



Downward Self-Reducibility in TFNP*

Prahladh Harsha[†] Daniel Mitropolsky[‡] Alon Rosen[§]

November 23, 2022

Abstract

A problem is *downward self-reducible* if it can be solved efficiently given an oracle that returns solutions for strictly smaller instances. In the decisional landscape, downward self-reducibility is well studied and it is known that all downward self-reducible problems are in PSPACE. In this paper, we initiate the study of downward self-reducible search problems which are guaranteed to have a solution — that is, the downward self-reducible problems in TFNP. We show that most natural PLS-complete problems are downward self-reducible and any downward self-reducible problem in TFNP is contained in PLS. Furthermore, if the downward self-reducible problem is in TFUP (i.e. it has a unique solution), then it is actually contained in UEOPPL, a subclass of CLS. This implies that if integer factoring is *downward self-reducible* then it is in fact in UEOPPL, suggesting that no efficient factoring algorithm exists using the factorization of smaller numbers.

1 Introduction

Perhaps the most surprising thing about Self-Reducibility is its longevity.

Eric Allender

Self-reducibility (sometimes also referred to as auto-reducibility) asks the question if a problem instance is easy to solve if one can solve *other* instances of the problem easily. More precisely, a computational problem is said to be self-reducible if there exists an efficient oracle machine that on input an instance of the problem solves it by making queries to an oracle that solves the same problem, with the significant restriction that it cannot query the oracle on the given instance. First introduced in the context of computability by Trakhtenbrot [Tra70] more than five decades ago, self-reducibility has proved to be immensely useful in computational complexity theory, especially in

*This work was initiated when the first and second authors were visiting the third author at Bocconi University and we are thankful to Bocconi University for their hospitality. We thank Pavel Hubáček, Eylon Yogev, and Omer Paneth for their comments on an earlier draft of this paper. We also thank the anonymous referees for several useful comments and informing us of END-OF-POTENTIAL-LINE and UNIQUEEOPL, the complete problems for the classes CLS and UEOPPL respectively, which simplified certain parts of our proof.

[†]Tata Institute of Fundamental Research, India. Email: prahladh@tifr.res.in. Research supported by the Department of Atomic Energy, Government of India, under project 12-R&D-TFR-5.01-0500.

[‡]Columbia University, USA. Email: mitropolsky@cs.columbia.edu. Supported in part by the Columbia-IBM center for Blockchain and Data Transparency, by LexisNexis Risk Solutions, and by European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant agreement No. 101019547).

[§]Bocconi University, Italy and Reichman University, Israel. Email: alon.rosen@unibocconi.it. Supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant agreement No. 101019547) and Cariplo CRYPTONOMEX grant.

the study of decision problems. Various notions of self-reducibility have been studied (for a detailed survey on self-reducibility, the reader is referred to Balcázar’s systematic study [Bal90], Selke’s comprehensive survey [Sel06] and Allender’s survey [All10]).

Downward self-reducibility, the key notion studied in this paper, refers to the kind of self-reducibility wherein the oracle machine is allowed to query the oracle only on instances strictly *smaller* than the given instance. Satisfiability, the prototype NP-complete problem as well as several other related NP-complete problems can be easily shown to be downward self-reducible. This simple observation led to plethora of search-to-decision reductions for NP-complete problems. All NP-complete problems are self-reducible and many natural NP-complete problems are also downward self-reducible. Furthermore, it is known that every downward self-reducible problem is contained in PSPACE (folklore). Another notion of self-reducibility that is extremely well-studied is random self-reducibility; wherein the oracle machine (which is randomized in this case) is only allowed to query the oracle on *random* instances of the problem, under a natural distribution on the input instance. There are problems in PSPACE which are known to be both downward self-reducible as well as random self-reducible, in particular permanent, one of the #P-complete problems. The downward and random self-reducibility of the permanent and other PSPACE-complete problems led to efficient interactive proofs for PSPACE.

Almost all the study of downward self-reducibility, to date, has been focused in the decisional landscape. In this work, we initiate the study of downward self-reducibility for search problems which are guaranteed to have a solution, that is problems in TFNP. This is motivated by the oft asked, but yet unresolved, question regarding integer factoring: is factoring an integer any easier if we know how to factor all *smaller* integers? Or equivalently, is factoring downward self-reducible?

1.1 Our results

Our focus will be on the class of search problems for which there is guaranteed to be a solution, more concisely referred to as TFNP. Within TFNP, there are several sub-classes of search problems such as PLS, PPAD, PPP, PPA, CLS, for which the totality – the fact that every instance has an efficiently verifiable solution – arises from some underlying combinatorial principle. These problems are typically defined in terms of natural complete problems. For instance, the class PLS, which is a subclass of TFNP, containing problems where totality is guaranteed by a “local-search” principle, is characterized by its complete problems ITER and SINK-OF-DAG. Our first result shows that several of these natural complete problems for PLS are in fact downward self-reducible.

Theorem 1.1. *ITER, SINK-OF-DAG are downward self-reducible.*

This result is similar to the corresponding situation for NP and PSPACE. Most of the naturally occurring NP-complete and PSPACE-complete problems are downward self-reducible. It is open if *all* NP-complete or PSPACE-complete problems are downward self-reducible. The situation is similar for PLS: it is open if every PLS-complete problem is downward self-reducible.

Given the above result, one may ask if there are problems outside PLS in TFNP which are downward self-reducible. Our next result rules this out.

Theorem 1.2. *Every downward self-reducible problem in TFNP is in PLS.*

That is, just as PSPACE is the ceiling for downward self-reducible problems in the decisional world, PLS contains all downward self-reducible problems in TFNP. An immediate corollary of these two results is that a downward self-reducible problem in TFNP is hard iff a problem in PLS is hard.

We then ask if we can further characterize the downward self-reducible problems in TFNP: what about search problems which are guaranteed to have an *unique* solution, more concisely denoted as TFUP. In this case, we show that the above containment of d.s.r problems in TFNP in the class PLS can be strengthened if the search problem is guaranteed to have an unique solution: the problem then collapses to UEOPL, a subclass of CLS (itself a sub-class of PLS), which is one of the lowest subclasses known in the TFNP hierarchy.

Theorem 1.3. *Every downward self-reducible problem in TFUP is in UEOPL.*

We now return to the question on downward self-reducibility of integer factoring. Let FACTOR denote the problem of finding a non-trivial factor of the given number if the number is composite or outputting “prime” if the number is prime and ALLFACTORS be the problem of listing *all* the non-trivial factors of a given number if it is composite or outputting “prime” if the number is prime. Clearly, ALLFACTORS is in TFUP. It is not hard to check that ALLFACTORS is downward self-reducible iff FACTOR downward self-reducible. Combining this with the above result, we get that factoring collapses to UEOPL if factoring is downward-self reducible. This result can be viewed as evidence against the downward self-reducibility of factoring.

Corollary 1.4. *If FACTOR or ALLFACTORS is downward self-reducible, then FACTOR and ALLFACTORS \in UEOPL.*

1.2 Related work

The class TFNP was introduced by Meggido and Papadimitriou [MP91]. In 1994, Papadimitriou initiated the study of syntactic subclasses of TFNP, i.e., search problems where totality is guaranteed by combinatorial principles, including PPAD [Pap94].

In the original 1990 paper, Meggido and Papadimitriou proved that if TFNP contains an NP-hard problem, then the polynomial hierarchy collapses to first level, i.e. $NP = co-NP$. Since this is generally believed not to be true, TFNP has emerged as an important potential source of “intermediate” problems – that is, problems that are not in P but not NP-complete. Since our work gives a new characterization of PLS-hardness, it is related to work that shows PLS-hardness from cryptographic assumptions, such as in [BPR15, GPS16, HY20]. In addition to hardness from cryptographic assumptions, Bitansky and Richter recently showed unconditional hardness of PLS relative to random oracles, as well as hardness of PLS from incrementally verifiable computation (though an instantiation of this is given through cryptographic assumptions) [BG20].

