Further Collapses in TFNP

Mika Göös
EPFL, Lausanne, Switzerland

Alexandros Hollender
University of Oxford, UK

Siddhartha Jain
EPFL, Lausanne, Switzerland

Gilbert Maystre
EPFL, Lausanne, Switzerland

William Pires
McGill University, Montreal, Canada

Robert Robere
McGill University, Montreal, Canada

Ran Tao
McGill University, Montreal, Canada

Abstract

We show EOPL = PLS ∩ PPAD. Here the class EOPL consists of all total search problems that reduce to the End-of-Potential-Line problem, which was introduced in the works by Hubáček and Yogev (SICOMP 2020) and Fearnley et al. (JCSS 2020). In particular, our result yields a new simpler proof of the breakthrough collapse CLS = PLS ∩ PPAD by Fearnley et al. (STOC 2021). We also prove a companion result SOPL = PLS ∩ PPADS, where SOPL is the class associated with the Sink-of-Potential-Line problem.

2012 ACM Subject Classification Theory of computation → Complexity classes; Theory of computation → Problems, reductions and completeness

Keywords and phrases TFNP, PPAD, PLS, EOPL

Digital Object Identifier 10.4230/LIPIcs.CCC.2022.33

Acknowledgements We thank Aviad Rubinstein for his many questions during e-mail correspondence, and the anonymous reviewers for their suggestions that helped improve the presentation of the paper.

1 Introduction

Our main results are two collapses of total NP search problem (TFNP) classes.

▶ **Theorem 1.** *EOPL = PLS ∩ PPAD.*

▶ **Theorem 2.** *SOPL = PLS ∩ PPADS.*

Let us explain what these collapses mean and how they fit into the diverse complexity zoo of search problem classes, as summarised in Figure 1. The classes PLS, PPAD, PPADS are classical. They were all introduced in the original pioneering works [17, 16, 20] that founded the theory of TFNP. To define these classes, it is most convenient to describe a canonical complete problem for each class. (See Section 2 for more formal definitions).

PLS: Sink-of-DAG (SoD). We are given *implicit access* to a directed graph $G = (V, E)$ that is acyclic, has out-degree at most 1, and has exponentially many nodes, $|V| = 2^n$. The graph is described by a poly(n)-sized circuit: for any node $v \in V$, we can compute its unique *successor* (out-neighbour) u, if any, and also an integer *potential*, which is guaranteed to increase along the direction of the edge (v, u). The goal is to find a *sink* node (in-degree ≥ 1, out-degree 0).

PPAD: End-of-Line (EoL). We are given access to a directed graph $G = (V, E)$ that has in/out-degree at most 1, and has $|V| = 2^n$ nodes. The graph is described by a poly(n)-sized circuit: for any $v \in V$, we can compute its *successor* u and *predecessor* u', if any. We are guaranteed that if v's successor is u, then u's predecessor is v, and vice versa. In addition, we are given the name of a *distinguished source* node v^* (in-degree 0, out-degree 1). The goal is to find any source or sink other than v^*.

PPADS: Sink-of-Line (SoL). Same as EoL except the goal is to find a sink.

$$\begin{tikzcd}
 & & & \mathsf{SOPL} \arrow[rr] & & \mathsf{PLS} \cap \mathsf{PPADS} \arrow[rr] \arrow[dr] \arrow[ll, bend left, "\text{Theorem 2}"] & & \mathsf{PLS} \arrow[dr] & \\
\mathsf{FP} \arrow[r] & \mathsf{UEOPL} \arrow[r] & \mathsf{EOPL} \arrow[rr] \arrow[ur] & & \mathsf{CLS} = \mathsf{PLS} \cap \mathsf{PPAD} \arrow[ur] \arrow[dr] \arrow[ll, bend left, "\text{Theorem 1}"] & & \mathsf{PPADS} \arrow[r] & \mathsf{PPP} \arrow[r] & \mathsf{TFNP} \\
 & & & & & \mathsf{PPAD} \arrow[rr] \arrow[ur] & & \mathsf{PPA} \arrow[ur] &
\end{tikzcd}$$

Figure 1 Class diagram for TFNP with new inclusions highlighted. An arrow A → B denotes A ⊆ B.

Modern classes

Research in the past decade has studied several relatively weak classes of search problems that lie below PLS and PPAD. The intersection class PLS ∩ PPAD is, of course, one immediate such example. This class, however, feels quite artificial at first glance. It does not seem to admit any "natural" complete problem. Motivated by this, Daskalakis and Papadimitriou [5] introduced the *continuous local search* class CLS ⊆ PLS ∩ PPAD, which, by its very definition, admits natural complete problems related to the local optimisation of continuous functions over the real numbers (computed by arithmetic circuits). The class CLS is exceptional in that it captures the complexity of *real continuous* optimisation problems, while most classical search problem classes are designed to capture *combinatorial principles*, often phrased in terms of directed graphs.

In order to understand CLS from a more combinatorial perspective, Hubáček and Yogev [14] and Fearnley, Gordon, Mehta, and Savani [9] introduced the class EOPL ⊆ CLS, whose complete problem is the namesake End-of-Potential-Line (EoPL) problem, defined below. (The paper [14] initially defined a more restricted "metered" version of this problem, but we use the formulation from [9], which they prove is equivalent to the one from [14].) It is also natural to define a *sink-only* version of EOPL as suggested by [12].

EOPL: End-of-Potential-Line (EoPL). We are given access to a directed graph $G = (V, E)$ that is acyclic, has in/out-degree at most 1, and has $|V| = 2^n$ nodes; that is, G is a disjoint union of directed paths. The graph is described by a poly(n)-sized circuit: for any node we can compute its successor and predecessor, if any, and also an integer potential, which is guaranteed to increase along the directed edges. In addition, we are given the name of a distinguished source v^*. The goal is to find any source or sink other than v^*.

SOPL: Sink-of-Potential-Line (SoPL). Same as EoPL except the goal is to find a sink.

It is comforting to know that the definition of EOPL is robust: Ishizuka [15] showed that a version of EoPL that guarantees poly(n) many distinguished sources is still equivalent (via polynomial-time reductions) to the above standard version with a single source.

Fearnley et al. [9] also defined a more restricted subclass $\mathsf{UEOPL} \subseteq \mathsf{EOPL}$ where the complete problem is Unique-EoPL, a version of EoPL with a *unique* directed path. They showed that this class contains many important search problems with unique witnesses, such as unique sink orientations, linear complementary problems, Arrival [6, 10]. Other problems known to lie in UEOPL are a restricted version of the Ham-Sandwich problem [3] and a pizza cutting problem [21]. Fearnley et al. [9] conjecture that $\mathsf{UEOPL} \neq \mathsf{EOPL}$.

