks also to the organisers of the 2015 "Algorithmic Perspective in Economics and Physics" research program at the Centre de Recerca Matemàtica (CRM), Barcelona, where this research was initiated.

A Some formal proofs and expressions

A.1 Proof of $S_C \to T_C$ (from proof of Theorem 1)

We noted that $S_C \to T_C$ is of the form:

$$\exists x A(x) \to \exists x, y (A(x) \vee B(x, y))$$

which is proved as follows. Using the distributive rule for quantifiers (8) this is equivalent to

$$\exists x A(x) \to \exists x A(x) \vee \exists x, y B(x, y). \tag{37}$$

Rule (1) of our proof system lets us write down lines containing $\vdash A \to A$ for any well-formed formula A, so we can write

$$\neg \exists x A(x) \to \neg \exists x A(x)$$

The antecedent strengthening rule (6) lets us deduce

$$(\neg \exists x A(x) \wedge \neg \exists x, y B(x, y)) \to \neg \exists x A(x)$$

Applying the rule of replacement $A \to B \equiv \neg B \to \neg A$, we have

$$\exists x A(x) \to \neg(\neg \exists x A(x) \wedge \neg \exists x, y B(x, y))$$

which (applying $\neg(A \wedge B) \equiv \neg A \vee \neg B$, and removal of double negation) is equivalent to (37).

A.2 Proof of $\neg S_C \to A_{2^n}$ (from proof of Theorem 1)

We want to prove $\neg S_C \to A_{2^n}$, and noting that $\neg S_C$ is equivalent to $\forall x(f(x) \neq 2^n)$, this is:

$$\forall x(f(x) \neq 2^n) \to \forall x(x \leq 2^n \to f_{2^n}(x) \leq 2^n - 1).$$

Let F be the expression $\forall x(f_{2^n}(x) = f(x))$, which we have as an extension axiom. So we would like to prove

$$F \to (\neg S_C \to A_{2^n})$$

which using modus ponens in conjunction with F would yield the desired result $\neg S_C \to A_{2^n}$. Equivalently, aim to prove $(F \wedge \neg S_C) \to A_{2^n}$, i.e.

$$(\forall x(f_{2^n}(x) = f(x)) \wedge \forall x(f(x) \neq 2^n)) \to A_{2^n}$$

i.e. by (8)

$$\forall x(f_{2^n}(x) = f(x) \wedge f(x) \neq 2^n) \to A_{2^n}$$

where A_{2^n} is $\forall x(x \leq 2^n \to f_{2^n}(x) \leq 2^n - 1)$.

By Lemma 3 it suffices to show that we can construct a polynomial-size circuit that takes $i \in [2^n]$ as input, and outputs a proof of

$$(x = i) \to \Big((f_{2^n}(x) = f(x) \wedge f(x) \neq 2^n)) \to (x \leq 2^n \to f_{2^n}(x) \leq 2^n - 1)\Big)$$

The tight-hand side of this expression (i.e., omitting the initial "$(x = i) \to$", can be seen to be a tautology over the $3n$ propositional variables x, $f(x)$, and $f_{2^n}(x)$, and can be proved to be equivalent to TRUE.

A.3 Proof of lines (24) from Theorem 1: $F_i \wedge A_i \to A_{i-1}$

We show how to formally prove $(F_i \wedge A_i) \to A_{i-1}$. Writing out this expression in full (we use line breaks and indentation indicate the priority of connectives in the expression, so the left-hand "$\to$" symbol has lowest priority), we have (38), where F_i appears in the first three lines of (38), and A_i and A_{i-1} appear in the fourth and fifth lines respectively.

$$\begin{array}{rl} & \forall x((x < i \wedge f_i(i) = i - 1 \wedge f_i(x) = i - 1) \to f_{i-1}(x) = i - 2) \\ \wedge & \forall x((x < i \wedge f_i(i) < i - 1 \wedge f_i(x) = i - 1) \to f_{i-1}(x) = f_i(i)) \\ \wedge & \forall x((x \geq i \vee f_i(x) < i - 1) \to f_{i-1}(x) = f_i(x)) \\ \wedge & \forall x(x \leq i \to f_i(x) \leq i - 1) \\ \to & \forall x(x \leq i - 1 \to f_{i-1}(x) \leq i - 2) \end{array} \tag{38}$$

We show that (38) is derivable via the proof system of Section 2.2. Applying the case analysis rule (4) with $B = f_i(i) = i-1$, (38) is derivable from the following two statements (the contents of lines $\ell'(A_i)$ and $\ell''(A_i)$ that are referred-to in (24) and discussed below (24)):

$$f_i(i) = i-1 \rightarrow (38) \tag{39}$$

$$f_i(i) \neq i-1 \rightarrow (38) \tag{40}$$

We omit the proof of (40), which is similar to the proof of (39); we focus on the details of the proof of (39).

