

Hierarchies Within TFNP: Building Blocks and Collapses

Surendra Ghentiyala*

Zeyong Li†

Abstract

In all well-studied TFNP subclasses (e.g. PPA, PPP etc.), the canonical complete problem takes as input a polynomial-size circuit $C : \{0,1\}^n \rightarrow \{0,1\}^m$ whose input-output behavior implicitly encodes an exponentially large object G , i.e. C is the succinct (polynomial-size) representation of the exponential size object G . The goal is to find some particular substructure in G which can be confirmed in polynomial time using queries to C .

While such formulations have proven fruitful in the TFNP literature, they are arguably insufficient to characterize much of TFNP. For example, for any object G whose succinct description requires one to factor integers, it seems we cannot represent it by a circuit C under the widely believed assumption that $\text{Factor} \notin \text{FP}$.

To address this, we initiate the study of classes of the form A^B where both A and B are TFNP subclasses. In particular, we define complete problems for these classes that take as input a circuit C which is allowed oracle gates to another TFNP class. For example, PPP^{PPA} would involve finding a collision in a circuit $C^{\text{PPA}} : [N] \rightarrow [N-1]$ where C has oracle gates to a PPA-complete problem. We can then iterate this construction to obtain hierarchies (e.g. $\text{PPP}^{(\text{PPP}^{\text{PPA}})}$). Here, we uncover a rich structure of hierarchies and collapses. The hierarchies we obtain are contained in TFNP and therefore much lower than the other generalization of TFNP subclasses (TFPH) recently defined in Kleinberg, Korten, Mitropolsky, and Papadimitriou (ITCS'21).

The exact definitions require some care since, unlike a class like PPP^{NP} , where the NP oracle defines a function, in PPP^{PPA} , the oracle is for a search problem with many possible solutions. To address this, we introduce a strict condition called *Internal Consistency* that underlies the definition. Informally, this condition enforces that, from the perspective of a bounded verifier, the oracle behaves like a function.

Beyond introducing definitions for TFNP oracle problems, our specific technical contributions include showing that several TFNP subclasses are self-low and hence their corresponding hierarchies collapse. In particular, $\text{PPA}^{\text{PPA}} = \text{PPA}$, $\text{PLS}^{\text{PLS}} = \text{PLS}$, and $\text{LOSSY}^{\text{LOSSY}} = \text{LOSSY}$. As an immediate consequence, we derive that when reducing to PPA, one can always assume access to PPA—and therefore factoring—oracle gates.

In addition to introducing a variety of hierarchies within TFNP that merit study in their own right, these ideas introduce a novel approach for classifying computational problems within TFNP. For example, we observe that the problem of deterministically generating large prime numbers, which has long resisted classification in a TFNP subclass, is in PPP^{PPP} under the Generalized Riemann Hypothesis.

*Cornell University. Email: sg974@cornell.edu. This work is supported in part by the NSF under Grants Nos. CCF-2122230 and CCF-2312296, a Packard Foundation Fellowship, and a generous gift from Google.

†University of Toronto. Email: zeyong1123@gmail.com. Most of this work was done while the author was at the National University of Singapore, supported by NRF grant NRF-NRFI09-0005.

Contents

1	Introduction	1
1.1	Our Contributions	3
1.2	Internal Consistency: Technical Nuisance or Structural Property?	4
1.3	Utility of TFNP Hierarchies	5
1.4	Open Questions	7
2	Preliminaries	7
2.1	TFNP	7
2.2	Some TFNP subclasses	8
3	Definition	9
3.1	Oracle Gates	9
3.2	Evaluating Oracle Circuits with Auxiliary Oracle Answers: C^*	10
3.3	The Full Definition	11
3.4	Another Helpful Evaluation: C_*	12
4	Robustness of Definition	13
4.1	Hierarchies Remain in TFNP	14
4.2	The Choice of Complete Problem	14
4.3	The Choice of Complete Problem (Continued)	16
4.4	Is Internal Consistency Necessary: an Example	16
5	Self-lowness	17
5.1	PPA Self-lowness	17
5.2	PLS Self-lowness	20
5.2.1	The PLS-Complete Problems	20
5.2.2	The Reduction	21
5.3	LOSSY Self-lowness	23
6	Further Applications	25
6.1	Number Theory	25

1 Introduction

The complexity class **TFNP** (Total Functions in **NP**) consists of search problems where a solution is guaranteed to exist and solutions can be verified in polynomial time [Pap94]. Although **TFNP** is a semantic class of problems, many syntactic subclasses of **TFNP** have been identified and studied, forming a rich ‘ecosystem’ within **TFNP**.

These syntactic subclasses typically correspond to the mathematical principle proving their totality. More specifically, the canonical complete problem for a **TFNP** subclass typically takes as input a polynomial-size circuit $C : \{0, 1\}^n \rightarrow \{0, 1\}^m$ encoding an exponentially large combinatorial object G . The goal is to find some particular substructure in G whose existence is guaranteed by the underlying combinatorial principle.

Take **PPP** (Polynomial Pigeonhole Principle) as an illustrative example. The subclass and its canonical complete problem **PIGEON** are defined as follows:

Definition 1.1 (**PIGEON**). Given a $\text{poly}(n)$ size circuit $C : \{0, 1\}^n \rightarrow \{0, 1\}^n$, output one of the following.

1. x s.t. $C(x) = 0^n$;
2. distinct x_1, x_2 s.t. $C(x_1) = C(x_2)$.

PPP is defined as all search problems which are many-one reducible to **PIGEON**.

Notice that a solution to **PIGEON** always exists since if C is surjective, then a type 1 solution exists, and if C is not surjective, a type 2 solution exists by the pigeonhole principle. **PIGEON** therefore always has a solution and is what we refer to as a total function problem. Furthermore, any solution to **PIGEON** is clearly efficiently verifiable since it simply requires evaluating C at most twice followed by checking equality of two polynomial length strings.

PPP is only one of many fundamental **TFNP** subclasses; others include **PPA**, **PPAD**, **PPADS**, and **PLS**. Despite being defined with respect to circuit problems (which are interpreted as unnatural problems), these subclasses prove to be important as they capture the complexity of many natural computational problems. To name a few:

- Computing a Nash equilibrium is complete for **PPAD** [DGP09, CDT09].
- **PPP** captures the complexity of many cryptographic primitives [SZZ18].
- **PLS** captures the complexity of searching for a local optimum [Kre90, Sch91].

While the identification and study of these subclasses under such formulations have greatly enhanced our understanding of **TFNP**, we argue that existing definitions are insufficient to characterize “most totality arguments in **TFNP**”. For example, the well-known problems of deterministically generating large primes (which exist by the Bertrand–Chebyshev theorem) and finding a monochromatic clique in the edge coloring of an exponentially large graph (which exists by Ramsey’s theorem) have yet to find a home inside **TFNP**. (Our hierarchies will allow us to find a home for the former problem in [Section 6](#)).

In fact, a very general class of constructive principles which seem very **TFNP**-like does not immediately have a home in any **TFNP** subclass. Consider the following scenario. Let $G : [2^n] \rightarrow [2^n - 1]$ be a function where $G(x)$ can be evaluated in $\text{poly}(n)$ time given access to a **FACTOR**

oracle (see [definition 2.15](#) for a formal definition of an oracle for factoring integers). The task is to find a collision in G , i.e. find $x \neq y$ such that $G(x) = G(y)$. On one hand, under the widely believed assumption that $\text{FACTOR} \notin \text{FP}$, we do not know how to evaluate G using a vanilla circuit and therefore cannot reduce collision finding in G to (some variant of) PIGEON . On the other hand, since both FACTOR and PIGEON are in TFNP , this task intuitively “belongs to” TFNP . It remains total since the pigeonhole principle tells us that G has a collision, regardless of whether it is efficiently computable or not. Moreover, one can efficiently verify a collision in G given x, y , and the solutions to the FACTOR oracle calls. (The observant reader may notice that this definition is more subtle than it appears upon first inspection since the output of a FACTOR gate is not well-defined for integers with multiple prime factors, but we will deal with this appropriately in [Section 3](#).)

This raises an unsettling possibility: there may exist rather natural problems in TFNP that we do not have the right tools to define, let alone study. A natural solution to this issue is to allow circuits with oracle gates to TFNP problems as input. Consider the above example: G can be succinctly represented by a polynomial-size oracle circuit $C^{\text{FACTOR}} : \{0, 1\}^n \rightarrow [2^n - 1]$ where the circuit is allowed oracle gates solving FACTOR . One could now interpret this circuit as an input and ask for a collision!

While oracle separations between TFNP subclasses have been well-studied [[BCE⁺95](#), [Mor](#), [BOM04](#), [GHJ⁺24](#), [JLRX24](#)], the idea of raising TFNP subclasses to oracles and studying the resulting complexity classes in their own right is relatively new. [[KKMP21](#)] introduced $\text{TFNP}^{\Sigma_i^P}$ as a total function analogue of PH . They also considered complexity classes like $\text{PPP}^{\Sigma_i^P}$, where the input circuit C in [definition 1.1](#) is allowed to have Σ_i^P oracle gates.

Although TFNP has only recently begun to be studied through the lens of oracles, oracles are ubiquitous in other areas of complexity theory. For example, the polynomial hierarchy PH , which can be defined in terms of oracles ($\Sigma_0^P = \text{P}$, $\Sigma_{i+1}^P = \text{NP}^{\Sigma_i^P}$), is related to a wide variety of complexity classes and important problems. Just to name a few, the Sipser-Lautemann theorem [[Lau83](#)] states that $\text{BPP} \subseteq \Sigma_2^P \cap \Pi_2^P$, and the circuit minimization problem is known to be in Σ_2^P but not known to be in NP . Furthermore, the assumption that the polynomial hierarchy does not collapse—that $\Sigma_i^P \neq \Sigma_{i+1}^P$ for any integer $i > 0$ —is now a standard assumption in complexity theory.

In this work, we introduce TFNP subclasses that have oracle access to some TFNP problem. For example, $\text{PPP}^{\text{PIGEON}}$ would have as its complete problem $\text{PIGEON}^{\text{PIGEON}}$ which is the same as PIGEON except that the input circuit C is allowed to have PIGEON oracle gates. While the intuition is straightforward, the actual definition requires some care since, unlike a Σ_i^P oracle gate, which encodes a decision problem with exactly one output, a PIGEON instance may have an exponential number of possible outputs since PIGEON defines a relation rather than a function. The behavior of the PIGEON oracle gate is therefore underspecified. Hence, it is somewhat unclear exactly how one should interpret the oracle gates unless the oracle problem is a search problem with unique solutions (like finding *all* prime factors of an integer). Informally, we resolve this issue by quantifying over all possible functions consistent with the relation defined by the oracle problem, and requiring the oracle answers exposed during verification to be consistent with one another. We call this condition *Internal Consistency*. See [Section 3](#) for a detailed discussion.

Under the definition above, one can naturally define hierarchies of subclasses within TFNP , such as the PPA hierarchy defined as $\text{PPA}^1 = \text{PPA}$, $\text{PPA}^i = \text{PPA}^{\text{PPA}^{i-1}}$ for $i > 0$, and ask about the complexity of these hierarchies. We further show that, under our definition, several classic TFNP subclasses are self-low. That is, their corresponding hierarchies collapse to the first level.

Our ideas explore a new dimension in the complexity landscape within TFNP . This opens up

the exciting possibility of analogies to PH. Our hierarchies hint at hitherto unasked questions about the structure of the most well-studied TFNP subclasses: Which subclass is self-low? How do the different subclasses interact with each other when given as an oracle? Are there Karp-Lipton type collapses within the TFNP hierarchies? What should standard assumptions look like in the TFNP hierarchies world (what is the equivalent of the assumption that PH does not collapse)?

1.1 Our Contributions

Definitions. Our first contribution is a robust definition for a new family of subclasses in TFNP. We propose a definition of problems taking the form of A^B when A is a TFNP circuit problem ([definition 3.1](#)) and B is any TFNP problem ([definition 3.5](#), [definition 3.7](#)). Informally, a circuit problem is one where the input consists of a circuit C and some other input a , and an answer to the problem can be verified using black-box queries to C . A^B takes as input a $\text{poly}(n)$ size circuit C^B with oracle gates for B and some other input a , and outputs a solution $y, w_{1,1}, \dots, w_{\text{poly}(n), \text{poly}(n)}$. Informally, one should view y as a solution to A on input (C^B, a) . However, verifying y as a solution requires evaluating C^B , which requires evaluating B oracle gates. This is where $w_{1,1}, \dots, w_{\text{poly}(n), \text{poly}(n)}$ come in. We use $w_{i,j}$ as the solution to the j^{th} B oracle gate query on the i^{th} time the verifier for A makes a query to C^B . In some sense, the solution to A^B on input (C^B, a) has the same form as the solution to A but also includes the auxiliary information required to evaluate C^B to verify a solution.