A surprising collapse

In a breakthrough, Fearnley, Goldberg, Hollender, and Savani [8] showed that, despite appearances to the contrary, $\mathsf{CLS} = \mathsf{PLS} \cap \mathsf{PPAD}$. This goes against the conjecture of Daskalakis and Papadimitriou [5] that the classes are distinct, a belief which underlied much of their original motivation for introducing CLS. The nontrivial direction of the collapse is a reduction from a canonical complete problem $\text{SoD} \curlywedge \text{EoL} \in \mathsf{PLS} \cap \mathsf{PPAD}$ (defined below) to a problem $\text{KKT} \in \mathsf{CLS}$, which involves computing a Karush–Kuhn–Tucker point of a smooth function. We may summarise the main technical result of Fearnley et al. [8] as

$$\text{SoD} \curlywedge \text{EoL} \ \leq \ \text{KKT} \qquad \text{which implies } \mathsf{PLS} \cap \mathsf{PPAD} \subseteq \mathsf{CLS}. \tag{1}$$

Here we use $\leq$ to denote a polynomial-time reduction between search problems. The operator $\curlywedge$ produces the *meet* of two search problems: the input to problem $\text{A} \curlywedge \text{B}$ is a pair (x, y) where x is an instance of A and y is an instance of B and the goal is to output either a solution to x or to y. Then $\text{SoD} \curlywedge \text{EoL}$ is the canonical (albeit "unnatural") complete problem for $\mathsf{PLS} \cap \mathsf{PPAD}$ [5].

Our new collapses

Our main results, Theorems 1 and 2, follow from two new reductions, the first one of which strengthens the reduction (1) from [8]:

$$\text{SoD} \curlywedge \text{EoL} \ \leq \ \text{EoPL} \qquad \text{which implies } \mathsf{PLS} \cap \mathsf{PPAD} \subseteq \mathsf{EOPL}, \tag{2}$$

$$\text{SoD} \curlywedge \text{SoL} \ \leq \ \text{SoPL} \qquad \text{which implies } \mathsf{PLS} \cap \mathsf{PPADS} \subseteq \mathsf{SOPL}. \tag{3}$$

These reductions are between purely combinatorially defined search problems. In the case of (2), this bypasses the continuous middle-man of CLS and makes our reduction relatively simple to describe. In particular, we get a new simpler proof of the breakthrough collapse of [8] by combining (2) with the inclusion $\mathsf{EOPL} \subseteq \mathsf{CLS}$ proved by [14]. Furthermore, the new collapse implies that problems related to Tarski's fixpoint theorem [7] and to a colourful version of Carathéodory's theorem [18] lie in EOPL.

A further surprise?

Given that the collapse $\mathsf{CLS} = \mathsf{PLS} \cap \mathsf{PPAD}$ was considered extremely surprising by most people, how surprised should we be by the further collapse

$$\mathsf{EOPL} = \mathsf{CLS} = \mathsf{PLS} \cap \mathsf{PPAD} \ ?$$

Fearnley et al. [9] wrote regarding EOPL vs. CLS that "we actually think it could go either way." In the wake of their breakthrough, the paper [8] explicitly conjectured $\mathsf{EOPL} \neq \mathsf{CLS}$.

For the authors of the present paper, the new collapse did come as an utter shock. When we began work on this project, our intuitions convinced us that, again, $\mathsf{EOPL} \neq \mathsf{CLS}$, a conjecture which had just found its way to the second author's PhD thesis [13, Section 7.5].

In our convictions, we set out to prove this separation in the *black-box model* where, instead of circuits, the directed graphs are described by black-box oracles. We tried in vain for nine months. The upshot is that Theorems 1 and 2 now crush this possibility, as they hold even in the black-box model.

2 A Unified View: The Grid Problem

In this section we formally define all the problems of interest. We take the unusual approach of defining a single problem (which we call the GRID problem) with various parameters which can be tweaked to obtain all of the problems we study in this paper. This mainly serves two purposes. First of all, it is particularly convenient for presenting our reductions, since it allows us to combine instances from different problems more easily. The second reason is that we believe that this unified view of seemingly very different problems is of independent interest.

The Grid problem

For $n \in \mathbb{N}$, let $[n] := \{1, 2, \dots, n\}$. We define a general problem on a grid $[N] \times [M]$, where N and M should be thought of as being (potentially) exponentially large. The problem involves A paths starting from column 1 ($[N] \times \{1\}$) and moving from column i ($[N] \times \{i\}$) to column $i+1$ ($[N] \times \{i+1\}$). On the last column ($[N] \times \{M\}$) there are at most B valid ends of paths. If paths are not allowed to merge, then by the Pigeonhole Principle $A > B$ ensures the existence of a solution, i.e., a path that does not end at a valid position on the last column. If paths are allowed to merge, then a solution is guaranteed to exist as long as $B = 0$. To make things more precise, the paths start from nodes 1 to A in the first column (i.e., $[A] \times \{1\}$), and the valid termination points are nodes 1 to B in the last column (i.e., $[B] \times \{M\}$).

In more detail, we are given a boolean circuit $S\colon [N] \times [M] \to [N] \cup \{\mathsf{null}\}$, the *successor circuit*, which allows us to efficiently compute the outgoing edge at a node. If $S(x, y) = \mathsf{null}$, then (x, y) does not have an outgoing edge. Otherwise, there is an outgoing edge from (x, y) to $(S(x, y), y + 1)$. The problem also has two parameters which are used to tweak the definition: r (*reversible*) and b (*bijective*). Intuitively, when $r = 1$, we change the representation of paths to make them *reversible*. Namely, in addition to the successor circuit S, we are also given access to a *predecessor circuit* $P\colon [N] \times [M] \to [N] \cup \{\mathsf{null}\}$, which, analogously to S, allows us to efficiently compute the incoming edge at a node. In particular, when $r = 1$, every node can have at most one incoming edge, i.e., two paths cannot merge. When $r = 1$, the other parameter b is used to introduce additional solutions. Namely, when $b = 1$, then we do not allow any new paths apart from the original A paths, and we also require that all B valid ends of paths are actually reached by a path. The combination $r = 0, b = 1$ is not allowed.

We use the term *sink* to refer to a node with at least one incoming edge but no outgoing edge. Similarly, a *source* is a node with an outgoing edge but no incoming edge. The formal definition of the problem is as follows.

