Unique End of Potential Line

John Fearnley[1], Spencer Gordon[2], Ruta Mehta[3], and Rahul Savani[1]

[1]University of Liverpool, {john.fearnley, rahul.savani}@liverpool.ac.uk
[2]California Institute of Technology, slgordon@caltech.edu
[3]University of Illinois at Urbana-Champaign, rutamehta@cs.illinois.edu

November 12, 2018

Abstract

This paper studies the complexity of problems in PPAD $\cap$ PLS that have *unique* solutions. Three well-known examples of such problems are the problem of finding a fixpoint of a contraction map, finding the unique sink of a Unique Sink Orientation (USO), and solving the P-matrix Linear Complementarity Problem (P-LCP). Each of these are promise-problems, and when the promise holds, they always possess unique solutions.

We define the complexity class UniqueEOPL to capture problems of this type. We first define a class that we call EOPL, which consists of all problems that can be reduced to EndOfPotentialLine. This problem merges the canonical PPAD-complete problem EndOfLine, with the canonical PLS-complete problem SinkOfDag, and so EndOfPotentialLine captures problems that can be solved by a line-following algorithm that also simultaneously decreases a potential function.

PromiseUEOPL is a promise-subclass of EOPL in which the line in the EndOfPotentialLine instance is guaranteed to be unique via a promise. We turn this into a non-promise class UniqueEOPL, by adding an extra solution type to EndOfPotentialLine that captures any pair of points that are provably on two different lines.

We show that UniqueEOPL $\subseteq$ EOPL $\subseteq$ CLS, and that all of our motivating problems are contained in UniqueEOPL: specifically USO, P-LCP, and finding a fixpoint of a Piecewise-Linear Contraction under an ℓ_p-norm all lie in UniqueEOPL. Until now, USO was not even known to lie in PPAD or PLS. Our results also imply that parity games, mean-payoff games, discounted games, and simple-stochastic games lie in UniqueEOPL.

All of our containment results are proved via a reduction to a problem that we call One-Permutation Discrete Contraction (OPDC). This problem is motivated by a discretized version of contraction, but it is also closely related to the USO problem. We show that OPDC lies in UniqueEOPL, and we are also able to show that OPDC is UniqueEOPL-complete.

Finally, using the insights from our reduction for Piecewise-Linear Contraction, we obtain the first polynomial-time algorithms for finding fixed points of contraction maps in fixed dimension for any ℓ_p norm, where previously such algorithms were only known for the ℓ_2 and ℓ_∞ norms. Our reduction from P-LCP to UniqueEOPL allows a technique of Aldous [2] to be applied, which in turn gives the fastest-known randomized algorithm for P-LCP.

This paper substantially revises and extends the work described in our previous preprint "End of Potential Line" arXiv:1804.03450 [25].

Contents

1 Introduction

Total function problems in NP. The complexity class TFNP contains search problems that are guaranteed to have a solution, and whose solutions can be verified in polynomial time [58]. While it is a semantically defined complexity class and thus unlikely to contain complete problems, a number of syntactically defined subclasses of TFNP have proven very successful at capturing the complexity of total search problems. In this paper, we focus on two in particular, PPAD and PLS. The class PPAD was introduced in [64] to capture the difficulty of problems that are guaranteed total by a parity argument. It has attracted intense attention in the past decade, culminating in a series of papers showing that the problem of computing a Nash-equilibrium in two-player games is PPAD-complete [11,16], and more recently a conditional lower bound that rules out a PTAS for the problem [66]. No polynomial-time algorithms for PPAD-complete problems are known, and recent work suggests that no such algorithms are likely to exist [4,33]. PLS is the class of problems that can be solved by local search algorithms (in perhaps exponentially-many steps). It has also attracted much interest since it was introduced in [47], and looks similarly unlikely to have polynomial-time algorithms. Examples of problems that are complete for PLS include the problem of computing a pure Nash equilibrium in a congestion game [23], a locally optimal max cut [67], or a stable outcome in a hedonic game [30].

Continuous Local Search. If a problem lies in both PPAD and PLS then it is unlikely to be complete for either class, since this would imply an extremely surprising containment of one class in the other. In their 2011 paper [17], Daskalakis and Papadimitriou observed that there are several prominent total function problems in PPAD $\cap$ PLS for which researchers have not been able to design polynomial-time algorithms. Motivated by this they introduced the class CLS, a syntactically defined subclass of PPAD$\cap$PLS. CLS is intended to capture the class of optimization problems over a continuous domain in which a continuous potential function is being minimized and the optimization algorithm has access to a polynomial-time continuous improvement function. They showed that many classical problems of unknown complexity are in CLS, including the problem of solving a simple stochastic game, the more general problems of solving a Linear Complementarity Problem with a P-matrix, finding an approximate fixpoint to a contraction map, finding an approximate stationary point of a multivariate polynomial, and finding a mixed Nash equilibrium of a congestion game.

CLS problems with unique solutions. In this paper we study an interesting subset of problems that lie within CLS, and have *unique* solutions.

Contraction. In this problem we are given a function $f : \mathbb{R}^d \to \mathbb{R}^d$ that is purported to be c-contracting, meaning that for all points $x, y \in [0,1]^n$ we have $d(f(x), f(y)) \leq c \cdot d(x,y)$, where c is a constant satisfying $0 < c < 1$, and d is a distance metric. Banach's fixpoint theorem states that if f is contracting, then it has a unique *fixpoint* [3], meaning that there is a unique point $x \in \mathbb{R}^d$ such that $f(x) = x$.

P-LCP. The *P-matrix linear complementarity problem* (P-LCP) is a variant of the linear complementarity problem in which the input matrix is a P-matrix [14]. An interesting property of this problem is that, if the input matrix actually is a P-matrix, then the problem is guaranteed to have a unique solution [14]. Designing a polynomial-time algorithm for P-LCP has been open for decades, at least since the 1978 paper of Murty [62] that provided exponential-time examples for *Lemke's algorithm* [54] for P-LCPs.

USO. A *unique sink orientation* (USO) is an orientation of the edges of an n-dimensional hypercube such that every face of the cube has a unique sink. Since the entire cube is a face of itself, this means that there is a unique vertex of the cube that is a sink, meaning that all edges are oriented inwards. The USO problem is to find this *unique* sink.

All of these problems are most naturally stated as *promise* problems. This is because we have no way of verifying up front whether a function is contracting, whether a matrix is a P-matrix, or whether an orientation is a USO. Hence, it makes sense, for example, to study the contraction problem where it is promised that the function f is contracting, and likewise for the other two.

However, each of these problems can be turned into non-promise problems that lie in TFNP. In the case of Contraction, if the function f is not contracting, then there exists a short certificate of this fact. Specifically, any pair of points $x, y \in \mathbb{R}^d$ such that $d(f(x), f(y)) > c \cdot d(x, y)$ give an explicit proof that the function f is not contracting. We call these *violations*, since they witness a violation of the promise that is inherent in the problem.

So Contraction can be formulated as the non-promise problem of either finding a solution, or finding a violation. This problem is in TFNP because in the case where there is not a unique solution, there must exist a violation of the promise. The P-LCP and USO problems also have violations that can be witnessed by short certificates, and so they can be turned into non-promise problems contained in the same way, and these problems also lie in TFNP.

For Contraction and P-LCP we actually have the stronger result that both problems are in CLS [17]. Prior to this work USO was not known to lie in any non-trivial subclass of TFNP, and placing USO into a non-trivial subclass of TFNP was identified as an interesting open problem by Kalai [51, Problem 6].

We remark that not every problem in CLS has the uniqueness properties that we identify above. For example, the KKT problem [17] lies in CLS, but has no apparent notion of having a unique solution. The problems that we identify here seem to share the special property that there is a natural promise version of the problem, and that promise problem always has a unique solution.

1.1 Our contribution

In this paper, we define a complexity class that naturally captures the properties exhibited by problems like Contraction, P-LCP, and USO. In fact, we define two new sub-classes of CLS.

End of potential line. The complexity class EOPL contains every problem that can be reduced in polynomial time to the problem EndOfPotentialLine, which we define in this paper (Definition 9). The EndOfPotentialLine problem unifies in an extremely natural way the circuit-based views of PPAD and of PLS. The canonical PPAD-complete problem is EndOfLine, a problem that provides us with an exponentially large graph consisting of lines and cycles, and asks us to find the end of one of the lines. The canonical PLS-complete problem provides us with an exponentially large DAG, whose acyclicity is guaranteed by the existence of a *potential function* that increases along each edge. The problem EndOfPotentialLine is an instance of EndOfLine that *also* has a potential function that increases along each edge.

So the class EOPL captures problems that admit a *single* combinatorial proof of their joint membership in the classes PPAD of fixpoint problems and PLS of local search problems, and is a combinatorially-defined alternative to the class CLS. We are able to show that EOPL $\subseteq$ CLS (Corollary 11), by providing a polynomial-time reduction from EndOfPotentialLine to the EndOfMeteredLine problem defined by Hubáček and Yogev [45], which they have shown to lie in CLS.

We remark that it is an interesting open problem to determine whether EOPL = CLS. The inspiration behind both classes was to capture problems in PPAD $\cap$ PLS. The class CLS does this by affixing a potential function to the PPAD-complete Brouwer fixpoint problem, while EOPL does this affixing a potential function to the PPAD-complete problem EndOfLine. The class EOPL is not the main focus of this paper, however.

Unique end of potential line. An EndOfPotentialLine instance consists of an exponentially large graph that contains only lines (the cycles that can appear in EndOfLine instances are ruled out by the potential function.) The problem explicitly gives us the start of one of these lines. A solution to the problem is a vertex that is at the end of any line, other than the given start vertex. We could find a solution by following the line from the start vertex until we find the other end, although that may take exponential time. There may be many other lines in the graph though, and the starts and ends of these lines are also solutions.

We define the promise-problem PromiseUniqueEOPL, in which it is promised that there is a unique line in the graph. This line must be the one that begins at the given starting vertex, and so the only solution to the problem is the other end of that line. Thus, if the promise is satisfied, the problem has a unique solution. We can define the promise-class PromiseUEOPL which contains all promise-problems that can be reduced in polynomial-time to PromiseUniqueEOPL.

We are not just studying promise problems in this paper, however. We can turn PromiseUniqueEOPL into a non-promise problem by defining appropriate violations. One might imagine that a suitable violation would be a vertex that is the start of a second line. Indeed, if we are given the promise that there is no start to a second line, then we do obtain the problem PromiseUniqueEOPL. However, with just this violation, we obtain a problem that is identical to EndOfPotentialLine, which is not what we are intending to capture.

Instead we add a violation that captures any pair of vertices v and u that are provably on different lines, even if v and u are in the middle of their respective lines. We do this by using the potential function: if v and u have the same potential, then they must be on different lines, and likewise if the potential of u lies between the potential of v and the potential of the successor of v. We formalise this as the problem UniqueEOPL (Definition 12), and we define the complexity class UniqueEOPL to contain all (non-promise) problems that can be reduced in polynomial-time to UniqueEOPL.

We have that UniqueEOPL $\subseteq$ EOPL by definition, since UniqueEOPL simply adds an extra type of violation to EndOfPotentialLine. We remark that this new violation makes the problem substantially different from EndOfPotentialLine. In EndOfPotentialLine only the starts and ends of lines are solutions, while in UniqueEOPL there are many more solutions whenever there is more than one line in the instance. As such, we view UniqueEOPL as capturing a distinct subclass of problems in EOPL, and we view it as the natural class for promise-problems in PPAD $\cap$ PLS that have unique solutions.

UEOPL containment results. We show that USO, P-LCP, and a variant of the Contraction problem all lie in UniqueEOPL. We define the concept of a *promise-preserving* reduction, which is a polynomial-time reduction between two problems A and B, with the property that if A is promised to have a unique solution, then the instance of B that is produced by the reduction will also have a unique solution. All of the reductions that we produce in this paper are promise-preserving, which means that whenever we show that a problem is in UniqueEOPL, we also get that the corresponding promise problem lies in PromiseUEOPL.

Theorem 1 (cf. Theorem 28)**.** *USO is in* UniqueEOPL *under promise-preserving reductions.*

For the USO problem, our UniqueEOPL containment result substantially advances our knowledge about the problem. Prior to this work, the problem was only known to lie in TFNP, and Kalai [51, Problem 6] had posed the challenge to place it in some non-trivial subclass of TFNP. Our result places USO in UniqueEOPL, EOPL, CLS, PPAD (and hence PPA and PPP), and PLS, and so we answer Kalai's challenge by placing the problem in *all* of the standard subclasses of TFNP.

This result is in some sense surprising. Although every face of a USO has a unique sink, the orientation itself may contain cycles, and so there is no obvious way to define a potential function for the problem. Moreover, none of the well-known algorithms for solving USOs [40, 74] have the line-following nature needed to produce an EndOfLine instance. Nevertheless, our result shows that one can produce an EndOfLine instance that has potential function for the USO problem.

Theorem 2 (cf. Theorem 38 and Theorem 39)**.** *There are two different variants of the P-LCP problem, both of which lie in* UniqueEOPL *under promise-preserving reductions.*

We actually provide two different promise-preserving reductions from P-LCP to UniqueEOPL. The issue here is that there are many possible types of violation that one can define for P-LCP. So far, the standard formulation of P-LCP either asks for a solution, or a *non-positive principle minor* of the input matrix. A matrix is a P-matrix if and only if all of its principle minors are positive, and so this is sufficient to define a total problem.

The reduction from P-LCP to EndOfLine can map the start and end of each line back to either a solution or a non-positive principle minor. Our problem is that the extra violations in the UniqueEOPL instance, corresponding to a proof that there are multiple lines, do not easily map back to non-positive principle minors. They do however map back to other short certificates that the input matrix is not a P-matrix. For example, a matrix is a P-matrix if and only if it does not reverse the sign of any non-zero vector [14]. So we can also formulate P-LCP as a total problem that asks for a solution, or a non-zero vector whose sign is reversed.

We study the following variants of P-LCP. The first variant asks us to either find a solution, or find a non-positive principle minor, or find a non-zero vector whose sign is reversed. The second variant asks us to either find a solution, or a non-positive principle minor, or a third type of violation whose definition is inspired by a violation of the USO property. In all cases, we either solve the problem, or obtain a short certificate that the input was not a P-matrix, although the format of these certificates can vary.

It is not clear whether these variants are equivalent under polynomial-time reductions, as one would need to be able to map one type of violation to the other efficiently. We remark that if one is only interested in the promise problem, then the choice of violations is irrelevant. Both of our reductions show that promise P-LCP lies in PromiseUEOPL.

Theorem 3 (cf. Theorem 34)**.** *Finding the fixpoint of a piecewise linear contraction map in the ℓ_p norm is in* UniqueEOPL *under promise-preserving reductions, for any $p \in \mathbb{N} \cup \{\infty\}$.*

For Contraction, we study contraction maps specified by *piecewise linear* functions that are contracting with respect to an ℓ_p norm. This differs the contraction problem studied previously [17], where the function is given by an arbitrary arithmetic circuit. To see why this is necessary, note that although every contraction map has a unique fixpoint, if we allow the function to be specified by an arbitrary arithmetic circuit, then there is no guarantee that the fixpoint is rational. So it is not clear whether finding the *exact* fixpoint of a contraction map even lies in FNP.

Prior work has avoided this issue by instead asking for an *approximate* fixpoint, and the problem of finding an approximate fixpoint of a contraction map specified by an arithmetic circuit lies in CLS [17]. However, if we look for approximate fixpoints, then we destroy the uniqueness property that we are interested in, because there are infinitely many approximate fixpoint solutions surrounding any exact fixpoint.

So, we study the problem where the function is represented by a LinearFIXP arithmetic circuit [22], which is a circuit in which the multiplication of two variables is disallowed. This ensures that, when the function actually is contracting, there is a unique rational fixpoint that we can produce. We note that this is still an interesting class of contraction maps, since it is powerful enough to represent simple-stochastic games [22].

We place this problem in UniqueEOPL via a promise-preserving reduction. Our reduction can produce multiple types of violation. In addition to the standard violation of a pair of points at which f is not contracting, our reduction sometimes produces a different type of violation (cf. Definition 32), which while not giving an explicit violation of contraction, still gives a short certificate that f is not contracting.

Theorem 4. *The following problems are in* UniqueEOPL

- *Solving a parity game.*
- *Solving a mean-payoff game.*
- *Solving a discounted game.*
- *Solving a simple-stochastic game.*
- *Solving the ARRIVAL problem.*

Finally, we observe that our results prove that several other problems lie in UniqueEOPL. The simple-stochastic game (SSG) problem is known to reduce to Contraction [22] and to P-LCP [39], and thus our two results give two separate proofs that the SSG problem lies in UniqueEOPL. It is known that discounted games can be reduced to SSGs [77], mean-payoff games can be reduced to discounted games [77], and parity games can be reduced to mean-payoff games [65]. So all of these problems lie in UniqueEOPL too. Finally, Gärtner et al. [35] noted that the ARRIVAL problem [20] lies in EOPL, and in fact their EndOfPotentialLine instance always contains exactly one line, and so the problem also lies in UniqueEOPL.

We remark that none of these are promise-problems. Each of them can be formulated so that they *unconditionally* have a unique solution. Hence, these problems seem to be easier than the problems captured by UniqueEOPL, since problems that are complete for UniqueEOPL only have a unique solution conditioned on the promise that there are no violations.

A UEOPL-complete problem. In addition to our containment results, we also give a UniqueEOPL-completeness result. Specifically, we show that *One-Permutation Discrete Contraction* (OPDC) is complete for UniqueEOPL.

OPDC is a problem that is inspired by both Contraction and USO. Intuitively, it is a discrete version of Contraction. The inputs to the problem are (a concise representation of) a discrete grid of points P covering the space $[0,1]^d$, and a set of *direction* functions $\mathcal{D} = D_{i=1\dots d}$, where each function D_i has the form $D_i : P \to \{\mathsf{up}, \mathsf{down}, \mathsf{zero}\}$. To discretize a map $f : [0,1]^d \to [0,1]^d$ we simply define D_i so that

- $D_i(p) = \mathsf{up}$ whenever $f(p)_i > p_i$,
- $D_i(p) = \mathsf{zero}$ whenever $f(p)_i = p_i$, and

- $D_i(p) = \mathsf{down}$ whenever $f(p)_i < p_i$.

So the direction function for dimension i simply points in the direction that f moves in dimension i. To solve the problem, we seek a point $p \in P$ such that $D_i(p) = \mathsf{zero}$ for all i, which corresponds to a fixpoint of f.

Why is the problem called *One Permutation* Discrete Contraction? This is due to some extra constraints that we place on the problem. An interesting property of a function f that is contracting with respect to an ℓ_p norm is that if we restrict the function to a *slice*, meaning that we fix some of the coordinates and let others vary, then the resulting function is still contracting. We encode this property into the OPDC problem, by insisting that slices should also have unique fixpoints when we ignore the dimensions not in the slice. However, we do not do this for all slices, but only for *i-slices* in which the *last* $d-i+1$ coordinates have been fixed. In this sense, our definition depends on the order of the dimensions. If we rearranged the dimensions into a different permutation, then we would obtain a different problem, so each problem corresponds to some particular permutation of the dimensions. The name of the problem was chosen to reflect this fact.

We remark that, although the problem was formulated as a discretization of Contraction, it is also closely related to the USO problem. Specifically, if we take the set of points P to be the $\{0, 1\}^n$ hypercube, then the direction functions actually specify an orientation of the cube. Moreover, the condition that every slice should have have unique fixpoint *exactly corresponds* to the USO property that every face should have a unique sink. However, since we only insist on this property for i-slices, OPDC can be viewed as a variant of USO in which *only some* of the faces have unique sinks.

Theorem 5 (cf. Theorem 21)**.** *OPDC lies in* UniqueEOPL *under promise-preserving reductions.*

OPDC actually plays a central role in the paper. We reduce Piecewise-Linear Contraction, USO, and P-LCP to it, and we then reduce OPDC to UniqueEOPL, as shown in Figure 1. The reduction from OPDC to UniqueEOPL is by far the most difficult step.

In the case where the promise is satisfied, meaning that the OPDC instance has a unique fixpoint, the reduction defines a single line that starts at the point $p \in P$ with $p_i = 0$ for all i, and ends at the unique fixpoint. This line walks around the grid, following the direction functions given by $\mathcal{D}$ in a specific manner that ensures that it will find the fixpoint, while also decreasing a potential function at each step. We need to very carefully define the vertices of this line, to ensure that the line is unique, and for this we crucially rely on the fact that every i-slice also has a unique fixpoint.

This does not get us all the way to UniqueEOPL though, because the line we describe above lacks a *predecessor* circuit. In UniqueEOPL, each vertex has a predecessor, a successor and a potential, but the line we construct only gives successors and potentials to each vertex. To resolve this, we apply the *pebbling game* reversibility argument introduced by Bitansky et al [4], and later improved by Hubáček and Yogev [45]. Using this technique allows us to produce a predecessor circuit, as long as there is exactly one line in the instance.

Our reduction also handles violations in the OPDC instance. The key challenge here is that the pebbling game argument assumes that there is exactly one line, and so far it has only been applied to promise-problems. We show that the argument can be extended to work with non-promise problems that may have multiple lines. This can cause the argument to break, specifically when multiple lines are detected, but we are able to show that these can be mapped back to violations in the OPDC instance.

Theorem 6 (cf. Theorem 25)**.** OPDC *is* UniqueEOPL*-complete under promise-preserving reductions, even when the set of points P is a hypercube.*

We show that OPDC is UniqueEOPL-hard by giving a polynomial-time promise-preserving reduction from UniqueEOPL to OPDC. This means that OPDC is UniqueEOPL-complete, and the variant of OPDC in which it is promised that there are no violations is PromiseUEOPL-complete.

Our reduction produces an OPDC instances where the set of points P is the boolean hypercube $\{0, 1\}^n$. In the case where the UniqueEOPL instance has no violations, meaning that it contains a single line, the reduction embeds this line into the hypercube. To do this, it splits the line in half. The second half is embedded into a particular sub-cube, while the first half is embedded into all other sub-cube. This process is recursive, so each half of the line is again split in half, and further embedded into sub-cubes. The reduction ensures that the only fixpoint of the instance corresponds to the end of the line. If the UniqueEOPL instance does have violations, then this embedding may fail. However, in any instance where the embedding fails, we are able to produce a violation for the original UniqueEOPL instance.

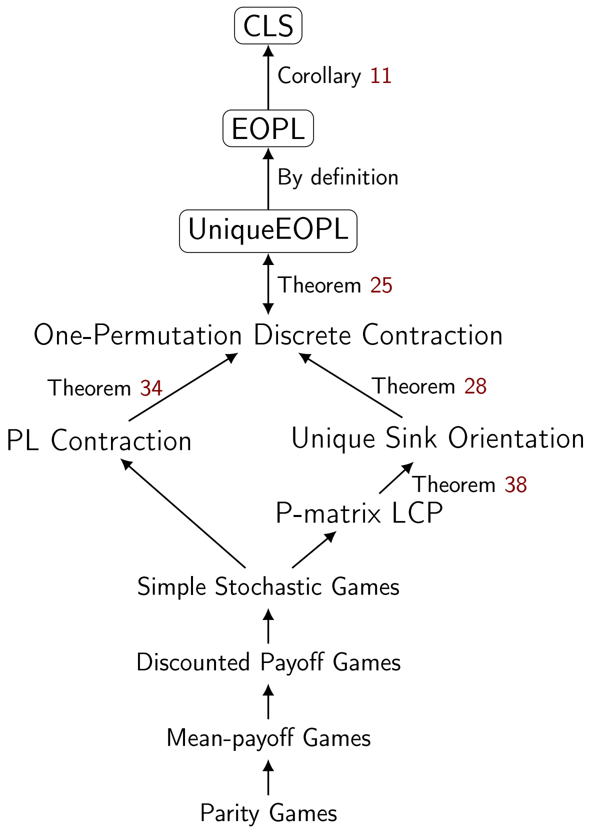


Figure 1: The relationship of UniqueEOPL to other classes and problems.

We remark that this hardness reduction makes significant progress towards showing a hardness result for Contraction and USO. As we have mentioned, OPDC is a discrete variant of Contraction, and when the set of points is a hypercube, the problem is also very similar to USO. The key difference is that OPDC insists that only i-slices should have a unique fixpoint, whereas Contraction and USO insist that *all* slices should have unique fixpoints. To show a hardness result for either of those two problems, one would need to produce an OPDC instance with that property.

New algorithms. Our final contributions are algorithmic and arise from the structural insights provided by our containment results. Using the ideas from our reduction from Piecewise-Linear Contraction to UniqueEOPL, we obtain the first polynomial-time algorithms for finding fixpoints of Piecewise-Linear Contraction maps in fixed dimension for any ℓ_p norm, where previously such algorithms were only known for ℓ_2 and ℓ_∞ norms. If this input is not a contraction map, then our algorithm may instead produce a short certificate that the function is not contracting.

We also show that these results can be extended to the case where the contraction map is given by a general arithmetic circuit. In this case, we provide polynomial-time algorithm that either finds an approximate fixpoint, or produces a short certificate that the function is not contracting.

An interesting consequence of our algorithms is that it is now unlikely that ℓ_p-norm Contraction in fixed-dimension is CLS-complete. This should be contrasted to the recent result of Daskalakis et al. [18], who showed the variant of the contraction problem where a metric is given as part of the input is CLS-complete, even in dimension 3. Our result implies that it is unlikely that this can be directly extended to ℓ_p norms, at least not without drastically increasing the number of dimensions in the instance.

Finally, as noted in [35], one of our reductions from P-LCP to EndOfPotentialLine allows a technique of Aldous [2] to be applied, which in turn gives the fastest known randomized algorithm for P-LCP.

1.2 Related work

CLS. Recent work by Hubáček and Yogev [45] proved lower bounds for CLS. They introduced a problem known as EndOfMeteredLine which they showed was in CLS, and for which they proved a query complexity lower bound of $\Omega(2^{n/2}/\sqrt{n})$ and hardness under the assumption that there were one-way permutations and indistinguishability obfuscators for problems in $\mathsf{P}_{/\mathsf{poly}}$. Another recent result showed that the search version of the Colorful Carathéodory Theorem is in $\mathsf{PPAD} \cap \mathsf{PLS}$, and left open whether the problem is also in CLS [60].

Until recently, it was not known whether there was a natural CLS-complete problem. In their original paper, Daskalakis and Papadimitriou suggested two natural candidates for CLS-complete problems, ContractionMap and P-LCP, which we study in this paper. Recently, two variants of ContractionMap have been shown to be CLS-complete. Whereas in the original definition of ContractionMap it is assumed that an ℓ_p or ℓ_∞ norm is fixed, and the contraction property is measured with respect to the metric induced by this fixed norm, in these two new complete variants, a metric [18] and meta-metric [24] are given as input to the problem[2].

P-LCP. Papadimitriou showed that P-LCP, the problem of solving the LCP or returning a violation of the P-matrix property, is in PPAD [64] using Lemke's algorithm. The relationship between Lemke's algorithm and PPAD has been studied by Adler and Verma [1]. Later, Daskalakis and Papadimitrou showed that P-LCP is in CLS [17], using the potential reduction method in [53]. Many algorithms for P-LCP have been studied, e.g., [52, 61, 62]. However, no polynomial-time algorithms are known for P-LCP, or for the promise version where one can assume that the input matrix is a P-matrix.

The best known algorithms for P-LCP are based on a reduction to Unique Sink Orientations (USOs) of cubes [73]. For an P-matrix LCP of size n, the USO algorithms of [74] apply, and give a deterministic algorithm that runs in time $O(1.61^n)$ and a randomized algorithm with expected running time $O(1.43^n)$. The application of Aldous' algorithm [2] to the UniqueEOPL instance that we produce from a P-matrix LCP takes expected time $2^{n/2} \cdot \text{poly}(n) = O(1.4143^n)$ in the worst case.

Unique Sink Orientations. In this paper we study USOs of cubes, a problem that was first studied by Stickney and Watson [73] in the context of P-matrix LCPs. A USO arising from a P-matrix may be cyclic. Motivating by Linear Programming, *acylic*, AUSOs have also been studied, both for cubes and general polytopes [37, 42]. Recently Gärtner and Thomas studied the computational complexity of recognizing USOs and AUSOs [38]. They found that the problem is coNP-complete for USOs and PSPACE-complete for AUSOs. A series of papers provide upper and lower bounds for specific algorithms for solving (A)USOs, including [28, 29, 40, 56, 69, 74, 75]. An AUSOs on a n-dimensional cube can be solved in subexponential time, by the RANDOM-FACET algorithm, which is essentially tight for this algorithm [34]. An almost quadratic lower bound on the number of vertex evaluations needed to solve a general USO is known [68]; unlike for AUSOs, the best running

[2] The result for meta-metrics appeared in an unpublished earlier version of this paper [24] that contained only a small fraction of the results from the current version. This result for meta-metrics is dropped from this current version given that it has been superseded, and is not important for our main message.

times known for general USOs, as for P-matrix LCPs, are exponential. To be best of our knowledge, we are first to study the general problem of solving a USO from a complexity-theoretic point of view.