Robustness of Definition. While such a definition appears straightforward on a high level, it is indeed delicate due to the fact that a TFNP problem defines a relation and not a function. In order to obtain a robust definition, we introduce an additional property called *Internal Consistency*, which intuitively enforces the oracle behave like a function from the perspective of the (polynomially bounded) verifier.

In particular, we showed that (under the stricter internally consistent requirement), this definition has several desirable properties, making it a robust choice for defining such classes. Assume A is a TFNP circuit problem ([definition 3.1](#)) and B is any TFNP problem, and A and B are the sets of total search problems reducible to A and B respectively. By defining A^B as the set of total search problems reducible to A^B under many-one reductions, we observe that A^B is in TFNP ([theorem 4.1](#)) and that A^B is robust to the choice of complete problem for A or B ([theorem 4.3](#), [theorem 4.6](#)) up to FP^B reductions.

Having defined TFNP oracle classes, we can now define whole hierarchies.

Definition 3.8. Let A be a TFNP subclass and A be the canonical A -complete circuit problem. We define $A^1 = A$, $A^i = A^{(A^{i-1})}$, and A^i as all problems which have a many-one reduction to A^i . The A hierarchy is defined as follows.

$$A^* = \bigcup_{i \in \mathbb{N}} A^i$$

Main Theorems. Our second contribution is a series of structural results:

Theorem 5.2. $\text{PPA}^* = \text{PPA}$.

Theorem 5.8. $\text{PLS}^* = \text{PLS}$.

Theorem 5.10. $\text{LOSSY}^* = \text{LOSSY}$.

Besides defining hierarchies within TFNP, we also study the structural properties of these new hierarchies. To our surprise, we find that—unlike the polynomial hierarchy—the PPA, PLS, and LOSSY hierarchies all collapse to their first levels. While these collapses might be perceived as straightforward conceptually (e.g. the conceptual idea behind $\text{PLS}^{\text{PLS}} = \text{PLS}$ can be found in the language of bounded arithmetic in [BK94]), the actual proof requires great care (again due to the fact that a TFNP problem defines a relation and not a function). In fact, each of these three classes requires different technical treatments that appeal to the unique properties of each class.

One consequence of our collapses is that, e.g., when reducing a problem to any of these classes, we are free to assume oracle access to any problem in that same class. This indicates that these three classes could be more powerful than previously imagined. These results also stand in stark contrast to the fact that PPP is not even Turing-closed under black-box reductions [FGPR24], a seemingly much weaker property than self-lowness.

Further Applications. In Section 6, we show how to apply our new TFNP subclasses to study the complexity of well-known problems. One of the consequences of the fact that PPA is self-low and that factoring is (likely) in PPA means that one can generally assume access to a factoring oracle when reducing a problem to a PPA-complete problem (assuming the Generalized Riemann Hypothesis). As an instantiation of this observation, we show [theorem 6.5](#): Let FACTOR be the problem of finding a non-trivial prime factor of an integer (or declaring none exist) and let WEAK-BERTRAND be the problem of generating a prime between 2^n and 2^{32n} given 1^n as input. The following two theorems may provide a new way of attacking the longstanding open problem of pinpointing the complexity of WEAK-BERTRAND in TFNP.

Theorem 6.5. *If WEAK-BERTRAND is in $\text{PPA}^{\text{FACTOR}}$, then the Generalized Riemann Hypothesis implies that WEAK-BERTRAND is in PPA.*

Theorem 6.4. *Under the Generalized Riemann Hypothesis, WEAK-BERTRAND is in $\text{LOSSY}^{\text{PPA}}$, $\text{LOSSY}^{\text{PPP}}$, $\text{PPADS}^{\text{PPA}}$, $\text{PPADS}^{\text{PPP}}$, and PPP^{PPP} .*

1.2 Internal Consistency: Technical Nuisance or Structural Property?

The Motivation. As mentioned earlier, the fact that a TFNP problem defines a relation rather than a function introduces complications in defining TFNP oracle classes. This is because the proof of totality of many classic TFNP subclasses relies on the circuit being a function! For example, it only makes sense to instantiate the pigeonhole principle in PIGEON if the underlying circuit defines a function. Otherwise one would need to contend with quirky statements such as $C^{\text{B}}(x) \neq C^{\text{B}}(x)$.

As such, we wish C^{B} to define a function (at least from the perspective of the verifier V'). There are at least two ways to enforce this: One is to ensure that the evaluation of the outer circuit (by the verifier V') outputs the same answer for the same input, regardless of the answers to the oracle queries. We call this property *External Consistency*. The other, which we use in this work, is to ensure that the answers to the oracle queries are consistent with one another. We call this property *Internal Consistency*.

It is not hard to see that Internal Consistency implies External Consistency, and it is therefore a stricter definition. One may wonder if this stricter definition is necessary, and if adopting this stricter definition buys us anything useful.

The Benefits. One major benefit of the stricter definition is that it is robust to the choice of complete problem for A or B (up to FP^B reductions). In particular, if there is a black-box reduction from A to A' and a reduction from B to B' , then there is a black-box reduction from A^B to $A'^{B'}$. This is not true under the weaker definition of External Consistency, and we provide an explicit counterexample in [Section 4.4](#). We believe that such robustness is desirable and serves as a strong argument for adopting the stricter definition.

Moreover, since our principal results are upper bounds/containments (instead of lower bounds/separations), proving them under the stricter internally consistent definition yields a stronger result.

The Downsides. The stronger requirement also has a genuine cost. As an example, we cannot directly apply Turing-closure of B and replace adaptive B queries by a single B query: two such queries may each be valid in isolation yet decode to different answers on a shared original query, violating internal consistency. Preserving internal consistency therefore requires more careful treatment of the oracle queries.

Structural Implications. One may still ask if the whole internal consistency requirement is a technical nuisance, and if there is a cleaner way to define TFNP oracle classes that retains all the desirable benefits and avoids the downsides.

We do not have a definitive answer to this question, but in the quest of proving self-lowness of PPA, PLS, and LOSSY under the internally consistent definition, we encountered an interesting phenomenon where three collapse proofs enforce internal consistency in rather different ways, using the involutive structure of PPA, the monotone structure in PLS, and the compression/decompression structure of LOSSY. (We encourage the reader to look at the proofs.)

It is somewhat surprising that enforcing internal consistency in different TFNP subclasses requires leveraging different properties of the underlying mathematical principles. We view this diversity as motivation for further study of whether internal consistency reflects a broader structural feature of TFNP subclasses, which could potentially lead to a better understanding of the structure of TFNP.

1.3 Utility of TFNP Hierarchies

We now discuss how our proposed theory of hierarchies fits into the existing TFNP literature and how it may act as a framework to further our understanding of TFNP.

Power of TFNP Our understanding of TFNP was previously relatively flat: focused on containments/separations/intersections between subclasses. We ask ourselves about the new dimension of hierarchies and primarily explore collapses and self-lowness in this work. As noted above, we hope that future work will use our definitions to explore topics beyond self-lowness. In particular, one can imagine asking if analogous theorems for other hierarchies (e.g. PH) hold in the TFNP world.

On a philosophical note, hierarchies within TFNP differ from PH in at least one important respect from our perspective. Although the notion of $TFNP^{TFNP}$ does not mean much formally as TFNP is not believed to have complete problems, our work can be taken to say something along the lines of $TFNP^{TFNP} = TFNP$ (this is very much believed not to be the case for NP, which forms the base of PH). In particular, we see in [Section 3](#) that oracle access to a TFNP problem does not

give you any power beyond TFNP since solutions to oracle queries always have witnesses (which is not true for NP).

One could draw an analogy from our self-lowness results to Turing-closure, a concept closely related to self-lowness. We say that a TFNP subclass A is Turing-closed if $FP^A = A$, or equivalently, the existence of a Turing reduction to A implies the existence of a many-one reduction to A . It is not hard to see that for any nontrivial class A , self-lowness ($A^A = A$) implicitly requires A to be Turing-closed in the first place, since simply evaluating the input circuit to A^A is a FP^A problem. In other words, self-lowness is a stronger property than Turing-closure under our definition (see [observation 4.5](#)) and the self-lowness of a class indicates that it has much more power than previously believed (more so than Turing-closure).

The Turing-closure of PLS, PPA, PPAD, and PPADS was shown in [\[BJ12\]](#). LOSSY was shown to be Turing-closed in [\[LPT24\]](#). On the other hand, [\[FGPR24\]](#) showed that PPP is not Turing-closed under black-box reductions.

Classification of problems and upper bounds We are now equipped with a more powerful tool for classifying computational problems into subclasses of TFNP. In particular, we are essentially free to use any TFNP subclass as a subroutine – just invoke the corresponding oracle. Moreover, the self-lowness of, e.g. PLS, tells us that if we are reducing to PLS, a PLS oracle is truly free.

[\[Kor22\]](#) considered the problem LOSSY with access to an MCSP (minimum circuit size problem) oracle and showed that solving $LOSSY^{MCSP}$ allows one to generate hard truth tables. This is closely related to our work, except MCSP is not a TFNP problem. We note that since MCSP is a decision problem, the reduction of [\[Kor22\]](#) to $LOSSY^{MCSP}$ does not need our definitions from [Section 3](#). [\[Kor22\]](#) also considered LOSSY with a factoring oracle. They showed that if one can solve $LOSSY^{ALLFACTOR}$ (where ALLFACTOR is the problem of outputting the full prime factorization of a number), then one can deterministically generate large primes, a longstanding open problem. Here again, the setting of [\[Kor22\]](#) is more straightforward than our setting since one can simply assume an oracle gate to a problem with unique solutions. They did not consider the case where the search problem associated to the oracle is simply in TFNP.

Separations and lower bounds We introduce new techniques for proving separations and lower bounds between subclasses of TFNP:

- If some subclasses form a true hierarchy, this gives us even more evidence that the class does not equal FP since this would collapse the hierarchy, much in the style of PH.
- [\[LLR24\]](#) studied the TFNP class $rwPHP(PLS)$ in the decision tree model, which consists of problems reducible to the retraction weak pigeonhole principle where the retraction function is in PLS. To a certain extent one could interpret this class as $LOSSY^{PLS}$ in our language, but in the decision-tree model, or in the fully black-box setting. [\[LLR24\]](#) showed that this class captures the problem of proving certain restricted lower bounds for Resolution proofs.

We do note that most separation results were proved in the decision-tree model, and our definition might not directly apply there. But we are hopeful that useful ideas could be borrowed between different models.

1.4 Open Questions

1. Are PPAD, PPADS, CLS, UEOP, and PPA_k (for $k \geq 3$) self-low?
2. The newly defined complexity class PLC (polynomial long choice) is meant to capture the combinatorial principle of the iterated pigeonhole principle [PYP22]. [PYP22] ask whether $\text{PLC} \subseteq \text{FP}^{\text{PPP}}$. The framework of hierarchies within TFNP allows us to ask more general questions. Is PLC contained in PPP^{PPP} , is PPP^{PPP} contained in PLC, or are the two classes incomparable?

2 Preliminaries

2.1 TFNP

We begin by formally defining a search problem and TFNP search problems.

Definition 2.1. A search problem is a binary relation $\mathcal{R} \subseteq \{0,1\}^* \times \{0,1\}^*$ where we say that y is a solution to x iff $(x, y) \in \mathcal{R}$.

Definition 2.2 (TFNP). A total NP search (TFNP) problem is a relation $\mathcal{R} \subseteq \{0,1\}^* \times \{0,1\}^*$ such that the following properties hold.

- Polynomial: For all $(x, y) \in \mathcal{R}$, $|y| \leq \text{poly}(|x|)$.
- Totality: For all inputs x , there is a solution o such that $(x, o) \in \mathcal{R}$.
- FNP membership: There exists a $\text{poly}(|x|, |o|)$ time algorithm V such that $V(x, o) = 1$ if and only if $(x, o) \in \mathcal{R}$.

When dealing with TFNP problems, we are generally concerned with many-one reductions. One should think of these as reductions with a single oracle call.

