Towards a Unified Complexity Theory of Total Functions*†

Paul W. Goldberg[1] and Christos H. Papadimitriou[2]

1 **Department of Computer Science, University of Oxford, Oxford, UK**
Paul.Goldberg@cs.ox.ac.uk
2 **Department of Computer Science, Columbia University, New York, USA**
christos@cs.columbia.edu

Abstract

The class TFNP, of NP search problems where all instances have solutions, appears not to have complete problems. However, TFNP contains various syntactic subclasses and important problems. We introduce a syntactic class of problems that contains these known subclasses, for the purpose of understanding and classifying TFNP problems. This class is defined in terms of the search for an error in a concisely-represented formal proof. Finally, the known complexity subclasses are based on existence theorems that hold for finite structures; from Herbrand's Theorem, we note that such theorems must apply specifically to finite structures, and not infinite ones.

1998 ACM Subject Classification F.1.3 Complexity Measures and Classes , F.4.1 Mathematical Logic

Keywords and phrases Computational complexity, first-order logic, proof system, NP search functions, TFNP

Digital Object Identifier 10.4230/LIPIcs.ITCS.2018.37

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 [21] 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 [26, 10, 8].

Presently, our understanding of the complexity of TFNP problems is a bit fragmented. Currently, our main means for deriving 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").

* This work was supported by NSF grant CCF-1408635.
† A full version of the paper is available at ECCC, https://eccc.weizmann.ac.il/report/2017/056/

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 [1]. They all have complete problems (actually, the most commonly used definition of, 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 [16]. 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 [27, 12]. Indeed, Pudlák ([25], Section 6) presents a similar result specifically for TFNP. Hence, this route for a unified complexity view of total functions is not available.

This paper aims to develop 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[1] (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 [3]. Fix a consistent deductive system – in this paper we use a propositional proof system that we call Q-EFF (for "quantified boolean formulae with extended Frege functions"; it allows lines of a proof to define new n-ary functions). 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 [19].

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

[1] The class should perhaps be called $\text{PTFNP}_{\text{Q-EFF}}$ since it is defined with respect to a deductive system Q-EFF that we introduce and use in our proofs here. Similar definitions with respect to other proof systems are possible. In this paper we just refer to it as PTFNP. A similar point applies to the problem Wrong Proof used to define PTFNP.

reference to the problems that are complete for the above complexity classes. Its proof system Q-EFF uses 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 these questions further. 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 6 that this is no coincidence: Herbrand's Theorem from 1930 [13, 6] 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.

1.1 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 [2], Arnold Beckmann and Sam Buss, working within the tradition of bounded arithmetic [3], 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. Our system Q-EFF differs in using propositional variables only, but arithmetic theories can be translated into propositional ones (see [18], Chapter 9).

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 [7] 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 [25] 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 [25] 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. [17] shows that any such TFNP problem has a *query complexity* proportional to the description-size of a problem instance. However, a reduction using the oracle model should allow a logical description of a circuit to be plugged in.

Finally, Hubáček et al. [14] show that hard-on-average NP problems lead to hard-on-average TFNP problems. The TFNP problems thus constructed are specific to the associated NP problems; our concern here, in contrast, is to identify a single easily-understood TFNP problem that generalises previous ones.

1.2 Background on propositional proofs and the pigeonhole principle

In 1979, Cook and Reckhow [9] 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 [9], 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.

[9] 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 [9]. 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[2]. 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). [9] gave polynomially-bounded extended Frege proofs of these expressions. Buss [5] subsequently gave polynomially-bounded Frege proofs of these, and in [4] quasi-polynomial size Frege proofs that are a reformulation of the extended Frege proofs of [9]. See [4] for a discussion of other candidate classes of formulae and progress that has been made on them.

Papadimitriou [23] 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

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

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? Our deductive system Q-EFF is first-order, but we require (exponentially many) lines that define the behaviour of additional function symbols (mapping n-bit strings to single-bit outputs). This seems to be a more powerful facility than the additional variable symbols allowed by extended Frege proofs (hence the name Q-EFF for "extended Frege functions"). 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.