Using the identity $A \rightarrow (B \rightarrow C) \equiv (A \wedge B) \rightarrow C$, (39) is equivalent to:

$$\begin{array}{rl} & f_i(i) = i-1 \\ \wedge & \forall x((x < i \wedge f_i(i) = i-1 \wedge f_i(x) = i-1) \rightarrow f_{i-1}(x) = i-2) \\ \wedge & \forall x((x < i \wedge f_i(i) < i-1 \wedge f_i(x) = i-1) \rightarrow f_{i-1}(x) = f_i(i)) \\ \wedge & \forall x(x \geq i \vee f_i(x) < i-1) \rightarrow f_{i-1}(x) = f_i(x)) \\ \wedge & \forall x(x \leq i \rightarrow f_i(x) \leq i-1) \\ \rightarrow & \quad \forall x(x \leq i-1 \rightarrow f_{i-1}(x) \leq i-2) \end{array} \tag{41}$$

Using the antecedent strengthening rule (6), (41) is inferable from the following expression, in which the third line of (41) has been omitted:

$$\begin{array}{rl} & f_i(i) = i-1 \\ \wedge & \forall x((x < i \wedge f_i(i) = i-1 \wedge f_i(x) = i-1) \rightarrow f_{i-1}(x) = i-2) \\ \wedge & \forall x(x \geq i \vee f_i(x) < i-1) \rightarrow f_{i-1}(x) = f_i(x)) \\ \wedge & \forall x(x \leq i \rightarrow f_i(x) \leq i-1) \\ \rightarrow & \quad \forall x(x \leq i-1 \rightarrow f_{i-1}(x) \leq i-2) \end{array} \tag{42}$$

By rule (9) (bringing a quantifier to the front) and simple manipulations, this is equivalent to the following expression in which the second line omits the subexpressions $f_i(i) = i-1$:

$$\begin{array}{rl} & f_i(i) = i-1 \\ \wedge & \forall x((x < i \wedge f_i(x) = i-1) \rightarrow f_{i-1}(x) = i-2) \\ \wedge & \forall x((x \geq i \vee f_i(x) < i-1) \rightarrow f_{i-1}(x) = f_i(x)) \\ \wedge & \forall x(x \leq i \rightarrow f_i(x) \leq i-1) \\ \rightarrow & \quad \forall x(x \leq i-1 \rightarrow f_{i-1}(x) \leq i-2) \end{array} \tag{43}$$

Equivalently (splitting the third line into two):

$$\begin{array}{rl} & f_i(i) = i-1 \\ \wedge & \forall x((x < i \wedge f_i(x) = i-1) \rightarrow f_{i-1}(x) = i-2) \\ \wedge & \forall x(x \geq i \rightarrow f_{i-1}(x) = f_i(x)) \\ \wedge & \forall x(f_i(x) < i-1 \rightarrow f_{i-1}(x) = f_i(x)) \\ \wedge & \forall x(x \leq i \rightarrow f_i(x) \leq i-1) \\ \rightarrow & \quad \forall x(x \leq i-1 \rightarrow f_{i-1}(x) \leq i-2) \end{array} \tag{44}$$

The above is derivable from the following (obtained by dropping the first and third lines, i.e. strengthening the antecedent):

$$\begin{array}{rl} & \forall x((x < i \wedge f_i(x) = i-1) \rightarrow f_{i-1}(x) = i-2) \\ \wedge & \forall x(f_i(x) < i-1 \rightarrow f_{i-1}(x) = f_i(x)) \\ \wedge & \forall x(x \leq i \rightarrow f_i(x) \leq i-1) \\ \rightarrow & \quad \forall x(x \leq i-1 \rightarrow f_{i-1}(x) \leq i-2) \end{array} \tag{45}$$

By the distributive rule for quantifiers (8) this is equivalent to:

$$\begin{array}{ll} \forall x & [(x < i \land f_i(x) = i-1) \to f_{i-1}(x) = i-2) \\ & \land\ (f_i(x) < i-1 \to f_{i-1}(x) = f_i(x)) \\ & \land\ (x \leq i \to f_i(x) \leq i-1)] \\ \to & \forall x(x \leq i-1 \to f_{i-1}(x) \leq i-2) \end{array} \tag{46}$$

Lemma 3 implies that the above follows if we prove for all j (restating Lemma 3 with j instead of i, to avoid a clash with the i being used in the current context):

$$\begin{array}{rl} (x = j) \to & [(x < i \land f_i(x) = i-1) \to f_{i-1}(x) = i-2) \\ & \land\ (f_i(x) < i-1 \to f_{i-1}(x) = f_i(x)) \\ & \land\ (x \leq i \to f_i(x) \leq i-1)] \\ \to & (x \leq i-1 \to f_{i-1}(x) \leq i-2) \end{array} \tag{47}$$