The connection between factoring and TFNP has been studied by Buresh-Oppenheim [BO06] and later by Jerabek [Jer16]. Their work shows that factoring is reducible to $PPA \cap PPP$. It is known that $PPAD \subseteq PPA \cap PPP$, but whether factoring is in PPAD is an open problem. This paper makes progress in the study of factoring and TFNP by showing that if a downward self-reduction for factoring exists, factoring is not only in PPAD but also in CLS. CLS was introduced by Daskalakis and Papadimitriou in 2011 [DP11], and in a recent breakthrough result, shown to be exactly the intersection $PPAD \cap PLS$ [FGHS21]. Recently, it was also shown that CLS is hard relative to a random oracle but with the additional assumption of #SAT hardness [CHKPRR19a]. More recently, in concurrent and independent work to ours, it was shown that UEOPL (and hence CLS) hardness can be based on any downward self-reducible problem family that additionally satisfied a randomized batching property, and for which the Fiat-Shamir heuristic protocol is unambiguously sound [BCHKLPR22]. This complements our results in interesting ways; we base hardness of UEOPL on any problem in unique TFNP that is d.s.r., while theirs is based on *any* problem that is d.s.r. (not necessarily total) but requires an additional batching property

and an assumption about the Fiat-Shamir heuristic for interactive protocols for that problem. The reductions in [Theorems 1.1](#) and [1.2](#) are reminiscent of the reductions in [\[CHKPRR19b\]](#) and [\[EFKP20\]](#) in their reliance on downward self-reducibility. The major differences between these works and our work are (i) these results work with specific d.s.r. problems (viz., sumcheck of low-degree polynomials and iterated squaring) while we work with any d.s.r. problem in TFNP; and (ii) these results rely on problems “outside” of NP while we rely on problems in TFNP. This allows us to do away with non-interactive proofs/arguments that the aforementioned works rely on (since a solution of a TFNP problem is itself a proof) and generalise/abstract out the assumption (i).

Organization

The rest of this paper is organized as follows. We first begin with some preliminaries in [Section 2](#) where we recall the definitions of various subclasses of problems in TFNP and the notion of downward self-reducibility. In [Section 3](#), we show that some of the classical PLS-complete problems such as ITER, SINK-OF-DAG are downward self-reducible, thus proving [Theorem 1.1](#). Next, in [Section 4](#), we show that every downward self-reducible problem in TFNP is contained in PLS, thus proving [Theorem 1.2](#). In [Section 5](#), we observe that if the search problem is in TFUP, then the reduction in [Section 4](#) can be strengthened to show that the problem is contained in CLS (thus, proving [Theorem 1.3](#)). This immediately yields [Corollary 1.4](#) on the consequence of the downward self-reducibility of factoring. We finally conclude with some closing thoughts in [Section 6](#).

2 Preliminaries

Notation. Throughout this paper $\|$ denotes concatenation of strings. We impose an ordering on strings in $\{0, 1\}^n$ lexicographically, e.g. for $x = 1011$ and $y = 1001$, $x > y$. Sometimes we will write $[T]$ to represent the subset of strings $\{0, 1\}^{\lceil \log(T) \rceil}$ that are at most T as binary numbers.

2.1 Search Problems

We begin by recalling the definitions of search problems, TFNP, and its subclasses.

Definition 2.1 (Search problem). A search problem is a relation $R \subset \{0, 1\}^* \times \{0, 1\}^*$. We will variously write xRy , $(x, y) \in R$ and $R(x, y) = 1$ to mean that (x, y) is in the relation R , or equivalently that y is a solution to x in the search problem R . A Turing Machine or algorithm M solves R if for any input x , $M(x)$ halts with a string y on the output tape such that $(x, y) \in R$.

Definition 2.2 (NP search problem). A search problem R is an NP search problem if (1) there exists a polynomial $p(\cdot)$ such that for every $x \in \{0, 1\}^*$, the set $Y_x = \{y \in \{0, 1\}^* : (x, y) \in R\} \subseteq \{0, 1\}^{p(n)}$ and (2) there exists a polynomial time Turing Machine M such that for all pairs of strings x, y , $M(x, y) = 1 \iff (x, y) \in R$. The class of NP search problems is denoted FNP.

Definition 2.3 (TFNP). A search problem R is *total* if for every $x \in \{0, 1\}^*$, there exists at least one y such that $(x, y) \in R$. The class of total NP search problems is called TFNP.

2.2 Local Search and Continuous Local Search

The study of total complexity is primarily interested in problems where totality – the fact that for any input there is a solution (that can be efficiently verified) – is the result of some combinatorial principle. A fundamental subclass of this kind is PLS, which intuitively captures problems whose

totality is guaranteed by a “local search”-like principle (that is, the principle that if applying some efficient step increases an objective but the objective is bounded, a local optimum must exist).

Definition 2.4 (SINK-OF-DAG). Given a successor circuit $S : \{0, 1\}^n \rightarrow \{0, 1\}^n$ such that $S(0^n) \neq 0^n$, and a valuation circuit $V : \{0, 1\}^n \rightarrow [2^m - 1]$ find $v \in \{0, 1\}^n$ such that $S(v) \neq v$ but $S(S(v)) = S(v)$ (a sink), or $V(S(v)) \leq V(v)$.

It is not hard to show that SINK-OF-DAG \in TFNP: since the range of V is finite, successively applying S must result in a sink of S , or a local maximum of V (note that exponentially many applications of S may be required to find this solution, so we cannot say that PLS is solvable in polynomial time). Any problem that is polynomial-time reducible to SINK-OF-DAG would also be in TFNP.

Definition 2.5 (PLS [JPY88]). The class PLS, for *polynomial local search*, consists of all search problems in TFNP that are polynomial-time reducible to SINK-OF-DAG.

In this paper, we will be interested in several other complete problems in PLS.

Definition 2.6 (ITER [Mor05]). Given a successor circuit $S : \{0, 1\}^n \rightarrow \{0, 1\}^n$ such that $S(0^n) > 0^n$, find $v \in \{0, 1\}^n$ such that $S(v) > v$, but $S(S(v)) \leq S(v)$.

Definition 2.7. ITER-WITH-SOURCE: Given a successor circuit $S : \{0, 1\}^n \rightarrow \{0, 1\}^n$ and a source $s \in \{0, 1\}^n$ such that $S(s) > s$, find $v \in \{0, 1\}^n$ such that $S(v) > v$, but $S(S(v)) \leq S(v)$.

Analogously to ITER versus ITER-WITH-SOURCE, one can define SINK-OF-DAG-WITH-SOURCE, where instead of the guarantee that $S(0^n) \neq 0^n$, we are given an additional input $s \in \{0, 1\}^n$ such that $S(s) \neq s$. The following is folklore:

Theorem 2.8. All of ITER, ITER-WITH-SOURCE, SINK-OF-DAG, SINK-OF-DAG-WITH-SOURCE are PLS-complete.

Proof. • **ITER \rightarrow SINK-OF-DAG:** Given an instance S of ITER, let $S' = S$ and $V' = S$; a solution v of (S', V') is trivially a solution of S .

• **SINK-OF-DAG \rightarrow ITER:** Given an instance (S, V) of SINK-OF-DAG, let $S' : \{0, 1\}^{2n} \rightarrow \{0, 1\}^{2n}$ be the circuit computing $S'(V(x)||x) = V(S(x))||S(x)$ and $S'(y||x) = y||x$ if $y \neq V(x)$, and source $s' = V(0^n)||0^n$. A solution v of S' must satisfy that either $S(S(v)) = S(v)$, or $V(S(S(v))) \leq S(v)$, so either v or $S(v)$ is a solution to S .