1.3 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 and 5 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 6 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 7.

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 Q-EFF in this paper, the language (and corresponding semantics) of interest is simple and straightforward, and it is not hard to check that it is 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.[3]

▶ **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 is 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.

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 deductive system that we describe in detail in the rest of this section.

2.1 The formulae and theorems of our system Q-EFF; 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 on the expressions used in our proofs.

[3] 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 is useful to have these extension-axiom lines that define new functions.

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 $\forall x, y(x < y \rightarrow \phi(x, y))$, or if y is a vector of propositional constants, it would mean $\forall x(x < y \rightarrow \phi(x, y))$. 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 $\text{TRUE} \vee (x_i \wedge y_i)$ and write a new line containing $\ell \vdash \text{TRUE} \vee (y_i \wedge x_i)$.
- "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 means $\forall x(f(x) \triangleq \phi(x))$, and the system can use $\forall x(f(x) = \phi(x))$ as a theorem.

Some of the rules we list below are redundant in the sense that they could be simulated using the others. We prefer 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.

As noted earlier, our extension axiom lines are somewhat novel. A standard extension-axiom line of an extended Frege proof may introduce a new propositional variable and set its value to equal some expression in terms of pre-existing values. Our extension-axiom rules (see rule (13)) allow us to define new *functions* via expressions that define their behaviour in

terms of pre-existing functions. So we call the proof system Q-EFF (for "extended Frege functions") on account of this novel feature.

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 proof line of the form $\ell \vdash A$ may state that A is derived from expression A', where A' is the theorem derived at line ℓ, via applying one of these basic manipulations to A', or to 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, 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 $A(\textsc{true})$ and $A(\textsc{false})$ be obtained by plugging in the constants $\textsc{true}$ and $\textsc{false}$ respectively in place of x_i, in $A(x_i)$. Then we have:

$$\begin{aligned} \exists x_i(A(x_i)) &\equiv A(\textsc{true}) \vee A(\textsc{false}) \\ \forall x_i(A(x_i)) &\equiv A(\textsc{true}) \wedge A(\textsc{false}) \end{aligned} \tag{7}$$

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

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

(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

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

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 is 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 is also 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 use the following rule, which can be simulated by previous ones:

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

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.[4] 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 is 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 3 shows how to construct such a proof. The three expressions in the statement of Lemma 3 correspond to the existence principles underlying PPP, PPA, and PLS (recall that PPAD and PPADS are special cases of PPP).

[4] 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.

The proofs of $\exists(x, x')\phi(x, x')$ are done separately for each TFNP problem of interest, in Sections 4 for PPP and 5 (but we leave to the full version, any detail on the proofs for PPA and PLS.). Section 3.1 introduces the general approach taken in Sections 4 and 5. Section 3.2 presents Lemma 3 that shows how to make a suitable proof of $\neg\exists(x, x')\phi(x, x')$.

3.1 Overview of the reductions of Sections 4 and 5

In Sections 4 and 5, 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 Q-EFF, 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 and 5 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 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 is 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 3.

Here we give some detail on the first reduction (from PIGEONHOLE CIRCUIT), leaving out further detail that appears in appendices of the full version of this paper. We leave to the full version the details on 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, Q-EFF 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 3.** *Suppose f is defined according to the above construction. Consider the expressions*[5]

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

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 3 are the principles underlying PPP, PPA, and PLS, used in Theorems 4, 6, 7. 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 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 $f_g(x) =$ TRUE/FALSE.

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

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 $(xx' = i) \to \phi'(x,x')$, which we then use to derive Φ'_i from Φ'_{i-1} in conjunction with $(xx' = i) \to \phi'(x,x')$. In particular we can derive $\Phi'_{i-1} \wedge ((xx' = i) \to \phi'(x,x'))$, equivalently $\forall xx'((xx' < i \to \phi'(x,x')) \wedge (xx' = i \to \phi'(x,x')))$, equivalently $\forall xx'((xx' < i \vee xx' = i) \to \phi'(x,x'))$, equivalently (via an equivalence shown in the full version of this paper), $\forall xx'(xx' \leq i \to \phi'(x,x'))$, which is the same as Φ'_i.