It can be checked that the right-hand side of (47), i.e. omitting the "$(x = j) \to$", is (for all i) a tautology over the vectors of propositional variables x, $f_i(x)$, and $f_{i-1}(x)$. This can be proved by a sequence of basic manipulations, but it's convenient to apply Lemma 2. This is done as follows. Suppose we take the right-hand side of (47), replace $f_i(x)$ and $f_{i-1}(x)$ with vectors of new variables y and z respectively, so we get the expression

$$\begin{array}{ll} & [(x < i \land y = i-1) \to z = i-2) \\ \land & (y < i-1 \to z = y) \\ \land & (x \leq i \to y \leq i-1)] \\ \to & (x \leq i-1 \to z \leq i-2) \end{array} \tag{48}$$

Given that this is a tautology over x, y, z, using Lemma 2 we write down a circuit-generated proof of a version of (48) that is preceded with $\forall x, y, z$. Then we can apply the universal instantiation rule (10) to replace y and z with $f_i(x)$ and $f_{i-1}(x)$.

A.4 Proof of lines (26) from Theorem 1: $A_i \land F_i \land C_{i-1} \to C_i$

Lines of type (26) contain formulae of the form $A_i \land F_i \land C_{i-1} \to C_i$, and (49) is such a line when written out in full. We give an overview of how to formally prove (49) without going into quite as much detail of individual formal steps as we did in Section A.3.

By way of some intuition, the first line of (49) contains A_i, saying that f_i maps elements of $[i]$ to elements of $[i-1]$. Implicit in that is the bottom line of (49), that states explicitly that two elements z and z' of $[i]$ are mapped to the same element of $[i-1]$. F_i, which defines how f_{i-1} is derived from f_i, appears in the second, third, and fourth lines of (49). C_{i-1} is given in the fifth line, and is an explicit statement of the collision for f_{i-1}: y and y' denote the colliding elements. Since we now have, in y and y', two identifiers or handles, for the colliding elements for f_{i-1}, it becomes possible to express z and z' in terms of y and y', in such a way that they provably satisfy the bottom line.

$$\begin{array}{ll} & \forall x \leq i(f_i(x) \leq i-1) \\ \land & \forall x((x < i \land f_i(i) = i-1 \land f_i(x) = i-1) \to f_{i-1}(x) = i-2) \\ \land & \forall x((x < i \land f_i(i) < i-1 \land f_i(x) = i-1) \to f_{i-1}(x) = f_i(i)) \\ \land & \forall x((x \geq i \lor f_i(x) < i-1) \to f_{i-1}(x) = f_i(x)) \\ \land & \exists y \neq y'(y \leq i-1 \land y' \leq i-1 \land f_{i-1}(y) = f_{i-1}(y') \land f_{i-1}(y) \leq i-2) \\ \to & \exists z \neq z'(z \leq i \land z' \leq i \land f_i(z) = f_i(z') \land f_i(z) \leq i-1) \end{array} \tag{49}$$

We begin with a slight simplification, motivated by the observation that the colliding elements z and z' that we are looking for, are supposed to occur in the range $[i]$. We can remove the subexpression "$x \geq i \vee$" from the fourth line to obtain the expression (50), which is stronger than (49) in the sense that (49) can be seen to be formally derivable from (50).

$$
\begin{array}{ll}
 & \forall x \leq i(f_i(x) \leq i-1) \\
\wedge & \forall x((x < i \wedge f_i(i) = i-1 \wedge f_i(x) = i-1) \rightarrow f_{i-1}(x) = i-2) \\
\wedge & \forall x((x < i \wedge f_i(i) < i-1 \wedge f_i(x) = i-1) \rightarrow f_{i-1}(x) = f_i(i)) \\
\wedge & \forall x((f_i(x) < i-1) \rightarrow f_{i-1}(x) = f_i(x)) \\
\wedge & \exists y \neq y'(y \leq i-1 \wedge y' \leq i-1 \wedge f_{i-1}(y) = f_{i-1}(y') \wedge f_{i-1}(y) \leq i-2) \\
\rightarrow & \exists z \neq z'(z \leq i \wedge z' \leq i \wedge f_i(z) = f_i(z') \wedge f_i(z) \leq i-1)
\end{array}
\tag{50}
$$

We proceed by case analysis (4) according to whether or not we have $f_i(i) = i-1$. Thus we want to prove $f_i(i) = i-1 \rightarrow (50)$ and $f_i(i) < i-1 \rightarrow (50)$. We don't need to consider the case $f_i(i) > i-1 \rightarrow (50)$ since that case is ruled out by the first line of (50) that contains A_i (i.e. $\forall x \leq i(f_i(x) \leq i-1)$).