• **X \rightarrow X-WITH-SOURCE, where X \in {ITER, SINK-OF-DAG}:** to specify the additional argument s , let $s = 0^n$.

• **X-WITH-SOURCE \rightarrow X, where X \in {ITER, SINK-OF-DAG}:** replace the successor circuit S of either problem with S' that computes $S'(0^n) = s$ and $S'(x) = x$ for $x \neq 0^n$. □

Another fundamental class in the study of total complexity is PPAD, which similar to PLS is a sink-(or source)-finding problem, but where the existence of sinks is not guaranteed by a consistently increasing valuation, but rather by the presence of a predecessor circuit. In both PLS and PPAD, a principle (either the increasing valuation or the predecessor circuit) ensures that loops are not possible along a path.

Definition 2.9 (END-OF-LINE). In END-OF-LINE (or EOL) problem, we are given a *successor* circuit $S : \{0, 1\}^n \rightarrow \{0, 1\}^n$ and a *predecessor* circuit $P : \{0, 1\}^n \rightarrow \{0, 1\}^n$ such that $S(0^n) \neq 0^n$ and $P(0^n) = 0^n$ and we need to output either another *source* node (v s.t. $S(v) \neq v$ and $P(v) = v$) or a *sink* node (v s.t. $P(v) \neq v$ but $S(v) = v$).

Definition 2.10 (PPAD [Pap94]). The class PPAD, for *polynomial parity arguments on directed graphs*, consists of all search problems in TFNP that are polynomial-time reducible to END-OF-LINE.

In this paper we will be interested in the class CLS, which was originally defined with respect to a *continuous* analogue of PLS and PPAD. However, it has recently been shown to have a much simpler complete problem which is the natural unification of PLS and PPAD:

Definition 2.11 (END-OF-POTENTIAL-LINE [FGMS20]). Given a *successor* circuit $S : \{0, 1\}^n \rightarrow \{0, 1\}^n$ and a *predecessor* circuit $P : \{0, 1\}^n \rightarrow \{0, 1\}^n$ such that $S(0^n) \neq 0^n$ and $P(0^n) = 0^n$, as well as a valuation circuit $V : \{0, 1\}^n \rightarrow [2^m - 1]$, find either

- a “PPAD-like solution”: v s.t. $S(v) \neq v = P(v)$ or $P(v) \neq v = S(v)$
- a “PLS-like solution”: v s.t. $S(v) \neq v$, $P(S(v)) = v$ and $V(S(v)) \leq V(v)$.

Definition 2.12 (CLS, defined in [DP11], completeness of END-OF-POTENTIAL-LINE proved in [GHJMPRT22]). The class CLS, for *continuous local search*, is the class of total search problems polynomial time reducible to END-OF-POTENTIAL-LINE.

A recent result surprised the TFNP community:

Theorem 2.13 ([FGHS21]). $\text{CLS} = \text{PLS} \cap \text{PPAD}$.

We think of END-OF-LINE and END-OF-POTENTIAL-LINE instances as defining an ordered “line” consisting of the vertices $0^n, S(0^n), S^2(0^n), \dots$. If there were exactly one “line” in the instance, the solution to these problems would be unique. We will be interested in a version of END-OF-POTENTIAL-LINE that allows an additional kind of solution, namely one which demonstrates the existence of at least two lines.

Definition 2.14 (UNIQUEEOPL [FGMS20]). Given an instance (S, P, V) of END-OF-POTENTIAL-LINE with vertex set $\{0, 1\}^n$, find a solution to the original problem *or* two points $u \neq v \in \{0, 1\}^n$ such that $x \neq S(x)$, $y \neq S(y)$ and either $V(x) = V(y)$ or $V(x) < V(y) < V(S(x))$.

Observe that either $V(x) = V(y)$ or $V(x) < V(y) < V(S(x))$ in the new type of solution above implies the existence of at least two “lines” in the graph represented by (S, P, V) . This problem leads to the definition of a natural subclass of CLS:

Definition 2.15 (UEOPL [FGMS20]). The class UEOPL is the class of total search problems polynomial time reducible to UNIQUEEOPL.

Proposition 2.16 ([FGMS20]). UEOPL is a subclass of CLS.

2.3 Downward-self-reducibility

Definition 2.17. A search problem R is *downward self-reducible* (d.s.r) if there is a polynomial time oracle algorithm for R that on input $x \in \{0, 1\}^n$ makes queries to an R -oracle of size strictly less than $n = |x|$.

In this definition of downward self-reducible, the number of *bits* of the input must be smaller in oracle calls. Indeed, both TQBF and SAT are d.s.r. in this sense by the usual trick of instantiating the first quantified variable x_1 , and *simplifying the resulting Boolean expression* to remove x_1 entirely. However, there is a related notion of downward self-reducibility which can be defined to certain classes of structured problems or languages, where the *number of variables* of the instance decreases in oracle calls, even if the number of bits of the sub-instances does not decrease. A circuit problem is one where inputs are always circuits $C : \{0, 1\}^n \rightarrow \{0, 1\}^m$ under some fixed representation of circuits, and solutions are inputs to the circuit of the form $s \in \{0, 1\}^n$.

Definition 2.18. A circuit problem is circuit-d.s.r. if there is a polynomial time oracle algorithm for R that on input $C : \{0, 1\}^n \rightarrow \{0, 1\}^m$ makes queries to an R -oracle on circuits with $\nu \leq n$ input bits, and $\mu \leq m$ output bits, and $\nu + \mu < n + m$ (at least one of ν or μ is smaller than n or m respectively).

Throughout the rest of the paper we will only consider circuit problems where the size of the input circuit $|C|$ is at least n and at most polynomial in n and m ; this way an algorithm's complexity is equivalent as a function of the number of inputs and outputs or of the actual input size (that is, $|C|$).

Definition 2.19. We say that a circuit problem is circuit-d.s.r. *with polynomial-blowup* if it is circuit d.s.r and furthermore on input C with $n + m$ inputs and outputs, it makes oracle queries on circuits of size at most $|C| + \text{poly}(nm)$.

The reader might immediately ask: what is the relation between these notions of self-reducibility? It might seem one is a weaker version of the other, but this does not seem to be the case. In fact, the authors initially believed this was indeed the case, but despite this intuition, attempts at proving it failed! For circuit problems in TFNP, it would be interesting if there any connection between these two different notion of downward self-reducibility.

3 Downward self-reducibility of PLS-complete problems

In this section, we initiate the study of downward self-reducibility in TFNP by showing that of the four (4) canonical PLS-complete problems, three (3) are circuit-d.s.r. and one is both circuit-d.s.r. and d.s.r. Showing that a circuit problem is d.s.r. is trickier, as it requires not only reducing the input and output bits but also *shrinking the entire circuit representation*.

Theorem 3.1. ITER-WITH-SOURCE is d.s.r. and circuit-d.s.r. with polynomial blowup.

The following definition will be useful for the proof of [Theorem 3.1](#):

Definition 3.2. Given an (acyclic) circuit $C : \{0, 1\}^n \rightarrow \{0, 1\}^m$, the input restriction of C on input bit $i \in [n]$ to value $b \in \{0, 1\}$, denoted $C^{i \rightarrow b}$, is the circuit obtained by setting the input i to b , and then removing input bit i by "propagating" its value through the circuit (the exact process is given following this definition). Similarly, the output restriction of C formed by excluding output bit $j \in [m]$, denoted $C_{\setminus j}$, is defined by removing the j -th output bit and propagating the effect throughout the circuit.