Definition 2.3. Let $\mathcal{R}$ and $\mathcal{Q}$ be search relations. A many-one reduction from $\mathcal{R}$ to $\mathcal{Q}$ consists of two polynomial-time algorithms f and g such that, for every $x, y \in \{0,1\}^*$,

$$(f(x), y) \in \mathcal{Q} \implies (x, g(x, y)) \in \mathcal{R}.$$

Alternatively, there is the notion of a Turing reduction, where one can make multiple oracle calls. Informally, we say that a class is Turing-closed if Turing reductions give us no more power than many-one reductions.

Definition 2.4 ([FGPR24]). We say that a search problem $\mathcal{R}$ is Turing-closed if any problem which is polynomial time reducible to $\mathcal{R}$ via multiple calls to an oracle for a problem in $\mathcal{R}$ is also polynomial time reducible to $\mathcal{R}$ using a single call to an oracle for a problem in $\mathcal{R}$. We say a complexity class with a complete problem is Turing-closed if its complete problem is Turing closed.

Quite crucially for us, a variety of important TFNP subclasses are known to be Turing-closed.

Lemma 2.5 ([BJ12]). PPA, PPAD, PPADS, and PLS are Turing-closed.

Lemma 2.6 ([LPT24]). LOSSY is Turing-closed.

2.2 Some TFNP subclasses

Here we review some TFNP subclasses and give some informal intuition regarding their structures. All of these classes of interest will involve a polynomial size circuit implicitly encoding an exponential size object. The goal will be to find (an efficiently verifiable) structure which must exist in this exponential size object. We note that we will freely switch between a binary string or set $(\{0, 1\}^n)$ and its integer representation $([2^n])$. We encourage the reader to not be overly concerned with this technical detail.

We begin by defining PPA with respect to the complete problem LONELY [BCE⁺95].

Definition 2.7. The problem LONELY is defined as follows. The input is a circuit $C : [N] \rightarrow [N]$ such that N is odd (which may be enforced syntactically). Output $w \in [N]$ such that either $C(w) = w$ or $C(C(w)) \neq w$. PPA is defined as all TFNP problems which are many-one reducible to the problem LONELY.

In the LONELY problem, C implicitly encodes a pairing of vertices $[N]$ where $C(x) = y$ means that x and y are paired. The goal is to output either an unpaired vertex (w such that $C(w) = w$) or a witness that C does not encode a pairing (w such that $C(C(w)) \neq w$). LONELY's totality follows from the fact that if N is odd, then no perfect pairing of all N vertices can exist. We will not do so here, but one can define the more general family of classes PPA_k (where PPA is equivalent to PPA_2) via the complete problem BIPARTITE-MOD-K [Pap94, Hol21].

Definition 2.8 (PPAD and ENDOFLINE). The problem ENDOFLINE is defined as follows. Given $S : \{0, 1\}^n \rightarrow \{0, 1\}^n, P : \{0, 1\}^n \rightarrow \{0, 1\}^n$ such that $S(0) \neq 0$ and $P(0) = 0$, output x such that $P(S(x)) \neq x$ or $x \neq 0$ such that $S(P(x)) \neq x$. PPAD is defined as all TFNP problems which are many-one reducible to ENDOFLINE.

The circuit for ENDOFLINE should be viewed as specifying a directed graph. The input gives us a successor circuit S and a predecessor circuit P . We say that the graph implicitly defined by S, P has an edge from x to y if $S(x) = y$ and $P(y) = x$. Notice that there is no edge leading to 0 in any such graph since $P(0) = 0$. Therefore, there must be a node x which has no outgoing edges, which implies $P(S(x)) \neq x$. This can be thought of as a sink of a line in the graph defined by S, P . Notice that ENDOFLINE also allows for a solution x such that $S(P(x)) \neq x$. This should be interpreted as the beginning of a new line in the graph implicitly encoded by S, P (one other than the one starting at 0), a node which has an edge out but no incoming edges.

Definition 2.9 (PPADS and SINKOFLINE). The problem SINKOFLINE is defined as follows. Given $S : \{0, 1\}^n \rightarrow \{0, 1\}^n, P : \{0, 1\}^n \rightarrow \{0, 1\}^n$ such that $S(0) \neq 0$ and $P(0) = 0$, output x such that $P(S(x)) \neq x$. PPADS is defined as all TFNP problems which are many-one reducible to SINKOFLINE.

SINKOFLINE is almost the same as ENDOFLINE except that we only allow one type of solution: a sink in the graph implicitly defined by S, P . Beginnings of a new line are no longer solutions. One can also define the following useful PPADS-complete problem (under blackbox reductions) which we will use in Section 6.

Definition 2.10. The problem INJECTIVE-PIGEON is defined as follows. Given $C : [2^n] \rightarrow [2^n - 1], D : [2^n - 1] \rightarrow [2^n]$, output x such that $D(C(x)) \neq x$.

Definition 2.11 (PLS and SINK-OF-DAG). The SINK-OF-DAG problem is defined as follows. Given $\text{poly}(n)$ size circuits $S : [2^n] \rightarrow [2^n]$ and $V : [2^n] \rightarrow [2^n]$ such that $S(0) \neq 0$, find v such that $S(v) \neq v$ and either $S(S(v)) = S(v)$ or $V(S(v)) \leq V(v)$. PLS is the set of all TFNP problems which are many-one reducible to SINK-OF-DAG.

SINK-OF-DAG should be viewed as encoding a local search problem. There are two circuits, a successor circuit and a value circuit. At every point v in the space, we hope that the successor function S leads us to a (different) point which has a higher value ($V(S(v)) > V(v)$). A solution to SINK-OF-DAG is a point such that this condition is violated ($S(v) \neq v$ but $V(S(v)) \leq V(v)$), or one which acts as a sink in the gradient ascent process ($S(v) \neq v$ but $S(S(v)) = S(v)$).

Definition 2.12 (PWPP and WEAK-PIGEON). The WEAK-PIGEON problem is defined as follows. Given a $\text{poly}(n)$ size circuit $C : \{0, 1\}^n \rightarrow \{0, 1\}^{n-1}$, output distinct $x_1, x_2 \in \{0, 1\}^n$ such that $C(x_1) = C(x_2)$. PWPP is the set of all TFNP problems which are many-one reducible to WEAK-PIGEON.

PWPP should be considered the algorithmic analogue of the weak pigeonhole principle. We know that a collision exists in C since C is compressing; WEAK-PIGEON asks us to find a collision.

Definition 2.13. The problem $f(n)$ -LOSSY is defined as follows. Given $C : \{0, 1\}^n \rightarrow \{0, 1\}^{f(n)}$ and $D : \{0, 1\}^{f(n)} \rightarrow \{0, 1\}^n$, output $x \in \{0, 1\}^n$ such that $D(C(x)) \neq x$. We refer to $(n/2)$ -LOSSY simply as LOSSY. LOSSY is defined as all TFNP problems which are many-one reducible to LOSSY.

[Kor22] defined LOSSY (which they call LOSSY CODE), but did not define the complexity class LOSSY. We believe this is the correct and natural definition for LOSSY. One should view the inputs to $f(n)$ -LOSSY as consisting of a compressor circuit C and a decompressor circuit D . The goal is to find a string that is not compressible by this compression scheme. Such a string must exist since the compression scheme is lossy. The following lemma shows that the compression factor of C and D does not matter (up to a polynomial factor).

Lemma 2.14 ([Kor22]). *$f(n)$ -LOSSY is many-one equivalent to LOSSY for any efficiently computable $f(n) < n$ and $f(n) = \text{poly}(n)$.*

Definition 2.15 (FACTOR). FACTOR is defined as follows. Given an n bit integer x , output 0^{n-1} if x is prime or $x \in \{0, 1\}$. Otherwise, output a non-trivial divisor $y \in \{0, 1\}^{n-1}$ where y divides x .

Notice that an n bit composite number has an $n - 1$ bit non-trivial divisor. Therefore, the size of the solution to FACTOR on an n bit input is bounded above by $n - 1$. Furthermore, as was shown in [AKS04], testing if x is prime can be done in polynomial time.

3 Definition

3.1 Oracle Gates

We will be considering some TFNP problem A given access to an oracle for a TFNP problem B. However, this notion will only make sense (at least as we define it) when A is what we term a “circuit problem”.

Definition 3.1. We say A is a circuit problem if the input to A is a $\text{poly}(n)$ size circuit $C : \{0, 1\}^n \rightarrow \{0, 1\}^{\text{poly}(n)}$ and possibly some other $\text{poly}(n)$ size input $a \in \{0, 1\}^{\text{poly}(n)}$.

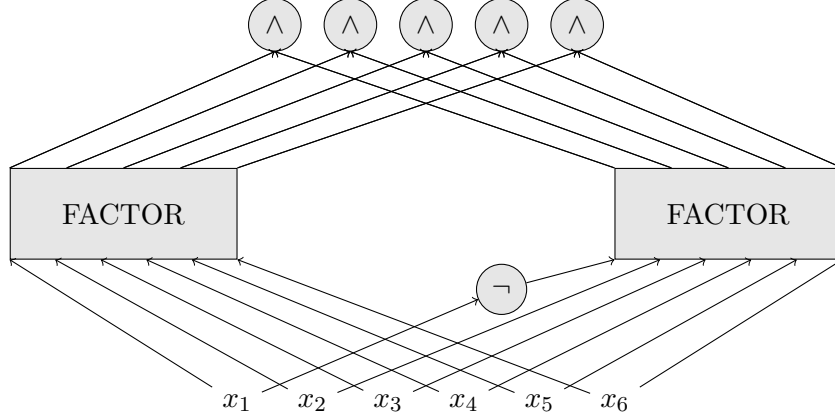


Figure 1: A $\text{WEAK-PIGEON}^{\text{FACTOR}}$ instance

Throughout, when a circuit problem is said to have a black-box verifier, we also require its totality to be black-box: the existence of a solution may depend only on the input-output behavior of the circuit, and not on the internal structure of the circuit.

We note that the complete problems for all the major TFNP classes are in fact circuit problems (Section 2.2). There is a minor subtlety that some problems, like SINK-OF-DAG, take as input more than one circuit S, V . In these cases, we treat the circuits as a single circuit SV , where $SV(x) = S(x) \parallel V(x)$. As an example, SINK-OF-DAG is a circuit problem since it has as input circuits S and V (which we treat as a single circuit). We assume a canonical gate evaluation order over circuits. Since B is a TFNP problem, the length of any solution on input x is bounded by some polynomial $p(|x|)$. We will insist that all B oracle gates G in C^B have the form $G : \{0, 1\}^z \rightarrow \{0, 1\}^{p(|z|)}$. This is to ensure that a valid solution always fits within the number of output wires of G .

Example 3.2. To illustrate some of the concepts around TFNP classes with oracles, we will re-view concepts by instantiating them with the $\text{WEAK-PIGEON}^{\text{FACTOR}}$ instance of Figure 1. For the remainder of this section, we refer to the circuit defined in Figure 1 as $T : \{0, 1\}^6 \rightarrow \{0, 1\}^5$.

3.2 Evaluating Oracle Circuits with Auxiliary Oracle Answers: C^*

We now reiterate why defining A^B for TFNP problems A and B is more challenging than defining A^B when B is a decision problem, like SAT. When the oracle gates in C^{SAT} are SAT gates, the behavior of the circuit on any input x is well-defined since every input to an oracle gate has a unique output (0 or 1). However, when the oracle gates in C^{PIGEON} are gates for a search problem (e.g., PIGEON), the behavior of the circuit C^{PIGEON} on an input x may not be well-defined, since the evaluation of the oracle gates could have many possible solutions. If the oracle gates of C are for a problem with unique solutions (e.g., finding all prime factors of a number), this problem does not occur, but we want to work with full generality. Towards addressing this issue, we define a helper function C^* which simulates C^B with auxiliary inputs that serve as solutions to its oracle gates.

Definition 3.3. Let $C^O : \{0, 1\}^n \rightarrow \{0, 1\}^m$ be an oracle circuit where O is a TFNP problem, C^O

contains t oracle gates, and the i^{th} oracle gate of C^O has an s_i -bit output. We now define

$$C^* : \{0, 1\}^n \times \prod_{i=1}^t \{0, 1\}^{s_i} \rightarrow \{0, 1\}^m \cup \{\perp\}.$$

C^* on input $(x, w_1, \dots, w_t)$ simulates the evaluation of C^O on x . At the i^{th} instance when C^O must evaluate an oracle gate, say on input u , we check if w_i has a suffix that is a valid solution to O on input u . If it is not, then $C^*(x, w_1, \dots, w_t) = \perp$ and the simulation terminates; otherwise, the simulation of C^O continues by using w_i as the output of oracle gate i . If the simulation does not terminate prematurely and obtains a simulated m -bit output of $C^O(x)$, $C^*(x, w_1, \dots, w_t)$ returns that output.