Contraction. The problem of computing a fixpoint of a continuous map $f : \mathcal{D} \mapsto \mathcal{D}$ with Lipschitz constant c has been extensively studied, in both continuous and discrete variants [9, 10, 19]. For arbitrary maps with $c > 1$, exponential bounds on the query complexity of computing fixpoints are known [8, 41]. In [6, 44, 72], algorithms for computing fixpoints for specialized maps such as weakly ($c = 1$) or strictly ($c < 1$) contracting maps are studied. For both cases, algorithms are known for the case of ℓ_2 and ℓ_∞ norms, both for absolute approximation ($||x - x^*|| \leq \epsilon$ where x^* is an exact fixpoint) and relative approximation ($||x - f(x)|| \leq \epsilon$). A number of algorithms are known for the ℓ_2 norm handling both types of approximation [43, 63, 71]. There is an exponential lower bound for absolute approximation with $c = 1$ [71]. For relative approximation and a domain of dimension d, an $O(d \cdot \log 1/\epsilon)$ time algorithm is known [43]. For absolute approximation with $c < 1$, an ellipsoid-based algorithm with time complexity $O(d \cdot [\log(1/\epsilon) + \log(1/(1-c))])$ is known [43]. For the ℓ_∞ norm, [70] gave an algorithm to find an ϵ-relative approximation in time $O(\log(1/\epsilon)^d)$ which is polynomial for constant d. In summary, for the ℓ_2 norm polynomial-time algorithms are known for strictly contracting maps; for the ℓ_∞ norm algorithms that are polynomial time for constant dimension are known. For arbitrary ℓ_p norms, to the best of our knowledge, no polynomial-time algorithms for constant dimension were known before this paper.

Infinite games. Simple Stochastic Games are related to Parity games, which are an extensively studied class of two-player zero-sum infinite games that capture important problems in formal verification and logic [21]. There is a sequence of polynomial-time reductions from parity games to mean-payoff games to discounted games to simple stochastic games [36, 39, 50, 65, 77]. The complexity of solving these problems is unresolved and has received much attention over many years (see, for example, [5, 13, 27, 28, 48, 77]). In a recent breakthrough [7], a quasi-polynomial time algorithm for parity games have been devised, and there are now several algorithms with this running time [7, 26, 49]. For mean-payoff, discounted, and simple stochastic games, the best-known algorithms run in randomized subexponential time [55]. The existence of a polynomial time algorithm for solving any of these games would be a major breakthrough. Simple stochastic games can also be reduced in polynomial time to Piecewise-Linear Contraction with the ℓ_∞ norm [22].

1.3 Future directions

A clear direction for future work is to show that further problems are UniqueEOPL-complete. We have several conjectures.

Conjecture 1. *USO is hard for* UniqueEOPL*.*

We think that, among our three motivating problems, USO is the most likely to be UniqueEOPL-complete. Our hardness proof for OPDC already goes some way towards proving this, since we showed that OPDC was hard even when the set of points is a hypercube. The key difference between OPDC on a hypercube and USO is that OPDC only requires that the faces corresponding to i-slices should have unique sinks, while USO requires that all faces should have unique sinks.

Conjecture 2. *Piecewise-Linear Contraction in an ℓ_p norm is hard for* UniqueEOPL*.*

Our OPDC hardness result also goes some way towards showing that Piecewise-Linear Contraction is hard, however there are more barriers to overcome here. In addition to the i-slice vs. all slice issue, we would also need to convert the discrete OPDC problem to the continuous contraction problem. Converting discrete problems to continuous fixpoint problems has been well-studied in the context of PPAD-hardness reductions [16, 59], but here the additional challenge is to carry out such a reduction while maintaining the contraction property.

Aside from hardness, we also think that the relationship between Contraction and USO should be explored further. Our formulation of the OPDC problem exposes significant similarities between the two problems, which until this point have not been recognised. Can we reduce USO to Contraction in polynomial time?

Conjecture 3. *P-LCP is hard for* UniqueEOPL.

Of all of our conjectures, this will be the most difficult to prove. Since P-LCP reduces to USO, the hardness of USO should be resolved before we attempt to show that P-LCP is hard. One possible avenue towards showing the hardness of P-LCP might be to reduce from Piecewise-Linear Contraction. Our UniqueEOPL containment proof for Piecewise-Linear Contraction makes explicit use of the fact that the problem can be formulated as an LCP, although in that case the resulting matrix is not a P-matrix. Can we modify the reduction to produce a P-matrix?

Conjecture 4. UniqueEOPL $\subset$ EOPL $=$ CLS.

The question of EOPL vs CLS is unresolved, and we actually think it could go either way. One could show that EOPL $=$ CLS by placing either of the two known CLS-complete Contraction variants into EOPL [18, 24]. If the two classes are actually distinct, then it is interesting to ask which of the problems in CLS are also in EOPL.

On the other hand, we believe that UniqueEOPL is a strict subset of EOPL. The evidence for this is that the extra violation in UniqueEOPL that does not appear in EndOfPotentialLine changes the problem significantly. This new violation will introduce many new solutions whenever there are multiple lines in the instance, and so it is unlikely, in our view, that one could reduce EndOfPotentialLine to UniqueEOPL. Of course, there is no hope to unconditionally prove that UniqueEOPL $\subset$ EOPL, but we can ask for further evidence to support the idea. For example, can oracle separations shed any light on the issue?

Finally, we remark that UniqueEOPL is the closest complexity class to FP, among all the standard sub-classes of TFNP. However, we still think that further subdivisions of UniqueEOPL will be needed. Specifically, we do not believe that simple stochastic games, or any of the problems that can be reduced to them, are UniqueEOPL-complete, since all of these problems have unique solutions unconditionally. Further research will be needed to classify these problems.

2 Unique End of Potential Line

In this section we define two new complexity classes called EOPL and UniqueEOPL. These two classes are defined by merging the definitions of PPAD and PLS, so we will begin by recapping those classes.

The complexity class PPAD contains every problem that can be reduced to EndOfLine [64].

Definition 7 (EndOfLine)**.** *Given Boolean circuits* $S, P : \{0,1\}^n \to \{0,1\}^n$ *such that* $P(0^n) = 0^n \neq S(0^n)$*, find one of the following:*

(E1) A point $x \in \{0,1\}^n$ *such that* $P(S(x)) \neq x$.

(E2) A point $x \in \{0,1\}^n$ *such that* $x \neq 0^n$ *and* $S(P(x)) \neq x$.

Intuitively, the problem defines an exponentially large graph where all vertices vhave in-degree and out-degree at most one. Each bit-string in $\{0,1\}^n$ defines a vertex, while the functions S and P define *successor* and *predecessor* functions for each vertex. A directed edge exists from vertex x and y if and only if $S(x) = y$ and $P(y) = x$. Any vertex x for which $P(S(x)) \neq x$ has no outgoing edge, while every vertex y with $S(P(x)) \neq x$ has no incoming edge.

The condition that $P(0^n) = 0^n \neq S(0^n)$ specifies that the vertex 0^n has no incoming edge, and so it is the start of a line. To solve the problem, we must find either a solution of type (E1), which is a vertex x that is the end of a line, or a solution of type (E2), which is a vertex x other than 0^n that is the start of a line. Since we know that the graph has at least one source, there must at least exist a solution of type (E1), and so the problem is in TFNP.

The complexity class PLS contains every problem that can be reduced to the SinkOfDag[3]

Definition 8 (SinkOfDag)**.** *Given a Boolean circuit $S : \{0,1\}^n \to \{0,1\}^n$ such that $S(0^n) \neq 0^n$, and a circuit $V : \{0,1\}^n \to \{0,1,\ldots,2^m - 1\}$ find:*

(S1) A vertex $x \in \{0,1\}^n$ such that $S(x) \neq x$ and either $S(S(x)) = x$ or $V(S(x)) \leq V(x)$.

Once again, the problem specifies an exponentially large graph on the vertex set $\{0,1\}^n$, but this time the only guarantee is that each vertex has out-degree one. The circuit S gives a successor function. In this problem, some bit-strings do not correspond to vertices in the graph. Specifically, if we have $S(x) = x$ for some bit-string $x \in \{0,1\}^n$, then x does not encode a vertex.

The second circuit V gives a *potential* to each vertex from the set $\{0,1,\ldots,2^m - 1\}$. An edge exists in the graph if and only if the potential *increases* along that edge. Specifically, there is an edge from x to y if and only if $S(x) = y$ and $V(x) < V(y)$. This restriction means that the graph must be a DAG.

To solve the problem, we must find a sink of the DAG, ie., a vertex that has no outgoing edge. Since we require that $S(0^n) \neq 0^n$, we know that the DAG has at least one vertex, and therefore it must also have at least one sink. This places the problem in TFNP.

End of potential line. We define a new problem called EndOfPotentialLine, which merges the two definitions of EndOfLine and SinkOfDag into a single problem.

Definition 9 (EndOfPotentialLine)**.** *Given Boolean circuits $S, P : \{0,1\}^n \to \{0,1\}^n$ such that $P(0^n) = 0^n \neq S(0^n)$ and a Boolean circuit $V : \{0,1\}^n \to \{0,1,\ldots,2^m - 1\}$ such that $V(0^n) = 0$ find one of the following:*

(R1) A point $x \in \{0,1\}^n$ such that $S(P(x)) \neq x \neq 0^n$ or $P(S(x)) \neq x$.

(R2) A point $x \in \{0,1\}^n$ such that $x \neq S(x)$, $P(S(x)) = x$, and $V(S(x)) - V(x) \leq 0$.

This problem defines an exponentially large graph where each vertex has in-degree and out-degree at most one (as in EndOfLine) that is also a DAG (as in SinkOfDag). An edge exists from x to y if and only if $S(x) = y$, $P(y) = x$, and $V(x) < V(y)$. As in SinkOfDag, only some bit-string encode vertices, and we adopt the same idea that if $S(x) = x$ for some bit-string x, then x does *not* encode a vertex.

So we have a single instance that is simultaneously an instance of EndOfLine and an instance of SinkOfDag. To solve the problem, it suffices to solve *either* of these problems. Solutions of type (R1) consist of vertices x that are either the end of a line, or the start of a line (excluding the case where $x = 0^n$). Solutions of type (R2) consist of any point x where the potential does not strictly increase on the edge between x and $S(x)$.

[3]While this is not the standard definition of PLS, it has been observed that the standard PLS-complete problem can easily be recast as a SinkOfDag instance [64].

The complexity class EOPL. We define the complexity class EOPL to consist of all problems that can be reduced in polynomial time to EndOfPotentialLine. By definition the problem lies in PPAD $\cap$ PLS, since one can simply ignore solutions of type (R2) to obtain an EndOfLine instance, and ignore solutions of type (R1) to obtain a SinkOfDag instance.

In fact we are able to show the stronger result that EOPL $\subseteq$ CLS. To do this, we reduce EndOfPotentialLine to the problem EndOfMeteredLine, which was defined by Hubáček and Yogev, who also showed that the problem lies in CLS [45]. The main difference between the two problems is that EndOfMeteredLine requires that the potential increases by *exactly* one along each edge. The reduction from EndOfMeteredLine to EndOfPotentialLine is straightforward. The other direction is more involved, and requires us to insert new vertices into the instance. Specifically, if there is an edge between a vertex x and a vertex y, but $V(y) \neq V(x)+1$, then we need to insert a new chain of vertices of length $V(y) - V(x) - 1$ between x and y, so that we can ensure that the potential always increases by exactly one along each edge. The full details are given in Appendix A, where the following theorem is proved.

Theorem 10. EndOfMeteredLine *and* EndOfPotentialLine *are polynomial-time equivalent.*

As we have mentioned, Hubáček and Yogev have shown that EndOfMeteredLine lies in CLS [45], so we get the following corollary.

Corollary 11. EOPL $\subseteq$ CLS.

Problems with unique solutions. The problems that we study in this paper all share a specific set of properties that cause them to produce an interesting subclass of EndOfPotentialLine instances. Each of the problems that we study have a *promise*, and if the promise is satisfied the problem has a *unique* solution.

For example, in the Contraction problem, we are given a function $f : [0,1]^d \to [0,1]^d$ that is promised to be *contracting*, meaning that $d(f(x), f(y)) \leq c \cdot f(x, y)$ for some positive constant $c < 1$ and some distance metric d. We cannot efficiently check whether f is actually contracting, but if it is, then Banach's fixpoint theorem states that f has a unique fixpoint [3]. If f is not contracting, then there will exist *violations* that can be witnessed by short certificates. For Contraction, a violation is any pair of points x, y such that $d(f(x), f(y)) > c \cdot f(x, y)$.

We can use violations to formulate the problem as a non-promise problem that lies in TFNP. Specifically, if we ask for either a fixpoint or a violation of contraction, then the contraction problem is total, because if there is no fixpoint, then the contrapositive of Banach's theorem implies that there must exist a violation of contraction.

Unique End of Potential Line. When we place this type of problem in EOPL, we obtain an instance with extra properties. Specifically, if the original problem has no violations, meaning that the promise is satisfied, then the EndOfPotentialLine instance will contain a *single* line that starts at 0^n, and ends at the unique solution of the original problem. This means that, if we ever find two distinct lines in our EndOfPotentialLine instance, then we immediately know that original instance fails to satisfy the promise.

We define the following problem, which is intended to capture these properties.

Definition 12 (UniqueEOPL). *Given Boolean circuits $S, P : \{0,1\}^n \to \{0,1\}^n$ such that $P(0^n) = 0^n \neq S(0^n)$ and a Boolean circuit $V : \{0,1\}^n \to \{0, 1, \ldots, 2^m - 1\}$ such that $V(0^n) = 0$ find one of the following:*

(U1) A point $x \in \{0,1\}^n$ such that $P(S(x)) \neq x$.

(UV1) A point $x \in \{0,1\}^n$ such that $x \neq S(x)$, $P(S(x)) = x$, and $V(S(x)) - V(x) \leq 0$.

(UV2) A point $x \in \{0,1\}^n$ such that $S(P(x)) \neq x \neq 0^n$.

(UV3) Two points $x, y \in \{0,1\}^n$, such that $x \neq y$, $x \neq S(x)$, $y \neq S(y)$, and either $V(x) = V(y)$ or $V(x) < V(y) < S(x)$.

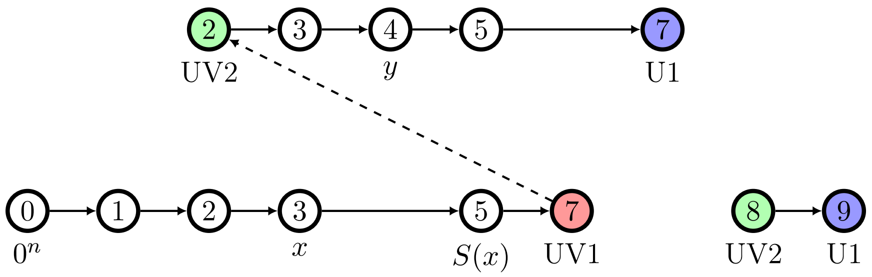


Figure 2: Example of solutions, including violations, for UniqueEOPL. This figure should be viewed in color. In this example we have 3 lines. The *main line* starts at 0^n and ends with a UV1 solution, when the successor of the red vertex, which has potential 7, has lower potential, 2. That successor is thus the start of another line, which is a UV2 solution. There is a final line of length one to the bottom right, which also has a start, which is another UV2 solution. This line does not intersect either of the other two lines in terms of the ranges of their potential values, so does not contribute to UV3 solutions. The main line and top line do intersect in terms of the ranges of their potential values and they contribute many UV3 solutions. We highlight one on the diagram with x, $S(x)$, and y, such that $V(x) < V(y) < V(S(x))$. There are two U1 solutions at the end of the top and bottom right lines. Finally, we note that if there were no violations, then there must be one line and a single U1 solution at the end of the main line.

We split the solutions into two types: proper solutions and violations. Solutions of type (U1) encode the end of any line, which are the proper solutions to the problem. There are three types of violation solution. Violations of type (UV1) are vertices at which the potential fails to strictly increase. Violations of type (UV2) ask for the start of any line, other than the vertex 0^n. Clearly, if there are two sources in the graph, then there are two lines.

Violations of type (UV3) give another witness that the instance contains more than one line. This is encoded by a pair of vertices x and y, with either $V(x) = V(y)$, or with the property that the potential of y lies between the potential of x and $S(x)$. Since we require the potential to strictly increase along every edge of a line, this means that y cannot lie on the same line as x, since all vertices before x in the line have potential strictly less than $V(x)$, while all vertices after $S(x)$ have potential strictly greater than $V(S(x))$.

We remark that (UV2) by itself already captures the property "there is a unique line", since if a second line cannot start, then it cannot exist. So why do we insist on the extra type of violation given by (UV3)? Violations of type (UV3) allow us to solve the problem immediately if we ever detect the existence of multiple lines. Note that this is not the case if we only have solutions of type (UV2), since we may find two vertices on two different lines, but both of them may be exponentially many steps away from the start of their respective lines.

By adding (UV3) solutions, we make the problem easier than EndOfPotentialLine (note that without UV3, the problem is actually the same as EndOfPotentialLine). This means that problems that can be reduced to UniqueEOPL have the very special property that, if at any point you detect the existence of multiple lines, either through the start of a second line, or through a violation in (UV3), then you *immediately* get a violation in the original problem without any extra effort. All of the problems that we study in this paper share this property.

The complexity class UniqueEOPL. We define the complexity class UniqueEOPL to be the class of problems that can be reduced in polynomial time to UniqueEOPL. We note that UniqueEOPL $\subseteq$ EOPL is trivial, since the problem remains total even if we disallow solutions of type UV3.

For each of our problems, it is also interesting to consider the complexity of the promise variant, in which it is guaranteed via a promise that no violations exist. We define PromiseUniqueEOPL to be the promise version of UniqueEOPL in which 0^n is the only start of a line (and hence there are no solutions that are type (UV2) or (UV3)). We define the complexity class PromiseUEOPL to be the class of promise problems that can be reduced in polynomial time to PromiseUniqueEOPL.

Promise-preserving reductions. The problem UniqueEOPL has the interesting property that, if it is promised that there are no violation solutions, then there must be a unique solution. All of the problems that we study in this paper share this property, and indeed when when we reduce them to UniqueEOPL, the resulting instance will have a unique line whenever the original problem has no violation solutions.

We formalise this by defining the concept of a *promise-preserving* reduction. This is a reduction between two problems A and B, both of which have proper solutions and violation solutions. The reduction is promise-preserving if, when it is promised that A has no violations, then the resulting instance of B also has no violations. Hence, if we reduce a problem to UniqueEOPL via a chain of promise-preserving reductions, and we know that there are no violations in the original problem, then there is a unique line ending at the unique proper solution in the UniqueEOPL instance.

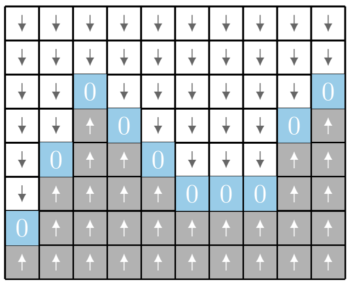
Note that this is more restrictive than a general reduction. We could in principle produce a reduction that took an instance of A, where it is promised that there are no violations, and produce an instance of B that sometimes contains violations. By using promise-preserving reductions, we are showing that our problems have the natural properties that one would expect for a problem in UniqueEOPL. Specifically, that the promise version has a unique solution, and that this can be found by following the unique line in the UniqueEOPL instance.

One added bonus is that, if we show that a problem is in UniqueEOPL via a chain of promise-preserving reductions, then we automatically get that the promise version of that problem, where it is promised that there are no violations, lies in PromiseUEOPL. Moreover, if we show that a problem is UniqueEOPL-complete via a promise-preserving reduction, then this also implies that the promise version of that problem is PromiseUEOPL-complete.

3 One-Permutation Discrete Contraction

The *One-Permutation Discrete Contraction* (OPDC) problem will play a crucial role in our results. We will show that the problem lies in UniqueEOPL, and we will then reduce both PL-Contraction and Grid-USO to OPDC, thereby showing that those problems also lie in UniqueEOPL. We will also show that UniqueEOPL can be reduced to OPDC, making this problem the first example of a non-trivial UniqueEOPL-complete problem.

Direction functions. The OPDC can be seen as a discrete variant of the continuous contraction problem. Recall that a contraction map is a function $f : [0, 1]^n \to [0, 1]^d$ that is contracting under a metric d, i.e., $d(f(x), f(y)) \leq c \cdot f(x, y)$ for all $x, y \in [0, 1]^d$ and some constant c satisfying $0 < c < 1$.



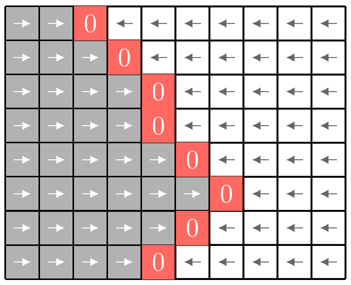


Figure 3: Left: A direction function for the up/down dimension. Right: A direction function for the left/right dimension. This figure should be viewed in color.

We will discretize the space by overlaying a grid of points on the $[0,1]^d$ cube. Let $[k]$ denote the set $\{0, 1, \ldots, k\}$. Given a tuple of grid widths $(k_1, k_2, \ldots, k_d)$, we define the set

$$P(k_1, k_2, \ldots, k_d) = [k_1] \times [k_2] \times \cdots \times [k_d].$$

We will refer to $P(k_1, k_2, \ldots, k_d)$ simply as P when the grid widths are clear from the context. Note that each point $p \in P$ is a tuple $(p_1, p_2, \ldots, p_d)$, where p_i is an integer between 0 and k_i, and this point maps onto the point $(p_1/k_1, p_2/k_2, \ldots, p_d/k_d) \in [0,1]^d$.

Instead of a single function f, in the discrete version of the problem we will use a family of *direction functions* over the grid P. For each dimension $i \leq d$, we have function $D_i : P \to \{\mathsf{up}, \mathsf{down}, \mathsf{zero}\}$. Intuitively, the natural reduction from a contraction map f to a family of direction functions would, for each point $p \in P$ and each dimension $i \leq d$ set:

- $D_i(p) = \mathsf{up}$ whenever $f(p)_i > p_i$,
- $D_i(p) = \mathsf{down}$ whenever $f(p)_i < p_i$, and
- $D_i(p) = \mathsf{zero}$ whenever $f(p)_i = p_i$.

In other words, the function D_i simply outputs whether $f(p)$ moves up, down, or not at all in dimension i. So a point $p \in P$ with $D_i(p) = \mathsf{zero}$ for all i would correspond to the fixpoint of f. Note, however, that the grid may not actually contain the fixpoint of f, and so there may be no point p satisfying $D_i(p) = \mathsf{zero}$ for all i.

A two-dimensional example. To illustrate this definition, consider the two-dimensional instance given in Figure 3, which we will use as a running example. It shows two direction functions: the figure on the left shows a direction function for the up-down dimension, which we will call dimension 1 and illustrate using the color blue. The figure on the right shows a direction function for the left-right dimension, which we will call dimension 2 and illustrate using the color red. Each square in the figures represents a point in the discretized space, and the value of the direction function is shown inside the box. Note that there is exactly one point p where $D_1(p) = D_2(p) = \mathsf{zero}$, which is the fixpoint that we seek.

Slices. We will frequently refer to subsets of P in which some of the dimensions have been fixed. A *slice* will be represented as a tuple $(s_1, s_2, \ldots, s_d)$, where each s_i is either

- a number in $[0,1]$, which indicates that dimension i should be fixed to s_i, or

- the special symbol $*$, which indicates that dimension i is free to vary.

We define $\mathsf{Slice}_d = ([0,1] \cup \{*\})^d$ to be the set of all possible slices in dimension d. Given a slice $s \in \mathsf{Slice}_d$, we define $P_s \subseteq P$ to be the set of points in that slice, ie., P_s contains every point $p \in P$ such that $p_i = s_i$ whenever $s_i \neq *$. We'll say that a slice $s' \in \mathsf{Slice}_d$ is a sub-slice of a slice $s \in \mathsf{Slice}_d$ if $s_j \neq * \implies s'_j = s_j$ for all $j \in [d]$.

An *i-slice* is a slice s for which $s_j = *$ for all $j \leq i$, and $s_j \neq *$ for all $j > i$. In other words, all dimensions up to and including dimension i are allowed to vary, while all other dimensions are fixed.

In our two-dimensional example, there are three types of i-slices. There is one 2-slice: the slice $(*, *)$ that contains every point. For each x, there is a 1-slice $(*, x)$, which restricts the left/right dimension to the value n. For each pair x, y there is a 0-slice (y, x), which contains only the exact point corresponding to x and y.

Discrete contraction maps. We can now define a one-permutation discrete contraction map. We say that a point $p \in P_s$ in some slice s is a *fixpoint* of s if $D_i(p) = \mathsf{zero}$ for all dimensions i where $s_i = *$. The following definition captures the promise version of the problem, and we will later give a non-promise version by formulating appropriate violations.

Definition 13 (One-Permutation Discrete Contraction Map)**.** *Let P be a grid of points over $[0,1]^d$ and let $\mathcal{D} = (D_i)_{i=1,\ldots,d}$ be a family of direction functions over P. We say that $\mathcal{D}$ and P form a one-permutation discrete contraction map if, for every i-slice s, the following conditions hold.*

1. *There is a unique fixpoint of s.*
2. *Let $s' \in \mathsf{Slice}_d$ be a sub-slice of s where some coordinate i for which $s_i = *$ has been fixed to a value, and all other coordinates are unchanged. If q is the unique fixpoint of s, and p is the unique fixpoint of s', then*
   - *if $p_i < q_i$, then $D_i(p) = \mathsf{up}$, and*
   - *if $p_i > q_i$, then $D_i(p) = \mathsf{down}$.*

The first condition specifies that each i-slice must have a unique fixed point. Since the slice $(*, *, \ldots, *)$ is an i-slice, this implies that the full problem also has a unique fixpoint.

The second condition is a more technical condition. It tells us that if we have found the unique fixpoint p of the $(i+1)$-slice s', and if this point is not the unique fixpoint of the i-slice s, then the direction function $D_i(p)$ tells us which way to walk to find the unique fixpoint of s. This is a crucial property that we will use in our reduction from OPDC to UniqueEOPL, and in our algorithms for contraction maps.

In our two-dimensional example, the first condition requires that every slice $(*, x)$ has a unique fixpoint, and this corresponds to saying that for every fixed slice of the left/right dimension, there is a unique blue point that is zero. The second condition requires that, if we are at some blue zero, then the red direction function at that point tells us the direction of the overall fixpoint. It can be seen that our example satisfies both of these requirements.

Note that both properties only consider i-slices. In the continuous contraction map problem with an L_p norm distance metric, *every* slice has a unique fixpoint, and so one may expect a discrete version of contraction to share this property. The problem is that the second property is very difficult to prove. Indeed, when we reduce PL-Contraction to OPDC in Section 4.2, we must carefully choose the grid size to ensure that both the first and second properties hold. In fact, our choice of grid size for dimension i will depend on the grid size of dimension $i+1$, which is why our definition only considers i-slices.

The name *One-Permutation* Discrete Contraction was chosen to emphasize this fact. The i-slices correspond to restricting dimensions in order, starting with dimension d. Since the order of the dimensions is arbitrary, we could have chosen any permutation of the dimensions, but we must choose *one* of these permutations to define the problem.

The OPDC problem. The OPDC problem is as follows: given a discrete contraction map $\mathcal{D} = (D_i(p))_{i=1,\ldots,d}$, find the unique point p such that $D_i(p) = \mathsf{zero}$ for all i. Note that we cannot efficiently verify whether $\mathcal{D}$ is actually a one-permutation discrete contraction map.

So, the OPDC problem is a promise problem, and we will formulate a total variant of it that uses a set of violations to cover the cases where $\mathcal{D}$ fails to be a discrete contraction map.

Definition 14 (OPDC)**.** *Given a tuple $(k_1, k_2, \ldots, k_d)$ and circuits $(D_i(p))_{i=1,\ldots,d}$, where each circuit $D_i : P(k_1, k_2, \ldots, k_d) \to \{\mathsf{up}, \mathsf{down}, \mathsf{zero}\}$, find one of the following*

(O1) A point $p \in P$ such that $D_i(p) = \mathsf{zero}$ for all i.

(OV1) An i-slice s and two points $p, q \in P_s$ with $p \neq q$ such that $D_j(p) = D_j(q) = \mathsf{zero}$ for all $j \leq i$.

(OV2) An i-slice s and two points $p, q \in P_s$ such that

- *$D_j(p) = D_j(q) = \mathsf{zero}$ for all $j < i$,*
- *$p_i = q_i + 1$, and*
- *$D_i(p) = \mathsf{down}$ and $D_i(q) = \mathsf{up}$.*

(OV3) An i-slice s and a point $p \in P_s$ such that

- *$D_j(p) = D_j(q) = \mathsf{zero}$ for all $j < i$, and either*
- *$p_i = 0$ and $D_i(p) = \mathsf{down}$, or*
- *$p_i = k_i$ and $D_i(p) = \mathsf{up}$.*

Solution type (O1) encodes a fixpoint, which is the proper solution of the discrete contraction map, while solution types (OV1) through (OV3) encode violations of the discrete contraction map property.