Propagating a restriction is intuitive, but the details are given for completeness. For an input restriction $C^{i \rightarrow b}$, initially, input bit i is replaced by the constant value b . In polynomial time the circuit can be partitioned into layers $1, \dots, L$ such that a gate in later ℓ has inputs in layers $\ell' < \ell$ and layer 1 are the input bits. For each layer $\ell = 2, \dots, L$, we modify every gate G in layer ℓ that has as at least one input that was replaced by a constant in an earlier layer:

- if $G = \text{NOT}$ and its input was replaced with constant β , replace G with constant $\neg\beta$;
- if $G \in \{\text{AND}, \text{OR}\}$ and both input bits ι, κ were replaced with constants, replace G with the value $(\iota G \kappa)$;
- if $G = \text{AND}$ and only input bit ι was set to constant β , if $\beta = 0$ replace G by 0, and if $\beta = 1$ remove G by replacing any outgoing wires from G into layers $\ell' > \ell$ with outgoing wires from κ ;
- if $G = \text{OR}$ and only input bit ι was set to constant β , if $\beta = 1$ replace G by 1, and if $\beta = 0$ remove G by replacing any outgoing wires from G into layers $\ell' > \ell$ with outgoing wires from κ .

For an output restriction $C_{\setminus j}$ we start by removing output bit j , and then for each layer $\ell \in \{L - 1, \dots, 1\}$, remove any gate which feeds only into gates that were removed in previous layers. Both input and output restrictions can be constructed in polynomial time. Both kinds of restriction have a smaller circuit representation in any reasonable representation of circuits¹, that is, $|C^{i \rightarrow b}| < |C|$ and $|C_{\setminus j}| < |C|$.

Proof of Theorem 3.1. The key observation is that we can first query a restriction of S to the first half of inputs (that is, those beginning with a 0) and either obtain a sink, or else a string whose successor is in the second half of inputs (those beginning with 1). With this new source, we can reduce the problem to finding a sink in the second half of inputs.

Concretely, given an instance (S, s) such that $S : \{0, 1\}^n \rightarrow \{0, 1\}^n$ and $S(s) > s$, define $S_0 = (S^{1 \rightarrow 0})_{\setminus 1}$ and $S_1 = (S^{1 \rightarrow 1})_{\setminus 1}$, i.e., the input restriction with the first bit set to 0 or 1 respectively, and the output restriction obtained by excluding the first output bit 1. These restrictions can be constructed efficiently and are necessarily smaller in total representation than S , and $S_0(x) = S(0||x)_{1:n-1}$, $S_1(x) = S(1||x)_{1:n-1}$ where the subscript denotes returning that range of bits.

Now, if $s > 2^{n-1}$, then $(S_1, s_{1:n-1})$ defines a valid smaller instance of ITER-WITH-SOURCE and given solution w to $(S_1, s_{1:n-1})$, $v = 1||w$ is a solution to (S, s) . Otherwise, assuming $s < 2^{n-1}$, we query the oracle for $(S_0, s_{1:n-1})$. Any solution w of this sub-instance for which $S(0||w) < 2^{n-1} - 1$ and $S(S(0||w)) < 2^{n-1} - 1$ directly yields a solution for S , namely $v = 0||w$. Otherwise, there are several possible kinds of "false" solutions. In each case we will have either a solution to S , or otherwise a vertex $\sigma \in \{0, 1\}^n$ such that $\sigma > 2^{n-1}$ such that $S(\sigma) > \sigma$ and in which case we query $(S_1, \sigma_{1:n-1})$ for solution z , and output $v = 1||z$.

Again, if w is the solution returned by the downward ITER oracle on $(S_0, s_{1:n-1})$:

1. If $S(0||w) \geq 2^{n-1}$, then either $S(S(0||w)) < S(0||w)$ (and then $0||w$ is a solution for (S, s)) or we set $\sigma = S(0||w)$ and continue as above.
2. If $S(0||w) < 2^{n-1}$ but $S(S(0||w)) \geq 2^{n-1}$, then either $S(S(S(0||w))) < S(S(0||w))$ (in which case $S(0||w)$ is a solution for (S, s)), or set $\sigma = S(S(0||w))$.

Note, that in addition to shrinking the circuit size, the oracle sub-instances have one less input and output bit, so ITER-WITH-SOURCE is also circuit-d.s.r. with sub-constant blowup. \square

Theorem 3.3. ITER, SINK-OF-DAG and SINK-OF-DAG-WITH-SOURCE are circuit-d.s.r. with polynomial blowup.

¹We note that is possible to come up with a "pathological" circuit representation such that restrictions with propagation of this kind result in a longer circuit representation, even though the number of nodes in the DAG encoded by the representation has decreased.

For the notion of circuit-d.s.r. to make sense for SINK-OF-DAG and SINK-OF-DAG-WITH-SOURCE, which as defined have several input circuits with the same domain, we think of the inputs to these problems as a single circuit which reuses the same input bits and concatenates the outputs. This works because although these problems are defined with multiple circuit inputs, a solution is a single input (that satisfies some relation depending on the circuits). That is, to be circuit-d.s.r. the oracle sub-instances must decrease the input size for *each* circuit, or the output size for *any* circuit.

Proof. 1. **ITER:** We follow roughly the same algorithm as in the proof of [Theorem 3.1](#) but with $s = 0^n$; when we query $S_0 = (S^{1 \rightarrow 0})_{\setminus 1}$ we either obtain a solution to the original instance, or a vertex $\sigma \in \{0, 1\}^n$ such that $\sigma \geq 2^{n-1}$ and $S(\sigma) > \sigma$. The problem is that we cannot just input-restrict the first bit to 1, since the source of this new circuit must be 0^{n-1} and $S_1(0^{n-1}) = S(2^{n-1})$ and it may not be that $\sigma = 2^{n-1}$; therefore, we modify $S_1 = (S^{1 \rightarrow 1})_{\setminus 1}$ to "redirect" the input 0^{n-1} to the new source by adding a component that checks if the input is 0^{n-1} , and if so feeds a hardcoded $\sigma_{1:n-1}$ into the rest of the circuit, and otherwise feeds in the input unchanged. This gadget can be implemented with $O(n)$ additional gates, so ITER is circuit-d.s.r with polynomial-blowup.

2. **Sink-of-DAG-with-source:** Let $V' = V_{\setminus 1}$ be the output restriction of the first bit; we query (S, V') and obtain solution w . If both $V(w) < 2^{m-1}$ and $V(S(w)) < 2^{m-1}$ then w is a solution to the original instance; otherwise, this gives us a $\sigma \in \{0, 1\}^n$ such that $V(\sigma) \geq 2^{m-1}$ and $S(\sigma) \neq \sigma$. For the next subquery, the valuation circuit will again be V' , but we also use a modified successor circuit S' : on input $x \in \{0, 1\}^n$ it checks whether $V(x) < V(\sigma)$ and if so outputs x , and otherwise outputs $S(x)$. We also set $s' = \sigma$. The modification to S' ensures no x for which $V(x) < V(\sigma)$ can be a solution to (S', V', s') . Since this algorithm only ever decreases the *output* bits of of the valuation circuit in oracle calls, it is necessary to check that it can be solved when output bits cannot be decreased any further, that is when $m = 1$; indeed, in that case the problem is trivial can be found by applying S just once. S' needs to be implemented carefully to ensure polynomial-blowup: instead of copying V , the output bits of V are moved down into a non-output layer, and S' compares these bits to the hardcoded $S(\sigma)$ ($n - 1$ of these bits are also reused directly to compute V'). This gadget can be implemented with $O(m)$ additional overhead.
3. **Sink-of-DAG:** The construction is exactly as above, except S' has an additional component which checks if $x = 0^n$, and if so feeds a hardcoded σ into the rest of the circuit for S' and V' . This gadget can be implemented with $O(n)$ additional overhead. \square

A natural question is whether every PLS-complete problem is d.s.r. (or circuit d.s.r. if it is a circuit problem). The difficulty with answering this question is that, as evidenced in this section, downward self-reducibility seems to depend on the details of the input representation of the problem. This is analogous to the case of downward self-reducibility in decision problems: SAT, and more generally TQBF, are known to be d.s.r., but the property is not known to be true for *all* NP- or PSPACE-complete problems. The same ideas to show any NP-complete problem has a search-to-decision self-reduction does not work when considering *downward* self-reducibility, either in the decisional landscape or in TFNP. If, say the reduction from SAT to A squares the input length, it is not clear how to use downward self-reducibility of SAT to get it for A .