How to formally prove $(xx' = i) \to \phi'(x,x')$: For each gate g of C, in the order in which the functions f_g are defined, we can prove a line saying

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

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 $(xx' = i) \to f_{g'}(x) = j_{g'}(x)$, $(xx' = i) \to f_{g''}(x) = j_{g''}(x)$.)

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

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

It then suffices to prove

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

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 $(xx' = i) \to \phi'(x,x')$. The circuit can be extended to a concise proof of (17). ◀

4 Reduction from PPP to Wrong Proof

In this section we establish the following result:

▶ **Theorem 4.** *Any problem that belongs to the complexity class* PPP *(which includes* PPAD *and* PPADS*) is reducible to* W*rong Proof (with respect to our deductive system* Q-EFF *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 5.** 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 4) 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 $f : [2^n] \to [2^n]$, which we construct using Q-EFF 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 3 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 the full version of this paper we show how to prove this using Q-EFF.

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 $F := f_{2^n}(x) \leftrightarrow f(x)$. In the full version we show in detail 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 [9], 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 the full version we give details on 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 we can derive containing $\neg S_C \rightarrow A_{2^n}$, 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 the full version we give details on 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 the full version we give some more detail on how to construct a formal proof of (26) using Q-EFF.

Putting it all together, we noted earlier that we have a sequence of lines containing $\neg S_C \to A_i$, 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) $\neg S_C \to C_{2^n}$; 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. ◀

5 PPA and PLS

The analogues of Theorem 4 for the classes PPA and PLS are stated below. We omit here the proofs, which are quite substantial but similar in spirit with that of Theorem 4, once appropriate adjustments have been made for the combinatorial idiosyncracies of these two classes. By the known inclusions, this covers all known syntactic subclasses of TFNP. The interested reader can see the details in the full version.

The analogue for PPA reduces from the problem Lonely as defined and shown PPA-complete in Beame et al. [1]; for PLS we make use of the problem Iter shown PLS-complete by Morioka [22] (Section 3.2).

▶ **Theorem 6.** *Any problem that belongs to the complexity class* PPA *is reducible to* W*rong Proof.*

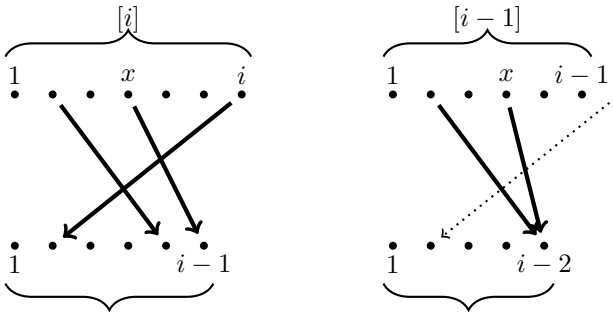
▶ **Theorem 7.** *Any problem that belongs to the complexity class* PLS *is reducible to* W*rong Proof.*

6 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."



$f_i([i]) = [i-1]$ $f_{i-1}([i-1]) = [i-2]$

"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.