$C^*(x, w_1, \dots, w_t)$ is the evaluation of C^O on x given that the output of oracle gate i is w_i . In other words, $(x, w_1, \dots, w_t)$ contains all the information necessary to lock in an evaluation of C^O on x .

We note that w_i may not itself be a solution to the query made to oracle gate i during the evaluation of $C^O(x)$, but some suffix of w_i is a valid solution. We do this because oracle gates have a fixed number of output wires but we still wish to allow for variable length solutions to queries to oracle gates. This is without loss of generality, as the circuit can determine for itself which suffix is a solution to the query it made to the oracle gate. Also, for the rest of the presentation, when we say w_i is a solution for some oracle query u , we mean that some suffix w'_i where $w_i = r \| w'_i$ is a solution to u . Moreover, for some circuit C_u defined on $\{0, 1\}^{|w'_i|}$, we abuse the notation and define $C_u(w_i) := r \| C_u(w'_i)$ for simplicity.

Example 3.4 (Continuation of [example 3.2](#)). *We now provide some intuition for what T^* does by probing it on a few values. We assume that the gates of T are evaluated from left to right. In the following examples, we freely switch between integers and their binary representations. Consider the evaluation of $T^*(10, 2, 17)$. We begin by simulating T on the input integer 10, which is 001010 in binary. This leads us to query 001010 at the left FACTOR gate. We see that 2 is indeed a factor of 10, so we continue the simulation of T by assuming that the LEFT oracle gate outputs 2 (00010 in binary). The simulation then queries 101010 (42 in binary) at the right FACTOR gate. We see that 17 is not a factor of 42, so $T^*(10, 2, 17) = \perp$.*

Consider the evaluation of $T^(10, 2, 21)$. Again, we begin by simulating T on the input integer 10, which is 001010 in binary. This leads us to query 001010 at the left FACTOR gate. We see that 2 is indeed a factor of 10, so we continue the simulation of T by assuming that the left oracle gate outputs 2 (00010 in binary). The simulation next queries 101010 (42 in binary) at the right FACTOR gate. We see that 21 is indeed a factor of 42, so the right FACTOR gate outputs 21 (10101 in binary). Therefore, $T^*(10, 2, 21) = 00010 \wedge 10101 = 00000$.*

3.3 The Full Definition

We are now ready to define what a solution to A^B looks like.

Definition 3.5. Let A and B be TFNP problems and A be a circuit problem with a black-box verifier $V : \{0, 1\}^n \times \{0, 1\}^{\text{poly}(n)} \rightarrow \{0, 1\}$, one which only queries the circuit input to A in a black-box manner. The input to A^B is defined the same as A except the circuit input C^B of A contains t B oracle gates. The verifier for A^B , $V' : \{0, 1\}^n \times \{0, 1\}^{\text{poly}(n)} \rightarrow \{0, 1\}$ is defined as follows. V'

takes as input $(C^B, a), (y, w_{1,1}, \dots, w_{\text{poly}(n),t})$. V' simulates the computation of $V((C^B, a), y)$ and on the i^{th} evaluation of C^B , say on input x , the black-box verifier V' receives $C^*(x, w_{i,1}, w_{i,2}, \dots, w_{i,t})$. If at any point, $C^*(x, w_{i,1}, w_{i,2}, \dots, w_{i,t}) = \perp$ or V' finds the sub-witnesses $w_{i,j}$ to not be *internally consistent* (which we define below), V' outputs 0. Otherwise, V' outputs the result of V after the simulation of V terminates.

We say that the $w_{1,1}, \dots, w_{\text{poly}(n),t}$ are not *internally consistent* (with respect to the simulation) if in the execution of V' , there exist two oracle queries made on the same input u and the simulation uses answer $w_{a,b}$ to the first oracle gate query and $w_{a',b'}$ as the answer to the second oracle gate query and $w_{a,b} \neq w_{a',b'}$.

In particular, a solution to A^B is a solution to A along with all the answers to the B oracle gate queries that are made when verifying such a solution. We allow any input/output of the B gates as long as it is actually a solution to B and the behavior is consistent across multiple evaluations of B . In some sense, we are quantifying over all instantiations (i.e. all functions consistent with the relation defined by B) of a B oracle. Under [definition 3.5](#), a reduction from some problem C to A^B should work for all instantiations of the oracle gates for B . An alternative perspective is that we are delegating the work of fixing the B oracle to the solution $y, w_{1,1}, \dots, w_{\text{poly}(n),t}$, particularly the $w_{1,1}, \dots, w_{\text{poly}(n),t}$.

Example 3.6 (Continuation of [example 3.4](#)). Using [definition 3.5](#), we see that a solution to $\text{WEAK-PIGEON}^{\text{FACTOR}}$ on input T would be distinct x_1, x_2 and internally consistent w_1, w_2, w_3, w_4 such that $T^*(x_1, w_1, w_2) \neq \perp$ and $T^*(x_1, w_1, w_2) = T^*(x_2, w_3, w_4)$. We will now try to find solutions to our $\text{WEAK-PIGEON}^{\text{FACTOR}}$ instance T . One seemingly obvious solution is $(x_1, x_2) = (0, 32), (w_1, w_2) = (0, 2), (w_3, w_4) = (8, 0)$. This appears to be a solution since $T^*(0, 0, 2) = 0$ and $T^*(32, 8, 0) = 0$. However, observe that this solution is not internally consistent. In particular, when evaluating $T^*(0, 0, 2)$, we assume that FACTOR on input 32 evaluates to 2, but when evaluating $T^*(32, 8, 0)$, we assume that FACTOR on input 32 evaluates to 8.

An actual solution for $\text{WEAK-PIGEON}^{\text{FACTOR}}$ on instance T (which we leave to the reader to confirm) is $(x_1, x_2) = (0, 32), (w_1, w_2) = (0, 2), (w_3, w_4) = (2, 0)$.

Definition 3.7. Let $A, B \subseteq \text{TFNP}$, let A and B be canonical complete problems for A and B respectively, and let A be a circuit problem with a black-box verifier. A^B is then defined as all problems which are many-to-one reducible to A^B .

With the definition of A^B in place, the definition of hierarchies follows naturally.

Definition 3.8. Let A be a TFNP subclass and A be the canonical A -complete circuit problem. We define $A^1 = A$, $A^i = A^{(A^{i-1})}$, and A^i as all problems which have a many-one reduction to A^i . The A hierarchy is defined as follows.

$$A^* = \bigcup_{i \in \mathbb{N}} A^i$$

3.4 Another Helpful Evaluation: C_*

We now define an alternative way to evaluate C^O on x that is significantly different from C^* . While C^* may be easier to reason about, C_* will be useful in enforcing internal consistency of our solutions.

Definition 3.9. Let $C^O : \{0, 1\}^n \rightarrow \{0, 1\}^m$ be an oracle circuit where O is a TFNP problem, C^O contains t oracle gates, and the i^{th} oracle gate of C^O has an s_i -bit output. We now define

$$C_* : \{0, 1\}^n \times \prod_{i=1}^t \{0, 1\}^{s_i} \rightarrow \{0, 1\}^m \cup (\{0, 1\}^{\lceil \log t \rceil} \times \{0, 1\}^{\text{poly}(n)}).$$

We maintain an ongoing set M , initially empty. C_* on input $(x, w_1, \dots, w_t)$ simulates the evaluation of C^O on x . At the i^{th} instance when C^O must evaluate an oracle gate G_i , say on input u , if 0 is a valid solution to G_i on u , we continue simulating C^O using 0 as the output of oracle gate G_i . Otherwise, we find the minimum j such that w_j is a solution for O on input u and update $M \leftarrow M \cup \{j\}$. We refer to this as the i^{th} witness used to evaluate $C_*(x, w_1, \dots, w_t)$. If no such j exists when evaluating oracle gate i , let k be the smallest index such that $k \notin M$ and we output (k, u) . We refer to this as an error output and it signals where the simulation fails. Otherwise, we continue simulating C^O by using w_j as the output of oracle gate i . If the simulation does not terminate prematurely and obtains a simulated m -bit output of $C^O(x)$, $C_*(x, w_1, \dots, w_t)$ returns that output. We refer to this as a result output. We say w_j was used in the evaluation of $C_*(x, w_1, \dots, w_t)$ if $j \in M$ when the procedure for C_* terminates.

The key difference between C^* and C_* is that instead of using the subsequent w_i as the solution to an oracle gate query in the simulation of C^O as we did in C^* , we find the first w_i which is sufficient to answer the oracle gate query and use that in the simulation of C^O . We refer to this as evaluation by first witnesses since we use the first solution possible whenever evaluating an oracle gate. We note that such evaluation naturally enforces internal consistency.

Example 3.10 (Continuation of [example 3.6](#)). *To understand how T_* is evaluated, consider $T_*(10, 2, 21)$. We begin by simulating T on the input 10, or 001010 in binary. We query the left FACTOR gate on 10 and find that 2 is indeed a factor of 10 (and in particular, the first value which is a factor among 2, 21), so we assume that the left FACTOR gate outputs 00010. We next evaluate the right FACTOR gate on 42 and find that 2 is a factor of 42 (and in particular, the first value which is a factor among 2, 21), so we continue the evaluation assuming that the right FACTOR gate outputs 2. This leads the simulation of T to output $00010 \wedge 00010 = 00010$, or 2. Therefore $T_*(10, 2, 21) = 2$. The observant reader will note that this is a different result from [Section 3.2](#), where $T^*(10, 2, 21) = 0$.*

Finally, let us consider an example where T_ outputs an error result. Consider $T_*(14, 7, 9)$. We query the left FACTOR gate on 14 and find that 7 is indeed a factor of 14 (and in particular, the first value which is a factor among 7, 9), so we assume that the left FACTOR gate outputs 00111. We also add 1 to our set M . We next evaluate the right FACTOR gate on 46 and find that neither 7 nor 9 is a factor of 46. We therefore terminate and $T_*(14, 7, 9) = (2, 46)$, indicating that w_2 is the next unused witness.*

4 Robustness of Definition

Having defined A^B for TFNP problems, we critique our definitions and show that they have several desirable properties.

4.1 Hierarchies Remain in TFNP

We first show that A^B and its corresponding class is in TFNP.

Theorem 4.1. *If A and B are TFNP problems and A is a circuit problem with a black-box verifier, then $A^B \in \text{TFNP}$.*

Proof. The efficiency of the verifier for A^B and the fact that A^B has polynomially bounded solutions are inherited from the efficiency of the verifier for A . To show totality, let us consider a problem B' defined as $\{(x, y) : y \text{ is the lexicographically first solution to } B \text{ on input } x\}$. Notice then that $A^{B'}$ is total since A is total. But of course, any solution to $A^{B'}$ is a solution to A^B . Therefore, A^B is total. $\square$

Observation 4.2. *Let $A, B \in \text{TFNP}$, let A and B be canonical complete problems for A and B respectively, and let A be a circuit problem with a black-box verifier. $A^B \subseteq \text{TFNP}$ and $A^* \subseteq \text{TFNP}$.*

4.2 The Choice of Complete Problem

To justify our definition, we now show that our choice for the complete problem for A does not matter when defining A^B as long as there exists a black-box reduction to/from that problem from/to the canonical complete problem for A (up to FP^B reductions). We note that [theorem 4.3](#) relies crucially on the fact that solutions to A^B are internally consistent and is the reason we insist on internal consistency in [definition 3.5](#).

Theorem 4.3. *Let A_1, A_2 and B be TFNP problems and A_1, A_2 be circuit problems. If A_1 is black-box many-one reducible to A_2 , then A_1^B is black-box many-one reducible to A_2^B under FP^B reductions.*

Proof. Let (f, g) be a pair of polynomial-time reduction algorithms such that on an A_1 instance C_1 , $f(C_1) = C_2$ generates an A_2 instance C_2 . And given a solution π_2 for C_2 , $g(C_2, \pi_2) = \pi_1$ outputs a solution π_1 for C_1 . In particular, we emphasize that (f, g) are black-box reduction algorithms. I.e., they evaluate C_1 on polynomially many inputs and construct C_2 as a circuit with C_1 -gates.

We define a pair of FP^B reduction algorithms (f_1^B, g_1^B) . In particular, the reduction algorithms maintain a polynomial-sized table T containing query-answer pairs to B .