Open Problem 3.4. Is every PLS-complete problem downward self-reducible?

4 Downward self-reducible problems are in PLS

In the previous section we proved that the canonical PLS-complete problems are downward self-reducible. The natural question is whether the reverse is true. Indeed, we are able to show the following theorem (restated from the introduction):

Theorem 1.2. *Every downward self-reducible problem in TFNP is in PLS.*

The “moral” of these results (Theorems 1.2 and 3.1 that some PLS-complete problems are d.s.r. and d.s.r. problems are in PLS) is the following corollary. An equivalent one exists in the decisional landscape, namely, that a hard language is d.s.r. iff there is a hard language in PSPACE.

Corollary 4.1. *A problem in PLS is hard if and only if a downward-self reducible problem in TFNP is hard. Here, “hard” can mean worst-case hard, or average-case hard with respect to some efficiently sampleable distribution.*

Proof. This is essentially restatement of Theorems 1.2 and 3.1 together. On one hand, if a problem in PLS is hard (either worst-case, or average-case with respect to some sampleable distribution) then any PLS complete problem is also hard (with the same notion of hardness carrying over); in particular, ITER \in PLS is hard. Conversely if a d.s.r. problem in TFNP is hard, then by Theorem 1.2 the same problem is in PLS. \square

Proof of Theorem 1.2. If R is downward self-reducible with algorithm A' , then there exists the following (inefficient) depth-first recursive algorithm A for R : on input x , simulate A' and for any oracle call to an instance x' where $|x'| < x$, run $A(x')$ to obtain a solution, and then continue the simulation of A' . The idea of reducing R to SINK-OF-DAG is to represent its intermediate states of this algorithm as vertices of a DAG. Of course, this graph may have exponential size.

Intuitively, we will represent the intermediate states of A as a list of the recursive instances invoked by A up to the current depth of recursion, along with the solutions to already-solved recursive instances. We can assume without loss of generality that on inputs of length n , A makes exactly $p(n)$ downward calls (this can be done by “padding” the algorithm with unused additional calls to some fixed sub-instance) and that solutions have size $q(n)$.

The SINK-OF-DAG vertex set will consist of strings $s \in \{0, 1\}^{P(n)}$ where $P(n) = n + q(n) + \sum_{i=1}^{n-1} p(n-i+1) \times (n-i+q(n-i))$ which we will call *states* or *nodes*. Note $P(n) < p(n)q(n)n^2$ and is hence polynomial size. We interpret a string $s \in \{0, 1\}^{P(n)}$ as a table $s[\cdot, \cdot]$ indexed by the depth of recursion $i \in \{0, \dots, n-1\}$, and index j of the recursive call at depth i , with $j \in [p(n-i+1)]$ (except for $i=0$ where j can only be 1). Each $s[i, j] \in \{0, 1\}^{n-i} \times B^{q(n-i)}$ can contain an instance of R , and possibly a solution. That is, $s[i, j]$ can be one of: (1) a pair (ξ, \perp) of an R -instance ξ and special symbol \perp , (2) a pair (ξ, γ) of an R -instance ξ and purported solution γ , or (3) the pair (\perp, \perp) representing no instance.

On input x , the starting state s_0 is defined by setting $s_0[0, 1] = (x, \perp)$ and $s[i, j] = (\perp, \perp)$ for all other i, j . We say that a state s is *valid* for input x based on the following recursive definition: $s[0, 1] = (x, \perp)$, and there exists $j \in [p(n)]$ (where $n = |x|$) such that

1. $s[1, j'] = (\perp, \perp)$ for all $j' > j$,
2. for $j' \leq j$, $s[1, j'] = (x_{j'}, y_{j'})$ where $y_{j'}$ is a correct solution for $x_{j'}$ (except for $j' = j$ where $y_{j'}$ can be \perp),
3. for each $j' \leq j$, $x_{j'}$ is the next $(n-1)$ -size subquery made by A on input x given previous subqueries $x_1, \dots, x_{j'-1}$ and respective solutions $y_1, \dots, y_{j'-1}$,

Algorithm 1: VALID? (Algorithm for whether state s is valid in [Theorem 1.2](#))

Input:

- State $s \in \{0, 1\}^{P(n)}$, $s[i, j] \in \{0, 1\}^{n-i} \times \{0, 1\}^{q(n-1)}$ for $i \in [n-1]$ and $j \in [p(n-i+1)]$.
- Input x to search problem R .

Output: TRUE if state s is valid for input x , else FALSE

Set $(\xi, y) \leftarrow s[0, 1]$

if $\xi \neq x$ then return FALSE;

if $y \neq \perp$ then return " $(x, y) \stackrel{?}{\in} R$ ";

Set $j \leftarrow 1$

while $(x_j, y_j) := s[1, j] \neq (\perp, \perp)$ do

 if x_j is not the next input queried by A after x_1, \dots, x_{j-1} with solutions y_1, \dots, y_{j-1}
 then return FALSE;

 if $y_j \neq \perp$ and $(x_j, y_j) \notin R$ then return FALSE;

 Set $s_{x_j} \in \{0, 1\}^{P(n-1)}$ as follows: $s_{x_j}[0, 1] = s[1, j]$, and $s_{x_j}[\iota, \kappa] = s[\iota + 1, \kappa]$ for
 $\iota \in [n-1]$.

 if VALID?(s_{x_j}, x_j) then break while loop;

 else return FALSE;

 Set $j \leftarrow j + 1$

for $j' \in \{j + 1, \dots, p(n)\}$ do

 if $s[1, j'] \neq (\perp, \perp)$ then return FALSE;

return TRUE.

4. if $y_j = \perp$, then letting $s_{x_j} \triangleq s[i', j']$ be the table defined by $s_{x_j}[0, 1] = (x_j, \perp)$ and $s[i', j'] = s[i' + 1, j']$ for $i' \in [n-2]$ and $j' \in p((n-1) - i' + 1)$, s_{x_j} is a valid state for input x_j .

See ?? 1 for the pseudocode for checking if a state is valid.

Condition 4 says that, letting s_{x_j} be the subtable of s corresponding to the computation of A to solve the instance x_j —that is, (x_j, \perp) in its first cell $s_{x_j}[0, 1]$, and the rest of $s[\iota, j]$ for $\iota > 2 - s_{x_j}$ is valid.

Validity can be checked efficiently by the recursive algorithm implicit to this definition; its time complexity is $T(n) \leq \text{poly}(p(n)) + T(n-1) = \text{poly}(n)$.