▶ **Definition 3.** *In the* GRID *problem, given N, M, A, B with $N \geq A > B \geq 0$ and $M \geq 2$, boolean circuits $S, P\colon [N] \times [M] \to [N] \cup \{\mathsf{null}\}$, and bits $r, b \in \{0, 1\}$, output any of the following:*

1. *$x \in [A]$ such that $S(x, 1) = \mathsf{null}$,* (missing pigeon/source)
2. *$x \in [N]$ such that $S(x, M - 1) > B$,* (invalid hole/sink)
3. *$x \in [N]$ and $y \in [M - 2]$ such that*
   $S(x, y) \neq \mathsf{null}$ and $S(S(x, y), y + 1) = \mathsf{null}$, (pigeon interception/sink)

4. *If $r = 1$ and $b = 1$:*

a. $(x, y) \in ([N] \times [M-1]) \setminus ([A] \times \{1\})$ *such that* $S(x, y) \neq \mathsf{null}$ *and* $P(x, y) = \mathsf{null}$, *or* (pigeon genesis/source)

b. $x \in [B]$ *such that* $P(x, M) = \mathsf{null}$. (empty hole/sink)

We also enforce the following two conditions syntactically:

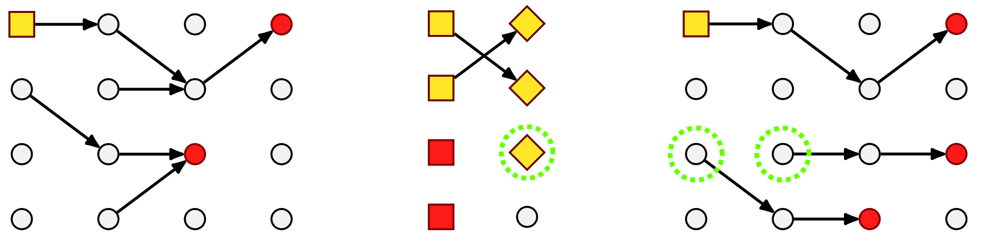
- *If $r = 0$, then $b = 0$ and $B = 0$.*
- *If $r = 1$, then the successor and predecessor circuits are consistent, which can be enforced as follows. The circuit S is replaced by the circuit $\overline{S}$, which on input (x, y) computes $x' := S(x, y)$ and outputs x', unless $(x', y+1) \notin [N] \times [M]$ or $P(x', y+1) \neq x$, in which case it outputs* null. *Similarly, the circuit P is replaced by the circuit $\overline{P}$, which on input (x, y) computes $x' := P(x, y)$ and outputs x', unless $(x', y-1) \notin [N] \times [M]$ or $S(x', y-1) \neq x$, in which case it outputs* null.

Canonical complete problems as special cases of Grid

As defined above, the inputs N, M, A, B of the Grid problem are completely unrestricted, apart from the natural restrictions $N \geq A > B \geq 0$ and $M \geq 2$. By imposing various additional restrictions on these inputs, we obtain the following canonical complete problems; see Figure 2. (Here inj-PHP/bij-PHP stand for Injective/Bijective Pigeonhole Principle.)

- SoD: $r = 0, b = 0, A = 1, B = 0$. (PLS-complete)
- SoPL: $r = 1, b = 0, A = 1, B = 0$. (SOPL-complete)
- EoPL: $r = 1, b = 1, A = 1, B = 0$. (EOPL-complete)
- inj-PHP: $r = 1, b = 0, M = 2, N = A = B + 1$. (PPADS-complete)
- bij-PHP: $r = 1, b = 1, M = 2, N = A = B + 1$. (PPAD-complete)

Note that beyond those restrictions, the inputs are left unrestricted. For example, in SoD, the input M can be very large, which is indeed needed for the problem to be PLS-complete.



(a) Sink-of-DAG (SoD). **(b)** inj-PHP. **(c)** Sink-of-Potential-Line (SoPL).

Figure 2 Examples of Grid problems. Square nodes are valid starts of paths (top-most A nodes in the first column) and diamonds are valid ends of paths (top-most B nodes in the last column). Solutions are drawn in red. However, for visual clarity we highlight the actual sinks rather than the *sink predecessors* as in Definition 3. Nodes with a null successor are drawn without an outgoing pointer. (2a) has parameters $(r = 0, b = 0, A = 1, B = 0)$ and defines an SoD instance. Only the successor circuit is drawn, as the predecessor circuit is not used by SoD. In particular, directed paths can *merge*, such as for node $(2, 3)$. (2b) has parameters $(r = 1, b = 0, A = N, B = N - 1)$ and defines an inj-PHP instance. The diamond with a green circle would be a solution of bij-PHP (with $b = 1$) but is not a solution of inj-PHP. (2c) has parameters $(r = 1, b = 0, A = 1, B = 0)$ and defines an SoPL instance. Sources with green circles would be solutions of an EoPL instance (with $b = 1$).

▶ Remark 4. Here we have slightly abused notation by calling these problems SoD, SoPL and EoPL even though their original definitions (in [16, 12, 9], respectively) do not use a grid structure, and instead come with an additional circuit computing the potential of any node. It is not too hard to see that these grid-versions of the problems are indeed polynomial-time equivalent to the original versions. The main idea is that the grid implicitly provides a potential value for every node (x, y), namely its column number y. Thus, given such a problem on a grid, it is easy to define a potential circuit by simply assigning the potential value y to any node (x, y) of the grid.

The other direction is slightly more involved. Consider an instance of one of the original problems with vertex set $V = [N]$ and potential values lying in $P = [M]$. Without loss of generality, we can assume that along any edge the potential increases by exactly one. Indeed, this was proved explicitly by [9] when they reduced EoPL to EoML, and the same idea applies to SoD and SoPL as well. The reduction to the grid-version of the problem is then obtained by identifying a vertex $x \in V$ that has potential $p \in P$ with the node (x, p) on the $[N] \times [M]$ grid.

The following is essentially folklore (see, e.g., [2]), so we only provide a brief proof sketch.

▶ **Lemma 5.** INJ-PHP *and* BIJ-PHP *are respectively* PPADS- *and* PPAD-*complete.*

Proof Sketch. To see that BIJ-PHP lies in PPAD, we can reduce to EoL (see, e.g., [4] for a formal definition) with vertex set $V = [N] \times [2]$ as follows: add a directed edge from node $(x, 2)$ to node $(x, 1)$ for all $x \leq A - 1$. By taking (x, A) as the distinguished source node, this yields an EoL instance with the same solutions as the original BIJ-PHP instance. On the other hand, given an instance of EoL with vertex set $V = [N]$ and distinguished source node N (without loss of generality), we construct an instance of BIJ-PHP on $[N] \times [2]$ as follows: for any isolated vertex $x \in [N]$, create an edge from $(x, 1)$ to $(x, 2)$; for any edge from x to y, create an edge from $(x, 1)$ to $(y, 2)$. This simple reduction proves the PPAD-hardness of BIJ-PHP. The exact same constructions can be used to prove that INJ-PHP is PPADS-complete, by reducing to and from the SoL problem (formally defined by Beame et al. [1], who call it SINK). ◀

In Section 6, we briefly explain how an extended version of the GRID problem can be used to also capture PPP, the class defined by Papadimitriou [20] to capture a version of the Pigeonhole Principle where edges can only be computed efficiently in the forward direction. We do not currently see any natural way of extending the definition of the GRID problem so that it also captures the class PPA.

3 Path-Pigeonhole Problems

In this section we use the GRID problem to define some interesting extensions of the two pigeonhole problems. Namely, we consider the case where, instead of just two columns, there are many columns. In a certain sense, this corresponds to allowing the pigeons to travel for a long time before reaching a hole. In particular, we can no longer efficiently tell in which hole a given pigeon will land. This allows us to show that the problems remain hard even when there are significantly more pigeons than holes. This fact, stated in Lemma 6 below, will be crucial to obtain our main result later.