On an A_1^B instance C_1^B , f_1^B starts by instantiating an empty lookup table T . Next, it simulates f : whenever f would evaluate C_1^B and hence query the oracle B on some input x , it checks if T contains the query-answer pair. If yes, it returns the answer stored in T . Otherwise, it uses its B oracle to obtain a query-answer pair, stores it in T and returns the answer. At the end of its simulation of f , it constructs a circuit C' (with C_1^B -gates). It further modifies C' as follows: the table T is hardwired into C' and any B gate in C' is modified to always look for an answer from T if available. The modified circuit C_2^B would be the output of f_1^B .

Given a solution (π_2, w_2) to C_2^B , g_1^B starts by simulating the verifier for A_2^B and adding the query-answer pairs from w_2 to the table T . Note that due to the additional modification in C_2^B , there will be no inconsistency between w_2 and T . Next, it simulates g and similarly maintains the table T in the same manner as f_1^B . At the end of the simulation, it outputs π_1 and oracle answers w_1 for verifying the solution.

Internal consistency follows from the use of table T and it remains to show the correctness of the reduction.

Consider any function b that is a restriction of B and is consistent with T . Since b is a function, we may view C_1^b as a vanilla circuit by hardwiring the truth table of b . Let $C_2^b := f(C_1^b)$. Since f_1^B simulates f and b is consistent with T , C_2^b is the same as C_2^B with the oracle gates switched and hence π_2 is a solution to C_2^b . Since g_1^B simulates g and by correctness of (f, g) , π_1 is a solution to C_1^b . Therefore, π_1 with the associated oracle answers w_1 must be a solution for C_1^B since C_1^b and C_1^B have the same behavior with respect to the verifier. $\square$

Corollary 4.4. *Let A_1, A_2, B , and C be TFNP problems. If all the following hold, then A_1^B reduces to C under many-one reductions.*

1. A_1 is black-box many-one reducible to A_2 .
2. A_2^B has a Turing reduction to C .
3. B has a Turing reduction to C .
4. C is Turing-closed.

Proof. By [theorem 4.3](#), A_1^B reduces to A_2^B under FP^B reductions. Since B has a Turing reduction to C , A_1^B reduces to A_2^B under FP^C reductions. This combined with the fact that A_2^B has a Turing reduction to C implies A_1^B reduces to C under FP^C reductions. Since C is Turing-closed, this implies that A_1^B reduces to C under many-one reductions. $\square$

Observation 4.5. *We note that PPP is likely not self-low since even evaluating the input circuit to PPP^{PPP} is an FP^{PPP} computation, and we do not believe $\text{FP}^{\text{PPP}} = \text{PPP}$ [FGPR24].*

We now show that our choice for the complete problem for B also does not matter as long as there exists a reduction to/from that problem from/to the canonical complete problem for B .

Theorem 4.6. *Let B_1, B_2 and A be TFNP problems. If B_1 is many-one reducible to B_2 , then A^{B_1} is many-one reducible to A^{B_2} .*

Proof. Let (f, g) be the many-to-one reduction from B_1 to B_2 . We now show a reduction from A^{B_1} to A^{B_2} . Let C^{B_1} be the input to A^{B_1} . Let G be a circuit with a B_2 oracle gate defined as follows. G on input x , computes $f(x)$, feeds $f(x)$ into a B_2 -gate, and applies g to the output of its B_2 -gate.

On an input $C_1^{B_1}$, the reduction from A^{B_1} to A^{B_2} simply replaces each B_1 gate with G and outputs the resulting circuit $C_2^{B_2}$. Given $(\pi, w_1, \dots, w_t)$ as a solution for $C_2^{B_2}$, the reduction outputs $(\pi, g(w_1), \dots, g(w_t))$ as a solution.

By the correctness of the reduction (f, g) , $g(w_i)$ are proper solutions to the B_1 oracle queries. Furthermore, by our construction of $C_2^{B_2}$, any evaluation on $C_1^{B_1}$ assisted by $g(w_i)$ would be exactly the same as evaluation on $C_2^{B_2}$ assisted by w_i . As such, $(\pi, g(w_1), \dots, g(w_t))$ is a valid solution to $C_1^{B_1}$.

It remains to verify that internal consistency is preserved. Assume, for the sake of contradiction, that $g(w_i) \neq g(w_j)$ are solutions to the same B_1 query x and hence $w_i \neq w_j$. However, w_i and w_j are solutions to the same B_2 query $f(x)$ in $C_2^{B_2}$. This contradicts the internal consistency of the solution $(\pi, w_1, \dots, w_t)$. $\square$

4.3 The Choice of Complete Problem (Continued)

We have shown that for the purpose of defining the complexity class A^B , the choice of complete problem chosen for A does not matter under FP^B reductions ([definition 3.7](#), [theorem 4.3](#)). However, it may matter if we are only considering many-one reductions rather than FP^B reductions. This makes the definition of A^B somewhat less robust than one might like.

However, we observe that this distinction is irrelevant for self-low classes since for any Turing-closed problem C , FP^C reductions to C and many-one reductions to C are equivalent.

Observation 4.7. *Let C, D be TFNP circuit problems that are black-box many-one reducible to each other. Let $C^1 = C$ and $C^i = C^{(C^{i-1})}$, similarly for D . If C is Turing-closed and C^C has a many-one reduction to C , then for all positive integers i , C^i is many-one equivalent to D^i under black-box reductions.*

Since we show that PPA, PLS, and LOSSY are self-low ([theorem 5.2](#), [theorem 5.8](#), [lemma 5.9](#)), the choice of complete problem for these classes does not matter at all, at least from the perspective of hierarchies.

4.4 Is Internal Consistency Necessary: an Example

We now present an example of what goes wrong if we do not enforce internal consistency in [definition 3.5](#): lifting of reductions as done in [theorem 4.3](#) breaks down. We will define two TFNP problems P and Q . P will reduce to Q via a many-to-one reduction, but without internal consistency we will not be able to lift this to a reduction from P^S to Q^S for some $S \in \text{TFNP}$.

Definition 4.8. The problem P is defined as follows. Given a circuit $C : [N] \rightarrow [N/10]$, output one of the following.

- (1) Two collisions of distinct values: distinct $x_1, x_2, y_1, y_2 \in [N]$ such that $C(x_1) = C(x_2) \neq C(y_1) = C(y_2)$.
- (2) A 5-wise collision: distinct $x_1, \dots, x_5 \in [N]$ such that $C(x_1) = C(x_2) = \dots = C(x_5)$.

Definition 4.9. The problem Q is defined as follows. Given a circuit $D : [N+3] \rightarrow [N/10]$, output one of the following.

- (1) Two 3-wise collisions of distinct values: distinct $x_1, x_2, x_3, y_1, y_2, y_3 \in [N+3]$ such that $D(x_1) = D(x_2) = D(x_3) \neq D(y_1) = D(y_2) = D(y_3)$.
- (2) An 8-wise collision: distinct $x_1, \dots, x_8 \in [N+3]$ such that $D(x_1) = D(x_2) = \dots = D(x_8)$.

It is not hard to verify that both problems are in TFNP. We note that in both problems, type-(1) solutions are the more interesting cases and type-(2) solutions are meant to ensure that the problems are total.

We now present a simple many-to-one reduction (f, g) from P to Q .

Theorem 4.10. *P is many-to-one reducible to Q .*

Proof. Given an instance $C : [N] \rightarrow [N/10]$ for P , the reduction constructs $D := f(C)$ as $D(x) := C(x \bmod N)$.

Next, we describe the second half of the reduction: the transformation g on a solution of $Q(D)$.

- (1) Suppose g receives a type-(1) solution $(x_1, x_2, x_3, y_1, y_2, y_3)$. Let $x'_i = x_i \bmod N$ and $y'_i = y_i \bmod N$ for $i \in \{1, 2, 3\}$. Since x_1, x_2, x_3 are distinct, there must exist $x'_i \neq x'_j$. Moreover $C(x'_i) = D(x_i) = D(x_j) = C(x'_j)$. By the exact same argument, g finds another colliding pair $y'_k \neq y'_\ell$ and $C(y'_k) = C(y'_\ell)$. The reduction outputs $x'_i, x'_j, y'_k, y'_\ell$ as a type-(1) solution.

We note that for any $i, j \in \{1, 2, 3\}$, we have $x'_i \neq y'_j$ since $C(x'_i) = D(x_i) \neq D(y_j) = C(y'_j)$. We will see that this guarantee does not hold in the absence of internal consistency when relative to an oracle S .

- (2) Suppose g receives a type-(2) solution $(x_1, \dots, x_8)$. It removes any $x_i \in \{N, N+1, N+2\}$ and outputs any five of the remaining x_i as a type-(2) solution. Note that at most 3 items are removed and each remaining x_i satisfies $C(x_i) = D(x_i)$.

□

We claim that the above reduction fails to reduce P^S to Q^S , in the following scenario. For simplicity assume that the circuit C^S has exactly one oracle gate. Now a type-(1) solution to Q^S may look like $D(0, w_0) = D(1, w_1) = D(2, w_2) \neq D(N, w'_0) = D(N+1, w'_1) = D(N+2, w'_2)$. Directly applying g leads to a solution looking like

$$C(0, w_0) = C(1, w_1) \neq C(1, w'_1) = C(2, w'_2) .$$

Notice that this violates the principle that C^S should behave like a function. We emphasize that we are not claiming that there is no reduction from P to Q which directly lifts to a reduction from P^S to Q^S for all $S \in \text{TFNP}$, merely that without internal consistency, we cannot guarantee that all such reductions lift.

5 Self-lowness

5.1 PPA Self-lowness

We show that PPA is self-low.

Lemma 5.1. $\text{LONELY}^{\text{LONELY}}$ reduces to LONELY under black-box many-one reductions.

Proof. Recall that we work with the definition of LONELY that consists of N vertices where N is odd. Moreover, for the oracle queries, we assume that each query is specified by a string in $Q := \{0, 1\}^{m_1} \setminus \{0\}$ and each solution to a query is specified by a string in $W := \{0, 1\}^{m_2} \setminus \{0\}$. And the base problem takes in strings from $S := \{0, 1\}^n \setminus \{0\}$ and outputs strings in S . All these assumptions are simply a matter of syntactic encoding choices and do not affect the final result. We therefore assume without loss of generality that the input to the reduction is a circuit $C^{\text{LONELY}} : S \rightarrow S$. The vertex set for the LONELY instance to which we reduce is then

$$S \times \prod_{i=1}^{2t} (Q \times W) ,$$

which has an odd number of vertices.

Next we construct the circuit C' , which has a very delicate behavior on input $(x_1, q_1, w_1, \dots, q_{2t}, w_{2t})$. Let $x_2 = C^*(x_1, w_1, \dots, w_t)$. C' simulates $C^*(x_1, w_1, \dots, w_t)$ and then

$C^*(x_2, w_{t+1}, \dots, w_{2t})$ as follows. Suppose that the simulation is about to make the i -th (where i starts from 1) oracle query (which will purportedly be answered by w_i). C' checks the following in order.

1. It checks if the query $q = q_i$. If not, C' returns $(x_1, q_1, w_1, \dots, q_i \oplus q, w_i, \dots, q_{2t}, w_{2t})$.
2. It then scans for a minimum $i' < i$ such that $q_i = q_{i'}$ but $w_i \neq w_{i'}$. If it finds one, C' returns $(x_1, q_1, w_1, \dots, q_i, w_i \oplus w_{i'}, \dots, q_{2t}, w_{2t})$.
3. It then checks if w_i is a solution to q_i . If not, C' returns $(x_1, q_1, w_1, \dots, q_i, C_i(w_i), \dots, q_{2t}, w_{2t})$, where C_i is the input circuit encoded in q_i .

If **Item 1**, **Item 2**, or **Item 3** is triggered at iteration i , we say condition 1, 2, or 3 was triggered on iteration i respectively. If the whole simulation finishes without any of the above checks being triggered, C' outputs $(x_2, q_{t+1}, w_{t+1}, \dots, q_{2t}, w_{2t}, q_1, w_1, \dots, q_t, w_t)$.

The reduction feeds C' to its LONELY oracle, and receives back an answer $(x_1, q_1, w_1, \dots, q_{2t}, w_{2t})$. If this is a type 1 solution to LONELY (i.e., $C'(x_1, q_1, w_1, \dots, q_{2t}, w_{2t}) = (x_1, q_1, w_1, \dots, q_{2t}, w_{2t})$), the reduction outputs $(x_1, w_1, \dots, w_t)$. Otherwise, the reduction outputs $(x_1, w_1, \dots, w_{2t})$.