$f_i(i) = i-1 \rightarrow (50)$ is equivalent to (after simplifying by removing the third line of (50), that assumes $f_i(i) < i-1$):

$$
\begin{array}{ll}
 & \forall x \leq i(f_i(x) \leq i-1) \\
\wedge & f_i(i) = i-1 \\
\wedge & \forall x \leq i-1(f_i(x) = i-1 \rightarrow f_{i-1}(x) = i-2) \\
\wedge & \forall x((f_i(x) < i-1) \rightarrow f_{i-1}(x) = f_i(x)) \\
\wedge & \exists y \neq y'(y \leq i-1 \wedge y' \leq i-1 \wedge f_{i-1}(y) = f_{i-1}(y') \wedge f_{i-1}(y) \leq i-2) \\
\rightarrow & \exists z \neq z'(z \leq i \wedge z' \leq i \wedge f_i(z) = f_i(z') \wedge f_i(z) \leq i-1)
\end{array}
\tag{51}
$$

We prove (51) using another application of case analysis, this time according to whether or not we have $\exists w < i(f_i(w) = i-1)$. In the case that $\exists w < i(f_i(w) = i-1)$, the bottom line of (51) follows from this and the rest of (51) by taking $z = i$ and $z' = w$. (See Figure 3, first example.) In the case of $\neg\exists w < i(f_i(w) = i-1)$ —which can be rewritten as $\forall x < i(f_i(x) \neq i-1)$— we derive the bottom line from this and the rest of (51) by taking $z = y$ and $z' = y'$ (the y and y' asserted to exist in the penultimate line). (We do this case in more detail in Section A.6.)

$f_i(i) < i-1 \rightarrow (50)$ is equivalent to

$$
\begin{array}{ll}
 & \forall x \leq i(f_i(x) \leq i-1) \\
\wedge & f_i(i) < i-1 \\
\wedge & \forall x \leq i-1(f_i(x) = i-1 \rightarrow f_{i-1}(x) = f_i(i)) \\
\wedge & \forall x((f_i(x) < i-1) \rightarrow f_{i-1}(x) = f_i(x)) \\
\wedge & \exists y \neq y'(y \leq i-1 \wedge y' \leq i-1 \wedge f_{i-1}(y) = f_{i-1}(y') \wedge f_{i-1}(y) \leq i-2) \\
\rightarrow & \exists z \neq z'(z \leq i \wedge z' \leq i \wedge f_i(z) = f_i(z') \wedge f_i(z) \leq i-1)
\end{array}
\tag{52}
$$

We prove (52) using another application of case analysis, again according to whether or not we have $\exists w < i(f_i(w) = i-1)$. Here we have to refine the case analysis further, according to whether w is unique: formally whether we have $\exists w \neq w'(w, w' < i \wedge f_i(w) = i-1 \wedge f_i(w' = i-1))$. (See Figure 3, second and third examples.) If so, w and w' can be used for z and z' in the bottom line of (52). If not, it should be inferable that the x and x' that collide for f_{i-1} also collide for f_i. If we have $\neg\exists w < i(f_i(w) = i-1)$, then it should also follow that the x and x' that collide for f_{i-1} also collide for f_i.

As a final note, an alternative approach to the case analysis to proving (50) in the case where $f_i(i) = i-1$, would be by further case analysis on y, y': consider a case where y and y' satisfy $f_{i-1}(y) = f_{i-1}(y') < i-2$ (in which case, choose $z = y$, $z' = y'$). In the alternative case of $f_{i-1}(y) = f_{i-1}(y') = i-2$, if $f_i(y) = i-1$ then $f_i(y) = f_i(i)$— choose $z = y, z' = i$. Similarly if $f_i(y') = i-1$ then $f_i(y') = f_i(i)$— choose $z = y', z' = i$.