The successor function S is also defined recursively. For invalid states the successor function returns the state itself; as they cannot be reached this makes them isolated nodes that cannot be a solution to the SINK-OF-DAG instance. For valid states, let $(x, y) = s[0, 1]$. If $y \neq \perp$ is a solution, then $S(s) := s$ (the node is a sink as it contains a solution to x). Otherwise, if $y = \perp$ (and s represents an intermediate state of the algorithm A), let $j \in p[n]$ be the index of the last subquery (of size $n-1$) encoded by s , exactly as in conditions (2)-(4) in the algorithm for validity, with corresponding cell $(x_j, y_j) = s[1, j]$. If $y_j \neq \perp$ and $j < p(n)$, we simulate A to obtain the next subquery x_{j+1} of size $n-1$ (given the previous subqueries x_1, \dots, x_j with solutions y_1, \dots, y_j) and set the $[1, j+1]$ -cell to (x_{j+1}, \perp) (that is, $S(s)[1, j+1] = (x_{j+1}, \perp)$ and $S(s)[i', j'] = s[i', j']$ for all other $(i', j') \neq (1, j+1)$). Note that this part is non-recursive (since by definition, x_{j+1} is the very next recursive call made by A) and requires polynomial time. If $y_j \neq \perp$ and $j = p(n)$, we similarly simulate A to obtain the final solution y to x since all the subqueries made by A on input x have been obtained with their answers; we set $S(s)[1, 0] = (x, y)$ and set all other cells to (\perp, \perp) . Finally

Algorithm 2: S (Algorithm for the successor circuit S in [Theorem 1.2](#))

Input:

- State $s \in \{0, 1\}^{P(n)}$ where $s[i, j] \in \{0, 1\}^{n-i} \times \{0, 1\}^{q(n-1)}$ for $i \in [n-1]$ and $j \in [p(n-i+1)]$.
- Input x to search problem R , and $n := |x|$.

Output: Successor state of state x on input x

```

if VALID?( $s, x$ ) =FALSE then return  $s$ ;
Set  $(x, y) \leftarrow s[0, 1]$ 
if  $y \neq \perp$  then return  $s$ ;
Let  $j \in [p(n)]$  such that  $\forall j' > j, s[1, j'] = (\perp, \perp)$ 
Set  $(x_j, y_j) \leftarrow s[1, j]$ 
if  $j = p(n)$  and  $y_j \neq \perp$  then
  Simulate  $A(x)$  after queries  $x_1, \dots, x_j$  with solutions  $y_1, \dots, y_j$  to obtain  $y$  such that
   $(x, y) \in R$ .
  Let  $s' \in \{0, 1\}^{P(n)}$  with  $s'[0, 1] = (x, y)$  and  $(\perp, \perp)$  elsewhere.
  return  $s'$ .
else if  $y_j \neq \perp$  then
  Simulate  $A(x)$  after queries  $x_1, \dots, x_j$  with solutions  $y_1, \dots, y_j$  to obtain next query
   $x_{j+1}$ .
  Let  $s' := s$ , but  $s'[1, j+1] := (x_{j+1}, \perp)$ .
  return  $s'$ .
else
  Let  $s_{x_j} \subset s$  be the subset of cells of  $s$  as indexed by  $s_{x_j}[0, 1] = s[1, j]$ , and
   $s_{x_j}[\iota, \kappa] = s[\iota+1, \kappa]$  for  $\iota \in [n-1]$ .
  Let  $s'_{x_j} = S(s_{x_j}, x_j)$ .
  Let  $s' := s$  with subrange  $s_{x_j}$  replaced by  $s'_{x_j}$ .
  return  $s'$ .

```

if $y_j = \perp$, s represents an intermediate state of A currently at a deeper level of recursion. Construct s_{x_j} as in the validity algorithm (s_{x_j} is a subset of the cells of s , namely (x_j, \perp) as cell $[0, 1]$, and the rest of s at levels 2 and beyond); let $\sigma_{x_j} = S(s_{x_j})$ be the successor of s_{x_j} when simulating A on s_{x_j} ; $S(s)$ returns s , replacing the subset of cells corresponding to s_{x_j} with σ_{x_j} . Observe that the time to compute the successor is at most the time to simulate A between recursive calls or the time to compute the successor for a smaller sub-instance, that is $T_s(n) \leq \text{poly}(n) + T_s(n-1) = \text{poly}(n)$. See ?? 2 for the pseudocode for checking if a state is valid.

Finally, for valid nodes, the potential circuit V will return how far the state is in the depth-first recursive algorithm. The position $\pi(s) \in \{0, 1\}^{cn^2}$ of a valid state s in the simulation of A can be computed as follows. Let j_i be the last cell at each level $i = 1, \dots, n-1$ that is not (\perp, \perp) (i.e., j_i is the last index such that $s[i, j_i]$ contains a subquery). If s is a solution (sink), $\pi(s) = \Pi(n)$, and otherwise $\pi(s) = 1 + \sum_{i=1}^{n-1} \min\{1, j_i\} + \max\{0, j_i - 1\} \times \Pi(n-i)$.

It is clear that the only valid node to not have a valid successor is the one corresponding to the final configuration of A , with solution y , a solution to the SINK-OF-DAG instance gives a solution to x (by reading $(x, y) = s[0, 1]$). \square

In [Sections 2.3](#) and [3](#) we discussed the related notion of circuit-d.s.r. in which the sub-instances

to the oracle need not be smaller, but the number of input or output bits decreases. While we do not know the exact connection between these notions, in particular whether they are equivalent for circuit problems, the following theorem establishes that it is enough for a TFNP problem to be downward self-reducible in either sense to guarantee membership in PLS.

Theorem 4.2. *Let R be a circuit problem in TFNP that is circuit-d.s.r. with polynomial blowup. Then $R \in \text{PLS}$.*

Proof. Let A' be the algorithm from the definition of circuit-d.s.r. As in d.s.r., A' can be used to construct an (inefficient) depth-first recursive algorithm A that solves R : simulate $A'(C)$, and on oracle calls to a sub-instance C' , run $A(C')$. Critically, A terminates even though $|C'|$ may be greater than $|C|$, because the sum of input and output bits $n + m$ decreases with each level of recursion.

The reduction to SINK-OF-DAG is the same as in the proof of [Theorem 1.2](#): the graph will represent intermediate states of the depth-first recursive algorithm A . However, the size of states will be different. Without loss of generality assume R makes exactly $p(|C|)$ sub-instance calls, and if R is circuit-d.s.r. with polynomial-blowup, we have that each sub-instance has size at most $|C| + (nm)^c$ for some $c > 0$. Hence, for a circuit of size $|C|$ we have $P(|C|) = |C| + n + \sum_{i=1}^{n+m-1} p(|C| + i(nm)^c)(|C| + i(nm)^c)n \leq \text{poly}(n, m)$ bits suffices to represent intermediate states. \square

Note that the size of representing intermediate states of A is where it is critical that R be circuit-d.s.r with polynomial blowup; if instead, for instance, we had the guarantee that sub-instances have size $f(|C|)$, then at depth- i sub-instances would have size $f^{(i)}(|C|)$ which even for the modest function $f(\alpha) = 2\alpha$ would give sub-instances of size $2^i|C|$, which are too big to represent as SINK-OF-DAG nodes.

Finally, we can give a characterization of PLS-hardness analogous to [Corollary 4.1](#):

Corollary 4.3. *A problem in PLS is hard if and only if there exists a hard circuit problem in TFNP that is circuit-d.s.r. with polynomial blowup.*

Proof. This is a restatement of [Theorems 2.8](#) and [4.2](#) together. \square

5 Unique TFNP and Downward self-reducibility

Several highly structured problems in TFNP have the additional property that for every input, the solution is *unique*. Most of these problems are algebraic in nature— such as factoring, and finding the discrete logarithm of an element of a cyclic group for a certain generator. We find, perhaps surprisingly, that d.s.r. problems with the additional condition of unique solutions are not only in PLS, but in CLS!