The reduction clearly runs in polynomial time, so we proceed to correctness. First, observe that if condition 2 is triggered at iteration i , then all earlier occurrences of the query q_i have the same recorded answer $w_{i'}$. Otherwise condition 2 would have been triggered even earlier. Next, if two inputs to C' share the same prefix up till the pair (q_i, w_i) , then the executions of C' on both inputs are identical at every iteration $j < i$ and make the same query q to the i -th oracle gate. We call this property prefix invariance.

Consider running $C'(x_1, q_1, w_1, \dots, q_{2t}, w_{2t})$. We now show correctness of the reduction by case analysis on how $C'(x_1, q_1, w_1, \dots, q_{2t}, w_{2t})$ is evaluated.

1. The evaluation of $C'(x_1, q_1, w_1, \dots, q_{2t}, w_{2t})$ triggers condition 1 on evaluation i .

$$C'(C'(x_1, q_1, w_1, \dots, q_{2t}, w_{2t})) = C'(x_1, q_1, w_1, \dots, q_i \oplus q, w_i, \dots, q_{2t}, w_{2t})$$

Let $\hat{q}$ be the value of q during iteration i of the evaluation of $C'(x_1, q_1, w_1, \dots, q_i \oplus q, w_i, \dots, q_{2t}, w_{2t})$. By the prefix invariance of C' , $\hat{q} = q$. But since $q \neq q_i$ and $q_i \neq 0$, $\hat{q} \neq q_i \oplus \hat{q}$. Therefore,

$$\begin{aligned} C'(x_1, q_1, w_1, \dots, q_i \oplus q, w_i, \dots, q_{2t}, w_{2t}) &= (x_1, q_1, w_1, \dots, q_i \oplus q \oplus q, w_i, \dots, q_{2t}, w_{2t}) \\ &= (x_1, q_1, w_1, \dots, q_{2t}, w_{2t}). \end{aligned}$$

This implies that $(x_1, q_1, w_1, \dots, q_{2t}, w_{2t})$ is not a solution to LONELY on C' . Therefore, this case cannot happen.

2. The evaluation of $C'(x_1, q_1, w_1, \dots, q_{2t}, w_{2t})$ triggers condition 2 on evaluation i .

$$C'(C'(x_1, q_1, w_1, \dots, q_{2t}, w_{2t})) = C'(x_1, q_1, w_1, \dots, q_i, w_i \oplus w_{i'}, \dots, q_{2t}, w_{2t})$$

Consider the evaluation of $C'(x_1, q_1, w_1, \dots, q_i, w_i \oplus w_{i'}, \dots, q_{2t}, w_{2t})$. Again, by prefix invariance, no condition is triggered during any iteration $j < i$. Moreover, $w_i \oplus w_{i'} \neq 0$ because

$w_i \neq w_{i'}$, and $w_i \oplus w_{i'} \neq w_{i'}$ because $w_i \neq 0$. Hence condition 2 is triggered again at iteration i , and the answer at the minimum qualifying index is $w_{i'}$. Therefore,

$$\begin{aligned} C'(x_1, q_1, w_1, \dots, q_i, w_i \oplus w_{i'}, \dots, q_{2t}, w_{2t}) &= (x_1, q_1, w_1, \dots, q_i, w_i \oplus w_{i'} \oplus w_{i'}, \dots, q_{2t}, w_{2t}) \\ &= (x_1, q_1, w_1, \dots, q_{2t}, w_{2t}). \end{aligned}$$

This implies that $(x_1, q_1, w_1, \dots, q_{2t}, w_{2t})$ is not a solution to LONELY on C' . Therefore, this case cannot happen.

3. The evaluation of $C'(x_1, q_1, w_1, \dots, q_{2t}, w_{2t})$ triggers condition 3 on iteration i . Note that this implies that $C_i(w_i) \neq w_i$ and $C_i(C_i(w_i)) = w_i$.

$$C'(C'(x_1, q_1, w_1, \dots, q_{2t}, w_{2t})) = C'(x_1, q_1, w_1, \dots, q_i, C_i(w_i), \dots, q_{2t}, w_{2t})$$

Similar reasoning as in the previous case tells us that the evaluation of $C'(x_1, q_1, w_1, \dots, q_i, C_i(w_i), \dots, q_{2t}, w_{2t})$ will not trigger any condition **Item 1**, **Item 2**, or **Item 3** on any iteration $j < i$ and will not trigger **Item 1** on iteration i .

We now show that it will also not trigger **Item 2** on iteration i . Say for the sake of contradiction that it did. This would imply that there is some $i' < i$ such that $q_{i'} = q_i$. But since **Item 2** did not trigger at iteration i of the execution of $C'(x_1, q_1, w_1, \dots, q_{2t}, w_{2t})$, we must have that $w_{i'} = w_i$. Since that execution passed iteration i' , $w_{i'}$ must be a valid solution to the LONELY instance $q_{i'}$. But since $q_i = q_{i'}$ and $w_i = w_{i'}$, w_i is a solution to q_i , which contradicts the fact that the evaluation of $C'(x_1, q_1, w_1, \dots, q_{2t}, w_{2t})$ triggers condition 3 on iteration i .

Therefore, the evaluation of $C'(x_1, q_1, w_1, \dots, q_i, C_i(w_i), \dots, q_{2t}, w_{2t})$ will check if $C_i(w_i)$ is a solution to q_i . It will find that it is not since $C_i(C_i(w_i)) \neq C_i(w_i)$ and $C_i(C_i(C_i(w_i))) = C_i(w_i)$, which both follow from the fact that w_i is not a solution to q_i .

$$\begin{aligned} C'(x_1, q_1, w_1, \dots, q_i, C_i(w_i), \dots, q_{2t}, w_{2t}) &= (x_1, q_1, w_1, \dots, q_i, C_i(C_i(w_i)), \dots, q_{2t}, w_{2t}) \\ &= (x_1, q_1, w_1, \dots, q_{2t}, w_{2t}). \end{aligned}$$

This implies that $(x_1, q_1, w_1, \dots, q_{2t}, w_{2t})$ is not a solution to LONELY on C' . Therefore, this case cannot happen.

4. The whole simulation of $C'(x_1, q_1, w_1, \dots, q_{2t}, w_{2t})$ finishes without triggering any of **Item 1**, **Item 2**, or **Item 3**. Let $x_3 = C^*(x_2, w_{t+1}, \dots, w_{2t})$. We distinguish whether $x_1 = x_2$. If $x_1 = x_2$, then the reduction clearly succeeds, and it outputs consistent subwitnesses because neither **Item 1** nor **Item 2** is ever triggered during the evaluation of $C'(x_1, q_1, w_1, \dots, q_{2t}, w_{2t})$. Now suppose that $x_1 \neq x_2$. We show that $x_1 \neq x_3$ by contradiction. Say $x_1 = x_3$. Then $(q_1, w_1, \dots, q_{2t}, w_{2t})$ is internally consistent, the simulation $C^*(x_1, w_1, \dots, w_t) = x_2$ makes oracle gate queries q_i which can be answered with subwitnesses w_i , and the simulation $C^*(x_2, w_{t+1}, \dots, w_{2t}) = x_1$ makes oracle gate queries q_{t+i} which can be answered with subwitnesses w_{t+i} . Consequently, after rotating the two transcripts, both simulations again pass all three checks. By the definition of C' , we therefore have

$$\begin{aligned} C'(C'(x_1, q_1, w_1, \dots, q_{2t}, w_{2t})) &= C'(x_2, q_{t+1}, w_{t+1}, \dots, q_{2t}, w_{2t}, q_1, w_1, \dots, q_t, w_t) \\ &= (x_1, q_1, w_1, \dots, q_{2t}, w_{2t}). \end{aligned}$$

Since $x_1 \neq x_2$, the displayed equality shows that applying C' twice returns to the original vertex. Hence this vertex is not a valid solution, a contradiction. So $x_1 \neq x_3$. Therefore, $(x_1, w_1, \dots, w_{2t})$ is a solution to $\text{LONELY}^{\text{LONELY}}$. Internal consistency is handled by the fact that neither **Item 1** nor **Item 2** is ever triggered during the evaluation of $C'(x_1, q_1, w_1, \dots, q_{2t}, w_{2t})$.

□

Theorem 5.2. $\text{PPA}^* = \text{PPA}$.

Proof. PPA is trivially contained in PPA^* .

lemma 5.1 lets us conclude that $\text{LONELY}^{\text{LONELY}}$ has a many-one reduction to LONELY. Therefore, $\text{PPA}^{\text{PPA}} = \text{PPA}$. This immediately collapses the whole PPA^* hierarchy to PPA by induction.

□

5.2 PLS Self-lowness

We now show that PLS is self-low. To do so, we work with the established PLS-complete problem ITER (introduced in [BK94] and [Mor]), ITERWITHSOURCE, and a variant we call ITER2.

5.2.1 The PLS-Complete Problems

To ensure syntactic totality of ITER, ITERWITHSOURCE, and ITER2, we say that 0 is a solution if S is the identity on the start vertex.

Definition 5.3. The problem ITER is defined as follows. The input is $S : [2^n] \rightarrow [2^n]$. Output any of the following solutions.

1. 0 if $S(0) = 0$,
2. x s.t. $S(x) > x$ and $S(S(x)) \leq S(x)$.

Definition 5.4. The problem ITERWITHSOURCE is defined as follows. The input is $S : [2^n] \rightarrow [2^n]$ and $\alpha \in [2^n]$. Output any of the following solutions.

1. 0 if $S(\alpha) \leq \alpha$,
2. x s.t. $S(x) > x$ and $S(S(x)) \leq S(x)$.

Definition 5.5. The problem ITER2 is defined as follows. The input is $S : [2^n] \rightarrow [2^n]$. Output any of the following solutions.

1. 0 if $S(0) = 0$,
2. x s.t. $S(x) < x$,
3. x s.t. $S(x) > x$ and $S(S(x)) \leq S(x)$.

Lemma 5.6. ITER2 is PLS-complete.

Proof. We first reduce ITER to ITER2. Given an instance S of ITER, we let $S'(x)$ be x if $S(x) < x$ and $S(x)$ otherwise. We feed S' to our ITER2 oracle to get back a solution y which we output. If $y = 0$ and $S'(0) = 0$, then $S(0) = 0$, so y is a valid solution to the original ITER instance. Otherwise, since $S'(x) \geq x$ for every x , the solution y must satisfy $S'(y) > y$ and $S'(S'(y)) \leq S'(y)$. We then have $S(y) = S'(y)$ and $S'(S(y)) \leq S(y)$. This implies that $S(S(y)) \leq S(y)$, since by construction $S' \geq S$ for all inputs. Therefore, y is a valid solution to ITER.

We now reduce ITER2 to ITER. Given an instance S of ITER2, the reduction calls its ITER oracle on S to get back an answer y which it outputs. If $y = 0$ and $S(0) = 0$, then y is a valid solution to ITER2. Otherwise, $S(y) > y$ and $S(S(y)) \leq S(y)$, so y is also a valid solution to ITER2. $\square$

5.2.2 The Reduction

We are now ready to show the main reduction.

Lemma 5.7. $\text{ITER}^{\text{ITER2}}$ reduces to ITERWITHSOURCE under black-box Turing reductions.

Proof. We show the reduction from $\text{ITER}^{\text{ITER2}}$ to ITERWITHSOURCE. Let S be the input to the reduction. We assume that S has t oracle gates, where the i^{th} oracle gate has an output of size s_i . Let S^2 be the circuit which simply composes S with itself. Note that S^2 has $2t$ oracle gates.

The reduction first computes $a_1, \dots, a_t$ such that $a_1, \dots, a_t$ are consistent for $S^*(0, a_1, \dots, a_t)$ and $S^*(0, a_1, \dots, a_t) \neq \perp$ by repeated calls to its ITERWITHSOURCE oracle. If $S^*(0, a_1, \dots, a_t) = 0$, the reduction outputs $(0, a_1, \dots, a_t)$. Otherwise, let $y_0 = S^*(0, a_1, \dots, a_t) > 0$ and define

$$\alpha = (0, a_1, \dots, a_t, 0^{s_1}, \dots, 0^{s_t}).$$

The reduction then constructs

$$S' : \{0, 1\}^n \times \prod_{j=1}^t \{0, 1\}^{s_j} \times \prod_{j=1}^t \{0, 1\}^{s_j} \rightarrow \{0, 1\}^n \times \prod_{j=1}^t \{0, 1\}^{s_j} \times \prod_{j=1}^t \{0, 1\}^{s_j}.$$

$S'(x, w_1, \dots, w_{2t})$ behaves as follows. Let $y = S^*(x, w_1, \dots, w_t)$ and if $y \neq \perp$, let $z = S^*(y, w_{t+1}, \dots, w_{2t})$. The behavior of S' is now split into four cases.