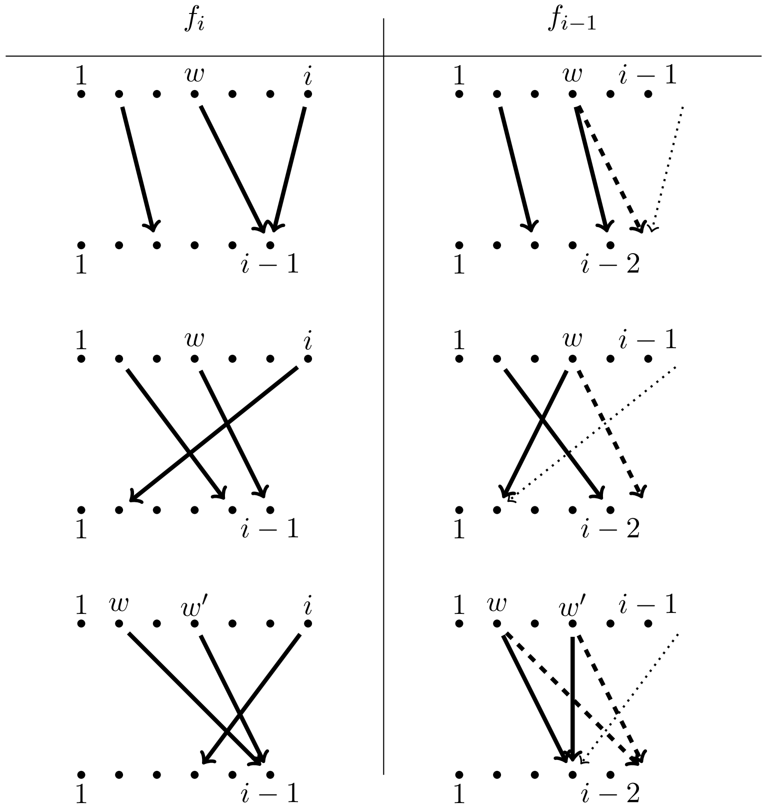


Figure 3: Illustration re proofs of (51), (52).

A.5 Proof of line (25) from Theorem 1: the formula $A_2 \to C_2$

Recall that $x^{(0)}$ denotes the n-vector $(0,\ldots,0)$, $x^{(1)}$ denotes the n-vector $(0,\ldots,0,1)$, and $x^{(2)}$ denotes the n-vector $(0,\ldots,0,1,0)$. Thus $x = x^{(2)}$ is an abbreviation for $x_1 = x_2 = \ldots = x_{n-2} = \text{FALSE}$; $x_{n-1} \vee x_n$; $\neg(x_{n-1} \wedge x_n)$.

Using rule (1), which allows us to write down $A \to A$ for any well-formed expression A, we can write

$$\vdash A_2 \to A_2$$

where recall A_2 is the expression $\forall x(x \leq x^{(2)} \to f_2(x) \leq x^{(1)})$.

Rename x to $\bar{x}$ in the right-hand occurrence of A_2, and bringing the quantifier to the front, we can deduce

$$\forall \bar{x}(A_2 \to (\bar{x} \leq x^{(2)} \to f_2(\bar{x}) \leq x^{(1)}))$$

Using universal instantiation (rule (10)), plugging in $x^{(1)}$ for $\bar{x}$ and then plugging in $x^{(2)}$ for $\bar{x}$ we can write down two lines containing the following expressions:

$$A_2 \to (x^{(1)} \leq x^{(2)} \to f_2(x^{(1)}) \leq x^{(1)})$$

$$A_2 \to (x^{(2)} \leq x^{(2)} \to f_2(x^{(2)}) \leq x^{(1)})$$

We can simplify these two expressions since $x^{(1)} \leq x^{(2)}$ and $x^{(2)} \leq x^{(2)}$ both evaluate to TRUE (using basic rules of replacement), to get

$$A_2 \to (f_2(x^{(1)}) \leq x^{(1)})$$

$$A_2 \to (f_2(x^{(2)}) \leq x^{(1)})$$

Then via conjunction introduction and $A \to (B \wedge C) \equiv (A \to B) \wedge (A \to C)$, we have

$$A_2 \to ((f_2(x^{(1)}) \leq x^{(1)}) \wedge (f_2(x^{(2)}) \leq x^{(1)}))$$

The right-hand side of the above can be shown to imply that $f_2(x^{(1)}) = f_2(x^{(2)})$, so we can combine with the the above to write down

$$A_2 \to ((f_2(x^{(1)}) \le x^{(1)}) \wedge (f_2(x^{(2)}) \le x^{(1)}) \wedge f_2(x^{(1)}) = f_2(x^{(2)})).$$

Insert into the RHS the expressions $x^{(1)} \le x^{(2)}$ and $x^{(2)} \le x^{(2)}$.

Use existential generalisation rule (12) twice (replacing occurrences of the constants $x^{(1)}$ and $x^{(2)}$ with existentially quantified variables x and x'). Push the existential quantifier into the RHS of the expression (using (9)), and we end up with the desired $A_2 \to C_2$.

A.6 Further details on the proof of (51)

Equation (51), in the case $\forall x < i(f_i(x) \ne i-1)$, is equivalent to

$$\begin{array}{rl} & \forall x \le i(f_i(x) \le i-2) \\ \wedge & f_i(i) = i-1 \\ \wedge & \forall x((f_i(x) < i-1) \to f_{i-1}(x) = f_i(x)) \\ \wedge & \exists y \ne y'(y \le i-1 \wedge y' \le i-1 \wedge f_{i-1}(y) = f_{i-1}(y') \wedge f_{i-1}(y) \le i-2) \\ \to & \exists z \ne z'(z \le i \wedge z' \le i \wedge f_i(z) = f_i(z') \wedge f_i(z) \le i-1) \end{array} \tag{53}$$

We can see that we would like to put z, z' equal to y, y' respectively.