Let $f \colon \mathbb{N} \to \mathbb{N}$ be a polynomial-time computable function with $f(t) > t$. In this section, we consider the following restrictions of GRID:

- PATH-INJ-PHP$_f$: $r = 1, b = 0, A = f(B)$.
- PATH-BIJ-PHP$_f$: $r = 1, b = 1, A = f(B)$.

The following lemma is an important ingredient for the proof of our main result.

▶ **Lemma 6.** *Let $f(t) > t$ be polynomial-time computable. There exists a reduction* INJ-PHP $\leq$ PATH-INJ-PHP$_f$ *that maps an instance with parameters $(A, B) = (T + 1, T)$ to an instance with parameters $(A, B) = (f(T), T)$ and $(N, M) = (f(T), f(T) - T + 1)$.*

Proof. The idea behind this reduction is very simple. Intuitively, we have the ability to "merge" $T + 1$ paths into T paths by using the INJ-PHP instance. Namely, we can go from having $T + 1$ paths on some column i to having only T paths on the next column $i + 1$, and such that finding a "mistake", i.e., a path that stops between the two columns, requires solving the INJ-PHP instance. In particular, if we start with some N paths, where $N \geq T+1$, then we can "merge" those paths into $N - 1$ paths by "merging" the first $T + 1$ paths into T paths, and leaving the remaining $N - (T+1)$ paths unchanged. Applying this idea repeatedly, we can "merge" $f(T)$ paths into just T paths in $f(T) - T$ steps. This results in an instance of PATH-INJ-PHP$_f$ with $f(T) - T + 1$ columns, where every solution yields a solution to the INJ-PHP instance; see Figure 3. More formally, let (S, P) denote an instance of INJ-PHP with parameters $A = T + 1$ and $B = T$. Without loss of generality, we can assume that no pigeon goes to the invalid hole, i.e., $S(x, 1) \neq T + 1$ for all $x \in [T + 1]$. Indeed, if there is such an edge, we can just remove it and this does not change the set of solutions; the node pointing to the invalid hole was a solution before, and now it is still a solution, because it has no successor. We construct an instance $(\widehat{S}, \widehat{P})$ of PATH-INJ-PHP$_f$ on the $[N] \times [M]$ grid, where $N = f(T)$, $M = f(T) - T + 1$, $A = f(T)$ and $B = T$. The successor circuit $\widehat{S}$ is defined as follows:

$$\widehat{S}(x, y) \;:=\; \begin{cases} S(x, 1) & \text{if } x \in [T + 1] \text{ and } y \in [M - 1], \\ x - 1 & \text{if } T + 2 \leq x \leq f(T) - y + 1 \text{ and } y \in [M - 1], \\ \mathsf{null} & \text{otherwise,} \end{cases}$$

and the predecessor circuit $\widehat{P}$ is then defined accordingly to be consistent with $\widehat{S}$. Both circuits can be constructed in polynomial time, given S and P, and given that f can be computed in polynomial time. It is straightforward to check that any solution of the constructed instance yields a solution to the original INJ-PHP instance. ◀

The same proof idea also yields that BIJ-PHP $\leq$ PATH-BIJ-PHP$_f$.

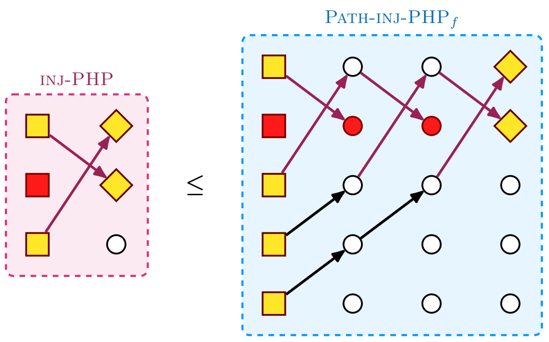


Figure 3 The reduction INJ-PHP $\leq$ PATH-INJ-PHP$_f$ in Lemma 6. We build a PATH-INJ-PHP$_f$ instance by chaining several identical INJ-PHP instances side-by-side. Note that a solution to the PATH-INJ-PHP$_f$ instance can be directly mapped to one for the original INJ-PHP instance.

4 SOPL = PLS ∩ PPADS

In this section, we prove Theorem 2, namely SOPL = PLS ∩ PPADS. To prove this we provide a reduction from a PLS ∩ PPADS-complete problem to the canonical SOPL-complete problem.

▶ **Lemma 7.** SoD $\curlywedge$ INJ-PHP $\leq$ SoPL.

Proof Sketch. There are two obstacles to a direct reduction from SoD to SoPL: (i) we can only compute edges in the forward direction (i.e., we only have access to a successor circuit), and (ii) multiple edges can point to the same node.

To resolve the first issue, we modify the original $[N] \times [M]$ grid of the SoD instance by taking N copies of each node. This ensures that there is a separate copy of each node v for each potential predecessor on the previous column. As a result, edges from different predecessor nodes will point to different copies of v. This means that predecessor nodes can now also be computed efficiently. Namely, in order to compute the predecessors of the ith copy of node v, it suffices to check whether in the original SoD instance the ith node on the previous column points to v. If that is the case, then all copies of this ith node are predecessors in the modified instance. Otherwise, there are no predecessors. However, the second issue remains: since we have made N copies of each node, there are also N copies of each predecessor node, and thus N edges pointing to the corresponding copy of v. This is not acceptable, since SoPL allows at most one incoming edge.

To overcome the second obstacle, we make use of the following high-level idea: use the INJ-PHP instance (which maps $K+1$ pigeons to K holes) to "hide" the fact that multiple edges can point to a single node. Unfortunately, we cannot use the INJ-PHP instance to hide the fact that N paths merge into a single node. But, if we take KN copies of each original node, instead of just N, then we have KN paths and K target nodes. By Lemma 6, the INJ-PHP instance can be turned into a PATH-INJ-PHP$_f$ instance that hides the fact that KN paths merge into K paths. Thus, we replace each original node v of the SoD instance by a gadget that has KN nodes in the left-most column and K nodes in the right-most column, and such that finding the sink of a path inside the gadget requires solving the INJ-PHP instance. Importantly, we only construct the paths inside this gadget when the original node v has a successor in the SoD instance. This ensures that when v is an isolated node, the corresponding gadget does not contain any edges. Figure 4 illustrates the construction for $N = 3$, $M = 4$ and $K = 2$.

Note that although we might have added many new sources to the graph (which are irrelevant for SoPL), it remains the case that from any sink of the new graph, we can extract either a solution to SoD or to INJ-PHP.