Solution type (OV1) witnesses a violation of the first property of a discrete contraction map, namely that each i-slice should have a unique fixpoint. A solution of type (OV1) gives two different points p and q in the same i-slice that are both fixpoints of that slice.

Solutions of type (OV2) witness violations of the first and second properties of a discrete contraction map. In these solutions we have two points p and q that are both fixpoints of their respective $(i-1)$-slices and are directly adjacent in an i-slice s. If there is a fixpoint r of the slice s, then this witnesses a violation of the second property of a discrete contraction map, which states that $D_i(p)$ and $D_i(q)$ should both point towards r, and clearly one of them does not. On the other hand, if slice s has no fixpoint, then p and q also witness this fact, since the fixpoint should be in-between p and q, which is not possible.

Solutions of type (OV3) consist of a point p that is a fixpoint of its $(i-1)$-slice but $D_i(p)$ points outside the boundary of the grid. These are clear violations of the second property, since $D_i(p)$

should point towards the fixpoint of the i-slice containing p, but that fixpoint cannot be outside the grid.

It is perhaps not immediately obvious that OPDC is a total problem. Ultimately we will prove this fact in the next section by providing a promise-preserving reducing from OPDC to UniqueEOPL. This will give us a proof of totality, and will also prove that, if the discrete contraction map has no violations, then it does indeed have a unique solution.

3.1 One-Permutation Discrete Contraction is in UniqueEOPL

In this section, we will show that One-Permutation Discrete Contraction lies in UniqueEOPL under promise-preserving reductions.

UFEOPL. Our reduction will make use of an intermediate problem that we call *unique forward EOPL*, which is a version of UniqueEOPL in which we only have a successor circuit S, meaning that no predecessor circuit P is given.

Definition 15 (UniqueForwardEOPL). *Given a Boolean circuits $S : \{0,1\}^n \to \{0,1\}^n$ such that $S(0^n) \neq 0^n$ and a Boolean circuit $V : \{0,1\}^n \to \{0,1,\ldots,2^m-1\}$ such that $V(0^n) = 0$ find one of the following:*

(UF1) A point $x \in \{0,1\}^n$ such that $S(x) \neq x$ and either $S(S(x)) = S(x)$ or $V(S(x)) \leq V(x)$.

(UFV1) Two points $x, y \in \{0,1\}^n$, such that $x \neq y$, $x \neq S(x)$, $y \neq S(y)$, and either $V(x) = V(y)$ or $V(x) < V(y) < V(S(x))$.

Without the predecessor circuit, this problem bears more resemblance to SinkOfDag than to EndOfPotentialLine. As in SinkOfDag, a bit-string x encodes a vertex if and only if $S(x) \neq x$, and an edge exists between vertices x and y if and only if $S(x) = y$ and $V(x) < V(y)$. The proper solution type (UF1) asks us to find a vertex that is a sink of the DAG, just as before.

The difference lies in the violation solution type (UFV1), which is the same as violation type (UV3) of UniqueEOPL. It asks for two vertices x and y that either have the same potential, or for which the potential of y lies strictly between the potential of x and the potential of $S(x)$. Note that this restriction severely constrains a SinkOfDag instance: if there are no violation solutions, then the DAG must consist of a single line that starts at 0^n, and ends at the unique solution of type (UF1). So in this sense, the problem really does capture instances of UniqueEOPL that lack a predecessor circuit.

The UniqueForwardEOPL problem will play a crucial role in our reduction. We will reduce OPDC to it, and we will then reduce it to UniqueEOPL.

An illustration of the reduction. Before we discuss the formal definition of the construction, we first give some intuition by describing the reduction for the two-dimensional example shown in Figure 3.

The reduction uses the notion of a *surface*. On the left side in Figure 4, we have overlaid the surfaces of the two direction functions from Figure 3. The surface of a direction function D_i is exactly the set of points $p \in P$ such that $D_i(p) = \mathsf{zero}$. The fixpoint p that we seek has $D_i(p) = \mathsf{zero}$ for all dimensions i, and so it lies at the intersection of these surfaces.

To reach the overall fixpoint, we walk along a path starting from the bottom-left corner, which is shown on the right-hand side of Figure 4. The path begins by walking upwards until it finds the blue surface. Once it has found the blue surface, it then there are two possibilities: either we have

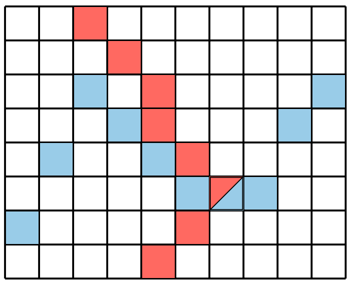
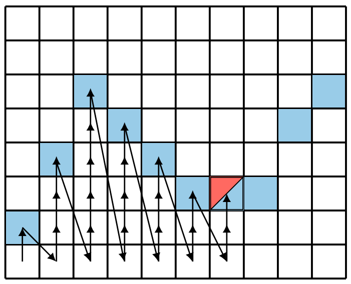



Figure 4: Left: The red and blue surfaces. Right: the path that we follow. This figure should be viewed in color.

found the overall fixpoint, in which case the line ends, or we have not found the overall fixpoint, and the red direction function tells us that the direction of the overall fixpoint is to the right.

If we have not found the overall fixpoint, then we move one step to the right, go back to the bottom of the diagram, and start walking upwards again. We keep repeating this until we find the overall fixpoint. This procedure gives us the line shown on the right-hand side of Figure 4.

The potential. How do we define a potential for this line? Observe that the dimension-two coordinates of the points on the line are weakly monotone, meaning that the line never moves to the left. Furthermore, for any dimension-two slice (meaning any slice in which the left/right coordinate is fixed), the dimension-one coordinate is monotonically increasing. So, if $p = (p_1, p_2)$ denotes any point on the line, if k denotes the maximum coordinate in either dimension, then the function

$$V(p_1, p_2) = k \cdot p_2 + p_1$$

is a function that monotonically increases along the line, which we can use as a potential function.

Uniqueness. To provide a promise-preserving reduction to UFEOPL, we must argue that the line is unique whenever the OPDC instance has no violations. Here we must carefully define what exactly a vertex on the line actually is, to ensure that no other line can exist. Specifically, we must be careful that only points that are to the left of the fixpoint are actually on the line, and that no "false" line exists to the right of the fixpoint.

Here we rely on the following fact: if the line visits a point with coordinate x in dimension 2, then it must have visited the point p on the blue surface in the slice defined by $x - 1$. Moreover, for that point p we must have $D_2(p) = \mathsf{up}$, which means that it is to the left of the overall fixpoint.

Using this fact, each vertex on our line will be a pair (p, q), where p is the current point that we are visiting, and q is either

- the symbol $-$, indicating that we are still in the first column of points, and we have never visited a point on the blue surface, or
- a point q that is on the blue surface, that satisfies $q_2 = p_2 - 1$ and $D_2(q) = \mathsf{up}$.

Hence the point q is always the last point that we visited on the blue surface, which provides a witness that we have not yet walked past the overall fixpoint.

When we finish walking up a column of points, and find the point on the blue surface, we overwrite q with the new point that we have found. This step is the reason why only a successor circuit can be given for the line, since the value that is overwritten cannot easily be computed by a predecessor circuit.

Violations. Our two-dimensional OPDC example does not contain any violations, but our reduction can still handle all possible violations in the OPDC instance. At a high level, there are two possible ways in which the reduction can go wrong if there are violations.

1. It is possible, that as we walk upwards in some column, we do not find a fixpoint, and our line will get stuck. This creates an end of line solution of type (UF1), which must be mapped back to an OPDC violation. In our two-dimensional example, this case corresponds to a column of points in which there is no point on the blue surface. However, if there is no point on the blue surface, then we will either

   - find two adjacent points p and q in that column with $D_1(p) = \mathsf{up}$ and $D_2(p) = \mathsf{down}$, which is a solution of type (OV2), or
   - find a point p at the top of the column with $D_1(p) = \mathsf{up}$, or a point q at the bottom of the column with $D_1(q) = \mathsf{down}$. Both of these are solutions of type (OV3).

   There is also the similar case where we walk all the way to the right without finding an overall fixpoint, in which case we will find a point p on the right-hand boundary that satisfies $D_1(p) = \mathsf{zero}$ and $D_2(p) = \mathsf{up}$, which is a solution of type (OV3).

2. The other possibility is that there may be more than one point on the blue surface in some of the columns. This will inevitably lead to multiple lines, since if q and q' are both points on the blue surface in some column, and p is some point in the column to the right of p and q, then (p, q) and (p, q') will both be valid vertices on two different lines.

   These can show up as violations of type (UFV1), which we map back to solutions of type (OV1). Specifically, the points p and q, which are given as part of the two vertices, are both fixpoints of the same slice, which is exactly what (OV1) asks for.

We can argue that our reduction is promise-preserving. This is because violation solutions in the UFEOPL instance are never mapped back to proper solutions of the OPDC instance. This means that, if we promise that the OPDC instance has no violations, then the resulting UFEOPL instance must also contain no violations.

The full reduction. Our reduction from OPDC to UniqueForwardEOPL generalizes the approach given above to d dimensions. We say that a point $p \in P$ is on the i*-surface* if $D_j(p) = \mathsf{zero}$ for all $j \leq i$. In our two-dimensional example we followed a line of points on the one-surface, in order to find a point on the two-surface. In between any two points on the one-surface, we followed a line of points on the zero-surface (every point is trivially on the zero-surface).

Our line will visit a sequence of points on the $(d-1)$-surface in order to find the point on the d-surface, which is the fixpoint. Between any two points on the $(d-1)$-surface the line visits a sequence of points on the $(d-2)$-surface, between any two points on the $(d-2)$-surface the line visits a sequence of points on the $(d-3)$-surface, and so on.

The line will follow the same pattern that we laid out in two dimensions. Every time we find a point on the i-surface, we remember it, increment our position in dimension i by 1, and reset our coordinates back to 0 for all dimensions $j < i$. Hence, a vertex will be a tuple $(p_0, p_1, \ldots, p_d)$, where each p_i is either

- the symbol $-$, indicating that we have not yet encountered a point on the i-surface, or
- the most recent point on the i-surface that we have visited.

This is a generalization of the witnessing scheme that we saw in two-dimensions.

The potential is likewise generalized so that the potential of a point p is proportional to $\sum_{i=1}^{d} k^i p_i$, where again k is some constant that is larger than the grid size. This means that progress in dimension i dominates progress in dimension j whenever $j < i$, which allows the potential to monotonically increase along the line.

We are also able to deal with all possible violations, using the ideas that we have described in two-dimensional case. Full details of this construction are given in Appendix B.1, where the following lemma is proved.

Lemma 16. *There is a polynomial-time promise-preserving reduction from* OPDC *to* UniqueForwardEOPL*.*

From UniqueForwardEOPL to UniqueForwardEOPL+1. The next step of the reduction is to slightly modify the UniqueForwardEOPL instance, so that the potential increases by exactly one in each step. Specifically we define the following problem

Definition 17 (UniqueForwardEOPL+1)**.** *Given a Boolean circuits $S : \{0,1\}^n \to \{0,1\}^n$ such that $S(0^n) \neq 0^n$ and a Boolean circuit $V : \{0,1\}^n \to \{0, 1, \ldots, 2^m - 1\}$ such that $V(0^n) = 0$ find one of the following:*

(UFP1) A point $x \in \{0,1\}^n$ such that $S(x) \neq x$ and either $S(S(x)) = S(x)$ or $V(S(x)) \neq V(x) + 1$.

(UFPV1) Two points $x, y \in \{0,1\}^n$, such that $x \neq y$, $x \neq S(x)$, $y \neq S(y)$, and $V(x) = V(y)$.

There are two differences between this problem and UniqueForwardEOPL. Firstly, an edge exists between x and y if and only if $S(x) = y$, and $V(y) = V(x) + 1$, and this is reflected in the modified definition of solution type (UFP1). Secondly, solution type (UFPV1) has been modified to only cover the case where we have two vertices x and y that have the same potential. The case where $V(x) < V(y) < V(S(x))$ is not covered, since in this setting this would imply $V(S(x)) > V(x) + 1$, which already gives us a solution of type (UFP1).

It is not difficult to reduce UniqueForwardEOPL to UniqueForwardEOPL+1, using the same techniques that we used in the reduction from EndOfPotentialLine to EndOfMeteredLine in Theorem 10. This gives us the following lemma, which is proved in Appendix B.2.

Lemma 18. *There is a polynomial-time promise-preserving reduction from* UniqueForwardEOPL *to* UniqueForwardEOPL+1*.*

UniqueForwardEOPL+1 to UniqueEOPL. The final step of the proof is to reduce UniqueForwardEOPL to UniqueEOPL. For this, we are able to build upon existing work. The following problem was introduced by Bitansky et al [4].

Definition 19 (SinkOfVerifiableLine [4])**.** *The input to the problem consists of a starting vertex $x_s \in \{0,1\}^n$, a target integer $T \leq 2^n$, and two boolean circuits $S : \{0,1\}^n \to \{0,1\}^n$, $W : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$. It is promised that, for every vertex $x \in \{0,1\}^n$, and every integer $i \leq T$, we have $W(x,i) = 1$ if and only if $x = S^{i-1}(x_s)$. The goal is to find the vertex $x_f \in \{0,1\}^n$ such that $W(x_f, T) = 1$.*

SinkOfVerifiableLine is intuitively very similar to UniqueForwardEOPL. In this problem, a single line is encoded, where as usual the vertices are encoded as bit-strings, and the circuit S gives the successor of each vertex. The difference in this problem is that the circuit W gives a way of verifying how far along the line a given vertex is. Specifically, $W(x, i) = 1$ if and only if x is the ith vertex on the line. Note that this is inherently a promise problem, since if $W(x, i) = 1$ for some i, we have no way of knowing whether x is *actually* i steps along the line, without walking all of those steps ourselves.

It was shown by Hubáček and Yogev [45] that SinkOfVerifiableLine can be reduced in polynomial time to EndOfMeteredLine, and hence also to EndOfPotentialLine (via Theorem 10). Moreover, the resulting EndOfPotentialLine instance has a unique line, so this reduction also reduces SinkOfVerifiableLine to UniqueEOPL. It is easy to reduce the promise version of UniqueForwardEOPL+1 to SinkOfVerifiableLine, since we can implement the circuit W so that $W(x, i) = 1$ if and only if $V(x) = i$.

However, the existing work only deals with the promise problem. Our contribution is to deal with violations. We show that, if one creates a SinkOfVerifiableLine instance from a UniqueForwardEOPL+1 instance, in the way described above, and applies the reduction of Hubáček and Yogev to produce a UniqueEOPL instance, then any violation can be mapped back to a solution in the original UniqueForwardEOPL+1 instance. Hence, we show the following lemma, whose full proof appears in Appendix B.3.

Lemma 20. *There is a polynomial-time promise-preserving reduction from* UniqueForwardEOPL *to* UniqueEOPL.

This completes the chain of promise-preserving reductions from OPDC to UniqueEOPL. Hence, we have shown the following theorem.

Theorem 21. OPDC *is in* UniqueEOPL *under polynomial-time promise-preserving reductions.*

3.2 One-Permutation Discrete Contraction is UniqueEOPL-hard

In this section we will show that One-Permutation Discrete Contraction is UniqueEOPL-complete, by giving a hardness result. Specifically, we give a reduction from UniqueEOPL to OPDC.

Modifying the line. The first step of the reduction is to slightly alter the UniqueEOPL instance. Specifically, we would like to ensure the following two properties.

1. Every edge increases the potential by exactly one. That is, $V(S(x)) = V(x) + 1$ for every vertex x.
2. The line has length exactly 2^n for some integer n. More specifically, we ensure that if x is the end of any line then we have $V(x) = 2^n - 1$. The start of the line given in the problem has potential 0, this ensures that the length of that line is exactly 2^n, although other lines may be shorter.

We have already developed a technique for ensuring the first property in the reduction from EOPL to EOML in Theorem 10, which can be reused here. Specifically, we introduce a chain of dummy vertices between any pair of vertices x and y with $S(x) = y$ and $V(y) > V(x) + 1$. The second property can be ensured by choosing n so that 2^n is larger than the longest possible line in the instance. Then, at every vertex x that is the end of a line, we introduce a chain of dummy vertices

$$(x, 0) \to (x, 1) \to \cdots \to (x, 2^n - V(x) - 1),$$

where $V(x, i) = V(x) + i$. The vertex $e = (x, 2^n - V(x) - 1)$ will be the new end of the line, and note that $V(e) = 2^n - 1$ as required. The full details of this are given in Appendix C.1, where the following lemma is shown.

Lemma 22. *Given a* UniqueEOPL *instance $L = (S, P, V)$, there is a polynomial-time promise-preserving reduction that produces a* UniqueEOPL *instance $L' = (S', P', V')$, where*

- *For every x and y with $y = S(x)$ and $x = P(y)$ we have $V(y) = V(x) + 1$, and*
- *There exists an integer n such that, x is a solution of type (U1) if and only if we have $V(x) = 2^n - 1$.*

For the remainder of this section, we will assume that we have a UniqueEOPL instance $L = (S, P, V)$ that satisfies that two extra conditions given by Lemma 22. We will use m to denote the bit-length of a vertex in L.

The set of points. We create an OPDC instance over a boolean hypercube with $m \cdot n$ dimensions, so our set of points is $P = \{0, 1\}^{mn}$. We will interpret each point $p \in P$ as a tuple $(v_1, v_2, \ldots, v_n)$, where each v_i is a bit-string of length m, meaning that each v_i can represent a vertex in L.

To understand the reduction, it helps to consider the case where there is a unique line in L. We know that this line has length exactly 2^n. The reduction repeatedly splits this line into two equal parts.

- Let L_1 denote the first half of L, which contains all vertices v with potential $0 \leq V(v) \leq 2^{n-1} - 1$.
- Let L_2 denote the second half of L, which contains all vertices v with potential $2^{n-1} \leq V(v) \leq 2^n - 1$.

Observe that L_1 and L_2 both contain exactly 2^{n-1} vertices.

The idea is to embed L_1 and L_2 into different sub-cubes of the $\{0, 1\}^{mn}$ point space. The line that we embed will be determined by the *last* element of the tuple. Let v be the vertex satisfying $V(v) = 2^{n-1}$, meaning v is the first element of L_2.

- We embed L_2 into the sub-cube $(*, *, \ldots, *, v)$.
- We embed a copy of L_1 into each sub-cube $(*, *, \ldots, *, u)$ with $u \neq v$.

Note that this means that we embed a single copy of L_2, but many copies of L_1. Specifically, there are 2^m possibilities for the final element of the tuple. One of these corresponds to the sub-cube containing L_2, while $2^m - 1$ of them contain a copy of L_1.

The construction is recursive. So we split L_2 into two lines $L_{2,1}$ and $L_{2,2}$, each containing half of the vertices of L_2. If w is the vertex satisfying $V(w) = 2^{n-1} + 2^{n-2}$, which is the first vertex of $L_{2,2}$, then we embed a copy of $L_{2,2}$ into the sub-cube $(*, *, \ldots, *, w, v)$, where v is the same vertex that we used above, and we embed a copy of $L_{2,1}$ into each sub-cube $(*, *, \ldots, *, u, v)$, where $u \neq w$. Likewise L_1 is split into two, and embedded into the sub-cubes of $(*, *, \ldots, *, u)$ whenever $u \neq v$.

Given a vertex $(v_1, v_2, \ldots, v_n)$, we can view the bit-string v_n as choosing either L_1 or L_2, based on whether $V(v_n) = 2^{n-1}$. Once that decision has been made, we can then view v_{n-1} as choosing one half of the remaining line. Since the original line L has length 2^n, and we repeat this process n times, this means that at the end of the process we will be left with a line containing a single vertex. So in this way, a point $(v_1, v_2, \ldots, v_n)$ is a representation of some vertex in L, specifically the vertex that is left after we repeatedly split the line according to the choices made by v_n through v_1.

We can compute the vertex represented by any tuple in polynomial time. Moreover, given a slice $(*, *, \ldots, *, v_i, v_{i+1}, \ldots, v_n)$ that fixes elements i through n of the tuple, we can produce, in polynomial time, a UniqueEOPL instance corresponding to the line that is embedded in that slice. This is formalised in the following lemma, whose proof is given in Appendix C.2.

Lemma 23. *There are polynomial-time algorithms for computing the following two functions.*

- *The function* $\text{decode}(v_1, v_2, \ldots, v_n)$ *which takes a point in P and returns the corresponding vertex of L.*
- *The function* $\text{subline}(v_i, v_{i+1}, \ldots, v_n, L)$*, which takes bit-strings v_i through v_n, representing the slice, $(*, *, \ldots, *, v_i, v_{i+1}, \ldots, v_n)$, and the instance $L = (S, P, V)$, and returns a new* UniqueEOPL *instance $L' = (S', P', V')$ which represents the instance that is to be embedded into this slice.*

Although we have described this construction in terms of a single line, the two polynomial time algorithms given by Lemma 23 are capable of working with instances that contain multiple lines. In the case where there are multiple lines, there may be two or more bit-strings x and y with $V(x) = V(y) = 2^{n-1}$. In that case, we will embed second-half instances into $(*, *, \ldots, x)$ and $(*, *, \ldots, y)$. This is not a problem for the functions decode and subline, although this may lead to violations in the resulting OPDC instance that we will need to deal with.

The direction functions. The direction functions will carry out the embedding of the lines. Since our space is $\{0, 1\}^{mn}$, we will need to define $m \cdot n$ direction functions D_1 through D_{mn}.

The direction functions $D_{m(n-1)+1}$ through D_{mn} correspond to the bits used to define v_n in a point $(v_1, v_2, \ldots, v_n)$. These direction functions are used to implement the transition between the first and second half of the line. For each point $p = (v_1, v_2, \ldots, v_n)$ we define these functions using the following algorithm.

1. In the case where $V(v_n) \neq 2^{n-1}$, meaning that $\text{decode}(p)$ is a vertex in the first half of the line, then there are two possibilities.

   (a) If $V(\text{decode}(p)) = 2^{n-1} - 1$, meaning that p is the last vertex on the first half of the line, then we orient the direction function of dimensions $(n-1) \cdot m$ through m towards the bit-string given by $S(\text{decode}(p))$. This captures the idea that once we reach the end of the first half of the line, we should then move to the second half, and we do this by moving towards the sub-cube $(*, *, \ldots, *, S(\text{decode}(p)))$. So for each i in the range $m(n-1) + 1 \leq i \leq mn$ we define

   $$D_i(p) = \begin{cases} \mathsf{up} & \text{if } p_i = 0 \text{ and } S(\text{decode}(p))_i = 1, \\ \mathsf{down} & \text{if } p_i = 1 \text{ and } S(\text{decode}(p))_i = 0, \\ \mathsf{zero} & \text{otherwise.} \end{cases}$$

   (b) If the rule above does not apply, then we orient everything towards 0. Specifically, we set

   $$D_i(p) = \begin{cases} \mathsf{down} & \text{if } p_i = 1, \\ \mathsf{zero} & \text{if } p_i = 0. \end{cases}$$

   This is an arbitrary choice: our reduction would work with any valid direction rules in this case.

2. If $V(v_n) = 2^{n-1}$, then we are in the second half of the line. In this case we set $D_i(p) = \mathsf{zero}$ for all dimensions i in the range $(n-1) \cdot m \leq i \leq m$. This captures the idea that, once we have entered the second half of the line, we should never leave it again.

We use the same idea recursively to define direction functions for all dimensions $i \leq m(n-1)$. This gives us a family of polynomial-time computable direction functions $\mathcal{D} = (D_i)_{i=1,\ldots,mn}$. The full details can be found in Appendix C.3.

The proof. We have now defined P and $\mathcal{D}$, so we have an OPDC instance. We must now argue that the reduction is correct. The intuitive idea is as follows. If we are at some point $p \in P$, and $\text{decode}(p) = v$ is a vertex that is not the end of a line, then there is some direction function D_i such that $D_i(p) \neq \mathsf{zero}$. We can see this directly for the case where $V(\text{decode}(p)) = 2^{n-1} - 1$, since the direction functions on dimensions $m(n-1)+1$ through mn will be oriented towards $S(V(\text{decode}(p))$, and so p will not be a solution.

As we show in the proof, the same property holds for all other vertices in the middle of the line. The end of the line will be a solution, because it will be encoded by the point $p = (v_1, v_2, \ldots, v_n)$, where each v_i is the first vertex on the second half of the line embedded in the sub-cube. Our direction functions ensure that $D_j(p) = \mathsf{zero}$ for all j for this point.

Our reduction must also deal with violations. Violations of type (OV3) are impossible by construction. In violations of type (OV1) and (OV2) we have two points p and q that are in the same i-slice. Here we specifically use the fact that violations can only occur within i-slices. Note that an i-slice will fix the last $mn - i$ bits of the tuple $(v_1, v_2, \ldots, v_n)$, which means that there will be an index j such that all v_l with $l > j$ are fixed. This allows us to associate the slice with the line $L' = \text{subline}(v_{j+1}, v_{j+2}, \ldots, v_n)$, and we know that both p and q encode vertices of L'. In both cases, we are able to recover two vertices in L' that have the same potential, and these vertices also have the same potential in L. So we get a solution of type (UV3). The details are rather involved, and we defer the proof to Appendix C.4, where the following lemma is proved.

Lemma 24. *There is a polynomial-time promise-preserving reduction from* UniqueEOPL *to* OPDC.

Thus, we have shown the following theorem.

Theorem 25. OPDC *is* $\mathsf{UniqueEOPL}$*-complete under promise-preserving reductions, even when the set of points P is a hypercube.*

Since we have shown bidirectional promise-preserving reductions between OPDC and UniqueEOPL, we also get that the promise version of OPDC is complete for $\mathsf{PromiseUEOPL}$.

4 UniqueEOPL containment results

4.1 Unique Sink Orientations

Unique sink orientations. Let $C = \{0, 1\}^n$ be an n-dimensional hypercube. An *orientation* of C gives a direction to each edge of C. We formalise this as a function $\Psi : C \to \{0, 1\}^n$, that assigns a bit-string to each vertex of C, with the interpretation that the i-th bit of the string gives an orientation of the edge in dimension i. More precisely, for each vertex $v \in C$ and each dimension i, let u be the vertex that is adjacent to v in dimension i.

- If $\Psi(v)_i = 0$ then the edge between v and u is oriented towards v.
- If $\Psi(v)_i = 1$ then the edge between v and u is oriented towards u.

Note that this definition does not insist that v and u agree on the orientation of the edge between them, meaning that $\Psi(v)_i$ and $\Psi(u)_i$ may orient the edge in opposite directions. However, this will be a violation in our set up, and a proper orientation should be thought of as always assigning a consistent direction to each edge.

A *face* is a subset of C in which some coordinates have been fixed. This can be defined using the same notation that we used for slices in OPDC. So a face $f = (f_1, f_2, \dots, f_n)$, where each f_i is either 0, 1, or $*$, and the sub-cube defined by f contains every vertex $v \in C$ such that $v_i = f_i$ whenever $f_i \neq *$.

A vertex $v \in C$ is a *sink* if all of the edges of v are directed towards v, meaning that $\Psi(v)_i = 0$ for all i. Given a face f, a vertex v is a *sink of f* if it is the sink of the sub-cube defined by f, meaning that $\Psi(v)_i = 0$ whenever $f_i = *$.

A *unique sink orientation* (USO) is an orientation in which *every* face has a unique sink. Since $f = (*, *, \dots, *)$ is a face, this also implies that the whole cube has a unique sink, and the USO problem is to find the unique sink.

Placing the problem in TFNP. The USO property is quite restrictive, and there are many orientations that are not USOs. Indeed, the overall cube may not have a sink at all, or it may have multiple sinks, and this may also be the case for the other faces of the cube. Fortunately, Szabó and Welzl have pointed out that if an orientation is not a USO, then there is a succinct witness of this fact [74].

Let $v, u \in C$ be two distinct vertices. We have that Ψ is a USO if and only if there exists some dimension i such that $v_i \neq u_i$ and $\Psi(v)_i \neq \Psi(u)_i$. Put another way, this means that if we restrict the orientation only to the sub-cube defined by any two vertices v and u, then the orientations of v and u must be different on that sub-cube. If one uses $\oplus$ to denote the XOR operation on binary strings, and $\cap$ to denote the bit-wise and-operation, then this condition can be written concisely as

$$(v \oplus u) \cap (\Psi(v) \oplus \Psi(u)) \neq 0^n.$$

Note that this condition also ensures that the orientation is consistent, since if v and u differ in only a single dimension, then the conditions states that they must agree on the orientation of the edge between them.

We use this condition to formulate the USO problem as a problem in TFNP.

Definition 26 (Unique-Sink-Orientation). *Given an orientation function $\Psi : \{0,1\}^n \to \{0,1\}^n \cup \{-\}$ find one of the following.*

(US1) A point $v \in \{0,1\}^n$ such that $\Psi(v)_i = 0$ for all i.

(USV1) A point $v \in \{0,1\}^n$ such that $\Psi(v) = -$.

(USV2) Two points $v, u \in \{0,1\}^n$ such that $v \neq u$ and $(v \oplus u) \cap (\Psi(v) \oplus \Psi(u)) = 0^n$.