Using the antecedent strengthening rule (6), it is sufficient to prove a version of the above where the subexpression "$\wedge f_i(i) = i-1$" is omitted, also the first and third lines imply $\forall x \le i(f_i(x) \le i-2 \wedge f_{i-1}(x) = f_i(x))$, so it's sufficient to prove:

$$\begin{array}{rl} & \forall x \le i(f_i(x) \le i-2 \wedge f_{i-1}(x) = f_i(x)) \\ \wedge & \exists y \ne y'(y \le i-1 \wedge y' \le i-1 \wedge f_{i-1}(y) = f_{i-1}(y') \wedge f_{i-1}(y) \le i-2) \\ \to & \exists z \ne z'(z \le i \wedge z' \le i \wedge f_i(z) = f_i(z') \wedge f_i(z) \le i-1) \end{array} \tag{54}$$

In order to apply Lemma 4 we need to make an extra copy of the universally-quantified variable vector x in (54); (54) is equivalent to:

$$\begin{array}{rl} & \forall x, x' \le i(f_i(x) \le i-2 \wedge f_{i-1}(x) = f_i(x) \wedge f_i(x') \le i-2 \wedge f_{i-1}(x') = f_i(x')) \\ \wedge & \exists y \ne y'(y \le i-1 \wedge y' \le i-1 \wedge f_{i-1}(y) = f_{i-1}(y') \wedge f_{i-1}(y) \le i-2) \\ \to & \exists z \ne z'(z \le i \wedge z' \le i \wedge f_i(z) = f_i(z') \wedge f_i(z) \le i-1) \end{array} \tag{55}$$

Lemma 4 says that it's sufficient to be able to generate, for all i, x, x', a proof of:

$$\begin{array}{rl} & x, x' \le i \to (f_i(x) \le i-2 \wedge f_{i-1}(x) = f_i(x) \wedge f_i(x') \le i-2 \wedge f_{i-1}(x') = f_i(x')) \\ \wedge & (x \ne x' \wedge x \le i-1 \wedge x' \le i-1 \wedge f_{i-1}(x) = f_{i-1}(x') \wedge f_{i-1}(x) \le i-2) \\ \to & (x \ne x' \wedge x \le i \wedge x' \le i \wedge f_i(x) = f_i(x') \wedge f_i(x) \le i-1) \end{array} \tag{56}$$

It can be checked that (56) is a tautology, so Lemma 2 can be used.

A.7 Proof of a technical equivalence used in Lemmas 1, 2

We show that the standard replacement rules of propositional logic allow us to prove that for any $i \in [2^n]$

$$x \le i-1 \vee x = i \equiv x \le i.$$

Put $k = i-1$. Note that for some $j \in [n]$,

$$i_1 = k_1, i_2 = k_2, \ldots, i_j = 1, k_j = \text{FALSE}, i_{j+1} = \text{FALSE}, k_{j+1} = \text{TRUE} \ldots i_n = \text{FALSE}, k_n = \text{TRUE}. \tag{57}$$

$x = i$ is an abbreviation for

$$x_1 = i_1 \wedge \overbrace{x_2 = i_2 \wedge \ldots \wedge x_n = i_n}^{E}. \tag{58}$$

$x \leq i - 1$ is an abbreviation for

$$\overbrace{\neg x_1 \wedge k_1}^{A} \vee \overbrace{(x_1 = k_1 \wedge \overbrace{(\neg x_2 \wedge k_2 \vee x_2 = k_2 \wedge (\ldots \neg x_n \wedge k_n) \ldots)}^{D})}^{B} \tag{59}$$

(for a non-strict inequality, we would insert $\vee x_n = k_n$ at the end.)
Similarly, $x \leq i$ is an abbreviation for

$$\neg x_1 \wedge i_1 \vee (x_1 = i_1 \wedge (\neg x_2 \wedge i_2 \vee x_2 = i_2 \wedge (\ldots \neg x_n \wedge i_n) \ldots)) \tag{60}$$

So we want to prove

$$(59) \vee (58) \equiv (60) \tag{61}$$

(59) $\vee$ (58) are of the form $(A \vee B) \vee C$, i.e. $A \vee (B \vee C)$ where $A = \neg x_1 \wedge i_1$ (assuming $j > 1$). We have $B = (x_1 = i_1) \wedge D$ and $C = (x_1 = i_1) \wedge E$, so $B \vee C \equiv (x_1 = i_1) \wedge (D \vee E)$. So the LHS of (61) can be written as

$$\neg x_1 \wedge i_1 \vee (x_1 = i_1 \wedge (D \vee E)).$$

D and E have the same structure as (59) and (58), so continue until we reach x_j:
(59) $\vee$ (58) is equivalent to

$$\neg x_1 \wedge i_1 \vee (x_1 = i_1 \wedge (\ldots (\neg x_{j-1} \wedge i_{j-1} \vee (x_{j-1} = i_{j-1} \wedge (D^j \vee E^j) \ldots)))) \tag{62}$$

where D^j is $\neg x_j \wedge k_j \vee (x_j = k_j \wedge (\neg x_{j+1} \wedge k_{j+1} \vee (\ldots) \ldots))$, and from (57) D^j is the expression $\neg x_j \wedge \text{FALSE} \vee (x_j = \text{FALSE} \wedge (\neg x_{j+1} \wedge \text{TRUE} \vee (\ldots) \ldots))$.