In the final construction, edges can indeed be computed in both directions efficiently. Namely, given any node, we can determine in polynomial time if it has an incoming and/or outgoing edge, as well as the identity of the potential predecessor and successor nodes. Here, we crucially use the fact that edges can be computed efficiently in both directions in the INJ-PHP instance. ◀

Proof. Let S be an instance of SoD on the grid $[N] \times [M]$. We are also given an instance of INJ-PHP with parameters $(A, B) = (K+1, K)$. Without loss of generality we can assume that $K = N$, because we can easily pad the SoD or INJ-PHP instance with additional rows without changing the set of solutions. By Lemma 6, we can reduce this INJ-PHP instance to a PATH-INJ-PHP$_{t^2}$ instance on the grid $[N^2] \times [M']$ with parameters $(A, B) = (N^2, N)$. Without loss of generality, we can assume that $M' = M$, because we can pad the SoD or PATH-INJ-PHP$_{t^2}$ instance with additional columns, if needed. This is not important for the reduction, but will be convenient.

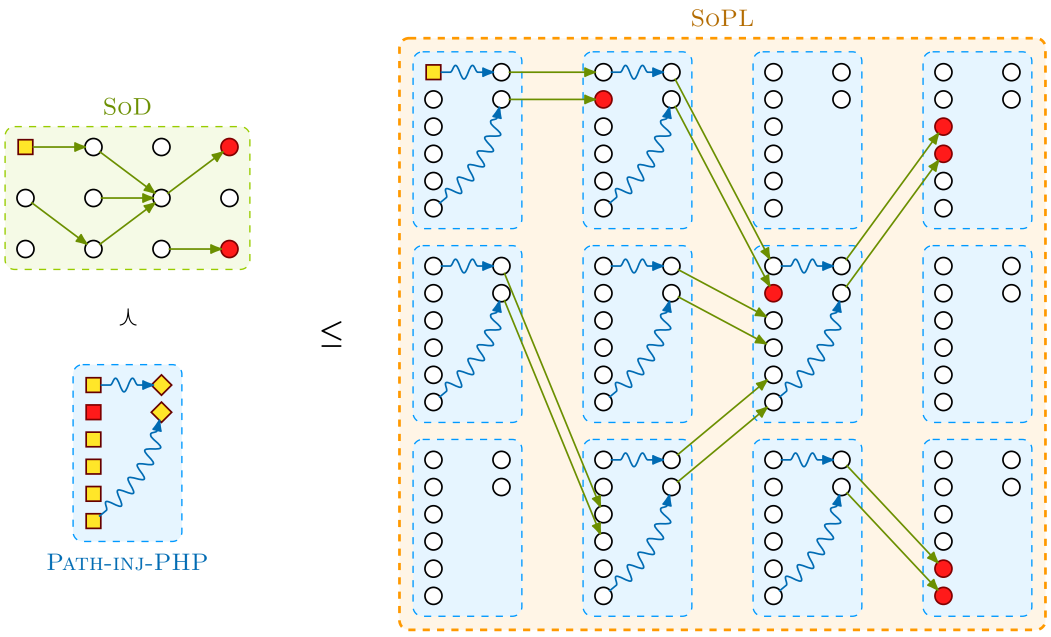


Figure 4 The reduction SoD ⅄ Path-inj-PHP ≤ SoPL in the proof of Lemma 7. Given instances of SoD and Path-inj-PHP, we build an SoPL instance whose solutions can be traced back to solutions of SoD ⅄ Path-inj-PHP. To overcome the issue of merging paths in SoD, the nodes of the SoD instance are replaced with a copy of a Path-inj-PHP gadget (in blue). Those gadgets are ultimately built out of the initial inj-PHP instance (not shown) using Lemma 6.

We will take N^2 copies of each node in the original SoD instance, and make M copies of each column. As a result, our SoPL instance will be defined on the $[N^3] \times [M^2]$ grid. It will be convenient to use some special notation to refer to points in this grid. For $\alpha \in [N^2]$ and $x \in [N]$, we use the notation (α, x) to denote the row $\alpha + (x-1) \cdot N^2 \in [N^3]$. This corresponds to indexing the αth copy of row x of the original instance. We also introduce some additional notation to index these $[N^2]$ copies: for $i, j \in [N]$, we let $[i,j] := i + (j-1) \cdot N \in [N^2]$. Thus, $([i,j], x)$ denotes the $[i,j]$th copy of row x. The "$[i,j]$" notation essentially subdivides $[N^2]$ into N blocks containing N values each, which will be useful for routing incoming edges to the correct copy of a node. Using the analogous subdivision also on the columns, the notation $(\alpha, x; k, y) \in [N^2] \times [N] \times [M] \times [M]$ denotes the node $(\alpha + (x-1) \cdot N^2, k + (y-1) \cdot M) \in [N^3] \times [M^2]$. In particular, the notation $([i,j], x; k, y)$ is well-defined.

The circuits $\widehat{S}, \widehat{P}$ of the SoPL instance on $[N^3] \times [M^2]$ are defined as follows:

$$\widehat{S}([i,j], x; k, y) := \begin{cases} ([i,x], S(x,y)) & \text{if } k = M \text{ and } j = 1, \\ (S'([i,j],k), x) & \text{if } k < M \text{ and } S(x,y) \neq \mathsf{null}, \\ \mathsf{null} & \text{otherwise} \end{cases}$$

$$\widehat{P}([i,j], x; k, y) := \begin{cases} ([i,1], j) & \text{if } k = 1 \text{ and } y > 1 \text{ and } S(j, y-1) = x, \\ (P'([i,j],k), x) & \text{if } k > 1 \text{ and } S(x,y) \neq \mathsf{null}, \\ \mathsf{null} & \text{otherwise} \end{cases}$$

where $(\alpha, z) \in [N^2] \times [N]$ is interpreted as an element in $[N^3]$ as above, and where we use the convention $(*, \mathsf{null}) = (\mathsf{null}, *) = \mathsf{null}$. Using the fact that S' and P' are consistent, it can be checked that $\widehat{S}$ and $\widehat{P}$ are also consistent.