Note that this formulation of the problem allows the orientation function to decline to give an orientation for some vertices, and this is indicated by setting $\Psi(v) = -$. Any such vertex is a violation of type (USV1). While this adds nothing interesting to the USO problem, we will use this in Section 4.3 when we reduce P-LCP *to* USO, since in some cases the reduction may not be able to produce an orientation at a particular vertex.

Assuming that there are no violations of type (USV1), it is easy to see that the problem is total. This is because every USO has a sink, giving a solution of type (US1), while every orientation that is not a USO has a violation of type (USV2).

Placing the problem in UniqueEOPL. We show that the problem lies in UniqueEOPL by providing a promise-preserving reduction from UNIQUE-SINK-ORIENTATION to OPDC. The reduction is actually not too difficult, because when the point set for the OPDC instance is a hypercube, the OPDC problem can be viewed as a less restrictive variant of USO. Specifically, USO demands that *every* face has a unique sink, while OPDC only requires that the i-slices should have unique sinks.

The reduction creates an OPDC instances on the same set of points, meaning that $P = C$. The direction functions simply follow the orientation given by Ψ. Specifically, for each $v \in P$ and each dimension i we define

$$D_i(v) = \begin{cases} \mathsf{zero} & \text{if } \Psi(v)_i = 0, \\ \mathsf{up} & \text{if } \Psi(v)_i = 1 \text{ and } v_i = 0, \\ \mathsf{down} & \text{if } \Psi(v)_i = 1 \text{ and } v_i = 1. \end{cases}$$

If $\Psi(v) = -$, then we instead set $D_i(v) = \mathsf{zero}$ for all i.

To prove that this is correct, we show that every solution of the OPDC instance can be mapped back to a solution of the USO instance. Any fixpoint of the OPDC instance satisfies $D_i(v) = \mathsf{zero}$ for all i, which can only occur if v is a sink, or if $\Psi(v) = -$. The violation solutions of OPDC can be used to generate a pair of vertices that constitute a (USV2) violation. We defer the details to Appendix D, where the following lemma is proved.

Lemma 27. *There is a polynomial-time promise-preserving reduction from* UNIQUE-SINK-ORIENTATION *to* OPDC.

Thus we have shown the following theorem.

Theorem 28. UNIQUE-SINK-ORIENTATION *is in* UniqueEOPL *under promise-preserving reductions.*

This proves the following new facts about the complexity of finding the sink of a USO.

Corollary 29. UNIQUE-SINK-ORIENTATION *is in* PPAD, PLS, *and* CLS.

4.2 Piecewise Linear Contraction Maps

In this section, we show that finding a fixpoint of a *piecewise linear* contraction map lies in UniqueEOPL. Specifically, we study contraction maps where the function f is given as a LinearFIXP *circuit*, which is an arithmetic circuit comprised of $\max, \min, +, -$, and $\times\zeta$ (multiplication by a constant) gates [22]. Hence, a LinearFIXP circuit defines a *piecewise linear* function.

Violations. Not every function f is contracting, and the most obvious way to prove that f is not contracting is to give a pair of points x and y that satisfy $\|f(x) - f(y)\|_p > c \cdot \|x - y\|_p$, which directly witness the fact that f is not contracting.

However, when we discretize Contraction in order to to reduce it to OPDC, there are certain situations in which we have a convincing proof that f is not contracting, but no apparent way to actually produce a violation of contraction. In fact, the discretization itself is non-trivial, so we will explain that first, and then define the type of violations that we will use.

The reduction. We are given a function $f : [0, 1]^n \to [0, 1]^n$, that is purported to be contracting with contraction factor c in the ℓ_p norm. We will produce an OPDC instance by constructing the point set P, and a family of direction functions $\mathcal{D}$.

The most complex step of the reduction is to produce an appropriate set of points P for the OPDC instance. This means we need to choose integers k_1, k_2, through k_d in order to define the point set $P(k_1, k_2, \ldots, k_d)$, where we recall that this defines a grid of integers, where each dimension i can take values between 0 and k_i. We will describe the method for picking k_1 through k_d after we have specified the rest of the reduction.

The direction functions will simply follow the directions given by f. Specifically, for every point $p \in P(k_1, k_2, \ldots, k_d)$, let p' be the corresponding point in $[0, 1]^n$, meaning that $p'_i = p_i/k_i$ for all i. For and every dimension i we define the direction function D_i so that

- if $f(p')_i > p'_i$ then $D_i(p) = \mathsf{up}$,
- if $f(p')_i < p'_i$ then $D_i(p) = \mathsf{down}$, and
- if $f(p')_i = p'_i$ then $D_i(p) = \mathsf{zero}$.

In other words, the function D_i simply checks whether $f(p')$ moves up, down, or not at all in dimension i. This completes the specification of the family of direction functions $\mathcal{D}$.

We must carefully choose k_1 through k_d to ensure that the fixpoint of f is contained within the grid. In fact, we need a stronger property: for every i-slice of the grid, if f has a fixpoint in that i-slice, then it should also appear in the grid. Recall that $p \in P$ is a fixpoint of some slice s if $D_i(p) = \mathsf{zero}$ for every i for which $s_i = *$. We can extend this definition to the continuous function f as follows: a point $x \in [0, 1]^d$ is a fixpoint of s if $(x - f(x))_i = 0$ for all i for which $s_i = *$, where we now interpret s as specifying that $x_i = s_i/k_i$ whenever $s_i \neq *$. We are able to show the following lemma, whose proof appears in Appendix F.1.

Lemma 30. *There exists integers $(k_1, k_2, \ldots, k_d)$ such that for every i-slice s we have that if $x \in [0, 1]^d$ is a fixpoint of s according to f, then there exists a point $p \in P(k_1, k_2, \ldots, k_d)$ such that*

- *p is a fixpoint of $\mathcal{D}$, and*
- *$p_i = k_i \cdot x_i$ for all i where $s_i = *$.*

Moreover, the number of bits needed to write down each k_i is polynomial in the number of bits needed to write down f.

This lemma states that we can pick the grid size to be fine enough so that all fixpoints of f in all i-slices are contained within the grid. The proof of this is actually quite involved, and relies crucially on the fact that we have access to a LinearFIXP representation of f. From this, we can compute upper bounds on the bit-length of any point that is a fixpoint of f. We also rely on the fact that we only need to consider i-slices, because our proof fixes the grid-widths one dimension at a time, starting with dimension d and working backwards.

The extra violation. The specification of our reduction is now complete, but we have still not fully defined the original problem, because we need to add an extra violation. The issue arises with solutions of type (OV2), where we have an i-slice s and two points p, q in s such that

- $D_j(p) = D_j(q) = \mathsf{zero}$ for all $j < i$,
- $p_i = q_i + 1$, and
- $D_i(p) = \mathsf{down}$ and $D_i(q) = \mathsf{up}$.

This means that p and q are both fixpoints of their respective slices $(i-1)$-slices, and are directly adjacent to each other in dimension i. We are able to show, in this situation, that if f is contracting, then f has a fixpoint for the slice s, and it must lie between p and q. The following lemma is shown in Appendix F.2.

Lemma 31. *If f is contracting, and we have two points p and q that are a violation of type (OV2), then there exists a point $x \in [0,1]^n$ in the slice s that satisfies all of the following.*

- *$(x - f(x))_j = 0$ for all $j \leq i$, meaning that x is a fixpoint of the slice s, and*
- *$q_i < k_i \cdot x_i < p_i$, meaning that x lies between p and q in dimension i.*

So if we have an (OV2) violation, and if f is contracting, then Lemma 31 implies that there is a fixpoint x of the slice s that lies strictly between p and q in dimension i. However, Lemma 30 says that all fixpoints of s lie in the grid, and since p and q are directly adjacent adjacent in the grid in dimension i, there is no room for x, so it cannot exist. The only way that this contradiction can be resolved is if f not actually contracting.

Hence, (OV2) violations give us a concise witness that f is not contracting. But the points p and q themselves may satisfy the contraction property. While we know that there must be a violation of contraction *somewhere*, we are not necessarily able to compute such a violation in polynomial time. To resolve this, we add the analogue of an (OV2) violation to the contraction problem.

Definition 32 (PL-CONTRACTION)**.** *Given a* LinearFIXP *circuit computing $f : [0,1]^d \to [0,1]^d$, a constant $c \in (0,1)$, and a $p \in \mathbb{N} \cup \{\infty\}$, find one of the following.*

(CM1) A point $x \in [0,1]^d$ such that $f(x) = x$.

(CMV1) Two points $x, y \in [0,1]^d$ such that $\|f(x) - f(y)\|_p > c \cdot \|x - y\|_p$.

(CMV2) A point $x \in [0,1]^d$ such that $f(x) \notin [0,1]^d$.

(CMV3) An i-slice s and two points $x, y \in [0,1]^d$ in s such that

- *$(f(x) - x)_j = (f(y) - y)_j = 0$ for all $j < i$,*
- *$k_i \cdot x_i = k_i \cdot y_i + 1$, where k_i is the integer given by Lemma 30 for the* LinearFIXP *circuit that computes f, and*
- *$f(x)_i < x_i$ and $f(y)_i > y_i$.*

Solution type (CM1) asks us to find a fixpoint of the map f, and there are two types of violation. Violation type (CMV1) asks us to find two points x and y that prove that f is not contracting with respect to the ℓ_p norm. Violation type (CMV2) asks us to find a point that f does not map to $[0,1]^d$. Note that this second type of violation is necessary to make the problem total, because it is possible that f is a contraction map, but the unique fixpoint of f does not lie in $[0,1]^d$.

Violations of type (CMV3) are the direct translation of (OV2) violations to contraction. Note that if f actually is contracting, and has a fixpoint in $[0,1]^n$, then no violations can exist. For (CMV3) violations, this fact is a consequence of Lemmas 30 and 31.

Correctness of the reduction. To prove that the reduction is correct, we must show that all solutions of the OPDC instance given by P and $\mathcal{D}$ can be mapped back to solutions of the original instance. Solutions of type (O1) give us a point p such that $D_i(p) =$ zero for all i, which by definition means that the point corresponding to p is a fixpoint of f. Violations of type (OV1) give us two points that are both fixpoints of the same slice s, which also means that they are both fixpoints of the slice s according to f, and it is not difficult to show that these two points violate contraction in an ℓ_p norm. Violations of type (OV3) are points that attempt to leave the $[0,1]^d$, and so give us a solution of type (CMV2). Violations of type (OV2) map directly to violations of type (CMV3), as we have discussed. So we have the following lemma, which is proved in Appendix F.3.

Lemma 33. *There is a polynomial-time promise-preserving reduction from* PL-Contraction *to* OPDC*.*

Theorem 34. PL-Contraction *is in* UniqueEOPL *under promise-preserving reductions.*

4.3 The P-Matrix Linear Complementarity Problem

In this section, we reduce P-LCP to Unique-Sink-Orientation and, separately, P-LCP to UniqueEOPL. Given that we show that Unique-Sink-Orientation reduces to UniqueEOPL (via OPDC) and is thus in UniqueEOPL, our direct reduction from P-LCP to UniqueEOPL is not needed to show that P-LCP is contained in UniqueEOPL. However, by reducing directly we can produce a UniqueEOPL instance with size linear in the size of our P-LCP instance, which is needed to obtain the algorithmic result in Section 5.2.

The direct reduction to UniqueEOPL relies heavily on the application of Lemke's algorithm to P-matrix LCPs, and our reduction to Unique-Sink-Orientation relies on the computation of *principal pivot transformations* of LCPs. Next we introduce the required concepts. Let $[d]$ denote the set $\{1, \dots, d\}$.

Definition 35 (LCP $(M, \boldsymbol{q})$)**.** *Given a matrix $M \in \mathbb{R}^{d \times d}$ and vector $\boldsymbol{q} \in \mathbb{R}^{d \times 1}$, find a $\mathbf{y} \in \mathbb{R}^{d \times 1}$ s.t.:*

$$\mathbf{w} = M\mathbf{y} + \boldsymbol{q} \geq 0; \quad \mathbf{y} \geq 0; \quad y_i \cdot w_i = 0, \ \forall i \in [d]. \tag{1}$$

In general, deciding whether an LCP has a solution is NP-complete [12], but if M is a P-matrix, as defined next, then the LCP $(M, \boldsymbol{q})$ has a *unique* solution for all $\boldsymbol{q} \in \mathbb{R}^{d \times 1}$.

The problem of checking if a matrix is a P-matrix is coNP-complete [15], so we cannot expect to be able to verify that an LCP instance $(M, \boldsymbol{q})$ is actually defined by a P-matrix M. Instead, we use succinct witnesses that M is not a P-matrix as a violation solution, which allows us to define total variants of the P-LCP problem that lie in TFNP, as first done by Megiddo [57, 58]. This approach has previously used to place the P-matrix problem in PPAD and CLS [17, 64].

Our paper is about problems with unique solutions. It is well known that a matrix M is a P-matrix *if and only if* for all $\boldsymbol{q} \in \mathbb{R}^{d \times 1}$, the LCP $(M, \boldsymbol{q})$ has a unique solution [14]. However, this characterization of a P-matrix is not directly useful for defining succinct violations: while two distinct solutions would be a succinct violation, there is no corresponding succinct witness for the case of no solutions. Next we introduce the three well-known succinct witnesses for M not being a P-matrix that we will use.

First, we introduce some further required notation. Restating (1), the LCP problem $(M, \boldsymbol{q})$ seeks a pair of non-negative vectors $(\mathbf{y}, \mathbf{w})$ such that:

$$I\mathbf{w} - M\mathbf{y} = \boldsymbol{q} \quad \text{and } \forall i \in [d] \text{ we have } \mathbf{y}_i \cdot \mathbf{w}_i = 0. \tag{2}$$

If $\boldsymbol{q} \geq 0$, then $(\mathbf{y}, \mathbf{w}) = (0, \boldsymbol{q})$ is a *trivial solution.* We identify a solution $(\mathbf{y}, \mathbf{w})$ with the set of components of $\mathbf{y}$ that are positive: let $\alpha = \{i \mid \mathbf{y}_i > 0, \ i \in [d]\}$ denote such a set of "basic variables". Going the other way, to check if there is a solution that corresponds to a particular $\alpha \subseteq [d]$, we try to perform a *principal pivot transformation*, re-writing the LCP by writing certain variables $\mathbf{y}_i$ as $\mathbf{w}_i$, and checking if in this re-written LCP there exists the trivial solution $(\mathbf{y}', \mathbf{w}') = (0, \boldsymbol{q}')$. To that end, we construct an $d \times d$ matrix A_α, where the ith column of A_α is defined as follows. Let e_i denote the ith unit column vector in dimension d, and let $M_{\cdot i}$ denote the ith column of the matrix M.

$$(A_\alpha)_{\cdot i} := \begin{cases} -M_{\cdot i} & \text{if } i \in \alpha, \\ e_i & \text{if } i \notin \alpha. \end{cases} \tag{3}$$

Then α corresponds to an LCP solution if A_α is non-singular and, the "new $\boldsymbol{q}$" i.e., $(A_\alpha^{-1}\boldsymbol{q})$, is non-negative. For a given $\alpha \subseteq [n]$, we define $\text{out}(\alpha) := -$ if $\det(A_\alpha) = 0$; note that this will not happen if M is really a P-matrix, but our general treatment here is made to deal with the non-promise problem, in which case a zero determinant will correspond to a violation[4]. If $\det(A_\alpha) \neq 0$, we define $\text{out}(\alpha)$ as a bit-string in $\{0,1\}^d$ as follows:

$$(\text{out}(\alpha))_i = \begin{cases} 1 & \text{if } (A_\alpha^{-1}\boldsymbol{q})_i < 0, \\ 0 & \text{if } (A_\alpha^{-1}\boldsymbol{q})_i \geq 0. \end{cases} \tag{4}$$

With this notation, a subset $\alpha \subseteq [d]$ corresponds to a solution of the LCP if $\text{out}(\alpha) = 0^d$. We will use $\text{out}(\alpha)$ both to define a succinct violation of the P-matrix property, and in our promise-preserving reduction from P-LCP to Unique-Sink-Orientation.

Next, we introduce three types of succinct violations that prove that a matrix M is not a P-matrix. Let $\text{char}(v)$ for $v \subseteq [d]$ denote the characteristic vector of v, i.e., $(\text{char}(v))_i = 1$ if $i \in v$ and 0 otherwise. As in the section on USOs, when we write $\oplus$ and $\cap$, here we mean the bit-wise XOR and bit-wise and-operations on bit-strings.

Definition 36 (P-LCP violations)**.** *For a given LCP $(M, \boldsymbol{q})$ in dimension d, each of the following provides a polynomial-size witness that M is not a P-matrix:*

(PV1) A set $\alpha \subseteq [d]$ such that the corresponding principal minor is non-positive, i.e., $\det(M_{\alpha\alpha}) \leq 0$.

(PV2) A vector $x \neq 0$ whose sign is reversed by M, that is, for all $i \in [d]$ we have $x_i(Mx)_i \leq 0$.

(PV3) Two distinct sets $\alpha, \beta \subset [d]$, with $(\text{char}(\alpha) \oplus \text{char}(\beta)) \cap (\text{out}(\alpha) \oplus \text{out}(\beta)) = 0^d$.

The violation PV1 corresponds to the standard definition of a P-matrix as having all positive principal minors. Megiddo [57, 58] used this violation to place to place the P-LCP problem in TFNP. The same violation was then used by Papadimitriou to put P-LCP in PPAD, because Lemke's algorithm is a PPAD-type complementary pivoting algorithm, which inspired PPAD, and it will return a non-positive principal minor if it fails to find an LCP solution.

The characterization of P-matrices as those that do not reverse the sign of any non-zero vector, as used for violation PV2, was first discovered by Gale and Nikaido [31]. The final violation, PV3, follows from the work of Stickney and Watson [73], who showed that P-matrix LCPs give rise to USOs, and from the work of Tzabo and Welzl [74, Lemma 2.3], who showed that this condition characterizes that "outmap" of USOs, as discussed in Section 4.1, hence the name of the function being "out".

Several other characterizations of P-matrices are known, some of which would provide alternative succinct violations [14, 46]. We have given the violations PV1–PV3 above, since we use the three for our promise preserving reductions from P-LCP to UniqueEOPL and to Unique-Sink-Orientation. In particular, for our reduction from P-LCP to Unique-Sink-Orientation we need violations of types PV1 and PV3, and for our reduction from P-LCP directly to UniqueEOPL we need violations of types PV1 and PV2. It is not immediately apparent how to convert violations of one type to another in polynomial time, and it is conceivable that allowing different sets of violations changes the complexity of the problem. We leave it as further work to further explore this.

[4]We could also define $\text{out}(\alpha) := -$ if $\det(A_\alpha) < 0$, since then we also get a P-matrix violation, however we choose to still perform a principal pivot transformation in this case since, ceteris paribus, it seems reasonable to prefer an LCP solution to a P-matrix violation when we reduce P-LCP to USO.

Definition 37 (P-LCP)**.** *Given an LCP* $(M, \boldsymbol{q})$*, find either:*

(Q1) $\mathbf{y} \in \mathbb{R}^{d\times 1}$ *that satisfies (1).*

(Q2) one of the violations (PV1)–(PV3).

In the promise version, we are promised that M *is a P-matrix and seek a solution of type (Q1).*

Reduction from P-LCP to Unique-Sink-Orientation. We are now ready to present our reduction to USO. The reduction is simple, and we present it in full detail here. For the LCP instance $\mathcal{I} = (M, \boldsymbol{q})$ in dimension d, we produce an instance $\mathcal{U}$ of Unique-Sink-Orientation also in dimension d.

We first need to deal with the possibility that $\boldsymbol{q}$ is degenerate. A P-matrix LCP has a degenerate $\boldsymbol{q}$ if $A_\alpha^{-1} \cdot \boldsymbol{q}$ has a zero entry for some $\alpha \subseteq [d]$. To ensure that this does not present a problem for our reduction, we use a standard technique known as lexicographic perturbation [14, Section 4.2]: In the reduction that follows we assume that we are using such a degeneracy resolution scheme.

The reduction associates each vertex v of the resulting USO with a set a $\alpha(v)$ of basic variables for the LCP, and then uses $\text{out}(\alpha)$ as the outmap at v. In detail, for a vertex $v \in \{0,1\}^d$ of $\mathcal{U}$, we define $\alpha(v) = \{i \mid v(i) = 1\}$, and set $\Psi(v) = \text{out}(\alpha(v))$. It immediately follows that:

- A solution of type (US1) in $\mathcal{U}$ is a solution of type (Q1) in $\mathcal{I}$.
- A solution of type (USV1) in $\mathcal{U}$ is a solution of type (PV1) in $\mathcal{I}$.
- A solution of type (USV2) in $\mathcal{U}$ is a solution of type (PV3) in $\mathcal{I}$.

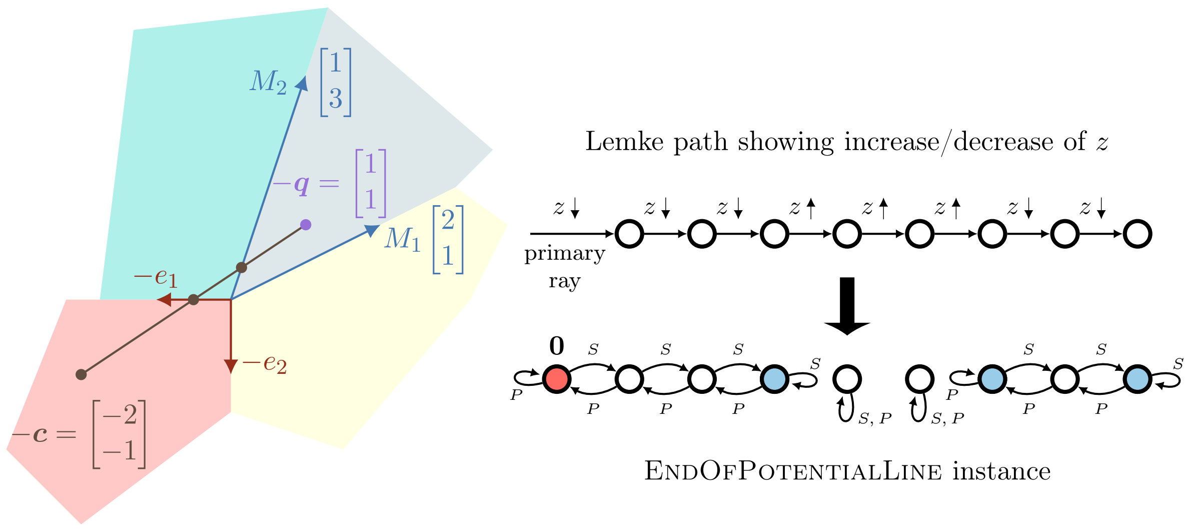
If M is actually a P-matrix then $\mathcal{U}$ will have exactly one solution of type (US1), and no violation solutions [73]. Thus our reduction is promise preserving, and we obtain the following.

Theorem 38. *There is a polynomial-time promise-preserving reduction from* P-LCP *with violations of type (PV1) and (PV3) to* Unique-Sink-Orientation*.*

Overview of reductions from P-LCP to EndOfPotentialLine and UniqueEOPL. Comparing UniqueEOPL and EndOfPotentialLine we see that: (U1) and (UV2) correspond to (R1), and (UV1) corresponds to (R2), so the only difference is that UniqueEOPL has the extra violation solution (UV3). Thus there is some extra work to do for our reduction to UniqueEOPL, to map (UV3) solutions back to a P-LCP solution.

Our reduction from P-LCP to the two problems produces the same instance. The reduction in both cases is based on Lemke's algorithm, which we describe in detail in Appendix G.1. For EndOfPotentialLine, we only need to use (PV1) violations. For UniqueEOPL, we only need to use (PV2) violations. Next we give a high-level description of the reduction, where for simplicity we just refer to a resulting EndOfPotentialLine instance. Full details of both reductions appear in Appendix G.

Lemke's algorithm introduces to the LCP an extra variable z and an extra positive vector $\boldsymbol{c}$, called a covering vector. It follows a path along edges of the new LCP polyhedron based on a complementary pivot rule that maintains an almost-complementary solution. In Figure 5, we give an example with $\boldsymbol{c} = (2,1)^\top$; in our reduction we take $\boldsymbol{c}$ to be the all ones vector $\mathbf{1}$. Geometrically, solving an LCP is equivalent to finding a *complementary cone*, corresponding to a subset of columns of M and the complementary unit vectors, that contains $-\boldsymbol{q}$. This is depicted on the left in Figure 5, which also shows Lemke's algorithm as inverting a piecewise linear map along the line from $-\boldsymbol{c}$ to



M_2 $\begin{bmatrix}1\\3\end{bmatrix}$
$-\boldsymbol{q} = \begin{bmatrix}1\\1\end{bmatrix}$
M_1 $\begin{bmatrix}2\\1\end{bmatrix}$
$-e_1$
$-e_2$
$-\boldsymbol{c} = \begin{bmatrix}-2\\-1\end{bmatrix}$
Lemke path showing increase/decrease of z
$z\downarrow$ $z\downarrow$ $z\downarrow$ $z\uparrow$ $z\uparrow$ $z\uparrow$ $z\downarrow$ $z\downarrow$
primary ray
$\mathbf{0}$
S S S S S S S
P P P P P P P
S, P S, P
EndOfPotentialLine instance

Figure 5: Left: geometric view of Lemke's algorithm as inverting a piecewise linear map. Right: construction of S and P for EndOfPotentialLine instance from the Lemke path, where the arrows on its edges indicate whether z increases or decreases along the edge. This figure should be viewed in color.

$-\boldsymbol{q}$. The algorithm pivots between the brown vertices at the intersections of complementary cones and terminates at $-\boldsymbol{q}$. The extra variable z can be normalized and then describes how far along the line from $\boldsymbol{c}$ to $-\boldsymbol{q}$ we are. P-matrix LCPs are exactly those where the complementary cones cover the whole space with no overlap, and then z decreases monotonically as Lemke's algorithm proceeds.

Each vertex along the Lemke path corresponds to a subset of $[d]$ of basic variables, and a possibly a unique duplicate label. A label $l \in [d]$ is said to be duplicate if $y_l = 0$ as well as $w_l = 0$. The vertices without a duplicate label has $z = 0$ and correspond to a solution of the LCP. To encode these subsets and the duplicate label, we consider bit strings of length $n = 2d$ that represent vertices in the EndOfPotentialLine instance. The first d bits encode the subset, and bits $(d+1, \ldots, 2d)$ bits encode the duplicate label, where bit $(d+l)$ is one if l is the duplicate label. Thus, "valid" vertices in EndOfPotentialLine instance can have at most one bit set to one among bits $(d+1)$ through $2d$. We ensure that "invalid" bit configurations form self-loops in the EndOfPotentialLine instance, and hence do not give rise to any solutions.

We use the following key properties of Lemke's algorithm as applied to a P-matrix:

1. If Lemke's algorithm does not terminate with a LCP solution (Q1), it provides a (Q2) solution.
2. For a P-matrix M, the extra variable z strictly decreases in each step of the algorithm.
3. Given a subset of $[d]$ and duplicate label l, we can efficiently decide if it corresponds to a vertex on a Lemke path.
4. By a result of Todd [76, Section 5], the Lemke path can be efficiently locally oriented.

Recall that LCP (1) has a trivial solution, namely $\mathbf{y} = 0$, if $\boldsymbol{q} \geq 0$. Therefore, wlog assume that $\min_{i\in[d]} q_i < 0$. The starting vertex of the Lemke path is the vertex $\boldsymbol{x}^0 = (\mathbf{y}^0, \mathbf{w}^0, z^0)$ with $\mathbf{y}^0 = 0$, $z^0 = |\min_{i\in[d]} q_i|$, and $\mathbf{w} = \boldsymbol{q} + z^0\mathbf{1}$. So 0^n is a start of line in the EndOfPotentialLine instance, we point the successor of 0^n to the bit configuration corresponding to $\boldsymbol{x}^0$. We then follow the line of bit configurations corresponding to the vertices traversed by Lemke's algorithm, updating z in each step. We use $(z^0 - z + 1)$ as the potential function – $(z^0 - z)$ to ensure increasing potential along the line, and $+1$ to ensure that only 0^n has zero potential. If we start with a P-LCP instance where M is actually a P-matrix then this reduction will produce a single line from $\boldsymbol{x}^0$ to the solution of the P-LCP, and z will monotonically decrease along this line. The main difficulty of the reduction is dealing with the case where M is not a P-matrix. This may cause Lemke's algorithm to terminate without an LCP solution. Another issue is that, even when Lemke's algorithm does find a solution, z may not decrease monotonically along the line.

In the former case, the first property above gives us a (Q2) solution for the P-LCP problem. In the latter case, we define any point on the line where z increases to be a self-loop, breaking the line at these points. Figure 5 shows an example, where the two vertices at which z increases are turned into self loops, thereby introducing two new solutions before and after the break. Both of these solutions give us a (Q2) solution for the P-LCP instance. The full details of the reduction are involved and appear in Appendix G. It is worth noting that, in the case where the input is actually a P-matrix, the resulting EndOfPotentialLine instance has a unique line, so our reduction in promise preserving. Moreover, our EndOfPotentialLine instance is a valid UniqueEOPL instance, and we can map back all violations so as to obtain the following.