E^j is $x_j = i_j \wedge \ldots \wedge x_n = i_n$. From (57) we have $i_j = \text{TRUE}$, and $i_{j+1}, \ldots, i_n = \text{FALSE}$, and so E^j is the expression $x_j = \text{TRUE} \wedge x_{j+1} = \text{FALSE} \ldots \wedge x_n = \text{FALSE}$, so $E^j \equiv x_j \wedge \neg x_{j+1} \wedge \ldots \wedge \neg x_n$.

At this point, we aim to manipulate the subexpression $D^j \vee E^j$ (using the rules of replacement) to obtain the expression (having similar structure to (60)):

$$\neg x_j \wedge i_j \vee (x_j = i_j \wedge (\neg x_{j+1} \wedge i_{j+1} \vee x_{j+1} = i_{j+1} \wedge (\ldots \neg x_n \wedge i_n) \ldots)). \tag{63}$$

From (57) we have that $i_j = \text{TRUE}$ and $i_{j+1} = \ldots = i_n = \text{FALSE}$, so (63) is equivalent to
$\neg x_j \wedge \text{TRUE} \vee (x_j = \text{TRUE} \wedge (\neg x_{j+1} \wedge \text{FALSE} \vee x_{j+1} = \text{FALSE} \wedge (\ldots \neg x_n \wedge \text{FALSE}) \ldots))$. At this stage it's hopefully clear that a further, rather tedious, sequence of replacement rules makes this the same as $D^j \vee E^j$.

A.8 Do we need extension axiom lines for new functions?

Should we be able to define Wrong Proof with respect to a proof system that is less feature-rich, but still allows us to reduce from Pigeonhole circuit, Lonely, and Iter? In particular, do we need extension axioms of type (13) that define new *functions*, or should we be able to make do with standard extended Frege lines that define new propositional variables? In addition, it may be that our usage of quantifiers is a syntactic convenience, and if we can get rid of those also, we would have purely propositional extended-Frege proofs. We discuss these possibilities with respect to the reduction of Theorem 1 (reducing Pigeonhole Circuit to Wrong Proof).

Line F_i (23) in the reduction from PIGEONHOLE CIRCUIT is the following:

$$f_{i-1}(x) \leftrightarrow \begin{cases} i-2 & \text{if } \ x < i \wedge f_i(i) = i-1 \wedge f_i(x) = i-1 \\ f_i(i) & \text{if } \ x < i \wedge f_i(i) < i-1 \wedge f_i(x) = i-1 \\ f_i(x) & \text{otherwise. } \ (\text{i.e. } x \geq i \vee f_i(x) < i-1) \end{cases} \tag{64}$$

Now, we could regard expressions like "$f_i(x)$" and "$f_{i-1}(x)$" as vectors of propositional variables, which we think of as self-contained, rather than a combination of two things, a function and a bit vector. We can replace each line F_i with a sequence of 2^n lines F_{ij}, $j \in [2^n]$, that define the bit graph of function f_i, where F_{ij} contains the expression:

$$f_{i-1}(j) \leftrightarrow \begin{cases} i-2 & \text{if } \ j < i \wedge f_i(i) = i-1 \wedge f_i(j) = i-1 \\ f_i(i) & \text{if } \ j < i \wedge f_i(i) < i-1 \wedge f_i(j) = i-1 \\ f_i(j) & \text{otherwise. } \ (\text{i.e. } j \geq i \vee f_i(j) < i-1) \end{cases} \tag{65}$$

Since F_{ij} defines a valuation for the bit-vector $f_{i-1}(j)$, it can be further decomposed into n standard extended Frege lines, one for each component of $f_{i-1}(j)$. In general, any expression of the form $\forall x(\phi(x))$ (where x is an n-vector of variables) may (in the context of circuit-generated proofs) be split into 2^n expressions of the form $\phi(j)$, $j \in [2^n]$.