In order to argue about the correctness of the reduction, consider any sink $([i,j],x;k,y)$ of the SoPL instance. If $2 \leq k \leq M-1$, then it must be that $([i,j],k)$ is a sink of the Path-inj-PHP$_{t^2}$ instance (S',P'). If $k=M$ and $j=1$, then $([i,j],x;k,y)$ cannot be a sink, since $\widehat{P}([i,j],x;k,y) \neq \mathsf{null}$ implies that $S(x,y) \neq \mathsf{null}$, and thus $\widehat{S}([i,j],x;k,y) \neq \mathsf{null}$. If $k=M$ and $j>1$, then $([i,j],k)$ is an invalid sink on the last column of the Path-inj-PHP$_{t^2}$ instance, and so in particular a solution. If $k=1$ and $S(x,y) \neq \mathsf{null}$, then $([i,j],k)$ is a missing source on the first column of the Path-inj-PHP$_{t^2}$ instance, and so again a solution. Finally, if $k=1$ and $S(x,y)=\mathsf{null}$, then it must be that $S(j,y-1)=x$ and thus (x,y) is a sink of the original SoD instance, and this is witnessed by the node $(j,y-1)$. ◀

5 EOPL = PLS ∩ PPAD

In this section, we prove Theorem 1, namely $\mathsf{EOPL} = \mathsf{PLS} \cap \mathsf{PPAD}$. The equality $\mathsf{SOPL} = \mathsf{PLS} \cap \mathsf{PPADS}$ (Theorem 2) proved in the previous section, together with the fact that $\mathsf{PPAD} \subseteq \mathsf{PPADS}$, immediately imply that

$$\mathsf{SOPL} \cap \mathsf{PPAD} \;=\; \mathsf{PLS} \cap \mathsf{PPAD}.$$

As a result, in order to prove Theorem 1, it suffices to give a reduction from an $\mathsf{SOPL} \cap \mathsf{PPAD}$-complete problem to an EOPL-complete problem:

▶ **Lemma 8.** SoPL $\curlywedge$ bij-PHP $\leq$ EoPL.

Proof Sketch. A very natural attempt at a reduction from SoPL to EoPL is to try to remove all undistinguished sources, i.e., all sources except the trivial one. Then, clearly, any EoPL-solution would have to be a sink, and thus also a solution to SoPL.

There is a simple trick that *almost* achieves this. First, make a *reversed* copy of the SoPL instance, i.e., reverse the direction of all edges, and the ordering of the potential. Note that sources of the original instance have now become sinks in the reversed copy, and vice versa. Then, for each source node v of the original graph, add an edge pointing from its copy $\overline{v}$ (which is a sink) to v.

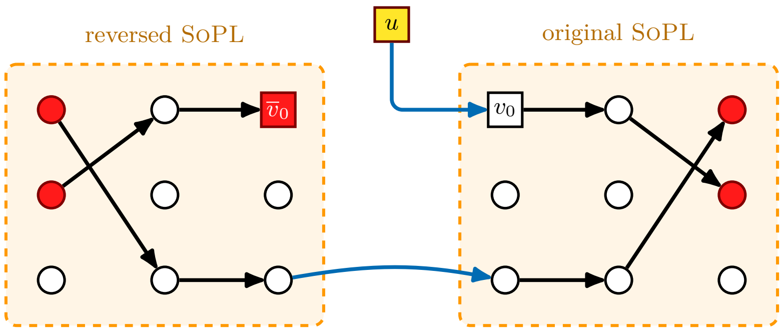


Figure 5 A naive attempt at a reduction SoPL $\leq$ EoPL. Although most solutions arise from the sinks of the original SoPL instance, a spurious solution is introduced at $\bar{v}_0$, which does not correspond to any sink of the original instance.

The only problem with this reduction is that we have eliminated *all* sources of the original graph, including the distinguished one. In particular, the distinguished source v_0 of the original instance is no longer a source, since there is an edge from its copy $\overline{v}_0$ to v_0. As a result, the reduction fails, because the instance of EoPL we have constructed does not have a distinguished source. Furthermore, we cannot hope to turn one of the new sources into a distinguished source, since any such source yields a solution to the original instance (where it is a sink).

In order to address this issue, we add a new node u and select it as our new distinguished source. Clearly, u is a solution of the instance, since it is a distinguished source that is not actually a source, but just an isolated node. Now, imagine that we remove the edge $(\overline{v}_0, v_0)$ and instead introduce an edge (u, v_0); see Figure 5. Then, u is no longer a solution, but $\overline{v}_0$ becomes a sink, and thus a solution, instead. In other words, the reduction can pick whether it wants u or $\overline{v}_0$ to be a solution by changing this edge. Of course, in both cases, the resulting instance is very easy to solve, but this minor observation already provides the idea for the next step.

Take k copies of the instance we have constructed (before adding u). There are now k copies $v_0^{(1)}, \dots, v_0^{(k)}$ of the original distinguished source, and k copies of the reverse copy $\overline{v}_0^{(1)}, \dots, \overline{v}_0^{(k)}$. Remove the edges $(\overline{v}_0^{(i)}, v_0^{(i)})$ for $i = 1, \dots, k$. If we now introduce the new distinguished source u, we have $k+1$ nodes that "need" an outgoing edge in order to not be solutions (namely, $u, \overline{v}_0^{(1)}, \dots, \overline{v}_0^{(k)}$) and k nodes that "need" an incoming edge (namely, $v_0^{(1)}, \dots, v_0^{(k)}$). Clearly, no matter how we introduce edges here, one of $u, \overline{v}_0^{(1)}, \dots, \overline{v}_0^{(k)}$ will not have an outgoing edge and will be a solution. However, we can use a BIJ-PHP instance to make it hard to find such a solution. Let K denote the parameter of the BIJ-PHP instance, i.e., $K+1$ points are mapped to K points. Then, we let $k := K$ and add edges between $u, \overline{v}_0^{(1)}, \dots, \overline{v}_0^{(k)}$ and $v_0^{(1)}, \dots, v_0^{(k)}$ according to the BIJ-PHP instance. An example of the construction is depicted in Figure 6.

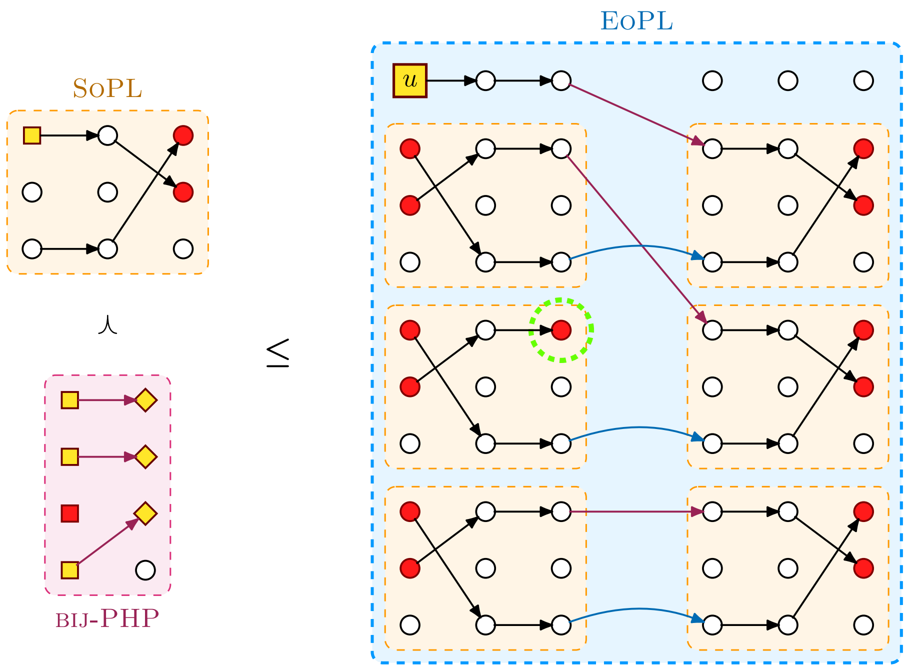


Figure 6 The reduction SoPL $\curlywedge$ BIJ-PHP $\leq$ EoPL in Lemma 8. The BIJ-PHP instance (in pink) connects the newly introduced source u together with the distinguished sources and sinks of the copied SoPL instances. Non-distinguished sources and sinks are connected with blue edges. The node circled in green corresponds to a solution of the BIJ-PHP instance.