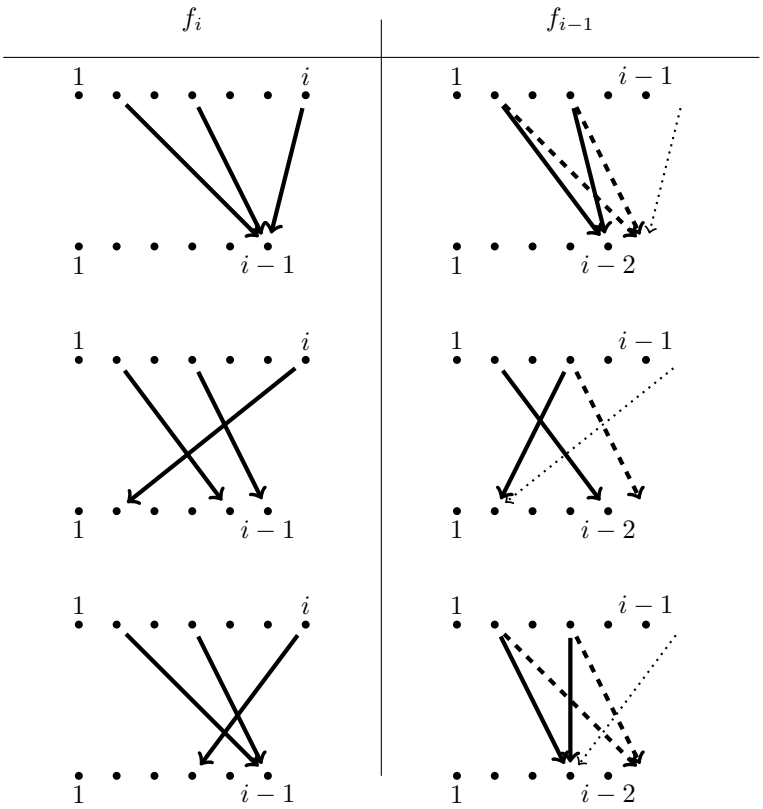


Figure 1 Construction of f_{i-1} from f_i (re proof of Theorem 4), such that from $f_i(x) < i$ for all $x \leq i$, we have $f_{i-1}(x) < i-1$ for all $x \leq i-1$. Dotted lines represent evaluations of f_{i-1} on i, and we are just interested in f_{i-1} on the domain $[i-1]$. Dashed lines are ones that 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}.

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 8.** *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. ◀

7 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.

Our understanding is that the present results are implicit in recent work in Bounded Arithmetic, but our system Q-EFF is of interest since it seems to allow more direct reductions. We also highlight the general topic of using more powerful logic to define more expressive versions of the Wrong Proof problem, and whether new TFNP problems can be defined as a result. With regard to the "rogue problems" such as Factoring and Ramsey, these have also been addressed in the Bounded Arithmetic literature, and similar results are obtainable for Ramsey [24, 15] and Factoring [20]. As noted in the Introduction, we are also working towards using our approach to placing these problems in PTFNP [11]. The topic of more powerful systems raises a general question of whether we reach a limit where no further TFNP problems can be expressed. This relates to the open problem in propositional proof complexity of whether there's a most powerful proof system, for which the answer is believed to be negative.

A related question is whether the reductions of the present paper should be modifiable to work with a *weaker* proof system than Q-EFF, and the work of Beckmann and Buss [2] indicates that this should be possible in principle, in particular that one should be able to use a system with extended Frege lines defining propositional variables rather than functions. With regard to the extended Frege function lines we use, it is tempting to try to define such

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? This approach does not seem applicable, at least in a direct and straightforward way.

Acknowledgements. 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.

References

1 Paul Beame, Stephen A. Cook, Jeff Edmonds, Russell Impagliazzo, and Toniann Pitassi. The relative complexity of NP search problems. *J. Comput. Syst. Sci.*, 57(1):3–19, 1998. doi:10.1006/jcss.1998.1575.

2 Arnold Beckmann and Sam Buss. The NP search problems of frege and extended frege proofs. *ACM Trans. Comput. Log.*, 18(2):11:1–11:19, 2017. doi:10.1145/3060145.

3 Sam Buss. *Bounded Arithmetic*. Bibliopolis, Naples, Italy, 1986. URL: www.math.ucsd.edu/~sbuss/ResearchWeb/BAthesis/.

4 Sam Buss. Quasipolynomial size proofs of the propositional pigeonhole principle. *Theor. Comput. Sci.*, 576:77–84, 2015. doi:10.1016/j.tcs.2015.02.005.

5 Samuel R. Buss. Polynomial size proofs of the propositional pigeonhole principle. *Journal of Symbolic Logic*, 52:916–927, 1987.

6 Samuel R. Buss. On Herbrand's Theorem. In *Logic and Computational Complexity*, volume 960 of *Lecture Notes in Computer Science*, pages 195–209. Springer, Berlin, Heidelberg, 1995. URL: www.math.ucsd.edu/~sbuss/ResearchWeb/herbrandtheorem/.