1. If $y = \perp$ or internal consistency is violated in $S^*(x, w_1, \dots, w_t)$, output $(x, w_1, \dots, w_{2t})$.
2. If $y \neq \perp$, the evaluation of $S^*(x, w_1, \dots, w_t)$ is internally consistent, and $y \leq x$, output $(x, w_1, \dots, w_{2t})$.
3. Consider when $y \neq \perp$, the evaluation of $S^*(x, w_1, \dots, w_t)$ is internally consistent, $y > x$, and one of the following occurs: $z = \perp$ or the evaluation of $S^{2*}(x, w_1, \dots, w_{2t})$ violates internal consistency. Let m be the smallest index in $\{t+1, \dots, 2t\}$ such that w_m is not a solution to oracle gate query m in the evaluation of $S^{2*}(x, w_1, \dots, w_{2t})$ or w_m violates internal consistency in the evaluation of $S^{2*}(x, w_1, \dots, w_{2t})$ because gate query m is the same as gate query i for some $i < m$ and $w_m \neq w_i$. If w_m violates internal consistency, output $(x, w_1, \dots, w_{m-1}, w_i, 0, \dots, 0)$. Otherwise, w_m is not a valid solution to an oracle gate query u which encodes an ITER2 query $H : \{0, 1\}^q \rightarrow \{0, 1\}^q$. In this case, output $(x, w_1, \dots, w_{m-1}, H(w_m), 0, \dots, 0)$.

4. Say $y \neq \perp$, $y > x$, $z \neq \perp$, and the evaluation of $S^{2*}(x, w_1, \dots, w_{2t})$ is internally consistent. Output $(y, w_{t+1}, \dots, w_{2t}, 0, \dots, 0)$.

The reduction calls its ITERWITHSOURCE oracle on S', α to get back an answer $(x, w_1, \dots, w_{2t})$ which would be the output of our reduction.

The reduction clearly runs in polynomial time since S' translates to a polynomial size circuit and all other operations run in polynomial time. We now show the correctness. Note that if $S^*(0, a_1, \dots, a_t) = 0$, the reduction is clearly correct since it returns a valid type 1 solution to $\text{ITER}^{\text{ITER}2}$. We therefore assume that $S^*(0, a_1, \dots, a_t) \neq 0$. We next show that $S'(\alpha) > \alpha$, ensuring that the final ITERWITHSOURCE oracle call will only return type 2 solutions to ITERWITHSOURCE. Consider the evaluation of $S'(\alpha)$. Since by construction, $S^*(0, a_1, \dots, a_t) \neq \perp$ and the evaluation of $S^*(0, a_1, \dots, a_t)$ is internally consistent, case 1 is not triggered during the evaluation of $S'(\alpha)$. Furthermore, since by assumption, $y \neq 0$, case 2 is not triggered during the evaluation of $S'(\alpha)$. If $S'(\alpha)$ is evaluated using case 3, then $S'(\alpha) > \alpha$ since case 3 can only replace a zero value in $(0, a_1, \dots, a_t, 0, \dots, 0)$ with a nonzero value. Finally, if $S'(\alpha)$ is evaluated using case 4, then $S'(\alpha) > \alpha$ by $y > x$. In all cases, $S'(\alpha) > \alpha$.

Let $v = (x, w_1, \dots, w_{2t})$ be the answer the final oracle call to ITERWITHSOURCE returns. By the fact that $S'(\alpha) > \alpha$, we know that v satisfies the following inequalities.

$$S'(v) > v \tag{1}$$

$$S'(S'(v)) \leq S'(v) \tag{2}$$

We will divide our proof of correctness by which cases are used to evaluate $S'(v)$ and $S'(S'(v))$. Before we dive into the case analysis, we note that S' evaluated using case 1 and case 2 is the identity and could not satisfy Equation (1). Hence $S'(v)$ has to be evaluated using either case 3 or case 4. Furthermore, since S' evaluated as case 4 is strictly increasing, $S'(S'(v))$ could not be evaluated as case 4.

1. $S'(v)$ is evaluated using case 3 and $S'(S'(v))$ is evaluated using case 1 or case 2. This case cannot happen. Since $S'(x, w_1, \dots, w_{2t})$ was evaluated using case 3, $S^*(x, w_1, \dots, w_t) \neq \perp$ and its evaluation is internally consistent. Let $S'(x, w_1, \dots, w_{2t}) = (x, w'_1, \dots, w'_{2t})$. Notice that $w_i = w'_i$ for all i in $[1, t]$. Therefore, $S^*(x, w'_1, \dots, w'_t) \neq \perp$ and its evaluation is internally consistent. Therefore, $S'(x, w'_1, \dots, w'_{2t})$ will not be evaluated using case 1 or case 2.
2. $S'(v)$ is evaluated using case 3 and $S'(S'(v))$ is evaluated using case 3. This case cannot happen. Let m be the identified index in $S'(x, w_1, \dots, w_{2t})$ and m' be the identified index in $S'(S'(x, w_1, \dots, w_{2t}))$. Note that $m \leq m'$. Consider first when $m < m'$. Then $S'(x, w_1, \dots, w_{2t}) = (x, w_1, \dots, w_{m-1}, \hat{w}_m, 0, \dots, 0)$ and $S'(x, w_1, \dots, w_{m-1}, \hat{w}_m, 0, \dots, 0) = (x, w_1, \dots, w_{m-1}, \hat{w}_m, 0, \dots, 0, \hat{w}_{m'}, 0, \dots, 0)$ for some $\hat{w}_m$ and $\hat{w}_{m'} \neq 0$. Therefore, $S'(S'(x, w_1, \dots, w_{2t})) > S'(x, w_1, \dots, w_{2t})$, violating Equation (2). If $m = m'$, this can only happen when w_m and $H(w_m)$ are not solutions to u . $S'(x, w_1, \dots, w_{2t}) = (x, w_1, \dots, w_{m-1}, H(w_m), 0, \dots, 0)$ and $S'(S'(x, w_1, \dots, w_{2t})) = (x, w_1, \dots, w_{m-1}, H(H(w_m)), 0, \dots, 0)$. Since w_m is not a solution to ITER2 on instance H and Equation (1) implies $H(w_m) > w_m$, we must have $H(H(w_m)) > H(w_m)$. Therefore, $S'(S'(x, w_1, \dots, w_{2t})) > S'(x, w_1, \dots, w_{2t})$, contradicting Equation (2).

3. $S'(v)$ is evaluated using case 4 and $S'(S'(v))$ is evaluated using case 1. This case cannot happen. In particular, since $S'(x, w_1, \dots, w_{2t})$ is evaluated using case 4, we know $S^*(y, w_{t+1}, \dots, w_{2t}) \neq \perp$ and is internally consistent.
4. $S'(v)$ is evaluated using case 4 and $S'(S'(v))$ is evaluated using case 2. Notice that $S'(x, w_1, \dots, w_{2t}) = (y, w_{t+1}, \dots, w_{2t}, 0, \dots, 0)$ where $S^*(y, w_{t+1}, \dots, w_{2t}) = z$. Therefore, $S'(S'(x, w_1, \dots, w_{2t}))$ being evaluated using case 2 means that $z \leq y$. The reduction therefore outputs a valid solution in this case.
5. $S'(v)$ is evaluated using case 4 and $S'(S'(v))$ is evaluated using case 3. This case cannot happen. Let $S'(x, w_1, \dots, w_{2t}) = (y, w_{t+1}, \dots, w_{2t}, 0, \dots, 0)$. Evaluation by case 3 implies that $S'(y, w_{t+1}, \dots, w_{2t}, 0, \dots, 0) = (y, w_{t+1}, \dots, w_{2t}, 0, \dots, 0, \hat{w}_m, 0, \dots)$. As such, $S'(S'(x, w_1, \dots, w_{2t})) > S'(x, w_1, \dots, w_{2t})$, contradicting [Equation \(2\)](#).

□

Theorem 5.8. $\text{PLS}^* = \text{PLS}$.

Proof. PLS is trivially contained in PLS^* .

We now show the other direction. Note that all the following conditions are satisfied.

1. SINK-OF-DAG has a black-box many-one reduction to ITER .
2. $\text{ITER}^{\text{ITER2}}$ has a Turing reduction to ITERWITHSOURCE by [lemma 5.7](#).
3. ITER2 is many-one reducible to ITER .
4. ITERWITHSOURCE is Turing-closed since PLS is Turing-closed.

Therefore, [corollary 4.4](#) tells us $\text{SINK-OF-DAG}^{\text{ITER2}}$ has a many-one reduction to ITERWITHSOURCE , which has a many-one reduction to SINK-OF-DAG . Finally, by [theorem 4.6](#), $\text{SINK-OF-DAG}^{\text{SINK-OF-DAG}}$ has a many-one reduction to $\text{SINK-OF-DAG}^{\text{ITER2}}$. Chaining these reductions gives us a reduction from $\text{SINK-OF-DAG}^{\text{SINK-OF-DAG}}$ to SINK-OF-DAG . Therefore, $\text{PLS}^{\text{PLS}} = \text{PLS}$. This immediately collapses the whole PLS^* hierarchy to PLS by induction.

□

5.3 LOSSY Self-lowness

We now show that LOSSY is self-low.

Lemma 5.9. $\text{LOSSY}^{\text{LOSSY}}$ reduces to $(n-1)\text{-LOSSY}$ under many-one reductions.

Proof. Let $C : \{0, 1\}^n \rightarrow \{0, 1\}^{n/2}$, $D : \{0, 1\}^{n/2} \rightarrow \{0, 1\}^n$ be the circuits that act as input to our $\text{LOSSY}^{\text{LOSSY}}$ problem. Without loss of generality, assume that C, D collectively have t LOSSY gates, and for some fixed polynomial $s(n) \geq 100 \log(t+100)$, every input to a LOSSY oracle gate takes the form $c_i : \{0, 1\}^s \rightarrow \{0, 1\}^{s/2}$, $d_i : \{0, 1\}^{s/2} \rightarrow \{0, 1\}^s$. This can be achieved by padding and applying [lemma 2.14](#).

We now construct

$$C' : \{0, 1\}^n \times \{0, 1\}^{s \cdot t} \rightarrow \{0, 1\}^{n+s \cdot t-1},$$

$$D' : \{0, 1\}^{n+s \cdot t-1} \rightarrow \{0, 1\}^n \times \{0, 1\}^{s \cdot t}.$$

Let $N := n + s \cdot t$. Let $D \circ C : \{0, 1\}^n \rightarrow \{0, 1\}^n$ be the composed circuit which consists of C followed by D . Let $x_2 = C_*(x_1, w_1, \dots, w_t)$ and $x_3 = (D \circ C)_*(x_1, w_1, \dots, w_t)$.

$C'(x_1, w_1, \dots, w_t)$ computes $x_3 = (D \circ C)_*(x_1, w_1, \dots, w_t)$. If x_3 is an error output (m, z) where z encodes a LOSSY instance $c : \{0, 1\}^s \rightarrow \{0, 1\}^{s/2}, d : \{0, 1\}^{s/2} \rightarrow \{0, 1\}^s$, C' outputs $(1, m, x_1, \dots, w_{m-1}, c(w_m), w_{m+1}, \dots, w_t, 0, \dots, 0) \in \{0, 1\}^{N-1}$. Otherwise, C' outputs $(0, x_2, w_1, \dots, w_t, 0, \dots, 0) \in \{0, 1\}^{N-1}$ where $x_2 = C_*(x_1, w_1, \dots, w_t)$.

Next, we now define D' . We first consider the case when the input to D' has the form $(0, x_2, w_1, \dots, w_t, 0, \dots, 0)$. D' computes $x_3 = D_*(x_2, w_1, \dots, w_t)$. If x_3 is an error output, D' outputs 0. Otherwise, D' outputs $(x_3, w_1, \dots, w_t)$. When the input to D' has the form $(1, m, x_1, \dots, w_t, 0, \dots, 0)$, D' computes $(D \circ C)_*(x_1, w_1, \dots, w_{m-1}, 0^s, w_{m+1}, \dots, w_t)$. If this results in a result output, D' outputs 0. Say it results in an error output (m', z) where z encodes a LOSSY instance $c : \{0, 1\}^s \rightarrow \{0, 1\}^{s/2}, d : \{0, 1\}^{s/2} \rightarrow \{0, 1\}^s$. D' outputs $(x_1, \dots, w_{m-1}, d(w_m), w_{m+1}, \dots, w_t)$.