Now, it is easy to check that any undistinguished source or any sink of the resulting graph must yield a solution to the BIJ-PHP instance or a solution of the SoPL instance. In particular, if u is not a source, then this yields a solution to BIJ-PHP. ◀

Proof. Let (S, P) be an instance of SoPL on the grid $[N] \times [M]$. Without loss of generality, we can assume that all sources occur on the first column, i.e., for any source $(x, y) \in [N] \times [M]$ it holds that $y = 1$. Indeed, by appropriately increasing N, for each source (x, y) we can add a path that starts on the first column and ends at (x, y), thus effectively "transferring" the source to the first column. Let (S', P') be an instance of BIJ-PHP on the grid $[K + 1] \times [2]$ that maps $K + 1$ pigeons to K holes.

We take K copies of the SoPL instance and K copies of the reversed SoPL instance, all together in a single grid. This grid will be of the form $[KN] \times [2M]$. For clarity, we will use the notation $(i, x; y) \in [K] \times [N] \times [2M]$ to denote the element $(x + (i - 1) \cdot N, y) \in [KN] \times [2M]$. The ith copy of the instance will be embedded in $\{i\} \times [N] \times ([2M] \setminus [M])$, while the ith reversed copy will be in $\{i\} \times [N] \times [M]$. Formally, we define new successor and predecessor circuits $\widehat{S}, \widehat{P}$ on $[KN] \times [2M]$ as follows:

$$\widehat{S}(i, x; y) := \begin{cases} (i, P(x, M - y + 1)) & \text{if } y \leq M, \\ (i, S(x, y - M)) & \text{if } y \geq M + 1 \end{cases}$$

$$\widehat{P}(i, x; y) := \begin{cases} (i, S(x, M - y + 1)) & \text{if } y \leq M, \\ (i, P(x, y - M)) & \text{if } y \geq M + 1 \end{cases}$$

where $(i, z) \in [K] \times [N]$ represents the element $z + (i - 1) \cdot N \in [KN]$, and where we use the convention $(i, \mathsf{null}) = \mathsf{null}$.

Since S and P are consistent, $\widehat{S}$ and $\widehat{P}$ are also consistent. Note that there are currently no edges between column M and column $M + 1$. In the second step of the reduction we add edges between these two columns as follows. For every $i \in [K]$ and $x \in [N] \setminus \{1\}$, if $(i, x; M+1)$ is a source, then we add an edge from $(i, x; M)$ to $(i, x; M+1)$. Note that in that case $(i, x; M)$ was a sink. The case where $x = 1$ is handled separately, because it corresponds to nodes that are copies of the distinguished source of the original SoPL instance. For any $i \in [K]$ and for $x = 1$, if $S'(i, 1) = j \neq \mathsf{null}$, we add an edge from $(i, 1; M)$ to $(j, 1; M + 1)$. Note that here we also use P' (which is assumed to be consistent with S') to implement this edge in $(\widehat{S}, \widehat{P})$.

Finally, we introduce a new special node u on column M which will act as our new distinguished source. If $S'(K + 1, 1) = j \neq \mathsf{null}$, then we add an edge from u to $(j, 1; M + 1)$. By extending the grid to be $[KN + 1] \times [2M]$, by renaming nodes and by "transferring" the source u to the first column as before, we can ensure that the distinguished source is $(1, 1)$.

It is easy to check that the new circuits $\widehat{S}, \widehat{P}$ can be constructed in polynomial time. For the correctness of the reduction, note that any source or sink that occurs on columns $[2M] \setminus \{M, M + 1\}$ must correspond to a sink of the original SoPL instance. On the other hand, any source or sink that occurs on column M or $M + 1$ must correspond to a solution of the BIJ-PHP instance (namely, a pigeon without a hole, or a hole without a pigeon). This completes the reduction. ◀

6 Discussion

As mentioned in the introduction, it remains open whether UEOPL $\overset{?}{=}$ EOPL. Separating the two classes in the black-box model would be an important first step towards pinning down the complexity of the various natural problems contained in UEOPL, since it would provide strong evidence that these problems are unlikely to be complete for PLS $\cap$ PPAD.

The techniques developed in this paper do not seem to yield any other major class collapse. Indeed, our reductions are all black-box, and the main classes are known to be distinct in that model [1, 19, 2]. A notable exception is the question of whether PLS is a subset of PPP, or even of PPADS. This remains open even in the black-box model.

In the remainder of this section we briefly present some observations about the path pigeonhole problems, as well as a further consequence of our reduction techniques: a version of SoD where paths are not allowed to merge turns out to be PLS $\cap$ PPP-complete.

Path-Pigeonhole problems

Lemma 6 in particular establishes that Path-inj-PHP$_f$ is PPADS-hard. Membership in PPADS can be shown by reducing to inj-PHP using a construction similar to the reduction from EoL to bij-PHP in the proof of Lemma 5.

The statement of Lemma 6 also holds for bij-PHP $\leq$ Path-bij-PHP$_f$, and the proof is essentially the same. This shows that Path-bij-PHP$_f$ is PPAD-hard. However, it is unclear whether Path-bij-PHP$_f$ lies in PPAD. Indeed, using the same idea as for Path-inj-PHP$_f$ $\leq$ inj-PHP yields an instance with $A \gg B$, and we cannot increase B artificially here (whereas this is possible in inj-PHP). Another way to state this is to say that we can reduce Path-bij-PHP$_f$ to an instance of EoL that has many distinguished source nodes, instead of just one. It is known that EoL with a polynomial number of distinguished sources remains PPAD-complete [11], but in general we will obtain an exponential number of such sources here.

Extending the Grid problem to capture PPP

The canonical PPP-complete problem is Pigeon-Circuit [20]: given a circuit mapping N pigeons to $N-1$ holes, find a *collision*, i.e., two pigeons that are mapped to the same hole. Importantly, unlike in inj-PHP or bij-PHP, we are not given a circuit to compute the mapping in the other direction, i.e., from holes to pigeons. In order to capture this problem, we extend the definition of Grid by introducing an additional parameter bit $c \in \{0,1\}$, which stands for *collision*. We also introduce a new solution type:

5. If $r = 0$ and $c = 1$: $x_1, x_2 \in [N]$ and $y \in [M-1]$ such that
$x_1 \neq x_2$ and $S(x_1, y) = S(x_2, y) \neq \mathsf{null}$, *(pigeon collision/merging)*

Furthermore, the syntactic condition "If $r = 0$, then $b = 0$ and $B = 0$" is replaced by the condition:

- If $r = 0$, then $b = 0$. If $r = 0$ and $c = 0$, then $B = 0$.