7 Samuel R. Buss and Alan S. Johnson. Propositional proofs and reductions between NP search problems. *Ann. Pure Appl. Logic*, 163(9):1163–1182, 2012. doi:10.1016/j.apal.2012.01.015.

8 Xi Chen, Xiaotie Deng, and Shang-Hua Teng. Settling the complexity of computing two-player nash equilibria. *J. ACM*, 56(3):14:1–14:57, 2009. doi:10.1145/1516512.1516516.

9 Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *J. Symb. Log.*, 44(1):36–50, 1979.

10 Constantinos Daskalakis, Paul W. Goldberg, and Christos H. Papadimitriou. The complexity of computing a nash equilibrium. *SIAM J. Comput.*, 39(1):195–259, 2009. doi:10.1137/070699652.

11 Matthew Greaves. *Classifying the computational complexity of the Ramsey and Factoring Problems*. MSc dissertation, University of Oxford, 2017.

12 Juris Hartmanis and Lane A. Hemachandra. Complexity classes without machines: On complete languages for UP. *Theor. Comput. Sci.*, 58:129–142, 1988. doi:10.1016/0304-3975(88)90022-9.

13 Jacques Herbrand. *Recherches sur la théorie de la démonstration*. 1930. URL: http://eudml.org/doc/192791.

14 Pavel Hubácek, Moni Naor, and Eylon Yogev. The journey from NP to TFNP hardness. *Electronic Colloquium on Computational Complexity (ECCC)*, 23:199, 2016. URL: http://eccc.hpi-web.de/report/2016/199.

15 Emil Jeřábek. Approximate counting by hashing in bounded arithmetic. *The Journal of Symbolic Logic*, 74(3):829–860, 2009.

16 Emil Jerábek. Integer factoring and modular square roots. *J. Comput. Syst. Sci.*, 82(2):380–394, 2016. doi:10.1016/j.jcss.2015.08.001.

17 Ilan Komargodski, Moni Naor, and Eylon Yogev. White-box vs. black-box complexity of search problems: Ramsey and graph property testing. *Electronic Colloquium on Computational Complexity (ECCC)*, 24:15, 2017. URL: https://eccc.weizmann.ac.il/report/2017/015.

18 Jan Krajíček. *Bounded arithmetic, propositional logic, and complexity theory*, volume 60 of *Encyclopedia of Mathematics and Its Applications*. Cambridge University Press, 1995.

19 Jan Krajícek. Implicit proofs. *J. Symb. Log.*, 69(2):387–397, 2004. doi:10.2178/jsl/1082418532.

20 Jan Krajícek and Pavel Pudlák. Some consequences of cryptographical conjectures for S^1_2 and EF. *Inf. Comput.*, 140(1):82–94, 1998. doi:10.1006/inco.1997.2674.

21 Nimrod Megiddo. A note on the complexity of P-matrix LCP and computing an equilibrium. Technical Report RJ6439, IBM Almaden Research Center, San Jose, 1988.

22 Tsuyoshi Morioka. Classification of search problems and their definability in bounded arithmetic. *Electronic Colloquium on Computational Complexity (ECCC)*, 8(082), 2001. URL: http://eccc.hpi-web.de/eccc-reports/2001/TR01-082/index.html.

23 Christos H. Papadimitriou. On the complexity of the parity argument and other inefficient proofs of existence. *J. Comput. Syst. Sci.*, 48(3):498–532, 1994. doi:10.1016/S0022-0000(05)80063-7.

24 Pavel Pudlák. Ramsey's theorem in bounded arithmetic. In *Proceedings of the 4th Workshop on Computer Science Logic*, CSL '90, pages 308–317, London, UK, UK, 1991. Springer LNCS 553.

25 Pavel Pudlák. On the complexity of finding falsifying assignments for herbrand disjunctions. *Arch. Math. Log.*, 54(7-8):769–783, 2015. doi:10.1007/s00153-015-0439-6.

26 Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978. doi:10.1145/359340.359342.

27 Michael Sipser. On relativization and the existence of complete sets. In *Proceedings of the 9th Colloquium on Automata, Languages and Programming*, pages 523–531, London, UK, 1982. Springer-Verlag.