Theorem 39. *There are a polynomial-time promise-preserving reduction from* P-LCP *with violations of type (PV1) to* EndOfPotentialLine*, and from* P-LCP *with violations of type (PV2) to* UniqueEOPL*, and thereby also to* EndOfPotentialLine*. Both reductions only incur a linear blowup in the size of the instance.*

5 Algorithms

5.1 Algorithms for Contraction Maps

An algorithm for PL-Contraction. The properties that we observed in our reduction from PL-Contraction to EndOfPotentialLine can also be used to give polynomial time algorithms for the case where the number of dimensions is constant. In our two-dimensional example, we relied on the fact that each dimension-two slice has a unique point on the blue surface, and that the direction function at this point tells us the direction of the overall fixpoint.

This suggests that a nested binary search approach can be used to find the fixpoint. The outer binary search will work on dimension-two coordinates, and the inner binary search will work on dimension-one coordinates. For each fixed dimension-two coordinate y, we can apply the inner binary search to find the unique point (x, y) that is on the blue surface. Once we have done so, $D_2(x, y)$ tells us how to update the outer binary search to find a new candidate coordinate y'.

This can be generalized to d-dimensional instances, by running d nested instances of binary search. Moreover, our algorithm can detect violations in the course of performing the binary search and is able to produce witnesses to the given function not being a contraction map. Thus, our algorithm solves the non-promise problem PL-Contraction, giving the following theorem, whose proof appears in Appendix H.3.

Theorem 40. *Given a* LinearFIXP *circuit C purporting to encode a contraction map $f : [0,1]^d \to [0,1]^d$ with respect to any ℓ_p norm, there is an algorithm to find a fixpoint of f or return a pair of points witnessing that f is not a contraction map in time that is polynomial in size(C) and exponential in d.*

An algorithm for Contraction. We are also able to generalize this to the more general Contraction problem, where the input is given as an arbitrary (non-linear) arithmetic circuit. Here the key issue is that the fixpoint may not be rational, and so we must find a suitably accurate approximate fixpoint. Our nested binary search approach can be adapted to do this.

Since we now deal with approximate fixpoints, we must cut off each of our nested binary search instances at an appropriate accuracy. Specifically, we must ensure that the solution is accurate enough so that we can correctly update the outer binary search. Choosing these cutoff points turns out to be quite involved, as we must choose different cutoff points depending on both the norm and the level of recursion, and moreover the ℓ_1 case requires a separate proof.

Again, the algorithm is able to detect violations of contraction during the course of the binary search, and thus solves the more general problem of either finding a fixpoint when the circuit defines a contraction map, or returning a pair of points that are not contracting. The details of this are deferred to Appendix H.4, where the following theorem is shown.

Theorem 41. *For a contraction map $f : [0,1]^d \to [0,1]^d$ under $\|\cdot\|_p$ for $2 \leq p < \infty$, there is an algorithm to compute a point $v \in [0,1]^d$ such that $\|f(v) - v\|_p < \varepsilon$ or return a pair of points (x, y) such that $\|f(x) - f(y)\|_p > c\,\|x - y\|_p$ in time $O(p^{d^2} \log^d(1/\varepsilon) \log^d(p))$.*

Actually, our algorithm treats the function as a black-box, and so it can be applied to any contraction map, with Theorem 41 giving the number of queries that need to be made.

5.2 Aldous' algorithm for PLCP

Aldous [2] analysed a simple randomized algorithm for solving local search problems. The algorithm randomly samples a large number of candidate solutions and then performs a local search from the best sampled solution. Aldous' algorithm can solve any problem in PLS and thus any problem in UniqueEOPL. In [35] it was noted that, because our reduction from our reduction from P-LCP to UniqueEOPL only incurs a linear blowup, that is, from an LCP in dimension n we produce an UniqueEOPL instance with $O(2^n)$ vertices, when we apply Aldous' algorithm to the resulting instance, the expected time is $2^{n/2} \cdot \text{poly}(n)$ in the worst case, which gives the fastest known running time for a randomized algorithm for P-LCP. Thus, we get the following corollary of Theorem 39.

Corollary 42. *There is a randomized algorithm for* P-LCP *that runs in expected time $O(1.4143^n)$.*

References

[1] Adler, I., and Verma, S. The linear complementarity problem, Lemke algorithm, perturbation, and the complexity class PPAD. Tech. rep., Manuscript, Depatrment of IEOR, University of California, Berkeley, CA 94720, 2011.

[2] Aldous, D. Minimization algorithms and random walk on the d-cube. *The Annals of Probability* (1983), 403–413.

[3] Banach, S. Sur les opérations dans les ensembles abstraits et leur application aux équations intégrales. *Fundamenta Mathematicae 3*, 1 (1922), 133–181.

[4] Bitansky, N., Paneth, O., and Rosen, A. On the cryptographic hardness of finding a Nash equilibrium. In *Proc. of FOCS* (2015), pp. 1480–1498.

[5] Björklund, H., Sandberg, S., and Vorobyov, S. A combinatorial strongly subexponential strategy improvement algorithm for mean payoff games. In *Proc. MFCS* (2004), pp. 673–685.

[6] Boonyasiriwat, C., Sikorski, K., and Xiong, C. A note on two fixed point problems. *J. Complexity 23*, 4-6 (2007), 952–961.

[7] Calude, C. S., Jain, S., Khoussainov, B., Li, W., and Stephan, F. Deciding parity games in quasipolynomial time. In *Proc. STOC* (2017), pp. 252–263.

[8] Chen, X., and Deng, X. On algorithms for discrete and approximate Brouwer fixed points. In *Proc. of STOC* (2005), pp. 323–330.

[9] Chen, X., and Deng, X. Matching algorithmic bounds for finding a Brouwer fixed point. *J. ACM 55*, 3 (2008), 13:1–13:26.

[10] Chen, X., and Deng, X. On the complexity of 2d discrete fixed point problem. *Theor. Comput. Sci. 410*, 44 (2009), 4448–4456.

[11] Chen, X., Deng, X., and Teng, S.-H. Settling the complexity of computing two-player Nash equilibria. *J. ACM 56*, 3 (2009), 14.

[12] Chung, S.-J. NP-completeness of the Linear Complementarity Problem. *Journal of Optimization Theory and Applications 60*, 3 (1989), 393–399.

[13] Condon, A. The complexity of stochastic games. *Information and Computation 96*, 2 (1992), 203–224.

[14] Cottle, R. W., Pang, J.-S., and Stone, R. E. *The Linear Complementarity Problem.* SIAM, 2009.

[15] Coxson, G. E. The P-matrix problem is co-NP-complete. *Mathematical Programming 64*, 1 (1994), 173–178.

[16] Daskalakis, C., Goldberg, P. W., and Papadimitriou, C. H. The complexity of computing a Nash equilibrium. *SIAM Journal on Computing 39*, 1 (2009), 195–259.

[17] Daskalakis, C., and Papadimitriou, C. Continuous Local Search. In *Proc. of SODA* (2011), pp. 790–804.

[18] Daskalakis, C., Tzamos, C., and Zampetakis, M. A Converse to Banach's Fixed Point Theorem and its CLS Completeness. In *Proc. of STOC* (2018).

[19] Deng, X., Qi, Q., Saberi, A., and Zhang, J. Discrete fixed points: Models, complexities, and applications. *Math. Oper. Res. 36*, 4 (2011), 636–652.

[20] Dohrau, J., Gärtner, B., Kohler, M., Matoušek, J., and Welzl, E. ARRIVAL: a zero-player graph game in $\mathsf{NP} \cap \mathsf{coNP}$. In *A journey through discrete mathematics*. Springer, Cham, 2017, pp. 367–374.

[21] Emerson, E. A., and Jutla, C. S. Tree automata, mu-calculus and determinacy. In *Proc. of FOCS* (1991), pp. 368–377.

[22] Etessami, K., and Yannakakis, M. On the complexity of nash equilibria and other fixed points. *SIAM J. Comput. 39*, 6 (2010), 2531–2597.

[23] Fabrikant, A., Papadimitriou, C., and Talwar, K. The complexity of pure Nash equilibria. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing (STOC)* (2004), ACM, pp. 604–612.

[24] Fearnley, J., Gordon, S., Mehta, R., and Savani, R. CLS: new problems and completeness. *CoRR abs/1702.06017* (2017).

[25] Fearnley, J., Gordon, S., Mehta, R., and Savani, R. End of potential line. *CoRR abs/1804.03450* (2018).

[26] Fearnley, J., Jain, S., Schewe, S., Stephan, F., and Wojtczak, D. An ordered approach to solving parity games in quasi polynomial time and quasi linear space. In *Proc. of SPIN* (2017), pp. 112–121.

[27] Fearnley, J., Jurdziński, M., and Savani, R. Linear complementarity algorithms for infinite games. In *International Conference on Current Trends in Theory and Practice of Computer Science* (2010), Springer, pp. 382–393.

[28] Fearnley, J., and Savani, R. The complexity of all-switches strategy improvement. In *Proc. of SODA* (2016), pp. 130–139.

[29] Friedmann, O., Hansen, T. D., and Zwick, U. A subexponential lower bound for the random facet algorithm for parity games. In *Proc. of SODA* (2011), pp. 202–216.

[30] Gairing, M., and Savani, R. Computing stable outcomes in hedonic games. In *Proc. of SAGT* (2010), pp. 174–185.

[31] Gale, D., and Nikaido, H. The Jacobian matrix and global univalence of mappings. *Mathematische Annalen 159*, 2 (1965), 81–93.

[32] Garg, J., Mehta, R., Sohoni, M., and Vazirani, V. V. A complementary pivot algorithm for market equilibrium under separable piecewise-linear concave utilities. *SIAM J. Comput. 44*, 6 (2015), 1820–1847.

[33] Garg, S., Pandey, O., and Srinivasan, A. Revisiting the cryptographic hardness of finding a Nash equilibrium. In *Annual Cryptology Conference* (2016), Springer, pp. 579–604.

[34] Gärtner, B. The random-facet simplex algorithm on combinatorial cubes. *Random Struct. Algorithms 20*, 3 (2002), 353–381.

[35] Gärtner, B., Hansen, T. D., Hubáček, P., Král, K., Mosaad, H., and Slívová, V. ARRIVAL: next stop in CLS. In *Proc. of ICALP* (2018), pp. 60:1–60:13.

[36] Gärtner, B., and Rüst, L. Simple stochastic games and P-matrix generalized linear complementarity problems. In *International Symposium on Fundamentals of Computation Theory* (2005), Springer, pp. 209–220.

[37] Gärtner, B., and Schurr, I. Linear programming and unique sink orientations. In *Proc. of SODA* (2006), pp. 749–757.

[38] Gärtner, B., and Thomas, A. The complexity of recognizing unique sink orientations. In *Proc. of STACS* (2015), pp. 341–353.

[39] Hansen, T. D., and Ibsen-Jensen, R. The complexity of interior point methods for solving discounted turn-based stochastic games. In *Conference on Computability in Europe* (2013), pp. 252–262.

[40] Hansen, T. D., Paterson, M., and Zwick, U. Improved upper bounds for random-edge and random-jump on abstract cubes. In *Proc. of SODA* (2014), pp. 874–881.

[41] Hirsch, M. D., Papadimitriou, C. H., and Vavasis, S. A. Exponential lower bounds for finding Brouwer fix points. *J. Complexity 5*, 4 (1989), 379–416.

[42] Hoke, K. W. Completely unimodal numberings of a simple polytope. *Discrete Applied Mathematics 20*, 1 (1988), 69–81.

[43] Huang, Z., Khachiyan, L., and Sikorski, K. Approximating fixed points of weakly contracting mappings. *J. Complexity 15* (1999), 200–213.

[44] Huang, Z., Khachiyan, L. G., and Sikorski, C. K. Approximating fixed points of weakly contracting mappings. *J. Complexity 15*, 2 (1999), 200–213.

[45] Hubáček, P., and Yogev, E. Hardness of continuous local search: Query complexity and cryptographic lower bounds. In *Proc. of SODA* (2017), pp. 1352–1371.

[46] Johnson, C. R., and Tsatsomeros, M. J. Convex sets of nonsingular and P-matrices. *Linear and Multilinear Algebra 38*, 3 (1995), 233–239.

[47] Johnson, D. S., Papadimitriou, C. H., and Yannakakis, M. How easy is local search? *Journal of Computer and System Sciences 37*, 1 (1988), 79–100.

[48] Jurdziński, M. Deciding the winner in parity games is in UP $\cap$ co-UP. *Information Processing Letters 68*, 3 (1998), 119–124.

[49] Jurdzinski, M., and Lazic, R. Succinct progress measures for solving parity games. In *Proc. of LICS* (2017), pp. 1–9.

[50] Jurdziński, M., and Savani, R. A simple P-matrix linear complementarity problem for discounted games. In *Conference on Computability in Europe* (2008), Springer, pp. 283–293.

[51] Kalai, G. Three puzzles on mathematics, computation, and games. *CoRR abs/1801.02602* (2018).

[52] Kojima, M., Megiddo, N., Noma, T., and Yoshise, A. *A unified approach to interior point algorithms for linear complementarity problems*, vol. 538. Springer Science & Business Media, 1991.

[53] Kojima, M., Megiddo, N., and Ye, Y. An interior point potential reduction algorithm for the linear complementarity problem. *Mathematical Programming 54*, 1-3 (1992), 267–279.

[54] Lemke, C. E. Bimatrix equilibrium points and mathematical programming. *Management science 11*, 7 (1965), 681–689.

[55] Ludwig, W. A subexponential randomized algorithm for the simple stochastic game problem. *Information and computation 117*, 1 (1995), 151–155.

[56] Matoušek, J., and Szabó, T. Random edge can be exponential on abstract cubes. In *Proc. of FOCS* (2004), pp. 92–100.

[57] Megiddo, N. *A note on the complexity of P-matrix LCP and computing an equilibrium.* IBM Thomas J. Watson Research Division, 1988.

[58] Megiddo, N., and Papadimitriou, C. H. On total functions, existence theorems and computational complexity. *Theoretical Computer Science 81*, 2 (1991), 317–324.

[59] Mehta, R. Constant rank bimatrix games are PPAD-hard. In *Proc. of STOC* (2014), pp. 545–554.

[60] Meunier, F., Mulzer, W., Sarrabezolles, P., and Stein, Y. The rainbow at the end of the line: A PPAD formulation of the colorful Carathéodory theorem with applications. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2017), pp. 1342–1351.

[61] Morris Jr, W. D. Randomized pivot algorithms for P-matrix linear complementarity problems. *Mathematical programming 92*, 2 (2002), 285–296.

[62] Murty, K. G. Computational complexity of complementary pivot methods. In *Complementarity and fixed point problems.* Springer, 1978, pp. 61–73.

[63] Nemirovsky, A., and Yudin, D. B. *Problem Complexity and Method Efficiency in Optimization.* Wiley, New York, 1983.

[64] Papadimitriou, C. H. On the complexity of the parity argument and other inefficient proofs of existence. *Journal of Computer and System Sciences 48*, 3 (1994), 498–532.

[65] Puri, A. Theory of hybrid systems and discrete event systems.

[66] Rubinstein, A. Settling the complexity of computing approximate two-player Nash equilibria. In *Proc. of FOCS* (2016), pp. 258–265.

[67] Schäffer, A. A., and Yannakakis, M. Simple local search problems that are hard to solve. *SIAM journal on Computing 20*, 1 (1991), 56–87.

[68] Schurr, I., and Szabó, T. Finding the sink takes some time: An almost quadratic lower bound for finding the sink of unique sink oriented cubes. *Discrete & Computational Geometry 31*, 4 (2004), 627–642.

[69] Schurr, I., and Szabó, T. Jumping doesn't help in abstract cubes. In *Proc. of IPCO* (2005), pp. 225–235.

[70] Shellman, S., and Sikorski, K. A recursive algorithm for the infinity-norm fixed point problem. *Journal of Complexity 19*, 6 (2003), 799 – 834.

[71] Sikorski, K. *Optimal solution of Nonlinear Equations.* Oxford Press, New York, 200.

[72] Sikorski, K. Computational complexity of fixed points. *Journal of Fixed Point Theory and Applications 6*, 2 (2009), 249–283.

[73] Stickney, A., and Watson, L. Digraph models of bard-type algorithms for the linear complementarity problem. *Mathematics of Operations Research 3*, 4 (1978), 322–333.

[74] Szabó, T., and Welzl, E. Unique sink orientations of cubes. In *Proc. of FOCS* (2001), pp. 547–555.

[75] Thomas, A. Exponential lower bounds for history-based simplex pivot rules on abstract cubes. In *Proc. of ESA* (2017), pp. 69:1–69:14.

[76] Todd, M. J. Orientation in complementary pivot algorithms. *Mathematics of Operations Research 1*, 1 (1976), 54–66.

[77] Zwick, U., and Paterson, M. The complexity of mean payoff games on graphs. *Theoretical Computer Science 158*, 1-2 (1996), 343–359.

A Proofs for Section 2: Equivalence of EOPL and EOML

First we recall the definition of EndOfMeteredLine, which was first defined in [45]. It is close in spirit to the problem EndOfLine that is used to define PPAD [64].

Definition 43 (EndOfMeteredLine [45]). *Given circuits $S, P : \{0,1\}^n \to \{0,1\}^n$, and $V : \{0,1\}^n \to \{0,\ldots,2^n\}$ such that $P(0^n) = 0^n \neq S(0^n)$ and $V(0^n) = 1$, find a string $\boldsymbol{x} \in \{0,1\}^n$ satisfying one of the following*

(T1) either $S(P(\boldsymbol{x})) \neq \boldsymbol{x} \neq 0^n$ or $P(S(\boldsymbol{x})) \neq \boldsymbol{x}$,

(T2) $\boldsymbol{x} \neq 0^n, V(\boldsymbol{x}) = 1$,

(T3) either $V(\boldsymbol{x}) > 0$ and $V(S(\boldsymbol{x})) - V(\boldsymbol{x}) \neq 1$, or $V(\boldsymbol{x}) > 1$ and $V(\boldsymbol{x}) - V(P(\boldsymbol{x})) \neq 1$.

EndOfMeteredLine is actually quite similar to EndOfPotentialLine. The main difference is that in EndOfMeteredLine, each edge must increase the potential by exactly 1, which is enforced by solution type T3.

A.1 EndOfMeteredLine to EndOfPotentialLine

Given an instance $\mathcal{I}$ of EndOfMeteredLine defined by circuits S, P and V on vertex set $\{0,1\}^n$ we are going to create an instance $\mathcal{I}'$ of EndOfPotentialLine with circuits S', P', and V' on vertex set $\{0,1\}^{(n+1)}$, i.e., we introduce one extra bit. This extra bit is essentially to take care of the difference in the value of potential at the starting point in EndOfMeteredLine and EndOfPotentialLine, namely 1 and 0 respectively.

Let $k = n+1$, then we create a potential function $V' : \{0,1\}^k \to \{0,\ldots,2^k - 1\}$. The idea is to make 0^k the starting point with potential zero as required, and to make all other vertices with first bit 0 be dummy vertices with self loops. The real graph will be embedded in vertices with first bit 1, i.e., of type $(1, \boldsymbol{u})$. Here by $(b, \boldsymbol{u}) \in \{0,1\}^k$, where $b \in \{0,1\}$ and $\boldsymbol{u} \in \{0,1\}^n$, we mean a k length bit string with first bit set to b and for each $i \in [2:k]$ bit i set to bit u_i.

Procedure $V'(b, \boldsymbol{u})$: If $b = 0$ then Return 0, otherwise Return $V(\boldsymbol{u})$.

Procedure $S'(b, \boldsymbol{u})$:

1. If $(b, \boldsymbol{u}) = 0^k$ then Return $(1, 0^n)$
2. If $b = 0$ and $\boldsymbol{u} \neq 0^n$ then Return $(b, \boldsymbol{u})$ (creating self loop for dummy vertices)
3. If $b = 1$ and $V(\boldsymbol{u}) = 0$ then Return $(b, \boldsymbol{u})$ (vertices with zero potentials have self loops)
4. If $b = 1$ and $V(\boldsymbol{u}) > 0$ then Return $(b, S(\boldsymbol{u}))$ (the rest follows S)

Procedure $P'(b, \boldsymbol{u})$:

1. If $(b, \boldsymbol{u}) = 0^k$ then Return $(b, \boldsymbol{u})$ (initial vertex points to itself in P').
2. If $b = 0$ and $\boldsymbol{u} \neq 0^n$ then Return $(b, \boldsymbol{u})$ (creating self loop for dummy vertices)
3. If $b = 1$ and $\boldsymbol{u} = 0^n$ then Return 0^k (to make $(0, 0^n) \to (1, 0^n)$ edge consistent)
4. If $b = 1$ and $V(\boldsymbol{u}) = 0$ then Return $(b, \boldsymbol{u})$ (vertices with zero potentials have self loops)
5. If $b = 1$ and $V(\boldsymbol{u}) > 0$ and $\boldsymbol{u} \neq 0^n$ then Return $(b, P(\boldsymbol{u}))$ (the rest follows P)

Valid solutions of EndOfMeteredLine of type T2 and T3 requires the potential to be strictly greater than zero, while solutions of EndOfPotentialLine may have zero potential. However, a solution of EndOfPotentialLine can not be a self loop, so we've added self-loops around vertices with zero potential in the EndOfPotentialLine instance. By construction, the next lemma follows:

Lemma 44. *S', P', V' are well defined and polynomial in the sizes of S, P, V respectively.*

Our main theorem in this section is a consequence of the following three lemmas.

Lemma 45. *For any $\boldsymbol{x} = (b, \boldsymbol{u}) \in \{0,1\}^k$, $P'(\boldsymbol{x}) = S'(\boldsymbol{x}) = \boldsymbol{x}$ (self loop) iff $\boldsymbol{x} \neq 0^k$, and $b = 0$ or $V(\boldsymbol{u}) = 0$.*

Proof. This follows by the construction of V', the second condition in S' and P', and third and fourth conditions in S' and P' respectively. ☐

Lemma 46. *Let $\boldsymbol{x} = (b, \boldsymbol{u}) \in \{0,1\}^k$ be such that $S'(P'(\boldsymbol{x})) \neq \boldsymbol{x} \neq 0^k$ or $P'(S'(\boldsymbol{x})) \neq \boldsymbol{x}$ (an (R1) type solution of* EndOfPotentialLine *instance $\mathcal{I}'$), then $\boldsymbol{u}$ is a solution of* EndOfMeteredLine *instance $\mathcal{I}$.*

Proof. The proof requires a careful case analysis. By the first conditions in the descriptions of S', P' and V', we have $\boldsymbol{x} \neq 0^k$. Further, since $\boldsymbol{x}$ is not a self loop, Lemma 45 implies $b = 1$ and $V'(1, \boldsymbol{u}) = V(\boldsymbol{u}) > 0$.

Case I. If $S'(P'(\boldsymbol{x})) \neq \boldsymbol{x} \neq 0^k$ then we will show that either $\boldsymbol{u}$ is a genuine start of a line other than 0^n giving a T1 type solution of EndOfMeteredLine instance $\mathcal{I}$, or there is some issue with the potential at $\boldsymbol{u}$ giving either a T2 or T3 type solution of $\mathcal{I}$. Since $S'(P'(1, 0^n)) = (1, 0^n)$, $\boldsymbol{u} \neq 0^n$. Thus if $S(P(\boldsymbol{u})) \neq \boldsymbol{u}$ then we get a T1 type solution of $\mathcal{I}$ and proof follows. If $V(\boldsymbol{u}) = 1$ then we get a T2 solution of $\mathcal{I}$ and proof follows.

Otherwise, we have $S(P(\boldsymbol{u})) = \boldsymbol{u}$ and $V(\boldsymbol{u}) > 1$. Now since also $b = 1$ $(1, \boldsymbol{u})$ is not a self loop (Lemma 45). Then it must be the case that $P'(1, \boldsymbol{u}) = (1, P(\boldsymbol{u}))$. However, $S'(1, P(\boldsymbol{u})) \neq (1, \boldsymbol{u})$ even though $S(P(\boldsymbol{u})) = \boldsymbol{u}$. This happens only when $P(\boldsymbol{u})$ is a self loop because of $V(P(\boldsymbol{u})) = 0$ (third condition of P'). Therefore, we have $V(\boldsymbol{u}) - V(P(\boldsymbol{u})) > 1$ implying that $\boldsymbol{u}$ is a T3 type solution of $\mathcal{I}$.

Case II. Similarly, if $P'(S'(\boldsymbol{x})) \neq \boldsymbol{x}$, then either $\boldsymbol{u}$ is a genuine end of a line of $\mathcal{I}$, or there is some issue with the potential at $\boldsymbol{u}$. If $P(S(\boldsymbol{u})) \neq \boldsymbol{u}$ then we get T1 solution of $\mathcal{I}$. Otherwise, $P(S(\boldsymbol{u})) = \boldsymbol{u}$ and $V(\boldsymbol{u}) > 0$. Now as $(b, \boldsymbol{u})$ is not a self loop and $V(\boldsymbol{u}) > 0$, it must be the case that $S'(b, \boldsymbol{u}) = (1, S(\boldsymbol{u}))$. However, $P'(1, S(\boldsymbol{u})) \neq (b, \boldsymbol{u})$ even though $P(S(\boldsymbol{u})) = \boldsymbol{u}$. This happens only when $S(\boldsymbol{u})$ is a self loop because of $V(S(\boldsymbol{u})) = 0$. Therefore, we get $V(S(\boldsymbol{u})) - V(\boldsymbol{u}) < 0$, i.e., $\boldsymbol{u}$ is a type T3 solution of $\mathcal{I}$. ☐

Lemma 47. *Let $\boldsymbol{x} = (b, \boldsymbol{u}) \in \{0,1\}^k$ be an (R2) type solution of the constructed* EndOfPotentialLine *instance $\mathcal{I}'$, then $\boldsymbol{u}$ is a type T3 solution of* EndOfMeteredLine *instance $\mathcal{I}$.*

Proof. Clearly, $\boldsymbol{x} \neq 0^k$. Let $\mathbf{y} = (b', \boldsymbol{u}') = S'(\boldsymbol{x}) \neq \boldsymbol{x}$, and observe that $P(\mathbf{y}) = \boldsymbol{x}$. This also implies that $\mathbf{y}$ is not a self loop, and hence $b = b' = 1$ and $V(\boldsymbol{u}) > 0$ (Lemma 45). Further, $\mathbf{y} = S'(1, \boldsymbol{u}) = (1, S(\boldsymbol{u}))$, hence $\boldsymbol{u}' = S(\boldsymbol{u})$. Also, $V'(\boldsymbol{x}) = V'(1, \boldsymbol{u}) = V(\boldsymbol{u})$ and $V'(\mathbf{y}) = V'(1, \boldsymbol{u}') = V(\boldsymbol{u}')$.

Since $V'(\mathbf{y}) - V'(\boldsymbol{x}) \leq 0$ we get $V(\boldsymbol{u}') - V(\boldsymbol{u}) \leq 0 \Rightarrow V(S(\boldsymbol{u})) - V(\boldsymbol{u}) \leq 0 \Rightarrow V(S(\boldsymbol{u})) - V(\boldsymbol{u}) \neq 1$. Given that $V(\boldsymbol{u}) > 0$, $\boldsymbol{u}$ gives a type T3 solution of EndOfMeteredLine. ☐

Theorem 48. *An instance of* EndOfMeteredLine *can be reduced to an instance of* EndOfPotentialLine *in linear time such that a solution of the former can be constructed in a linear time from the solution of the latter.*

A.2 EndOfPotentialLine to EndOfMeteredLine

In this section we give a linear time reduction from an instance $\mathcal{I}$ of EndOfPotentialLine to an instance $\mathcal{I}'$ of EndOfMeteredLine. Let the given EndOfPotentialLine instance $\mathcal{I}$ be defined on vertex set $\{0,1\}^n$ and with procedures S, P and V, where $V : \{0,1\}^n \to \{0, \ldots, 2^m - 1\}$.

Valid Edge. We call an edge $\boldsymbol{u} \to \boldsymbol{v}$ valid if $\boldsymbol{v} = S(\boldsymbol{u})$ and $\boldsymbol{u} = P(\boldsymbol{v})$.

We construct an EndOfMeteredLine instance $\mathcal{I}'$ on $\{0,1\}^k$ vertices where $k = n + m$. Let S', P' and V' denotes the procedures for $\mathcal{I}'$ instance. The idea is to capture value $V(\boldsymbol{x})$ of the potential in the m least significant bits of vertex description itself, so that it can be gradually increased or decreased on valid edges. For vertices with irrelevant values of these least m significant bits we will create self loops. Invalid edges will also become self loops, e.g., if $\mathbf{y} = S(\boldsymbol{x})$ but $P(\mathbf{y}) \neq \boldsymbol{x}$ then set $S'(\boldsymbol{x}, .) = (\boldsymbol{x}, .)$. We will see how these can not introduce new solutions.

In order to ensure $V'(0^k) = 1$, the $V(S(0^n)) = 1$ case needs to be discarded. For this, we first do some initial checks to see if the given instance $\mathcal{I}$ is not trivial. If the input EndOfPotentialLine instance is trivial, in the sense that either 0^n or $S(0^n)$ is a solution, then we can just return it.