The PPP-complete problem Pigeon-Circuit is then obtained by setting $r = 0, c = 1, M = 2, N = A = B + 1$. In fact, Grid remains in PPP even if we just set $r = 0, c = 1$ and leave the other parameters unfixed. This can be shown by using a construction similar to the reduction from EoL to bij-PHP in the proof of Lemma 5.

SoD without merging

What is the complexity of SoD if paths are not allowed to merge? In other words, what is the complexity of the Grid problem with parameters $r = 0, c = 1, A = 1, B = 0$? Clearly, this restricted version still lies in PLS, and by the previous paragraph it also lies in PPP. Using the ideas developed in this paper, it can be shown that the problem is in fact PLS $\cap$ PPP-complete. To see this, note that using the simple construction in the proof of Lemma 6 we can reduce Pigeon-Circuit to a path-version of the problem where $f(T)$ pigeons are mapped to T holes. Then, the construction in the proof of Lemma 7 can be used to reduce SoD $\curlywedge$ Pigeon-Circuit to SoD without merging.

References

1 Paul Beame, Stephen Cook, Jeff Edmonds, Russell Impagliazzo, and Toniann Pitassi. The relative complexity of NP search problems. *Journal of Computer and System Sciences*, 57(1):3–19, 1998. doi:10.1006/jcss.1998.1575.

2 Joshua Buresh-Oppenheim and Tsuyoshi Morioka. Relativized NP search problems and propositional proof systems. In *Proceedings of the 19th IEEE Conference on Computational Complexity (CCC)*, pages 54–67, 2004. doi:10.1109/CCC.2004.1313795.

3 Man-Kwun Chiu, Aruni Choudhary, and Wolfgang Mulzer. Computational Complexity of the α-Ham-Sandwich Problem. In *47th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 31:1–31:18, 2020. doi:10.4230/LIPIcs.ICALP.2020.31.

4 Constantinos Daskalakis, Paul Goldberg, and Christos Papadimitriou. The complexity of computing a Nash equilibrium. *SIAM Journal on Computing*, 39(1):195–259, 2009. doi:10.1137/070699652.

5 Constantinos Daskalakis and Christos Papadimitriou. Continuous local search. In *Proceedings of the 22nd Symposium on Discrete Algorithms (SODA)*, pages 790–804, January 2011. doi:10.1137/1.9781611973082.62.

6 Jérôme Dohrau, Bernd Gärtner, Manuel Kohler, Jiří Matoušek, and Emo Welzl. ARRIVAL: A zero-player graph game in NP ∩ coNP. In *A Journey Through Discrete Mathematics*, pages 367–374. Springer, 2017. doi:10.1007/978-3-319-44479-6_14.

7 Kousha Etessami, Christos Papadimitriou, Aviad Rubinstein, and Mihalis Yannakakis. Tarski's theorem, supermodular games, and the complexity of equilibria. In *Proceedings of the 11th Innovations in Theoretical Computer Science Conference (ITCS)*, volume 151, pages 18:1–18:19, 2020. doi:10.4230/LIPIcs.ITCS.2020.18.

8 John Fearnley, Paul Goldberg, Alexandros Hollender, and Rahul Savani. The complexity of gradient descent: CLS = PPAD ∩ PLS. In *Proceedings of the 53rd Symposium on Theory of Computing (STOC)*, pages 46–59, 2021. doi:10.1145/3406325.3451052.

9 John Fearnley, Spencer Gordon, Ruta Mehta, and Rahul Savani. Unique end of potential line. *Journal of Computer and System Sciences*, 114:1–35, 2020. doi:10.1016/j.jcss.2020.05.007.

10 Bernd Gärtner, Thomas Dueholm Hansen, Pavel Hubácek, Karel Král, Hagar Mosaad, and Veronika Slívová. ARRIVAL: Next stop in CLS. In *45th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 107, pages 60:1–60:13. Schloss Dagstuhl, 2018. doi:10.4230/LIPICS.ICALP.2018.60.

11 Paul Goldberg and Alexandros Hollender. The Hairy Ball problem is PPAD-complete. *Journal of Computer and System Sciences*, 122:34–62, 2021. doi:10.1016/j.jcss.2021.05.004.

12 Mika Göös, Pritish Kamath, Robert Robere, and Dmitry Sokolov. Adventures in monotone complexity and TFNP. In *Proceedings of the 10th Innovations in Theoretical Computer Science Conference (ITCS)*, volume 124, pages 38:1–38:19, 2018. doi:10.4230/LIPIcs.ITCS.2019.38.

13 Alexandros Hollender. *Structural results for total search complexity classes with applications to game theory and optimisation.* PhD thesis, University of Oxford, 2021. URL: https://ora.ox.ac.uk/objects/uuid:67e2d80b-76bf-4b49-9b7d-8bbd91633dd7.

14 Pavel Hubáček and Eylon Yogev. Hardness of continuous local search: Query complexity and cryptographic lower bounds. *SIAM Journal on Computing*, 49(6):1128–1172, 2020. doi:10.1137/17m1118014.

15 Takashi Ishizuka. The complexity of the parity argument with potential. *Journal of Computer and System Sciences*, 120:14–41, September 2021. doi:10.1016/j.jcss.2021.03.004.

16 David Johnson, Christos Papadimitriou, and Mihalis Yannakakis. How easy is local search? *Journal of Computer and System Sciences*, 37(1):79–100, August 1988. doi:10.1016/0022-0000(88)90046-3.

17 Nimrod Megiddo and Christos Papadimitriou. On total functions, existence theorems and computational complexity. *Theoretical Computer Science*, 81(2):317–324, April 1991. doi:10.1016/0304-3975(91)90200-L.

18 Frédéric Meunier, Wolfgang Mulzer, Pauline Sarrabezolles, and Yannik Stein. The rainbow at the end of the line—a PPAD formulation of the colorful Carathéodory theorem with applications. In *Proceedings of the 28th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1342–1351, 2017. doi:10.1137/1.9781611974782.87.

19 Tsuyoshi Morioka. Classification of search problems and their definability in bounded arithmetic. Master's thesis, University of Toronto, 2001. URL: https://www.collectionscanada.ca/obj/s4/f2/dsk3/ftp04/MQ58775.pdf.

20 Christos Papadimitriou. On the complexity of the parity argument and other inefficient proofs of existence. *Journal of Computer and System Sciences*, 48(3):498–532, June 1994. doi:10.1016/s0022-0000(05)80063-7.

21 Patrick Schnider. The Complexity of Sharing a Pizza. In *32nd International Symposium on Algorithms and Computation (ISAAC)*, pages 13:1–13:15, 2021. doi:10.4230/LIPIcs.ISAAC.2021.13.