By contrast, existentially quantified expressions don't allow this treatment. Consider an expression like $\exists x, x'(f_i(x) = x^{(1)} \vee f_i(x) = f_i(x'))$. This is equivalent to the exponentially-long expression

$$\bigvee_{x,x'} (f_i(x) = x^{(1)} \vee f_i(x) = f_i(x')).$$

The advantage of this expression is that it treats subexpressions of the form "$f_i(x)$" as self-contained (vectors of) propositional variables, as opposed to functions acting on propositional variables. Since this exponentially-long expression is highly structured (a disjunction of many small clauses), it seems feasible to extend the definition of circuit-generated proof, so as to allow such expressions to constitute lines of a proof, and the corresponding circuit would take as input, a line number and a clause number, and compute the relevant clause. What's needed is inference rules that use these "long" lines, which are efficiently checkable for errors. One kind of inference rule that looks sound and may do what we want, would say that if we have two long disjunction lines $\bigvee_i \mathcal{C}_i$ and $\bigvee_i \mathcal{C}'_i$ where $\mathcal{C}_i$ and $\mathcal{C}'_i$ are clauses, and we want to deduce $\bigvee_i \mathcal{C}'_i$ from $\bigvee_i \mathcal{C}_i$, using additional information F, then it suffices to say that given any clause $\mathcal{C}_i$ and F, we can infer some clause $\mathcal{C}'_i$. A circuit-generated proof would just need to be able to output all of these smaller results (i.e. prove that $\mathcal{C}_i, F \vdash \mathcal{C}'_j$ for some j) and if any were erroneous, such an error could be verified, alternatively, if none were erroneous then we have a valid proof.

References

[1] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Annals of Mathematics* 160 (2): pp. 781–793 (2004).

[2] P. Beame, S. Cook, J. Edmonds, R. Impagliazzo and T. Pitassi. The relative complexity of NP search problems. *27th ACM Symposium on Theory of Computing* pp. 303–314 (1995).

[3] A. Beckmann, Sam Buss. "The NP Search Problems of Frege and Extended Frege Proofs," preliminary draft December 2015.

[4] Sam Buss. "Bounded Arithmetic," Bibliopolis, Naples, Italy, 1986, www.math.ucsd.edu/∼sbuss/ResearchWeb/BAthesis/

[5] Sam Buss "On Herbrand's Theorem," in Maurice, Daniel; Leivant, Raphaël, *Logic and Computational Complexity*, Lecture Notes in Computer Science, Berlin, New York: Springer-Verlag, pp. 195–209, (1995). www.math.ucsd.edu/-∼sbuss/ResearchWeb/herbrandtheorem/

[6] S. Buss. Polynomial size proofs of the propositional pigeonhole principle. *Journal of Symbolic Logic*, **52** pp. 916–927 (1987).

[7] S. Buss. Quasipolynomial size proofs of the Propositional Pigeonhole Principle. *Theoretical Computer Science.* **576**, pp. 77–84 (2015).

[8] S.R. Buss and A.S. Johnson. Propositional Proofs and Reductions between NP Search Problems. *Annals of Pure and Applied Logic.* **163**(9) pp. 1163–1182 (2012).

[9] X. Chen, X. Deng and S.-H. Teng. Settling the complexity of computing two-player Nash equilibria. *Journal of the ACM* **56**(3) pp. 1–57 (2009).

[10] S.A. Cook and R.A. Reckhow. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic* **44**(1) pp. 36–50 (1979).

[11] C. Daskalakis, P.W. Goldberg and C.H. Papadimitriou. The Complexity of Computing a Nash Equilibrium. *SIAM Journal on Computing* **39**(1) pp. 195–259 (2009).

[12] J. Hartmanis and L.A. Hemachandra. Complexity classes without machines: on complete languages for UP. *Theoretical Computer Science*, **58**(1–3) 129–142 (1988).

[13] J. Herbrand. *Recherches sur la théorie de la démonstration*, PhD thesis, Université de Paris, 1930.

[14] E. Jeřábek. Integer factoring and modular square roots. *J. Comput. System Sci.* **82**(2) pp. 380–394 (2016).

[15] I. Komargodski, M. Naor, and E. Yogev. White-Box vs. Black-Box Complexity of Search Problems: Ramsey and Graph Property Testing. *ECCC* Report TR17-015 (2017).

[16] Jan Krajíček. Implicit Proofs. *Journal of Symbolic Logic*, **69**(2), pp. 387–397 (2004).

[17] Nimrod Megiddo. A note on the complexity of P-matrix LCP and computing an equilibrium. *Res. Rep. RJ6439, IBM Almaden Research Center, San Jose.* pp. 1–5 (1988).

[18] C.H. Papadimitriou. On the complexity of the parity argument and other inefficient proofs of existence. *J. Comput. System Sci.* **48** pp. 498–532 (1994).

[19] P. Pudlák. On the complexity of finding falsifying assignments for Herbrand disjunctions. *Arch. Math. Logic*, 54, pp. 769–783 (2015).

[20] R. Rivest, A. Shamir and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM* **21**(2) pp. 120–126 (1978).

[21] Michael Sipser. "On relativization and the existence of complete sets," *Proceedings of the 9th Colloquium on Automata, Languages and Programming* pp. 523–531. Springer-Verlag, (1982).

ECCC ISSN 1433-8092

https://eccc.weizmann.ac.il