Lemma 49. *If 0^n or $S(0^n)$ are not solutions of* EndOfPotentialLine *instance $\mathcal{I}$ then $0^n \to S(0^n) \to S(S(0^n))$ are valid edges, and $V(S(S(0^n)) \geq 2$.*

Proof. Since both 0^n and $S(0^n)$ are not solutions, we have $V(0^n) < V(S(0^n)) < V(S(S(0^n)))$, $P(S(0^n)) = 0^n$, and for $\boldsymbol{u} = S(0^n)$, $S(P(\boldsymbol{u})) = \boldsymbol{u}$ and $P(S(\boldsymbol{u})) = \boldsymbol{u}$. In other words, $0^n \to S(0^n) \to S(S(0^n))$ are valid edges, and since $V(0^n) = 0$, we have $V(S(S(0^n)) \geq 2$. $\square$

Let us assume now on that 0^n and $S(0^n)$ are not solutions of $\mathcal{I}$, and then by Lemma 49, we have $0^n \to S(0^n) \to S(S(0^n))$ are valid edges, and $V(S(S(0^n)) \geq 2$. We can avoid the need to check whether $V(S(0))$ is one all together, by making 0^n point directly to $S(S(0^n))$ and make $S(0^n)$ a dummy vertex.

We first construct S' and P', and then construct V' which will give value zero to all self loops, and use the least significant m bits to give a value to all other vertices. Before describing S' and P' formally, we first describe the underlying principles. Recall that in $\mathcal{I}$ vertex set is $\{0,1\}^n$ and possible potential values are $\{0, \ldots, 2^m - 1\}$, while in $\mathcal{I}'$ vertex set is $\{0,1\}^k$ where $k = m + n$. We will denote a vertex of $\mathcal{I}'$ by a tuple $(\boldsymbol{u}, \pi)$, where $\boldsymbol{u} \in \{0,1\}^n$ and $\pi \in \{0, \ldots, 2^m - 1\}$. Here when we say that we introduce an *edge* $\boldsymbol{x} \to \mathbf{y}$ we mean that we introduce a valid edge from $\boldsymbol{x}$ to $\mathbf{y}$, i.e., $\mathbf{y} = S'(\boldsymbol{x})$ and $\boldsymbol{x} = P(\mathbf{y})$.

- Vertices of the form $(S(0^n), \pi)$ for any $\pi \in \{0,1\}^m$ and the vertex $(0^n, 1)$ are dummies and hence have self loops.
- If $V(S(S(0^n)) = 2$ then we introduce an edge $(0^n, 0) \to (S(S(0^n)), 2)$, otherwise
  - for $p = V(S(S(0^n))$, we introduce the edges $(0^n, 0) \to (0^n, 2) \to (0^n, 3) \ldots (0^n, p-1) \to (S(S(0^n)), p)$.
- If $\boldsymbol{u} \to \boldsymbol{u}'$ valid edge in $\mathcal{I}$ then let $p = V(\boldsymbol{u})$ and $p' = V(\boldsymbol{u}')$
  - If $p = p'$ then we introduce the edge $(\boldsymbol{u}, p) \to (\boldsymbol{u}', p')$.
  - If $p < p'$ then we introduce the edges $(\boldsymbol{u}, p) \to (\boldsymbol{u}, p+1) \to \ldots \to (\boldsymbol{u}, p'-1) \to (\boldsymbol{u}', p')$.
  - If $p > p'$ then we introduce the edges $(\boldsymbol{u}, p) \to (\boldsymbol{u}, p-1) \to \ldots \to (\boldsymbol{u}, p'+1) \to (\boldsymbol{u}', p')$.
- If $\boldsymbol{u} \neq 0^n$ is the start of a path, i.e., $S(P(\boldsymbol{u})) \neq \boldsymbol{u}$, then make $(\boldsymbol{u}, V(\boldsymbol{u}))$ start of a path by ensuring $P'(\boldsymbol{u}, V(\boldsymbol{u})) = (\boldsymbol{u}, V(\boldsymbol{u}))$.

- If $\boldsymbol{u}$ is the end of a path, i.e., $P(S(\boldsymbol{u})) \neq \boldsymbol{u}$, then make $(\boldsymbol{u}, V(\boldsymbol{u}))$ end of a path by ensuring $S'(\boldsymbol{u}, V(\boldsymbol{u})) = (\boldsymbol{u}, V(\boldsymbol{u}))$.

Last two bullets above remove singleton solutions from the system by making them self loops. However, this can not kill all the solutions since there is a path starting at 0^n, which has to end somewhere. Further, note that this entire process ensures that no new start or end of a paths are introduced.

Procedure $S'(\boldsymbol{u}, \pi)$.

```
1. If ($\boldsymbol{u} = 0^n$ and $\pi = 1$) or $\boldsymbol{u} = S(0^n)$ then Return $(\boldsymbol{u}, \pi)$.
2. If $(\boldsymbol{u}, \pi) = 0^k$, then let $\boldsymbol{u}' = S(S(0^n))$ and $p' = V(\boldsymbol{u}')$.
   (a) If $p' = 2$ then Return $(\boldsymbol{u}', 2)$ else Return $(0^n, 2)$.
3. If $\boldsymbol{u} = 0^n$ then
   (a) If $2 \leq \pi < p' - 1$ then Return $(0^n, \pi + 1)$.
   (b) If $\pi = p' - 1$ then Return $(S(S(0^n)), p')$.
   (c) If $\pi \geq p'$ then Return $(\boldsymbol{u}, \pi)$.
4. Let $\boldsymbol{u}' = S(\boldsymbol{u})$, $p' = V(\boldsymbol{u}')$, and $p = V(\boldsymbol{u})$.
5. If $P(\boldsymbol{u}') \neq \boldsymbol{u}$ or $\boldsymbol{u}' = \boldsymbol{u}$ then Return $(\boldsymbol{u}, \pi)$
6. If $\pi = p = p'$ or ($\pi = p$ and $p' = p + 1$) or ($\pi = p$ and $p' = p - 1$) then Return $(\boldsymbol{u}', p')$.
7. If $\pi < p \leq p'$ or $p \leq p' \leq \pi$ or $\pi > p \geq p'$ or $p \geq p' \geq \pi$ then Return $(\boldsymbol{u}, \pi)$
8. If $p < p'$, then if $p \leq \pi < p' - 1$ then Return $(\boldsymbol{u}, \pi + 1)$. If $\pi = p' - 1$ then Return $(\boldsymbol{u}', p')$.
9. If $p > p'$, then if $p \geq \pi > p' + 1$ then Return $(\boldsymbol{u}, \pi - 1)$. If $\pi = p' + 1$ then Return $(\boldsymbol{u}', p')$.
```

Procedure $P'(\boldsymbol{u}, \pi)$.

```
1. If ($\boldsymbol{u} = 0^n$ and $\pi = 1$) or $\boldsymbol{u} = S(0^n)$ then Return $(\boldsymbol{u}, \pi)$.
2. If $\boldsymbol{u} = 0^n$, then
   (a) If $\pi = 0$ then Return $0^k$.
   (b) If $\pi < V(S(S(0^n)))$ and $\pi \notin \{1, 2\}$ then Return $(0^n, \pi - 1)$.
   (c) If $\pi < V(S(S(0^n)))$ and $\pi = 2$ then Return $0^k$.
3. If $\boldsymbol{u} = S(S(0^n))$ and $\pi = V(S(S(0^n))$ then
   (a) If $\pi = 2$ then Return $(0^n, 0)$, else Return $(0^n, \pi - 1)$.
4. If $\pi = V(\boldsymbol{u})$ then
   (a) Let $\boldsymbol{u}' = P(\boldsymbol{u})$, $p' = V(\boldsymbol{u}')$, and $p = V(\boldsymbol{u})$.
   (b) If $S(\boldsymbol{u}') \neq \boldsymbol{u}$ or $\boldsymbol{u}' = \boldsymbol{u}$ then Return $(\boldsymbol{u}, \pi)$
   (c) If $p = p'$ then Return $(\boldsymbol{u}', p')$
```

```
        (d) If $p' < p$ then Return $(\boldsymbol{u}', p-1)$ else Return $(\boldsymbol{u}', p+1)$
  5. Else % when $\pi \neq V(\boldsymbol{u})$
        (a) Let $\boldsymbol{u}' = S(\boldsymbol{u})$, $p' = V(\boldsymbol{u}')$, and $p = V(\boldsymbol{u})$
        (b) If $P(\boldsymbol{u}') \neq \boldsymbol{u}$ or $\boldsymbol{u}' = \boldsymbol{u}$ then Return $(\boldsymbol{u}, \pi)$
        (c) If $p' = p$ or $\pi < p < p'$ or $p < p' \leq \pi$ or $\pi > p > p'$ or $p > p' \geq \pi$ then Return $(\boldsymbol{u}, \pi)$
        (d) If $p < p'$, then If $p < \pi \leq p' - 1$ then Return $(\boldsymbol{u}, \pi - 1)$.
        (e) If $p > p'$, then if $p > \pi \geq p' + 1$ then Return $(\boldsymbol{u}, \pi + 1)$.
```

As mentioned before, the intuition for the potential function procedure V' is to return zero for self loops, return 1 for 0^k, and return the number specified by the lowest m bits for the rest.

Procedure $V'(\boldsymbol{u}, \pi)$. Let $\boldsymbol{x} = (\boldsymbol{u}, \pi)$ for notational convenience.

```
  1. If $\boldsymbol{x} = 0^k$, then Return 1.
  2. If $S'(\boldsymbol{x}) = \boldsymbol{x}$ and $P'(\boldsymbol{x}) = \boldsymbol{x}$ then Return 0.
  3. If $S'(\boldsymbol{x}) \neq \boldsymbol{x}$ or $P'(\boldsymbol{x}) \neq \boldsymbol{x}$ then Return $\pi$.
```

The fact that procedures S', P' and V' give a valid EndOfMeteredLine instance follows from construction.

Lemma 50. *Procedures S', P' and V' gives a valid* EndOfMeteredLine *instance on vertex set $\{0,1\}^k$, where $k = m + n$ and $V' : \{0,1\}^k \rightarrow \{0, \ldots, 2^k - 1\}$.*

The next three lemmas shows how to construct a solution of EndOfPotentialLine instance $\mathcal{I}$ from a type T1, T2, or T3 solution of constructed EndOfMeteredLine instance $\mathcal{I}'$. The basic idea for next lemma, which handles type T1 solutions, is that we never create spurious end or start of a path.

Lemma 51. *Let $\boldsymbol{x} = (\boldsymbol{u}, \pi)$ be a type T1 solution of constructed* EndOfMeteredLine *instance $\mathcal{I}'$. Then $\boldsymbol{u}$ is a type (R1) solution of the given* EndOfPotentialLine *instance $\mathcal{I}$.*

Proof. Let $\Delta = 2^m - 1$. In $\mathcal{I}'$, clearly $(0^n, \pi)$ for any $\pi \in 1, \ldots, \Delta$ is not a start or end of a path, and $(0^n, 0)$ is not an end of a path. Therefore, $\boldsymbol{u} \neq 0^n$. Since $(S(0^n), \pi), \forall \pi \in \{0, \ldots, \Delta\}$ are self loops, $\boldsymbol{u} \neq S(0^n)$.

If to the contrary, $S(P(\boldsymbol{u})) = \boldsymbol{u}$ and $P(S(\boldsymbol{u})) = \boldsymbol{u}$. If $S(\boldsymbol{u}) = \boldsymbol{u} = P(\boldsymbol{u})$ then $(\boldsymbol{u}, \pi)$, $\forall \pi \in \{0, \ldots, \Delta\}$ are self loops, a contradiction.

For the remaining cases, let $P'(S'(\boldsymbol{x})) \neq \boldsymbol{x}$, and let $\boldsymbol{u}' = S(\boldsymbol{u})$. . There is a valid edge from $\boldsymbol{u}$ to $\boldsymbol{u}'$ in $\mathcal{I}$. Then we will create valid edges from $(\boldsymbol{u}, V(\boldsymbol{u}))$ to $(S(\boldsymbol{u}), V(S(\boldsymbol{u}))$ with appropriately changing second coordinates. The rest of $(\boldsymbol{u}, .)$ are self loops, a contradiction.

Similar argument follows for the case when $S'(P'(\boldsymbol{x})) \neq \boldsymbol{x}$. □

The basic idea behind the next lemma is that a T2 type solution in $\mathcal{I}'$ has potential 1. Therefore, it is surely not a self loop. Then it is either an end of a path or near an end of a path, or else near a potential violation.

Lemma 52. *Let $\boldsymbol{x} = (\boldsymbol{u}, \pi)$ be a type T2 solution of $\mathcal{I}'$. Either $\boldsymbol{u} \neq 0^n$ is start of a path in $\mathcal{I}$ (type (R1) solution), or $P(\boldsymbol{u})$ is an (R1) or (R2) type solution in $\mathcal{I}$, or $P(P(\boldsymbol{u}))$ is an (R2) type solution in $\mathcal{I}$.*

Proof. Clearly $\boldsymbol{u} \neq 0^n$, and $\boldsymbol{x}$ is not a self loop, i.e., it is not a dummy vertex with irrelevant value of π. Further, $\pi = 1$. If $\boldsymbol{u}$ is a start or end of a path in $\mathcal{I}$ then done.

Otherwise, if $V(P(\boldsymbol{u})) > \pi$ then we have $V(\boldsymbol{u}) \leq \pi$ and hence $V(\boldsymbol{u}) - V(P(\boldsymbol{u})) \leq 0$ giving $P(\boldsymbol{u})$ as an (R2) type solution of $\mathcal{I}$. If $V(P(\boldsymbol{u})) < \pi = 1$ then $V(P(\boldsymbol{u})) = 0$. Since potential can not go below zero, either $P(\boldsymbol{u})$ is an end of a path, or for $\boldsymbol{u}'' = P(P(\boldsymbol{u}))$ and $\boldsymbol{u}' = P(\boldsymbol{u})$ we have $\boldsymbol{u}' = S(\boldsymbol{u}'')$ and $V(\boldsymbol{u}') - V(\boldsymbol{u}'') \leq 0$, giving $\boldsymbol{u}''$ as a type (R2) solution of $\mathcal{I}$. □

At a type T3 solution of $\mathcal{I}'$ potential is strictly positive, hence these solutions are not self loops. If they correspond to potential violation in $\mathcal{I}$ then we get a type (R2) solution. But this may not be the case, if we made S' or P' self pointing due to end or start of a path respectively. In that case, we get a type (R1) solution. The next lemma formalizes this intuition.

Lemma 53. *Let $\boldsymbol{x} = (\boldsymbol{u}, \pi)$ be a type T3 solution of $\mathcal{I}'$. If $\boldsymbol{x}$ is a start or end of a path in $\mathcal{I}'$ then $\boldsymbol{u}$ gives a type (R1) solution in $\mathcal{I}$. Otherwise $\boldsymbol{u}$ gives a type (R2) solution of $\mathcal{I}$.*

Proof. Since $V'(\boldsymbol{x}) > 0$, it is not a self loop and hence is not dummy, and $\boldsymbol{u} \neq 0^n$. If $\boldsymbol{u}$ is start or end of a path then $\boldsymbol{u}$ is a type (R1) solution of $\mathcal{I}$. Otherwise, there are valid incoming and outgoing edges at $\boldsymbol{u}$, therefore so at $\boldsymbol{x}$.

If $V((S(\boldsymbol{x})) - V(\boldsymbol{x}) \neq 1$, then since potential either remains the same or increases or decreases exactly by one on edges of $\mathcal{I}'$, it must be the case that $V(S(\boldsymbol{x})) - V(\boldsymbol{x}) \leq 0$. This is possible only when $V(S(\boldsymbol{u})) \leq V(\boldsymbol{u})$. Since $\boldsymbol{u}$ is not an end of a path we do have $S(\boldsymbol{u}) \neq \boldsymbol{u}$ and $P(S(\boldsymbol{u})) = \boldsymbol{u}$. Thus, $\boldsymbol{u}$ is a type T2 solution of $\mathcal{I}$.

If $V((\boldsymbol{x}) - V(P(\boldsymbol{x})) \neq 1$, then by the same argument we get that for $(\boldsymbol{u}'', \pi'') = P(\boldsymbol{u})$, $\boldsymbol{u}''$ is a type (R2) solution of $\mathcal{I}$. □

Our main theorem follows using Lemmas 50, 51, 52, and 53.

Theorem 54. *An instance of* EndOfPotentialLine *can be reduced to an instance of* EndOfMeteredLine *in polynomial time such that a solution of the former can be constructed in a linear time from the solution of the latter.*

B Proofs for Section 3.1: OPDC to UEOPL

B.1 Proof of Lemma 16

Throughout this proof, we will fix $\mathcal{D} = (D_i)_{i=1,\ldots,d}$ to be the direction functions, and $P = P(k_1, k_2, \ldots, k_d)$ to be the set of points used in the OPDC instance. We will produce a UniqueForwardEOPL instance $L = (S, V)$.

The circuit S. A vertex of the line is a tuple $(p_0, p_1, p_2, \ldots, p_d)$, where each $p_i \in P \cup \{-\}$ is either a point or a special symbol, $-$, that is used to indicate an unused element. We use $\mathsf{vert} = (P \cup \{-\})^{d+1}$ to denote the set of possible vertices. Only some of the tuples are valid encodings for a vertex. To be valid, a vertex $(p_0, p_1, p_2, \ldots, p_d)$ must obey the following rules:

1. If $p_i \neq -$, then $D_j(p_i) = \mathsf{zero}$ for all $j \leq i$. This means that if p_i is a point, then it must be a point on the i-surface.
2. If $p_i \neq -$, then we must have $D_{i+1}(p_i) \neq \mathsf{down}$.
3. If $p_i \neq -$ and $p_j = -$ and $i < j$, then we must have $(p_i)_{j+1} = 0$.

4. If $p_i \neq -$ and $p_j \neq -$ and $i < j$, then we must have $(p_i)_{j+1} = (p_j)_{j+1} + 1$.

We define the function IsVertex : $\mathsf{vert} \to \{\mathtt{true}, \mathtt{false}\}$ that determines whether a given $v \in \mathsf{vert}$ is a valid encoding of a vertex, by following the rules laid out above. This can clearly be computed in polynomial time.

The initial vertex, which will be mapped to the bit-string 0^n, will be $(p_{\text{init}}, -, \dots, -)$, where $p_{\text{init}} = (0, 0, \dots, 0)$ is the all zeros point in P.

Given a vertex encoding $v = (p_0, p_1, p_2, \dots, p_d) \in \mathsf{vert}$, the circuit S carries out the following operations. If IsVertex(v) is $\mathtt{false}$, then $S(v) = v$, indicating that v is indeed not a vertex. Otherwise, we use the following set of rules to determine the successor of v. Let i be the smallest index such that $p_i \neq -$.

1. If $i = d$ then our vertex has the form $v = (-, \dots, -, p_d)$, and p_d is on the d-surface, meaning that it is a solution to the discrete contraction map. So we set $S(v) = v$ to ensure that this is a solution.

2. If $D_{i+1}(p_i) = \mathsf{zero}$, then we define $S(v) = v'$ where

$$v'_j = \begin{cases} - & \text{if } j < i+1, \\ p_i & \text{if } j = i+1, \\ p_j & \text{if } j > i+1. \end{cases}$$

   This operation overwrites the point in position $i + 1$ with p_i, and sets position i to $-$. All other components of v are unchanged.

3. If $D_{i+1}(p_i) \neq \mathsf{zero}$ and $i > 0$ then let q be the point such that

$$(q)_j = \begin{cases} 0 & \text{if } j < i+1, \\ (p_i)_{i+1} + 1 & \text{if } j = i+1, \\ (p_i)_j & \text{if } j > i+1. \end{cases}$$

   (a) If q is a point in P, then we define $S(v) = (q, p_1, p_2, \dots, p_d)$.

   (b) Otherwise, we must have that $(p_i)_{i+1} = k_{i+1}$, meaning that p_i is the last point of the grid. This means that we have a solution of type (OV3), since the fact that IsVertex$(v) = \mathtt{true}$ implies that $D_{i+1}(p_i) = \mathsf{up}$. So we set $S(v) = v$.

4. If $D_{i+1}(p_i) \neq \mathsf{zero}$ and $i = 0$ then let q be the point such that $(q)_j = (p_0)_j$ for all $j > 1$, and $(q)_1 = (p_0)_1 + 1$.

   (a) If q is in the point set P, then we define $S(v) = (q, p_1, p_2, \dots, p_d)$.

   (b) If q is not in P, then we again have a solution of type (OV3), since $(p_0)_1 = k_1$, and $D_1(p_0) = \mathsf{up}$ from the fact that IsVertex$(v) = \mathtt{true}$. So we set $S(v) = v$.

The potential function. To define the potential function, we first define an intuitive potential that uses a tuple of values ordered lexicographically, and then translate this into a circuit V that produces integers. We'll call the tuple of values the lexicographic potential associated with a vertex v, and denote it using LexPot(v). To define the lexicographic potential, we'll need to introduce an auxiliary function Potential : $(P \cup \{-\}) \times \{0, \dots, d+1\} \to \mathbb{Z}$ given by

$$\text{Potential}(p, i) = \begin{cases} (p)_{i+1} + 1 & \text{if } p \neq -, \\ 0 & \text{otherwise.} \end{cases}$$

The lexicographic potential of $v = (p_0, p_1, \dots, p_d) \in \mathsf{vert}$ is the following:

$$\text{LexPot}(v) = (\text{Potential}(p_0, 0), \text{Potential}(p_1, 1), \dots, \text{Potential}(p_{d-1}, d-1)) .$$

Note that the $\text{LexPot}(v) \in \mathbb{Z}^d$, since the definition ignores p_d.

Let $\prec_d$ be the ordering on tuples from $\mathbb{Z}^d$ where they are compared lexicographically from right to left, so that $(0,0) \prec_2 (1,0) \prec_2 (0,1) \prec_2 (1,1)$, for example. Our potential function will be defined by the tuples given by LexPot and the order $\prec_{d+1}$. We omit the subscript from $\prec$ whenever it is clear from the context.

Given a vertex $v = (p_0, p_1, \dots, p_d) \in \mathsf{vert}$, let $\text{LexPot}(v) = (l_0, \dots, l_{d-1})$. To translate from lexicographically ordered tuples to integers in a way that preserves the ordering, we pick some integer k such that $k > k_i$ for all i, meaning that k is larger than the grid-width used in every dimension, which implies that $l_j < k$ for all j. We now take a weighted sum of the coordinates of $\text{LexPot}(v)$ where the weight for coordinate i is k^i, so that the ith coordinate dominates coordinates 0 through $i - 1$. The final potential value $V : \mathsf{vert} \to \mathbb{Z}$ is then given by

$$V(v) = \sum_{i=0}^{d-1} k^i l_i,$$

which can be easily computed in polynomial time. This completes the definition of the UniqueForwardEOPL problem (S, V).

Lemma 55. *Every solution of the* UniqueForwardEOPL *instance* (S, V) *can be used to find a solution of the* OPDC *instance given by* $\mathcal{D}$ *and* P*. Furthermore, solutions of type* (O1) *are only ever mapped onto solutions of type* (UF1)*.*

Proof. We begin by considering solutions of type (UF1). Let $x = (p_0, p_1, \dots, p_d)$ and let $y = S(x)$, and suppose that x is a (UF1) solution. This means that $S(x) \neq x$ and either $S(y) = y$ or $V(y) \leq V(x)$. We first suppose that $S(y) = y$, and note that in this case we must have $\text{IsVertex}(x) = \texttt{true}$ while $\text{IsVertex}(y) = \texttt{false}$. We have the following cases based on the rule used to determine $S(x)$.

1. If $S(x)$ is determined by the first rule in the definition of S, then this means that $p_d \neq -$. Since $\text{IsVertex}(x) = \texttt{true}$, this means that $D_i(p_d) = \mathsf{zero}$ for all i, which means that p_d is a solution of type (O1).

2. If $S(x)$ is determined by the second rule in the definition of S, then there are two cases. Let i be the smallest index such that $p_i \neq -$.

   (a) If $D_{i+2}(p_i) = \mathsf{down}$ then we have a solution of type (OV2) or (OV3).

      i. If $p_{i+2} \neq -$ then p_i and p_{i+2} are solution of type (OV2). Specifically, this holds because $D_{i+2}(p_{i+2}) = \mathsf{up}$ while $D_{i+2}(p_i) = \mathsf{down}$, and $(p_i)_{i+2} = (p_{i+2})_{i+2} + 1$ holds because $\text{IsVertex}(x) = \texttt{true}$. Moreover, $D_j(p_i) = D_j(p_{i+2}) = \mathsf{zero}$, for all $j < i + 2$, where in particular the fact that $D_{i+1}(p_i) = \mathsf{zero}$ is given by the fact that we are in the second case of the definition of S.

      ii. If $p_{i+2} = -$ then this means that $(p_i)_{i+2} = 0$. Since $D_{i+2}(p_i) = \mathsf{down}$ this gives us a solution of type (OV3).

   (b) If $D_{i+2}(p_i) \neq \texttt{down}$, then we argue that this case is impossible. Specifically, we will show that IsVertex$(y) = \texttt{true}$, meaning that $S(y) \neq y$. To do this, we will prove that the four conditions of IsVertex hold for y. Note that y differs from x only in positions i and $i+1$, and that position i of y is $-$. So we only need to consider the conditions imposed by IsVertex when the point p_i is placed in position $i+1$.

      i. The first condition of IsVertex is that p_i should be on the $(i+1)$-surface, which is true because the second rule of S explicitly checks that $D_{i+1}(p_i) = \texttt{zero}$, while the fact that IsVertex$(x) = \texttt{true}$ guarantees that $D_j(p_i) = \texttt{zero}$ for all $j < i+1$.
      ii. The second condition requires that $D_{i+2}(p_i) \neq \texttt{down}$, which is true by assumption.
      iii. Every constraint imposed by the third and fourth conditions also holds for p_i in x, and so the fact that IsVertex$(x) = \texttt{true}$ implies that these conditions hold for y.

3. If $S(x)$ is determined by the third rule defining S, then we have two cases. Since the third rule was used, we know that $y = (q, p_1, p_2, \dots, p_d)$, with the definition of q being given in the third rule.

   (a) If q is not in P, then we have a solution of type (OV3), as described in the algorithm for S.
   (b) If $q \in P$ and $D_1(q) = \texttt{down}$, then we have a solution of type (OV3), since $q_1 = 0$ by definition.
   (c) If $q \in P$ and $D_1(q) \neq \texttt{down}$ and $q \in P$, then we argue that the case is impossible, and we prove this by showing that IsVertex$(y) = \texttt{true}$. Note that y differs from x only in the position occupied by q, and so this is the only point for which we need to prove the conditions, since all the other points satisfy the conditions by the fact that IsVertex$(x) = \texttt{true}$.

      i. The first requirement of IsVertex(y) holds trivially, since the only new requirement is that q is on the 0-surface, and every point is on the 0-surface by definition.
      ii. The second requirement is that $D_1(q) \neq \texttt{down}$, which is true by assumption.
      iii. The third and fourth conditions place constraints on certain coordinates of q. For coordinates $j < i$, the third condition requires that $q_j = 0$, which is true by definition, while the fourth condition is inapplicable. For coordinates $j \geq i$, the constraints imposed by the third and fourth conditions hold because $q_j = (p_i)_j$ in these coordinates, and p_i also satisfies these constraints.

4. If $S(x)$ is determined by the fourth rule, then we have two cases. Let $y = (q, p_1, p_2, \dots, p_d)$ be the value of y produced by the fourth rule, where the definition of q is given in that rule.

   (a) If q is not in P, then we have a solution of type (OV3), as described in the algorithm for S.
   (b) If $q \in P$ and $D_1(q) = \texttt{down}$ then we have a solution of type (OV2). Specifically, the points p_0 and q provide the violation since $(q)_1 = (p_0)_1 + 1$, while $D_1(p_0) = \texttt{up}$ and $D_1(q) = \texttt{down}$. The fact that both q and p_0 belong to the same 1-slice is guaranteed by the definition of q.
   (c) If $q \in P$ and $D_1(q) \neq \texttt{down}$ then we again argue that IsVertex$(y) = \texttt{true}$, making this case impossible. The reasoning is the same as the reasoning used in case 3b.

We now proceed to the case where we have a solution $x = (p_0, p_1, \dots, p_d)$ of type (UF1) and the vertex $y = S(x)$ satisfies $S(y) \neq y$. In this case, we must have $V(y) \leq V(x)$. We argue that this is impossible, and again we will do a case analysis based on the rule used to determine the output of the circuit S.

1. If $S(x)$ is determined by the first rule then we have $\text{IsVertex}(y) = \texttt{false}$, which is not possible in this case.

2. If $S(x)$ is determined by the second rule, then we can prove that $V(y) > V(x)$. This is because y differs from x only in positions i and $i+1$, and because $(p_i)_{i+1} = (p_{i+1})_{i+1} + 1$ by the fourth rule of $\text{IsVertex}(x)$. Hence

   $$k^i \cdot \text{Potential}(-, i) + k^{i+1} \cdot \text{Potential}(p_i, i+1) > k^i \cdot \text{Potential}(p_i, i) + k^{i+1} \cdot \text{Potential}(p_{i+1}, i+1),$$

   where we are using the fact that $k^{i+1} > k^i \cdot \text{Potential}(p_i, i)$. Thus, $V(y) > V(x)$.