Definition 5.1 ([\[HV21\]](#)). TFUP, for “unique” TFNP, is the class of problems in TFNP where for each input x there is exactly one solution y .

Definition 5.2. SINK-OF-VERIFIABLE-LINE (SVL): given a successor circuit $S : \{0, 1\}^n \rightarrow \{0, 1\}^n$, source $s \in \{0, 1\}^n$, target index $T \in [2^n]$ and verifier circuit $V : \{0, 1\}^n \times [T] \rightarrow \{0, 1\}$ with the guarantee that for $(x, i) \in \{0, 1\}^n \times [T]$, $V(x, i) = 1$ iff $x = S^{i-1}(s)$, find the string $v \in \{0, 1\}^n$ s.t. $V(v, T) = 1$.

SVL is a *promise* problem; in particular, there is no way to efficiently check the guarantee about the verifier circuit V . A solution is guaranteed to exist if the promise is true, and the solution must be unique. It is possible to modify the problem (by allowing solutions that witness an “error” of the verifier circuit) in order to obtain a search problem in TFNP, but for our purposes SVL is sufficient:

Theorem 5.3 ([FGMS20]). *SVL is polynomial-time reducible to UEOPL.*

Bitansky, Paneth and Rosen [BPR15], building on the results of Abbot, Kane and Valiant [AKV04], used a reversible pebbling game (also known as the east model) to make the computation of the successor circuit reversible and show that SVL is polynomial-time reducible to PPAD (see [AKV04, Section 4.3] and [BPR15, Section 3]). Hubaček and Yogev [HY20] then strengthened this result by showing that the same reduction shows SVL is polynomial-time reducible to CLS. Finally, in recent work this was strengthened by observing that the same reduction works for UEOPL [FGMS20].

Lemma 5.4. *Every d.s.r. problem and every circuit-d.s.r. circuit problem in TFUP reduces to SVL.*

Proof. Given a d.s.r. (or circuit-d.s.r.) problem R , we construct a graph with successor circuit S exactly as in the proof of Theorem 1.2 or Theorem 1.2. The key observation is that if R has unique solutions, we can construct a verifier circuit V as in Definition 5.2 for which the SVL promise holds true.

Let $\Pi(n)$ denote the total length of the path from the initial state s_0 to the unique sink in the SINK-OF-DAG instance from the earlier theorem proofs; we have $\Pi(n) = p(n) \times \Pi(n-1) = p(n) \times p(n-1) \cdots \times p(1)$. While $\Pi(n)$ may be exponential it can be represented with $\sum \log(p(n-i)) = O(n^2)$ bits; that is, the distance of each valid node from s_0 in the graph can be represented with cn^2 for some constant c . The position $\pi(s) \in \{0, 1\}^{cn^2}$ of a valid state s in the simulation of A can be computed as follows. Let j_i be the last cell at each level $i = 1, \dots, n-1$ that is not (\perp, \perp) (i.e., j_i is the last index such that $s[i, j_i]$ contains a subquery). If s is a solution (sink), $\pi(s) = \Pi(n)$, and otherwise $\pi(s) = 1 + \sum_{i=1}^{n-1} \min\{1, j_i\} + \max\{0, j_i - 1\} \times \Pi(n-i)$. Equivalently this can be computed recursively; using the sub-table notation from Theorem 1.2, where j is the last cell at level 1 that is not (\perp, \perp) , $\pi(s) = 1 + (j-1)\Pi(n-1) + \pi(s_{x_j})$. Either way, Π and π can be computed efficiently. Since the sink is the $\Pi(n)$ -th state, the SVL instance will have target $T = \Pi(n)$.

Critically, since $R \in \text{TFUP}$, for each $i \in [\Pi(n)]$ there is exactly one state $s \in \{0, 1\}^{P(n)}$ such that $\pi(s) = i$: to see this, for every sub-instance x' invoked by A , there is a unique solution y' , and given a sequence of previous sub-instances and solutions at a given level of recursion, the next sub-instance is uniquely determined by A . We define $V(s, i) = 1$ if s is a valid state (as in the proof of Theorem 1.2) and $\pi(s) = i$. \square

As a consequence of Theorem 5.3 and the previous lemma, we have the following theorem (restated from the introduction).

Theorem 1.3. *Every downward self-reducible problem in TFUP is in UEOPL.*

This result yields an interesting application to the study of number-theoretic total problems such as factoring. It is an open problem whether there exists an algorithm for factoring that uses oracle calls on smaller numbers. The following observations are evidence that the answer may be no.

Definition 5.5. ALLFACTORS is the problem of, given an integer, listing its non-trivial factors in increasing order, or “prime” if it is prime. FACTOR is the problem of returning a non-trivial factor, or “prime” if it is prime.

ALLFACTORS is in TFUP. ALLFACTORS and FACTOR are almost the same problem: FACTOR is Cook reducible to ALLFACTORS with one oracle call and ALLFACTORS is Cook reducible to FACTOR with $\log(\text{input size})$ oracle calls. A consequence of Theorem 1.3 is the following:

Corollary 1.4. *If FACTOR or ALLFACTORS is downward self-reducible, then FACTOR and ALLFACTORS \in UEOPL.*

Proof. If either problem is downward self-reducible, so is the other. Given a downward self-reduction A of FACTOR, to solve ALLFACTORS on input n , run A (replacing its downward oracle calls with those for ALLFACTORS) to obtain factor m , and then run the oracle on $\frac{n}{m}$. Given a downward self-reduction B of ALLFACTORS, to solve FACTOR on input n run B and return any factor—replace downward oracle calls to ALLFACTORS with ones to FACTOR and at most $\log(\log(n))$ additional downward queries. Hence if either problem is d.s.r., ALLFACTORS \in UEOPL, but since FACTOR \leq_p ALLFACTORS, FACTOR \in UEOPL. \square

FACTOR \in UEOPL, and hence in CLS, would be a surprising consequence to TFNP community. While it has been shown that FACTOR can be reduced to the TFNP-subclasses PPA and PPP, the intersection of which contains PPAD, it has been a long outstanding question whether FACTOR $\stackrel{?}{\in}$ PPAD. An algorithm for factoring N which queries numbers at most $N/2$ (this is what it means for the input size to shrink by at least one bit, but note that any number that could contain a non-trivial factor of N is at most $N/2$) would immediately place FACTOR not just in PPAD, but also in PLS (since CLS = PPAD \cap PLS)!

6 Discussion and Open Problems

In this paper we have initiated the study of downward self-reducibility, and more broadly self-reducibility, in TFNP. Naturally, a host of questions remain:

- What is the relationship between downward self-reducibility and circuit-downward self-reducibility for circuit problems in TFNP?
- What other problems in PLS are downward self-reducible? Is it possible every PLS-complete problem is downward self-reducible? As a first step, it would be helpful to prove that the suite of PLS-complete local constraint/clause maximization problems (where a specific neighborhood function is specified as part of the problem, such as FLIP or KERNIGHAN-LIN) are d.s.r.
- Another important notion of self-reducibility is that of *random* self-reducibility (r.s.r.): a problem R is r.s.r. if it has a worst-to-average case reduction. The details vary but one definition is R is r.s.r. if it has a randomized Las Vegas algorithm if there exists an algorithm solving R on $1/\text{poly}(n)$ -fraction of inputs. Some very important algebraic problems in TFNP, such as the discrete logarithm, and RSA inversion, are r.s.r. These problems are also in PPP, so a natural starting point is to ask whether all r.s.r. problems in TFNP are in PPP.