The reduction feeds C', D' to its LOSSY oracle to get back $(v_1, w_1, \dots, w_t)$. The reduction outputs v_1 as well as witnesses used to evaluate $(D \circ C)_*(v_1, w_1, \dots, w_t)$ in order.

The reduction clearly runs in polynomial time. Notice also that since c compresses by at least $s/2 \geq 50 \log(t + 100)$ bits and m requires exactly $\log_2(t)$ bits to specify, C' compresses by at least 1 bit. To show correctness, let $v_2 = C_*(v_1, w_1, \dots, w_t)$ and $v_3 = (D \circ C)_*(v_1, w_1, \dots, w_t)$. We consider two cases.

1. v_3 is a result output. Then $C'(v_1, w_1, \dots, w_t) = (0, v_2, w_1, \dots, w_t, 0, \dots, 0)$. Notice that by construction, $D'(0, v_2, w_1, \dots, w_t, 0, \dots, 0) = (v_3, w_1, \dots, w_t)$. By assumption, $v_1 \neq v_3$. Therefore, v_1 as well as the witnesses among $w_1, \dots, w_t$ used to evaluate $D(C(v_1))$ are a solution to our $\text{LOSSY}^{\text{LOSSY}}$ instance. The witnesses are all consistent since the first valid w_i among $w_1, \dots, w_t$ is used to evaluate the oracle gates at every step.
2. v_3 is an error output (m, z) . This cannot happen. Say z encodes a LOSSY instance $c : \{0, 1\}^s \rightarrow \{0, 1\}^{s/2}, d : \{0, 1\}^{s/2} \rightarrow \{0, 1\}^s$. Then $C'(v_1, w_1, \dots, w_t) = (1, m, v_1, \dots, c(w_m), \dots, w_t, 0, \dots, 0)$. Notice that since $m \notin M$ (recall [definition 3.9](#)) for the evaluation of $v_3 = (D \circ C)_*(v_1, w_1, \dots, w_t)$, $(D \circ C)_*(v_1, w_1, \dots, w_{m-1}, 0^s, w_{m+1}, \dots, w_t)$ should have the exact same behaviour as $(D \circ C)_*(v_1, w_1, \dots, w_t)$ and output (m, z) and w_m is not a solution to LOSSY on c, d . This is because C_* always tries 0 as a solution to an oracle gate. As such, $D'(1, m, v_1, w_1, \dots, c(w_m), \dots, w_t, 0, \dots, 0)$ evaluates to $(v_1, w_1, \dots, d(c(w_m)), \dots, w_t) = (v_1, w_1, \dots, w_t)$. Therefore, $(v_1, w_1, \dots, w_m, \dots, w_t)$ is not a solution to our oracle call to LOSSY on C', D' .

□

Theorem 5.10. $\text{LOSSY}^* = \text{LOSSY}$.

Proof. LOSSY is trivially contained in LOSSY^* .

$\text{LOSSY}^{\text{LOSSY}}$ has a many-one reduction to $(n - 1)$ -LOSSY by [lemma 5.9](#), which reduces to LOSSY by [lemma 2.14](#). Chaining the reductions tells us $\text{LOSSY}^{\text{LOSSY}}$ reduces to LOSSY. Therefore, $\text{LOSSY}^{\text{LOSSY}} = \text{LOSSY}$. This immediately collapses the whole LOSSY^* hierarchy to LOSSY by induction.

□

6 Further Applications

In this section, we demonstrate the potential of our new definitions of TFNP subclasses with TFNP oracles for developing a better understanding of important computational problems.

6.1 Number Theory

Notably, our result that PPA is self-low provides a potential way to classify the problem of deterministically generating large primes inside TFNP. We first define the necessary problems.

Definition 6.1 (WEAK-BERTRAND). Given a string 1^n , output a $32n$ bit prime p such that $p > 2^n$.

We refer to this problem as WEAK-BERTRAND since Bertrand's postulate tells us that there always exists a prime between 2^n and 2^{n+1} . The problem BERTRAND would ask us to find such a prime. In WEAK-BERTRAND, we are asking for a prime between 2^n and 2^{32n} . We now review relevant results.

Lemma 6.2 ([Jer16]). *Under the Generalized Riemann Hypothesis, FACTOR is in PPA and PPP.*

Lemma 6.3 ([Kor22]). WEAK-BERTRAND reduces to $\text{LOSSY}^{\text{FACTOR}}$ ¹.

We are able to leverage lemma 6.2 and lemma 6.3 to give a classification of WEAK-BERTRAND into our newly defined classes. Furthermore, the fact that PPA is self-low and FACTOR is in PPA (under the Generalized Riemann Hypothesis) suggests that WEAK-BERTRAND (or even BERTRAND) may be easily reducible to a PPA-complete problem.

Theorem 6.4. *Under the Generalized Riemann Hypothesis, WEAK-BERTRAND is in $\text{LOSSY}^{\text{PPA}}$, $\text{LOSSY}^{\text{PPP}}$, $\text{PPADS}^{\text{PPA}}$, $\text{PPADS}^{\text{PPP}}$, and PPP^{PPP} .*

Proof. Both $\text{WEAK-BERTRAND} \in \text{LOSSY}^{\text{PPA}}$ and $\text{WEAK-BERTRAND} \in \text{LOSSY}^{\text{PPP}}$ follow directly by combining lemma 6.3 with lemma 6.2. Observing that LOSSY has a trivial black-box reduction to the PPADS-complete problem INJECTIVEPIGEON then implies WEAK-BERTRAND is in both $\text{PPADS}^{\text{PPA}}$ and $\text{PPADS}^{\text{PPP}}$. Similarly, the fact that INJECTIVEPIGEON has a trivial black-box reduction to PIGEON implies that WEAK-BERTRAND is in PPP^{PPP} . □

Theorem 6.5. *If WEAK-BERTRAND is in $\text{PPA}^{\text{FACTOR}}$, then the Generalized Riemann Hypothesis implies that WEAK-BERTRAND is in PPA.*

Proof. Apply lemma 6.2 and theorem 5.2. □

¹ Technically, [Kor22] showed this for LOSSY given access to an oracle which outputs all prime factors of a number rather than one non-trivial factor. We observe that one can obtain all factors of a number by simply applying FACTOR multiple times. Furthermore, Korten's reduction makes further FACTOR queries after finding an incompressible string in the $\text{LOSSY}^{\text{FACTOR}}$ instance, but we note that these can trivially be encoded into the $\text{LOSSY}^{\text{FACTOR}}$ instance.

Acknowledgements

The authors would like to thank Karthik Gajulapalli, Sidhant Saraogi, and Noah Stephens-Davidowitz for many helpful discussions and feedback on an earlier draft of this manuscript. The authors are grateful to Mika Göös and Valentin Imbach for helpful discussions which encouraged them to include further exposition about the utility of internal consistency and during which a bug in the PPA self-lowness proof was discovered. The authors would also like to thank the anonymous referees for useful comments.

AI Disclosure

GPT 5.6 Sol, GPT 6, and Gemini 3.1 Pro were used to identify and correct several proof bugs. The authors (without AI) discovered that the previous proof of PPA self-lowness had a bug and devised a new proof using the same underlying idea. AI was used to fully elaborate on this proof (which AI correctly identified as incomplete). AI was used to identify and close several other important gaps. These included a bug in the PLS self-lowness proof concerning the choice of initial vertex, an observation which led to [Footnote 1](#), and inconsistencies regarding encoding of circuits and witnesses. AI was also used for literature review and copy-editing. The authors assume full responsibility for correctness and originality.

References

- [AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. Primes is in p. *Annals of mathematics*, pages 781–793, 2004. [9](#)
- [BCE⁺95] Paul Beame, Stephen Cook, Jeff Edmonds, Russell Impagliazzo, and Toniann Pitassi. The relative complexity of np search problems. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 303–314, 1995. [2](#), [8](#)
- [BJ12] Samuel R Buss and Alan S Johnson. Propositional proofs and reductions between np search problems. *Annals of Pure and Applied Logic*, 163(9):1163–1182, 2012. [6](#), [7](#)
- [BK94] Samuel R. Buss and Jan Krajíček. An application of boolean complexity to separation problems in bounded arithmetic. *Proceedings of the London Mathematical Society*, s3-69(1):1–21, 1994. [4](#), [20](#)
- [BOM04] Josh Buresh-Oppenheim and Tsuyoshi Morioka. Relativized np search problems and propositional proof systems. In *Proceedings. 19th IEEE Annual Conference on Computational Complexity, 2004.*, pages 54–67. IEEE, 2004. [2](#)
- [CDT09] Xi Chen, Xiaotie Deng, and Shang-Hua Teng. Settling the complexity of computing two-player nash equilibria. *J. ACM*, 56(3), May 2009. [1](#)
- [DGP09] Constantinos Daskalakis, Paul W. Goldberg, and Christos H. Papadimitriou. The complexity of computing a nash equilibrium. *SIAM Journal on Computing*, 39(1):195–259, 2009. [1](#)

- [FGPR24] Noah Fleming, Stefan Grosser, Toniann Pitassi, and Robert Robere. Black-box ppp is not turing-closed. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*, pages 1405–1414, 2024. 4, 6, 7, 15
- [GHJ⁺24] Mika Göös, Alexandros Hollender, Siddhartha Jain, Gilbert Maystre, William Pires, Robert Robere, and Ran Tao. Separations in proof complexity and tfnp. *Journal of the ACM*, 71(4):1–45, 2024. 2
- [Hol21] Alexandros Hollender. The classes ppa-k: Existence from arguments modulo k. *Theoretical Computer Science*, 885:15–29, 2021. 8
- [Jeř16] Emil Jeřábek. Integer factoring and modular square roots. *Journal of Computer and System Sciences*, 82(2):380–394, 2016. 25
- [JLRX24] Siddhartha Jain, Jiawei Li, Robert Robere, and Zhiyang Xun. On pigeonhole principles and ramsey in tfnp. In *2024 IEEE 65th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 406–428. IEEE, 2024. 2
- [KKMP21] Robert Kleinberg, Oliver Korten, Daniel Mitropolsky, and Christos Papadimitriou. Total functions in the polynomial hierarchy. In *12th Innovations in Theoretical Computer Science Conference (ITCS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021. 2
- [Kor22] Oliver Korten. Derandomization from time-space tradeoffs. In *37th Computational Complexity Conference (CCC 2022)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2022. 6, 9, 25
- [Kre90] Mark W. Krentel. On finding and verifying locally optimal solutions. *SIAM J. Comput.*, 19(4):742–749, June 1990. 1
- [Lau83] Clemens Lautemann. Bpp and the polynomial hierarchy. *Information Processing Letters*, 17(4):215–217, 1983. 2
- [LLR24] Jiawei Li, Yuhao Li, and Hanlin Ren. Metamathematics of resolution lower bounds: A tfnp perspective, 2024. 6
- [LPT24] Jiayu Li, Edward Pyne, and Roei Tell. Distinguishing, predicting, and certifying: On the long reach of partial notions of pseudorandomness. In *2024 IEEE 65th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1–13, 2024. 6, 7
- [Mor] T Morioka. Classification of search problems and their definability in bounded arithmetic, master’s thesis, university of toronto, toronto, canada, 2001. 2, 20
- [Pap94] Christos H. Papadimitriou. On the complexity of the parity argument and other inefficient proofs of existence. *J. Comput. Syst. Sci.*, 48:498–532, 1994. 1, 8
- [PYP22] Amol Pasarkar, Mihalis Yannakakis, and Christos Papadimitriou. Extremal combinatorics, iterated pigeonhole arguments, and generalizations of ppp. *arXiv preprint arXiv:2209.07625*, 2022. 7

- [Sch91] Alejandro A. Schäffer. Simple local search problems that are hard to solve. *SIAM Journal on Computing*, 20(1):56–87, 1991. 1
- [SZZ18] Katerina Sotiraki, Manolis Zampetakis, and Giorgos Zirdelis. PPP-Completeness with Connections to Cryptography . In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 148–158, Los Alamitos, CA, USA, October 2018. IEEE Computer Society. 1