3. If $S(x)$ is determined by the third rule, then note that we must have $q \in P$. We again argue that $V(y) > V(x)$. Specifically, observe that $V(y) = V(x) + \text{Potential}(q, 0) = V(x) + 1$.

4. If $S(x)$ is determined by the fourth rule, then note that we must have $q \in P$, and we also have $V(y) > V(x)$. Specifically

   $$\begin{aligned} V(y) &= V(x) - \text{Potential}(p_0, 0) + \text{Potential}(q, 0) \\ &= V(x) - (p_0)_1 + q_1 \\ &= V(x) + 1. \end{aligned}$$

Finally, we move to the case where we have a solution of type (UFV1). In this case we have two vertices $x = (p_0, p_1, \dots, p_d)$ and $y = (q_0, q_1, \dots, q_d)$ for which $\text{IsVertex}(x) = \text{IsVertex}(y) = \texttt{true}$, and for which $V(x) \leq V(y) < V(S(x))$. We will once again perform a case analysis over the possible cases of S.

1. $S(x)$ cannot be determined by the first case in the definition of S, because that case only applies when $\text{IsVertex}(y) = \texttt{false}$.

2. If $S(x)$ is determined by the second case in the definition of S, then we observe that we have $\text{LexPot}(x) = (0, \dots, 0, v_i, v_{i+1}, \dots, v_d)$ and $\text{LexPot}(S(x)) = (0, \dots, 0, 0, v_{i+1} + 1, v_{i+2} \dots, v_d)$, ie., the ith element of $\text{LexPot}(x)$ is replaced by 0, and the $(i+1)$th element is replaced by $v_{i+1}+1$. Note also that elements $i+2$ through d of $\text{LexPot}(S(x))$ agree with the corresponding elements of $\text{LexPot}(x)$.

   Since we have $V(x) \leq V(y) < V(S(x))$ this means that $\text{LexPot}(x) \prec \text{LexPot}(y) \prec \text{LexPot}(S(x))$. Hence, $\text{LexPot}(y)$ must have the form $(v'_0, v'_1, \dots, v'_i, v_{i+1}, v_{i+2}, \dots, v_d)$, meaning that it agrees with $\text{LexPot}(x)$ and $\text{LexPot}(S(x))$ on elements $i+1$ through d. Furthermore, we must have $v'_i \geq v_i$.

   Let j be the largest index satisfying $j \leq i$ and $v'_j \neq v_j$. Note that such a j must exist, since otherwise we would have $x = y$, which would contradict the fact that $x \neq y$ in any solution of type UFV1. We claim that p_j and q_j form a solution of type (OV1).

   Let s be the j-slice that satisfies $s_l = *$ for all $l \leq j$ and $s_l = (p_j)_l$ for all $l > j$. The point p_j lies in the slice s by definition. We claim that q_j also lies in this slice, which follows from the fact that $v_l = v'_l$ for all $l > j$, combined with the third and fourth properties of $\text{IsVertex}(x)$ and $\text{IsVertex}(y)$. Note also that $(p_j)_j \neq (q_j)_j$, and hence $p_j \neq q_j$.

Finally, the first property of $\text{IsVertex}(x)$ and $\text{IsVertex}(y)$ imply that $D_l(p_j) = \mathsf{zero}$ and $D_l(q_j) = \mathsf{zero}$ for all $l \leq j$. So p_j and q_j are two distinct fixpoint of the j-slice s, meaning that we have a solution of type (OV1).

3. We claim that $S(x)$ cannot be determined by the third rule in the definition of S. In this case we have $\text{LexPot}(x) = (0, v_1, v_2, \ldots, v_d)$, since the case is only applicable when $p_0 = -$. Observe that $\text{LexPot}(S(x)) = (1, v_1, v_2, \ldots, v_d)$, and that there is no possible tuple t that satisfies $\text{LexPot}(x) \prec t \prec \text{LexPot}(S(x))$. This means that we must have $V(y) = V(x)$, but this is only possible if $y = x$, due to the constraints placed by the third and fourth properties of IsVertex. Hence this case is not possible, since $y = x$ is specifically ruled out in a solution of type UVF1.

4. For similar reasons, we claim that $S(x)$ cannot be determined by the fourth rule. In this case we have $\text{LexPot}(x) = (v_0, v_1, v_2, \ldots, v_d)$, and $\text{LexPot}(S(x)) = (v_0 + 1, v_1, v_2, \ldots, v_d)$, which again means that there cannot be a tuple t satisfying $\text{LexPot}(x) \prec t \prec \text{LexPot}(S(x))$. Hence we would have $V(x) = V(y)$ and $x = y$ as in the previous case, which is impossible.

To complete the proof, we note that solutions of type (O1) are only ever mapped onto solutions of type (UF1). □

This proves that our reduction from OPDC to UniqueForwardEOPL is correct. The fact that the reduction is promise-preserving follows from the fact that all solutions of type (O1) are mapped onto solutions of type (UF1). Hence, if we promise that there are no violations in the OPDC instance, then the resulting UFEOPL instance can only have solutions of type (UF1). This completes the proof of Lemma 16.

B.2 Proof of Lemma 18

We provide a polynomial-time promise-preserving reduction from UniqueForwardEOPL to UniqueForwardEOPL+1. This reduction uses essentially the same idea as the reduction from EndOfPotentialLine to EndOfMeteredLine given in Theorem 10.

Let $L = (S, V)$ be an instance of UniqueForwardEOPL, and let n be the bit-length of the strings used to represent vertices in L. If x is a vertex in L such that $V(S(x)) > V(x) + 1$, then we introduce new vertices between x and $S(x)$, each of which increases in potential by 1.

Formally, our UniqueForwardEOPL+1 instance will be denoted as $L' = (S', V')$. Each vertex in this instance will be a pair (v, i), where v is a vertex from L, and i is an integer between 0 and 2^n. The circuit S' is defined in the following way. For each vertex (v, i) we use the following algorithm.

1. If $S(v) = v$, then $S'(v, i) = (v, i)$, meaning that if v is not a vertex in L, then (v, i) is not a vertex in L' for all i.
2. If $S(v) \neq v$ and $V(S(v)) > V(v) + i + 1$, then $S'(v, i) = (v, i + 1)$.
3. If $S(v) \neq v$ and $V(S(v)) = V(v) + i + 1$, then $S'(v, i) = (S(v), 0)$.
4. If $S(v) \neq v$ and $V(S(v)) < V(v) + i + 1$, then $S'(v, i) = (v, i)$.

The last three conditions add a new sequence of vertices between any edge (x, y) where $V(y) > V(y) + 1$. Specifically, this sequence is

$$(x, 0) \to (x, 1) \to \cdots \to (x, V(y) - V(x) - 1) \to (y, 0).$$

Any pair (v, i) that is not used in such a sequence has $S'(v, i) = (v, i)$, meaning that it is not a vertex.

The potential function is defined as follows. For each vertex (v, i), we use the following algorithm.

1. If $S(v, i) = (v, i)$, then the potential of (v, i) is irrelevant.

2. Otherwise, we set $V'(v, i) = V(v) + i$.

This completes the specification of the reduction.

Clearly the reduction can be carried out in polynomial time. We now prove that this is a promise-preserving reduction.

Lemma 56. *Every solution of L' can be mapped to a solution of L. Furthermore, solutions of type (UF1) of L are only ever mapped onto solutions of type (UFP1) in L'.*

Proof. We start by considering solutions of type (UFP1). Let $x = (v, i)$ be a vertex, and let $y = (u, j)$ be the vertex with $y = S(x)$. In a solution of type (UFP1) we have that $S'(x) \neq x$ and either $S'(y) = y$ or $V'(y) \neq V'(x) + 1$. Hence, there are two cases to consider.

1. If $S'(y) = y$ then we must have that $u = S(v)$ and $j = 0$, since if (v, i) is a vertex, then S' will always either give $(v, i + 1)$ or $(S(v), 0)$, and it suffices to note that if $S'(v, i) = (v, i + 1)$ then $S'(v, i + 1) \neq (v, i + 1)$.

   So, we have $y = (S(v), 0)$, and $S'(y) = y$. This can only occur in the case where either $S(S(v)) = S(v)$, or $V(S(S(v))) < V(S(v))$. Both of these cases yield a solution of type (UF1) for L.

2. We claim that the case $V'(y) \neq V'(x) + 1$ is not possible. Since we have already dealt with the case where $S'(y) = y$, we can assume that y is a vertex. Note that $S'(x)$ cannot be determined by cases 1 or 4 of the algorithm, since in those cases x would not be a vertex. If $S'(x)$ is determined by case 2 of the algorithm, then we have that $V'(y) = V'(x) + 1$ by definition. If $S'(x)$ is determined by case 3 of the algorithm, then we have

   $$V'(x) = V(v) + (V(u) - V(v) - 1) = V'(y) - 1$$

   Hence, this case is impossible by construction.

Thus, we have dealt with all possible solutions of type (UFP1).

We now consider a violation of type (UFPV1). In this case we have two vertices $x = (v, i)$ and $y = (u, j)$ such that $x \neq y$, $x \neq S'(x)$, $y \neq S'(y)$, and $V'(x) = V'(y)$. We claim that this gives us a solution of type (UFV1). If $i = j$ then $V(v) = V(u)$, and so we have a solution of type (UFV1) in L. Assume, without loss of generality, that $i \leq j$. We have

$$V(v) + i = V'(v, i) = V'(u, j) = V(u) + j,$$

which implies that $V(u) \leq V(v)$. Furthermore, we have that

$$V(S(u)) > V(u) + j \geq V(v) + i \geq V(v),$$

where the first inequality arises from the fact that (u, j) is a valid vertex in L'. Hence, we have shown that $V(u) \leq V(v) < V(S(u))$, which is exactly a solution of type (UFV1) in L. □

The lemma implies that our reduction is correct. The fact that it is promise-preserving follows from the fact that solutions of type (UF1) are only ever mapped onto solutions of type (UFP1). Hence, if it is promised that L only has (UF1) solutions, then L' must only have (UFP1) solutions. This completes the proof of Lemma 18.

B.3 Proof of Lemma 20

The reduction of Hubáček and Yogev [45] relies on the *pebbling game* technique that was first applied by Bitansky et al [4]. Let $L = (S, W, x_s, T)$ be an SinkOfVerifiableLine instance. The pebbling game is played by placing *pebbles* on the vertices of this instance according to the following rules.

- A pebble may be placed or removed from the starting vertex x_s at any time.
- A pebble may be placed or removed on a vertex $x \neq x_s$ if and only if there is a pebble on a vertex y with $S(y) = x$.

Given n pebbles, how far can we move along the line by following these rules? The answer is that we can place a pebble on vertex $2^n - 1$ by applying the following *optimal strategy*. The strategy is recursive. In the base case, we can place a pebble on vertex $2^1 - 1 = 1$ by placing our single pebble on the successor of x_s. In the recursive step, where we have n pebbles, we use the following approach:

1. Follow the optimal strategy for $n-1$ pebbles, in order to place a pebble on vertex $2^{n-1} - 1$.
2. Place pebble n on the vertex 2^{n-1}.
3. Follow the optimal strategy for $n-1$ pebbles *backwards* in order to reclaim the first $n-1$ pebbles.
4. Follow the optimal strategy for $n-1$ pebbles forwards, but starting from the vertex 2^{n-1}. This ends by placing a pebble on vertex $2^{n-1} + 2^{n-1} - 1 = 2^n - 1$.

Step 3 above relies on the fact that the pebbling game is *reversible*, meaning that we can execute any sequence of moves backwards as well as forwards. At the beginning of Step 4, we have a single pebble on vertex 2^{n-1}, and we follow the optimal strategy again, but using 2^{n-1} as the starting point.

The reduction from UniqueForwardEOPL to UniqueEOPL. To reduce UniqueForwardEOPL to UniqueEOPL, we play the optimal strategy for the pebbling game. Note that, since that since every step of the pebbling game is reversible, this gives us a predecessor circuit. We will closely follow the reduction given by Bitansky et al [4] from SinkOfVerifiableLine to EndOfLine. Specifically, we will reduce an instance $L = (S, V)$ of UniqueForwardEOPL to an instance $L' = (S', P', V')$ of UniqueEOPL.

A vertex in L' will be a tuple of pairs $((v_1, a_1), (v_2, a_2), \dots, (v_n, a_n))$ describing the state of the pebbling game. Each v_i is a bit-string, while each a_i is either

- the special symbol $-$, implying that pebble i is not used and that the bit-string v_i should be disregarded, or
- an integer such that $a_i = V(v_i)$, meaning that pebble i is placed on the vertex v_i.

Bitansky et al [4] have produced circuits S' and P' that implement the optimal strategy of the pebbling game for pebbles encoded in this way. The only slight difference is that they reduce from SinkOfVerifiableLine, but we can apply their construction by creating the circuit W so that $W(v, a) = 1$ if and only if $V(v) = a$. We refer the reader to their work for the full definition of these two circuits.

Hubáček and Yogev [45] built upon this reduction by showing that it is possible to give a potential function V' for the resulting instance. Specifically, their potential encodes how much progress we have made along the optimal strategy, which it turns out, can be computed purely from the current configuration of the pebbles. Their construction also guarantees that the potential at each vertex always increases by 1, meaning that we have $V(S(x)) = V(x) + 1$ whenever $S(x)$ and x are both vertices. We refer the reader to their work for the full definition of the circuit V'.

Violations. So far, we have a reduction from the promise version of UniqueForwardEOPL to UniqueEOPL, which entirely utilizes prior work. Specifically, every solution of type (U1) in L' will map back to a solution of type (UFP1) in L.

Our contribution is to handle the violations, thereby giving a promise-preserving reduction from the non-promise version of UniqueForwardEOPL to the non-promise version of UniqueEOPL.

Lemma 57. *Every violation in L' can be mapped back to a violation of L.*

Proof. There are three types of violation in L'.

1. Violations of type (UV1), which are edges where the potential decreases, are not possible, since the reduction of Hubáček and Yogev ensures that $V'(S'(x)) = V'(x) + 1$ whenever x and $S'(x)$ are both vertices.

2. In violations of type (UV2) we have a vertex $((v_1, a_1), (v_2, a_2), \dots, (v_n, a_n))$ that is the start of a second line. This means that, for some reason, we are not able to execute the optimal strategy backwards from this vertex. There are two possibilities

   (a) The optimal strategy needs to place a pebble on the successor of some vertex v_i, but it cannot because v_i is the end of a line. This means that either $S(v_i) = v_i$ or that $V(S(v_i)) \neq V(v_i) + 1$, and in either case we have a solution of type (UFP1) for L.

   (b) The optimal strategy needs to remove the pebble v_i, but it cannot, because it does not have a pebble on a vertex u with $S(u) = u$. By construction, there will be some pebble v_j with $a_j = a_i - 1$, but in this case we have $S(v_j) \neq v_i$. This means that we have two lines, and specifically we have that v_i and $S(v_j)$ are two vertices with the same potential, since $V(v_j) = a_j$ and $V(S(v_j)) = V(v_j) + 1$. This gives us a solution of type (UFPV1).

3. In violations of type (UV3) we have two distinct vertices $x = ((v_1, a_1), (v_2, a_2), \dots, (v_n, a_n))$ and $y = ((u_1, b_1), (u_2, b_2), \dots, (u_n, b_n))$ with $V'(x) \leq V'(y) < V'(S'(x))$. Since the reduction ensures that $V'(S'(x)) = V'(x) + 1$ this means that x and y have the same potential. The reduction of Hubáček and Yogev ensures that, if two vertices have the same potential, then they refer to the same step of the optimal strategy, meaning that $a_i = b_i$ for all i. This means that any pair of vertices v_i and u_i with $a_i \neq -$ is a pair of vertices with $V(v_i) = V(u_i)$, and so a solution of type (UFPV1). To see that such a pair must exist, it suffices to note that the only vertex with $a_i = -$ for all i is the start of the line, and there cannot be two distinct vertices with this property.

□

The lemma above proves that the reduction is correct, but it does not directly prove that it is promise-preserving. Specifically, in case 2a of the proof we show that some violations of (UV2) are mapped back to solutions of type (UFP1). This, however, is not a problem, because we can argue that case 2a of the proof can only occur if there is more than one line in L'.

Specifically, if we are at some vertex $x = ((v_1, a_1), (v_2, a_2), \ldots, (v_n, a_n))$ and $P(x)$ needs to place a pebble on $S(v_i)$, then v_i cannot be the furthest point in the pebble configuration, meaning that there is some v_j with $a_j \neq -$ and $a_j > a_i$. This can be verified by inspecting the recursive definition of the optimal strategy. But note that if v_i is the end of a line, and v_j is a vertex with $V(v_j) > V(v_i)$, then L' must contain more than one line.

This allows us to argue that the reduction is promise-preserving, since if L is promised to have no violations, then it must contain exactly one line, and if L contains exactly one line, then all proper solutions of L are mapped onto proper solutions of L'. Thus L' will contain no violations. This completes the proof of Lemma 20.

C Proofs for Section 3.2: UniqueEOPL to OPDC

C.1 Proof of Lemma 22

This construction is very similar to the one used in the proof of Lemma 18 given in Appendix B.2, although here we must deal with the predecessor circuit, and ensure that the end of each line has the correct potential. Let $L = (S, P, V)$ be an instance of UniqueEOPL, and let k be the bit-length of the vertices used in L.

We will create an instance $L' = (S', P', V')$ in the following way. Each vertex in L' will be a pair (v, i), where v is a vertex in L, and i is an integer satisfying $i \leq 2^n$, where $n = k + 1$. Hence, a vertex in L' can be represented by a bit-string of length $n + k$.

The circuit S' is defined as follows. Given a vertex (v, i), we execute the following algorithm.

1. If $S(v) = v$, then $S'(v, i) = (v, i)$, meaning that if v is not a vertex in L, then (v, i) is not a vertex in L' for all i.

2. If $v \neq P(S(v))$ and $V(v) + i > 2^n - 1$, then $S'(v, i) = S'(v, i)$, meaning that (v, i) is not a vertex in the instance.

3. If $v \neq P(S(v))$ and $V(v) + i = 2^n - 1$, then $S'(v, i) = 0^k$, which makes (v, i) an end of line solution.

4. If $v \neq P(S(v))$ and $V(v) + i < 2^n - 1$, then $S'(v, i) = (v, i + 1)$.

5. If $S(v) \neq v = P(S(v))$ and $V(S(v)) > V(v) + i + 1$, then $S'(v, i) = (v, i + 1)$.

6. If $S(v) \neq v = P(S(v))$ and $V(S(v)) = V(v) + i + 1$, then $S'(v, i) = (S(v), 0)$.

7. If $S(v) \neq v = P(S(v))$ and $V(S(v)) < V(v) + i + 1$, then $S'(v, i) = (v, i)$, meaning that (v, i) is not a vertex.

The last three conditions add a new sequence of vertices between any edge (x, y) where $V(y) > V(y) + 1$. Specifically, this sequence is

$$(x, 0) \to (x, 1) \to \cdots \to (x, V(y) - V(x) - 1) \to (y, 0).$$

Conditions 2 through 4 introduce a new line starting at $(x, 0)$, whenever x is the end of a line in the original instance. This line has the form

$$(x, 0) \to (x, 1) \to \cdots \to (x, 2^n - V(x) - 1).$$

where $(x, 2^n - V(x) - 1)$ is the new end of line. Observe that this line is always non-empty, since $2^n - 1 = 2^{k+1} - 1 > 2^k \geq V(x)$.

The predecessor circuit P' walks the line backwards, which is easy to construct, since we can either follow the predecessor circuit of L, or we can walk backwards along any of the lines that we have introduced. Specifically, given a vertex (v, i), we use the following algorithm to implement P'.

1. If $S(v) = v$, then the value returned by $P'(v, i)$ is irrelevant, since (v, i) is not a vertex.
2. If $v \neq P(S(v))$ and $V(v) + i > 2^n - 1$, then again the value of $P'(v, i)$ is irrelevant, since (v, i) not a vertex.
3. If $v \neq P(S(v))$ and $V(v) < V(v) + i \leq 2^n - 1$, then $P'(v, i) = (v, i - 1)$, meaning that if we are on the new line at a solution, then we walk the line backwards.
4. If $v \neq P(S(v))$ and $V(v) + i = V(v)$, then $P'(v, i) = (P(v), V(v) - V(P(v)))$, meaning that if we are at $(v, 0)$, then we move to the end of the line between $(P(v), 0)$ and $(v, 0)$.
5. If $S(v) \neq v = P(S(v))$ and $i = 0$, then $P'(v, i) = (P(v), V(v) - V(P(v)))$.
6. If $S(v) \neq v = P(S(v))$ and $V(S(v)) < V(v) + i + 1$, then the value returned by $P'(v, i)$ is irrelevant, since (v, i) is not a vertex.
7. If $S(v) \neq v = P(S(v))$ and cases 5 and 6 do not apply, then $P'(v, i) = (v, i - 1)$.

Cases 2 through 4 deal with the new line introduced at the end of each line. while cases 5 through 7 deal with the new lines introduced between the vertices $(x, 0)$ and $(S(x), 0)$.

The potential function V' is defined so that $V'(v, i) = V(v) + i$.

The two properties that we need to satisfy hold by construction for L'. Every edge is constructed so that $V'(S(x)) = V'(x) + 1$, whenever x is a vertex, and the new lines starting at vertices $(v, 0)$, where v is the end of a line in L, ensure that $V'(x) = 2^n - 1$ if and only if x is the end of a line in L'. The following lemma shows that the reduction is correct.

Lemma 58. *Every solution of L' can be mapped back to a solution of L.*

Proof. We enumerate the types of solutions in UniqueEOPL.

1. A solution of type (U1) in L' is a vertex $x = (v, i)$ such that $P'(S'(x)) \neq x$. By construction this can only occur if $P(S(v)) \neq v$, so this gives us a solution of type (U1) for L.
2. A solution of type (UV1) gives us a vertex $x = (v, i)$ where $P'(S'(x)) = x$ and $V'(S'(x)) \leq V'(x)$. By construction, this can only occur if $V(S(v)) \leq V(v)$, so we have a solution of type (UV1) for L.
3. A solution of type (UV2) gives us a vertex $x = (v, i)$ where $S'(P'(x)) \neq x$. By construction, this can only happen if $S(P(x)) \neq p$ and $i = 0$. This gives us a solution of type (UV2) for L.

4. A solution of type (UV3) gives us a pair of vertices $x = (v, i)$ and $y = (u, j)$ such that $x \neq y$, and either $V(x) = V(y)$, or $V(x) < V(y) < V(S(x))$. Note that the latter case is impossible here, since by construction we have $V(S(x)) = V(x) + 1$ whenever x is a vertex, so we must have $V(x) = V(y)$.

   If $i = j$, then $V(v) = V(u)$, and so v and u form a solution of type (UV3) for L. Otherwise, we will suppose, without loss of generality, that $i > j$, which means that $V(v) < V(u)$. Moreover, we have $V(S(v)) > V(v) + i = V(u) + j \geq V(u)$. Hence we have shown that $V(v) < V(u) < V(S(v))$, and so we have that v and u form a solution of type (UV3) for L.

□

The lemma above implies that the reduction is correct. To see that it is promise-preserving, it suffices to note that proper solutions of L are only ever mapped onto proper solutions of L'. Therefore, if it is promised that L has no violations, then L' will also have no violations. This completes the proof of Lemma 22.

C.2 Proof of Lemma 23

The proof requires us to define two polynomial-time algorithms.

Splitting lines around a vertex. We begin by defining the subline function. We will use two sub-functions that split an instance in two based on a particular vertex. Let $L = (S, P, V)$ be a UniqueEOPL instance, and v be some vertex of L.

1. We define the function FirstHalf(v, L) to return an UniqueEOPL instance (S', P', V') by removing every vertex with potential greater than or equal to $V(v)$. Specifically, the S' and P' circuits will check whether $V(x) \geq 2^{n-1}$ for each vertex x, and if so, then they will set $S'(x) = P'(x) = x$, which ensures that x is not on the line. For any vertex x with $V(x) < 2^{n-1}$, we set $S'(x) = S(x)$, $P'(x) = P(x)$ and $V'(x) = V(x)$. Note that S', P', and V' can all be produced in polynomial time if we are given (S, P, V).

2. We define the function SecondHalf(v, L) to return a UniqueEOPL instance (S', P', V') by removing every vertex with potential strictly less than $V(v)$. For the circuits S' and P', this is done in the same way as the previous case, but this time the circuits will check whether $V(x) < 2^{n-1}$. The function V' is defined so that $V'(x) = V(x) - 2^{n-1}$, thereby ensuring that the potentials are in the range $[0, 2^{n-1})$. Finally, the string 0^n is remapped to represent the vertex v, which is the start of the second half of the line. In this case, we are able to compute S', P', and V' in polynomial time if we are given (S, P, V) and the bit-string v.

We remark that, although we view these functions as intuitively splitting a line, the definitions still work if L happens to contain multiple lines. Each line in the instance will be split based on the potential of v.

The subline function. The subline function is defined recursively, based on the number of bit-strings that are given to the function. In the base case we are given a line L and zero bit-strings, in which case subline$(L) = L$.

For the recursive step, assume that we are given bit-strings v_i through v_n. Let $L' = (S', P', V') =$ subline$(v_{i+1}, v_{i+2}, \ldots, v_n, L)$.

- If $V'(v_i) \neq 2^{i-1}$ then we set $\text{subline}(v_i, v_{i+1}, \dots, v_n, L) = \text{FirstHalf}(L')$.
- If $V'(v_i) = 2^{i-1}$ then we set $\text{subline}(v_i, v_{i+1}, \dots, v_n, L) = \text{SecondHalf}(L')$.

Note that since FirstHalf and SecondHalf can be computed out in polynomial time, this means that subline can also be computed in polynomial time.

An important property of the reduction is that the output of $\text{subline}(v_i, v_{i+1}, \dots, v_n)$ is a UniqueEOPL instance in which the longest possible line has length 2^{i-1}. This can be proved by induction. For the base case note that $\text{subline}(L)$ allows potentials between 0 and $2^n - 1$. Each step of the recursion cuts this in half, meaning that $\text{subline}(v_i, v_{i+1}, \dots, v_n, L)$ allows potentials between 0 and $2^i - 1$. Since each edge in a line must always strictly increase the potential, this means that the longest possible line in $\text{subline}(v_i, v_{i+1}, \dots, v_n, L)$ has length 2^{i-1}. This holds even if the instance has multiple lines.

The decode function. Given a point $(v_1, v_2, \dots, v_n)$, let $L' = \text{subline}(v_1, v_2, \dots, v_n, L)$. As we have argued above, we know that L' is an instance in which the longest possible line has length $2^{1-1} = 1$. Hence, the starting vertex of L', which is by definition v_1, is the end of a line. So we set $\text{decode}(v_1, v_2, \dots, v_n) = v_1$. Since subline can be computed in polynomial time, this means that decode can also be computed in polynomial time. This completes the proof of Lemma 23.

C.3 The formal definition of the direction functions.

We will extend the approach given in the main body to all dimensions. Let $p = (v_1, v_2, \dots, v_n)$ be a point. For the dimensions j in the range $m(i-1) + 1 \leq j \leq mi$ we use the following procedure to determine D_j. Let $L' = (S', P', V') = \text{subline}(v_{i+1}, v_{i+2}, \dots, v_n, L)$.

1. In the case where $V'(v_i) \neq 2^{i-1}$, meaning that $\text{decode}(p)$ is a vertex in the first half of L', then there are two possibilities.

   (a) If $V'(\text{decode}(p)) = 2^{i-1} - 1$, meaning that p is the last vertex on the first half L', then we orient the direction function towards the bit-string given by $S(\text{decode}(p))$. So we set

   $$D_j(p) = \begin{cases} \mathsf{up} & \text{if } p_j = 0 \text{ and } S(\text{decode}(p)) = 1, \\ \mathsf{down} & \text{if } p_j = 1 \text{ and } S(\text{decode}(p)) = 0, \\ \mathsf{zero} & \text{otherwise.} \end{cases}$$

   (b) If the rule above does not apply, then we orient everything towards 0 by setting

   $$D_i(p) = \begin{cases} \mathsf{down} & \text{if } p_j = 1, \\ \mathsf{zero} & \text{if } p_j = 0. \end{cases}$$

2. If $V'(v_i) = 2^{i-1}$, then we are in the second half of the line. In this case we set $D_i(p) = \mathsf{zero}$.

Observe that this definition simply extends the idea presented in the main body to all dimensions, using exactly the same idea.

C.4 Proof of Lemma 24

To prove this lemma, we must map every solution of the OPDC instance given by P and $\mathcal{D}$ to a solution of the UniqueEOPL instance $L = (S, P, V)$. We do this by enumerating the solution types for OPDC.

1. In solutions of type (O1) we have a point $p = (v_1, v_2, \ldots, v_n)$ such that $D_i(p) = \mathsf{zero}$ for all i. Since the dimensions corresponding to v_n are all zero, we know that decode(p) is in the second half of the line, and since the dimensions corresponding to v_{n-1} are all zero, we know that decode(p) is in the last quarter of the line. Applying this reasoning for all dimensions allows us to conclude that $V(\text{decode}(p)) = 2^n - 1$. Lemma 22 implies that this can be true if and only if decode(p) is the end of a line, which is a solution of type (U1).