References

- [AKV04] TIM ABBOT, DANIEL KANE, and PAUL VALIANT. *On algorithms for Nash equilibria*, 2004. (manuscript). 14
- [All10] ERIC ALLENDER. *New surprises from self-reducibility*. In FERNANDO FERREIRA, HELIA GUERRA, ELVIRA MAYORDOMO, and JOÃO RASGA, eds., *Programs, Proofs, Processes (Abstract and Handout Booklet, 6th Conference on Computability in Europe (CiE))*, pages 1–5. Centre for Applied Mathematics and Information Technology, Dept. of Mathematics, University of Azores, Portugal, 2010. 2

- [Bal90] JOSÉ L. BALCÁZAR. *Self-reducibility*. J. Comput. Syst. Sci., 41(3):367–388, 1990. 2
- [BCHKLPR22] NIR BITANSKY, ARKA RAI CHOUDHURI, JUSTIN HOLMGREN, CHETHAN KAMATH, ALEX LOMBARDI, OMER PANETH, and RON D. ROTHBLUM. *PPAD is as hard as LWE and iterated squaring*. In EIKE KILTZ and VAIKUNTANATHAN VINOD, eds., *Proc. 20th International Theory of Crypt. Conf. (TCC)*, volume 13747 of *LNCS*. Springer, 2022. (to appear). [eprint.iacr:2022/1272](#). 3
- [BG20] NIR BITANSKY and IDAN GERICHTER. *On the cryptographic hardness of local search*. In THOMAS VIDICK, ed., *Proc. 11th Innovations in Theor. Comput. Sci. (ITCS)*, volume 151 of *LIPICs*, pages 6:1–6:29. Schloss Dagstuhl, 2020. [eprint.iacr:2020/013](#). 3
- [BO06] JOSHUA BURESH-OPPENHEIM. *On the TFNP complexity of factoring*, 2006. (manuscript). 3
- [BPR15] NIR BITANSKY, OMER PANETH, and ALON ROSEN. *On the cryptographic hardness of finding a Nash equilibrium*. In VENKATESAN GURUSWAMI, ed., *Proc. 56th IEEE Symp. on Foundations of Comp. Science (FOCS)*, pages 1480–1498. 2015. [ecc:2015/TR15-001](#), [eprint.iacr:2014/1029](#). 3, 14
- [CHKPRR19a] ARKA RAI CHOUDHURI, PAVEL HUBÁČEK, CHETHAN KAMATH, KRZYSZTOF PIETRZAK, ALON ROSEN, and GUY N. ROTHBLUM. *Finding a Nash equilibrium is no easier than breaking Fiat-Shamir*. In MOSES CHARIKAR and EDITH COHEN, eds., *Proc. 51st ACM Symp. on Theory of Computing (STOC)*, pages 1103–1114. 2019. [ecc:2019/TR19-074](#), [eprint.iacr:2019/549](#). 3
- [CHKPRR19b] ———. *PPAD-hardness via iterated squaring modulo a composite*, 2019. (manuscript). [eprint.iacr:2019/667](#). 4
- [DP11] CONSTANTINOS DASKALAKIS and CHRISTOS H. PAPADIMITRIOU. *Continuous local search*. In DANA RANDALL, ed., *Proc. 22nd Annual ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 790–804. 2011. 3, 6
- [EFKP20] NAOMI EPHRAIM, CODY FREITAG, ILAN KOMARGODSKI, and RAFAEL PASS. *Continuous verifiable delay functions*. In ANNE CANTEAUT and YUVAL ISHAI, eds., *Proc. 39th Annual International Conf. on the Theory and Appl. of Cryptographic Tech (EUROCRYPT), Part III*, volume 12107 of *LNCS*, pages 125–154. Springer, 2020. [eprint.iacr:2019/619](#). 4
- [FGHS21] JOHN FEARNLEY, PAUL W. GOLDBERG, ALEXANDROS HOLLENDER, and RAHUL SAVANI. *The complexity of gradient descent: $CLS = PPAD \cap PLS$* . In SAMIR KHULLER and VIRGINIA VASSILEVSKA WILLIAMS, eds., *Proc. 53rd ACM Symp. on Theory of Computing (STOC)*, pages 46–59. 2021. [arXiv:2011.01929](#). 3, 6
- [FGMS20] JOHN FEARNLEY, SPENCER GORDON, RUTA MEHTA, and RAHUL SAVANI. *Unique end of potential line*. J. Comput. Syst. Sci., 114:1–35, 2020. (Preliminary version in *46th ICALP*, 2019). [arXiv:1811.03841](#). 6, 14
- [GHJMPRT22] MIKA GÖÖS, ALEXANDROS HOLLENDER, SIDDHARTHA JAIN, GILBERT MAYSTRE, WILLIAM PIRES, ROBERT ROBERE, and RAN TAO. *Further collapses in TFNP*. In SHACHAR LOVETT, ed., *Proc. 37th Comput. Complexity Conf.*, volume 234 of *LIPICs*, pages 33:1–33:15. Schloss Dagstuhl, 2022. [arXiv:2202.07761](#), [ecc:2022/TR22-018](#). 6
- [GPS16] SANJAM GARG, OMKANT PANDEY, and AKSHAYARAM SRINIVASAN. *Revisiting the cryptographic hardness of finding a Nash equilibrium*. In MATTHEW ROBshaw and JONATHAN KATZ, eds., *Proc. 36th Annual International Cryptology Conf. (CRYPTO), Part II*, volume 9815 of *LNCS*, pages 579–604. Springer, 2016. [eprint.iacr:2015/1078](#). 3
- [HV21] PAVEL HUBÁČEK and JAN VÁCLAVEK. *On search complexity of discrete logarithm*. In FILIPPO BONCHI and SIMON J. PUGLISI, eds., *Proc. 46th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 202 of *LIPICs*, pages 60:1–60:16. Schloss Dagstuhl, 2021. [arXiv:2107.02617](#). 13

- [HY20] PAVEL HUBÁČEK and EYLON YOGEV. *Hardness of continuous local search: Query complexity and cryptographic lower bounds*. SIAM J. Comput., 49(6):1128–1172, 2020. (Preliminary version in *28th SODA*, 2017). [eccc:2016/TR16-063](#). 3, 14
- [Jer16] EMIL JERÁBEK. *Integer factoring and modular square roots*. J. Comput. Syst. Sci., 82(2):380–394, 2016. [arXiv:1207.5220](#). 3
- [JPY88] DAVID S. JOHNSON, CHRISTOS H. PAPADIMITRIOU, and MIHALIS YANNAKAKIS. *How easy is local search?* J. Comput. Syst. Sci., 37(1):79–100, 1988. (Preliminary version in *26th FOCS*, 1985). 5
- [Mor05] TSUYOSHI MORIOKA. *Logical approaches to the complexity of search problems: : proof complexity, quantified propositional calculus, and bounded arithmetic*. Ph.D. thesis, University of Toronto, Canada, 2005. 5
- [MP91] NIMROD MEGIDDO and CHRISTOS H. PAPADIMITRIOU. *On total functions, existence theorems and computational complexity*. Theoret. Comput. Sci., 81(2):317–324, 1991. 3
- [Pap94] CHRISTOS H. PAPADIMITRIOU. *On the complexity of the parity argument and other inefficient proofs of existence*. J. Comput. Syst. Sci., 48(3):498–532, 1994. 3, 6
- [Sel06] JOACHIM SELKE. *Autoreducibility and Friends About Measuring Redundancy in Sets*. Master’s thesis, Gottfried-Wilhelm-Leibniz-Universität Hannover Fakultät für Elektrotechnik und Informatik Institut für Theoretische Informatik, 2006. 2
- [Tra70] BORIS AVRAAMOVICH TRAKHTENBROT. *Об автоматовности (Russian) [On Autoreducibility]*. Dokl. Akad. Nauk SSSR, 192(6):1224–1227, 1970. (English translation in *Soviet Math. Dokl.* 11:814–817, 1970). 1