2. In solutions of type (OV1) we have two fixed points of a single i-slice. More specifically, we have an i-slice s and two points $p = (v_1, v_2, \ldots, v_n)$ and $q = (u_1, u_2, \ldots, u_n)$ in the slice s with $p \neq q$ such that $D_j(p) = D_j(q) = \mathsf{zero}$ for all $j \leq i$. Let v_j be the bit-string that uses dimension i, and let $L' = (S', P', V') = \text{subline}(v_{j+1}, v_{j+2}, \ldots, v_n, L)$. Since p and q both lie in s, we know that $v_l = u_l$ for all $l > j$. Observe that, since $D_l(p) = D_l(q) = \mathsf{zero}$ for all $l \leq j$, we can use the same reasoning as we did in case 1 to argue that v_1 through v_{j-1} encode a vertex that is at the end of the sub-line embedded into $(*, *, \ldots, v_j, v_{j+1}, \ldots, v_n)$, and u_1 through u_{j-1} encode a vertex that is at the end of the sub-line embedded into $(*, *, \ldots, u_j, v_{j+1}, \ldots, v_n)$. There are multiple cases to consider.

   (a) If $v_j = u_j$, then we have that $\text{subline}(v_j, v_{j+1}, \ldots, v_n, L) = \text{subline}(u_j, u_{j+1}, \ldots, u_n, L)$. Since $p \neq q$, there must be some pair of vertices v_l and u_l such that $v_l \neq u_l$, and note that since p and q both at the end of their respective sub-lines, we have $V'(v_l) = V'(u_l)$, which also implies that $V(v_l) = V(u_l)$, which is a solution of type UV3.

   (b) If $v_j \neq u_j$, and $D_l(p) = \mathsf{zero}$ for all l in the range $m(i-1)+1 \leq l \leq mi$, then $\text{subline}(v_j, v_{j+1}, \ldots, v_n, L)$ is the second half of L', while $\text{subline}(u_j, u_{j+1}, \ldots, u_n, L)$ is the first half. Since q represents a vertex at the end of the corresponding line, we have that $S(\text{decode}(q))$ is the first vertex on the next half of the L', meaning that $V'(S(\text{decode}(q))) = 2^{j-1}$. Moreover since p is on the second-half of the line, we have that $V'(v_j) = 2^{j-1}$, so we have $V(S(\text{decode}(q))) = V(v_j)$. This is a solution of type (UV3) so long as $S(\text{decode}(q)) \neq v_j$.

   To prove that this is the case, recall that the direction functions for q in the dimensions corresponding to u_j always point towards $S(\text{decode}(q))$. Since $u_j \neq v_j$, and since u_j and v_j lie in the same slice s, we know that they disagree on some dimension $l \leq i$. But we also have that $D_l(q) = \mathsf{zero}$ for all $l \leq i$, which can only occur if $S(\text{decode}(q))$ disagrees with v_j in some dimension. Hence we have shown that $S(\text{decode}(q)) \neq v_j$.

   (c) If $v_j \neq u_j$, and $D_l(q) = \mathsf{zero}$, for all l in the range $m(i-1)+1 \leq l \leq mi$, then this is entirely symmetric to the previous case.

   (d) In the last case, we have $v_j \neq u_j$, an index l_1 in the range $m(i-1)+1 \leq l_1 \leq mi$ with $D_{l_1}(p) \neq \mathsf{zero}$, and an index l_2 in the range $m(i-1)+1 \leq l_2 \leq mi$ with $D_{l_2}(p) \neq \mathsf{zero}$. Thus $\text{subline}(v_j, v_{j+1}, \ldots, v_n, L) = \text{subline}(u_j, u_{j+1}, \ldots, u_n, L)$, meaning that both points lie at the end of the first half of L'. Thus the direction functions at p point towards $S(\text{decode}(p))$, while the direction functions at q point towards $S(\text{decode}(q))$. Moreover we have $V'(S(\text{decode}(p))) = V'(S(\text{decode}(q))$, so to obtain a solution of type (UV3), we need to prove that $S(\text{decode}(p))) \neq S(\text{decode}(q)$.

This follows from the fact that $v_j \neq v_u$, and v_j and v_u's membership in the slice s. This means that they disagree on some dimension $l < i$, but since $D_a(p) = D_a(q) = \textsf{zero}$ for all $a < i$, we must have $S(\text{decode}(p))) \neq S(\text{decode}(q)$.

3. For solutions of type (OV2), we have two points $p = (v_1, v_2, \ldots, v_n)$ and $q = (u_1, u_2, \ldots, u_n)$ that lie in an i-slice s with the following properties.

   - $D_j(p) = D_j(q) = \textsf{zero}$ for all $j < i$,
   - $p_i = q_i + 1$, and
   - $D_i(p) = \textsf{down}$ and $D_i(q) = \textsf{up}$.

   As with case 2, let v_j be the bit-string that uses dimension i, and let $L' = \text{subline}(v_{j+1}, v_{j+2}, \ldots, v_n) = \text{subline}(u_{j+1}, u_{j+2}, \ldots, u_n)$. Since $D_i(p) \neq \textsf{zero}$ and $D_i(q) \neq \textsf{zero}$, we must have that p and q are both points on the first half of L'. Furthermore, decode(p) and decode(q) must both be at the end of the first half, since $D_l(p) = D_l(q) = \textsf{zero}$ for all $l < i$. Hence, $V'(\text{decode}(p)) = V'(\text{decode}(q)) = 2^{i-1} - 1$, which also implies that $V(\text{decode}(p)) = V(\text{decode}(q))$. So to obtain a solution of type (UV3), we just need to prove that $\text{decode}(p) \neq \text{decode}(q)$.

   To prove this, we observe that the direction function in dimension i should be oriented towards $S(\text{decode}(p))$ for the point p, and $S(\text{decode}(q))$ for the point q. However, since $D_i(p)$ and $D_i(q)$ both point away from their points, and since $p_i \neq q_i$, this must mean that $S(\text{decode}(p)) \neq S(\text{decode}(q))$, which also implies that $\text{decode}(p) \neq \text{decode}(q)$.

4. We claim that solutions of type (OV3) are impossible. A solution of type (OV3) requires that we have a point p with either $p_i = 0$ and $D_i(p_i) = \textsf{down}$, or $p_i = 1$ and $D_i(p_i) = \textsf{up}$. Our construction never sets $D_i(p_i) = \textsf{down}$ when $p_i = 0$, and it never sets $D_i(p_i) = \textsf{up}$ when $p_i = 1$. So solutions of type (OV3) cannot occur.

To see that the reduction is promise preserving, it suffices to note that we only ever map solutions of type (U1) onto solutions of type (O1). Thus, if the original instance has only solutions of type (U1), then the resulting OPDC instance will have solutions of type (O1).

D Proofs for Section 4.1: USO to OPDC

D.1 Proof of Lemma 27

Every solution of the OPDC instance can be mapped back to a solution of the USO instance. We prove this by enumerating all possible types of solution.

1. In solutions of type (O1) we are given a point $p \in P$ such that $D_i(p) = \textsf{zero}$ for all i. If $\Psi(p) \neq -$, then this means that p is a sink, and so it is solution of type (US1). If $\Psi(p) = -$, then this means that p is a solution of type (USV1).

2. In solutions of type (OV1), we have an i-slice s and two points $p, q \in P_s$ with $p \neq q$ such that $D_j(p) = D_j(q) = \textsf{zero}$ for all $j \leq i$. If $\Psi(p) = -$ or $\Psi(q) = -$, then we have a solution of type (USV1). Otherwise, this means that p and q are both sinks of the face corresponding to s, and specifically this means that they have the same out-map on this face. So this gives us a solution of type (USV2).

3. In solutions of type (OV2) we have an i-slice s and two points $p, q \in P_s$ such that

- $D_j(p) = D_j(q) = \mathsf{zero}$ for all $j < i$,
- $p_i = q_i + 1$, and
- $D_i(p) = \mathsf{down}$ and $D_i(q) = \mathsf{up}$.

Note that in this case we must have $\Psi(p) \neq -$ and $\Psi(q) \neq -$. If we restrict the cube to the face defined by s, note that $\Psi(p)_j = \Psi(q)_j = 0$ for all dimensions $j \neq i$, and $\Psi(p)_i = \Psi(q)_i = 1$. Hence p and q have the same out-map on the face defined by s, which gives us a solution of type (USV2).

4. Solutions of type (OV3) are impossible for the instance produced by our reduction. In these solutions we have a point p with $p_i = 0$ and $D_i(p) = \mathsf{down}$, or a point q with $q_j = 1$ and $D_j(q) = \mathsf{up}$. Since our reduction never does this, solutions of type (OV3) cannot occur.

To see that the reduction is promise-preserving, it suffices to note that solutions of type (US1) are only ever mapped onto solutions of type (O1). Thus, if the USO instance has no violations, then the resulting OPDC instance also has no violations. This completes the proof of Lemma 27.

E Prerequisites for Appendix F and H

E.1 Slice Restrictions of Contraction Maps

Our algorithms will make heavy use of the concept of a *slice restriction* of a contraction map, which we describe here. We define the set of fixed coordinates of a slice $s \in \mathsf{Slice}_d$, $\text{fixed}(s) = \{i \in [d] \mid s_i \neq *\}$ and the set of free coordinates, $\text{free}(s) = [d] \setminus \text{fixed}(s)$. Thus, an i-slice is a slice $s \in \mathsf{Slice}_d$ for which $\text{free}(s) = [i]$. We'll say that a slice is a *k-dimensional slice* if $|\text{free}(s)| = k$.

We can define the *slice restriction* of a function $f : [0,1]^d \to [0,1]^d$ with respect to a slice $s \in \mathsf{Slice}_d$, denoted $f_{|s}$, to be the function obtained by fixing the coordinates fixed(s) according to s, and keeping the coordinates of free(s) as arguments. To simplify usage of $f_{|s}$ we'll formally treat $f_{|s}$ as a function with d arguments, where the coordinates in fixed(s) are ignored. Thus, we define $f_{|s} : [0,1]^d \to [0,1]^d$ by

$$f_{|s}(x) = f(y) \quad \text{where } y_i = \begin{cases} s_i & \text{if } i \in \text{fixed}(s) \\ x_i & \text{if } i \in \text{free}(s). \end{cases}$$

Let $\text{free}(s) = \{i_1, \ldots, i_k\}$. We'll also introduce a variant of $f_{|s}$ when we want to consider the slice restriction as a lower-dimensional function, $\tilde{f}_{|s} : [0,1]^d \to [0,1]^{|\text{free}(s)|}$ defined by

$$\tilde{f}_{|s}(x) = \left(\left(f_{|s}(x)\right)_{i_1}, \ldots, \left(f_{|s}(x)\right)_{i_k} \right).$$

We can also define slice restrictions for vectors in the natural way:

$$\left(x_{|s}\right)_i = \begin{cases} s_i & \text{if } s_i \neq * \\ x_i & \text{otherwise.} \end{cases}$$

Finally, we'll use $\tilde{x}_{|s}$ to denote projection of x onto the coordinates in free(s):

$$\tilde{x}_{|s} = (x_{i_1}, \ldots, x_{i_k}).$$

We now extend the definition of a contraction map to a slice restriction of a function in the obvious way. We say that $\tilde{f}_{|s}$ is a contraction map with respect to a norm $\|\cdot\|$ with Lipschitz constant c if for any $x, y \in [0,1]^d$ we have

$$\left\| \tilde{f}_{|s}(x) - \tilde{f}_{|s}(y) \right\| \leq c \left\| \tilde{x}_{|s} - \tilde{y}_{|s} \right\|.$$

We'll also introduce some notation to remove clutter. We'll define

$$\Delta_i(p) \triangleq f(p)_i - p_i \, \forall i \in [d]$$

and use this notation when the function f is clear from the context. Note that we'll only use $\Delta_i(p)$ when considering the free coordinates of a slice restriction, so that $\Delta_i(p)$ doesn't depend on whether the most recent context has us considering $f_{|s}$ or f.

Slice restrictions will prove immensely useful through the following observations:

Lemma 59. *Let $f : [0,1]^d \to [0,1]^d$ be a contraction map with respect to $\|\cdot\|_p$ with Lipschitz constant $c \in (0,1)$. Then for any slice $s \in \mathsf{Slice}_d$, $f_{|s}$ is also a contraction map with respect to $\|\cdot\|_p$ with Lipschitz constant c.*

Proof. For any two vectors $x, y \in [0,1]^d$ we have

$$\begin{aligned}\left\| \tilde{f}_{|s}(x) - \tilde{f}_{|s}(y) \right\|_p &\leq \left\| f(x_{|s}) - f(y_{|s}) \right\|_p \\ &\leq c \left\| x_{|s} - y_{|s} \right\|_p \\ &= c \left\| \tilde{x}_{|s} - \tilde{y}_{|s} \right\|_p\end{aligned}$$

□

Since slice restrictions of contraction maps are themselves contraction maps in the sense defined above, they have unique fixpoints, up to the coordinates of the argument which are fixed by the slice and thus ignored. We'll nevertheless refer to the *unique fixpoint of a slice restriction of a contraction map*, which is the unique point $x \in [0,1]^d$ such that

$$\tilde{f}_{|s}(x) = \tilde{x}_{|s} \quad \text{and} \quad x = x_{|s}.$$

Lemma 60. *Let $f : [0,1]^d \to [0,1]^d$ be a contraction map with respect to $\|\cdot\|_p$ with Lipschitz constant $c \in (0,1)$. Let $s, s' \in \mathsf{Slice}_d$ be such that $\text{fixed}(s') = \text{fixed}(s) \cup \{i\}$ and $s_j = s'_j$ for all $j \in \text{fixed}(s)$. Let $x, y \in [0,1]^d$ be the unique fixpoints of $\tilde{f}_{|s}$ and $\tilde{f}_{|s'}$, respectively. Then $(x_i - y_i)\Delta_i(y) > 0$.*

Proof. We'll prove this by contradiction. Without loss of generality, assume towards a contradiction that $x_i \leq y_i$ and that $f(y)_i > y_i$. Then we have

$$\begin{aligned}\|f(y) - f(x)\|_p^p &= \|(f(y)_1, \ldots, f(y)_d) - (f(x)_1, \ldots, f(x)_d)\|_p^p \\ &= \sum_{i \in \text{fixed}(s)} |s_i - s_i|^p + \sum_{j \in \text{free}(s')} |y_j - x_j|^p + |f(y)_i - x_i|^p \\ &> \sum_{i \in \text{fixed}(s)} |s_i - s_i|^p + \sum_{j \in \text{free}(s')} |y_j - x_j|^p + |y_i - x_i|^p \\ &= \|y - x\|_p^p\end{aligned}$$

which contradicts the fact that f is a contraction map. The lemma follows. □

E.2 Approximate Fixpoint Lemmas

In this section we state and prove the key lemmas that will be used both in our reduction of Contraction to EndOfPotentialLine and our algorithms for finding approximate fixpoints. The lemmas reflect the intuition that sufficiently good approximate fixpoints on $(k-1)$-dimensional slices can be used to obtain slightly worse approximate fixpoints for k-dimensional slices for a notion of approximation to be formalized shortly. By choosing our approximation guarantees at each level appropriately we can ensure that we end up with a approximate fixpoint for the original function.

To formalize this intuition we define an approximate fixpoint of a given dimension with respect to some ℓ_p-norm and a slice. For any fixed ℓ_p-norm, i-slice s, and dimension parameter $k \leq i$ we'll say that a point $x \in s$ is a (s, ℓ_p, k)-*approximate fixpoint* if

$$|\Delta_j(x)| \leq \varepsilon_j(p, d), \quad \forall j \in [k]$$

where $(\varepsilon_j)_{j=1}^k$ is a constant depending only on the ℓ_p norm and the dimension d. To make the dependence on p and d explicit, we'll write $\varepsilon_j(p, d)$ instead of ε_j.

We'll define an *ε-approximate fixpoint of a contraction map f w.r.t. an ℓ_p-norm* to be a point $x \in [0,1]^d$ such that $\|f(x) - x\|_p \leq \varepsilon$.

For each different ℓ_p norm and dimension d of our contraction map, we will choose a different sequence $(\varepsilon_i(p, d))_{i=1}^d$ such that if a point x on a $(k-1)$-slice satisfies $|\Delta_j(x)| \leq \varepsilon_j(p, d)j$ for all $j < k$, either $|\Delta_k(x)| \leq \varepsilon_k(p, d)$ or the sign of $\Delta_k(x)$ indicates the direction of the unique fixpoint of that slice. There are two distinct cases to consider for $(\varepsilon_i(p, d))_{i=1}^d$:

- For $p = 1$, we choose $\varepsilon_i(1, d)_i^{(1)} \triangleq \varepsilon / 2^{2(d+1-i)}$.
- For $2 \leq p < \infty$, we choose $\varepsilon_i(p, d) \triangleq \varepsilon^{p^{(d+1-i)}} (dp)^{-2\sum_{j=0}^{d+1-i} p^j}$.

The key property satisfied by the choices of ε is captured in the following:

Lemma 61. *For any $p \in \mathbb{N}$, and any $d \in \mathbb{N}, k \in \mathbb{N}$ with $k \leq d$,*

$$\sum_{i=1}^{k-1} p\varepsilon_i(p, d) \leq \varepsilon_k(p, d)^p.$$

Proof. There are two cases to consider. For $p = 1$,

$$\sum_{i=1}^{k-1} \varepsilon_i(1, d) = \sum_{i=1}^{k-1} \varepsilon 2^{-2(d+1-i)} < \varepsilon 2^{-2(d+1-k)} = \varepsilon_k(1, d).$$

The interesting case is where $p \geq 2$. Here we observe that

$$\begin{aligned}
\sum_{i=1}^{k-1} p\varepsilon_i(p,d) &= \sum_{i=1}^{k-1} p \frac{\varepsilon^{p^{d+1-i}}}{(dp)^{2p^0+2p^1+\cdots+2p^{d+1-i}}} \\
&< \sum_{i=1}^{k-1} p \frac{\varepsilon^{p^{d+1-(k-1)}}}{(dp)^{2p^0+2p^1+\cdots+2p^{d+1-(k-1)}}} \\
&< dp \frac{\varepsilon^{p^{d+1-(k-1)}}}{(dp)^{2p^0+2p^1+\cdots+2p^{d+1-(k-1)}}} \\
&= \frac{\varepsilon^{p^{d+1-(k-1)}}}{(dp)^{2p^1+2p^2+\cdots+2p^{d+1-(k-1)}+1}} \\
&= \left(\frac{\varepsilon^{p^{d+1-k}}}{(dp)^{2p^0+2p^1+\cdots+2p^{d+1-k}}} \right)^p / (dp) \\
&= (\varepsilon_k(p,d)) / (dp) \\
&< \varepsilon_k(p,d).
\end{aligned}$$

□

We now prove lemmas showing that these choices of $(\varepsilon_i(p,d))_{i=1}^d$ ensure that $k-1$-dimensional fixpoints can be used to find k-dimensional fixpoints.

Lemma 62 (Approximate Fixpoints Give Directions)**.** *Let $s \in \mathsf{Slice}_d$ be a k-slice, for some $k \leq d$. Let $f : [0,1]^d \to [0,1]^d$ be a contraction map with respect to $\|\cdot\|_p$. Let x^* be the unique fixpoint of $\tilde{f}_{|s}$. Given any $(s, \ell_p, k-1)$-approximate fixpoint $v \in [0,1]^d$, either $|\Delta_k(v)| \leq \varepsilon_k(p,d)$ or $\Delta_k(v)(x^*_k - v_k) > 0$.*

Proof. By definition, v satisfies $|\Delta_i(v)| \leq \varepsilon_i$ for all $i \leq k-1$. Assume that $|\Delta_k(v)| > \varepsilon_k$. Without loss of generality, let $\Delta_k(v) > \varepsilon_k$. Assume towards a contradiction that $x_k^* \leq v_k$. Then

$$\begin{aligned}
\left\|\tilde{f}_{|s}(v) - \tilde{x^*}_{|s}\right\|_p^p &= \sum_{j=1}^{k} \left|f(v)_j - x_j^*\right|^p && (5)\\
&> \varepsilon_k^p + |v_k - x_k^*|^p + \sum_{j=1}^{k-1} \left|f(v)_j - x_j^*\right|^p && (6)\\
&\geq \varepsilon_k^p + |v_k - x_k^*|^p + \sum_{j=1}^{k-1} \left(\left|v_j - x_j^*\right| - \varepsilon_j\right)^p && (7)\\
&= \varepsilon_k^p + \sum_{j=1}^{k-1} |v_k - x_k^*|^p - \sum_{j=1}^{k-1} \left[\left|v_j - x_j^*\right|^p - \left(\left|v_j - x_j^*\right| - \varepsilon_j\right)^p\right] && (8)\\
&= \left\|\tilde{v}_{|s} - \tilde{x^*}_{|s}\right\|_p^p + \varepsilon_k^p - \sum_{j=1}^{k-1} \left[\left|v_j - x_j^*\right|^p - \left(\left|v_j - x_j^*\right| - \varepsilon_j\right)^p\right] && (9)\\
&\geq \left\|\tilde{v}_{|s} - \tilde{x^*}_{|s}\right\|_p^p + \varepsilon_k^p - \sum_{j=1}^{k-1} p\varepsilon_j && (10)\\
&> \left\|\tilde{v}_{|s} - \tilde{x^*}_{|s}\right\|_p^p. && (11)\\
& && (12)
\end{aligned}$$

Line 10 uses the fact that $a^p - (a-b)^p \leq 1 - (1-b)^p \leq pb$ for $p > 1$, $a, b \in (0,1)$. Line 11 follows directly from Lemma 61. Again we have a contradiction since $\left\|\tilde{f}_{|s}(v) - \tilde{x^*}_{|s}\right\|_1 \geq \left\|\tilde{v}_{|s} - \tilde{x^*}_{|s}\right\|_1$ and $\tilde{f}_{|s}$ is a contraction map. The lemma follows. □

In order for our algorithms and reductions to produce approximate fixpoints we'll need to ensure that we can find an approximate fixpoint somewhere in our discretization of the unit hypercube. We'll first establish that if we don't have an approximate fixpoint, we have witnesses to our function not being a contraction map.

Lemma 63 (Two Nearby Opposing Pivots Give Violations). *Let $s \in \mathsf{Slice}_d$ be a k-slice, for some $k \leq d$. Let $v^{(\ell)}, v^{(h)} \in [0,1]^d$ be $(s, \ell_p, k-1)$-approximate fixpoints of f such that $v_k^{(h)} - v_k^{(\ell)} < \varepsilon_k(p,d)/2$ and both $\Delta_k(v^{(h)}) \geq \varepsilon_k(p,d)$ and $\Delta_k(v^{(\ell)}) \leq -\varepsilon_k(p,d)$. Then $\left\|f(v^{(h)}) - f(v^{(\ell)}\right\|_p \geq \left\|v^{(h)} - v^{(\ell)}\right\|_p$, and $v^{(h)}, v^{(\ell)}$ witness that f is not a contraction map.*

Proof. Under the assumptions of the lemma, we have

$$\left\|\tilde{f}_{|s}\left(v^{(h)}\right)-\tilde{f}_{|s}\left(v^{(\ell)}\right)\right\|_p^p-\left\|\widetilde{v^{(h)}}_{|s}-\widetilde{v^{(\ell)}}_{|s}\right\|_p^p=\sum_{j=1}^{k-1}\left[\left|f\left(v^{(h)}\right)_j-f\left(v^{(\ell)}\right)_j\right|^p-\left|v_j^{(h)}-v_j^{(\ell)}\right|^p\right] \tag{13}$$

$$=\left[\left|f\left(v^{(h)}\right)_k-f\left(v^{(\ell)}\right)_k\right|^p-\left|v_k^{(h)}-v_k^{(\ell)}\right|^p\right] \tag{14}$$

$$+\sum_{j=1}^{k-1}\left[\left|f\left(v^{(h)}\right)_j-f\left(v^{(\ell)}\right)_j\right|^p-\left|v_j^{(h)}-v_j^{(\ell)}\right|^p\right]$$

$$\geq\left[\left|f\left(v^{(h)}\right)_k-f\left(v^{(\ell)}\right)_k\right|-\left|v_k^{(h)}-v_k^{(\ell)}\right|\right]^p \tag{15}$$

$$+\sum_{j=k+1}^{d}\left[\left(\left|v_j^{(h)}-v_j^{(\ell)}\right|-2\varepsilon_j(p,d)\right)^p-\left|v_j^{(h)}-v_j^{(\ell)}\right|^p\right]$$

$$\geq(3\varepsilon_k(p,d)/2)^p-\sum_{j=k+1}^{d}2p\varepsilon_j(p,d) \tag{16}$$

$$>0. \tag{17}$$

Here, line 16 mirrors line 10 from Lemma 62 (which holds for $\varepsilon_i(p,d)<1/2$, which is the case here). Line 17 follows from Lemma 61 when $p\geq 2$ (since $(3/2)^p>2$ in that case) and from the observation that $\frac{3}{2}\varepsilon_k(1,d)\geq 2\sum_{j=1}^{k-1}\varepsilon_j(1,d)$ for $p=1$.

Observing that

$$\left\|f\left(v^{(h)}\right)-f\left(v^{(\ell)}\right)\right\|_p\geq\left\|\tilde{f}_{|s}\left(v^{(h)}\right)-\tilde{f}_{|s}\left(v^{(\ell)}\right)\right\|_p$$

completes the proof. □

The following lemma establishes that when we have two $(s,\ell_p,k-1)$-approximate fixpoints on the same k-slice sandwiching the true k-dimensional fixpoint in a sufficiently small interval, one of the two $(s,\ell_p,k-1)$-approximate fixpoints must be an (s,ℓ_p,k)-approximate fixpoint.

Lemma 64 (One of Two Adjacent Pivots is a Fixpoint)**.** *Let $s\in\mathsf{Slice}_d$ be a k-slice, for some $k\leq d$. Let f be a contraction map with respect to $\|\cdot\|_p$. Let x^* be the unique fixpoint of $\tilde{f}_{|s}$ and let $v^{(\ell)},v^{(h)}\in[0,1]^d$ be $(s,\ell_p,k-1)$-approximate fixpoints such that $v_k^{(h)}-v_k^{(\ell)}<\varepsilon_k(p,d)/2$ and the unique fixpoint x^* of $\tilde{f}_{|s}$ satisfies*

$$v_k^{(\ell)}<x_k^*<v_k^{(h)}.$$

Then either $v^{(h)}$ or $v^{(\ell)}$ is an (s,ℓ_p,k)-approximate fixpoint of f.

Proof. By the contrapositive of Lemma 62, $\Delta_k\left(v^{(h)}\right)<\varepsilon_k(p,d)$ and $\Delta_k\left(v^{(\ell)}\right)>-\varepsilon_k(p,d)$. If either $\Delta_k\left(v^{(h)}\right)>-\varepsilon_k(p,d)$ or $\Delta_k\left(v^{(\ell)}\right)<\varepsilon_k(p,d)$ we're done since $v^{(h)}$ or $v^{(\ell)}$ is an (s,ℓ_p,k)-approximate fixpoint, respectively. The only way we can fail to have a fixpoint is if both $\Delta_k\left(v^{(h)}\right)\leq-\varepsilon_k(p,d)$ and $\Delta_k\left(v^{(\ell)}\right)\geq\varepsilon_k(p,d)$. By Lemma 63, this cannot happen when f is a contraction map, and so one of the two points must be an (s,ℓ_p,k)-approximate fixpoint. □

Finally, by choosing $\varepsilon_k(p,d)$ as above, an (s,ℓ_p,d)-approximate fixpoint is an ε-approximate fixpoint:

Lemma 65. *Any (s,ℓ_p,d)-approximate fixpoint x is an ε-approximate fixpoint.*

Proof. When $p = 1$

$$\begin{aligned}\|f(v) - v\| &= \sum_{i=1}^{d} |\Delta_i(v)| \\ &\leq \sum_{i=1}^{d} \varepsilon_i(p,d) \\ &= \sum_{i=1}^{d} \varepsilon / 2^{2(d+1-i)} \\ &= \sum_{i=1}^{d} \varepsilon / 2^{2i} \\ &< \varepsilon.\end{aligned}$$

When $p \geq 2$

$$\begin{aligned}\|f(v) - v\|_p^p &= \sum_{i=1}^{d} |\Delta_i(v)|^p \\ &\leq \sum_{i=1}^{d} \varepsilon_i^p(p,d) \\ &= \sum_{i=1}^{d} \varepsilon^{p^{(d+1-i)}} (dp)^{-2\sum_{j=0}^{d+1-i} p^j} \\ &= \sum_{i=1}^{d} \varepsilon^{p^i} (dp)^{-2\sum_{j=0}^{i} p^j} \\ &< \varepsilon^p.\end{aligned}$$

□

F Proofs for Section 4.2: PL-Contraction to OPDC

F.1 Proof of Lemma 30

We'll define the bit-length of an integer $n \in \mathbb{Z}$, denoted $b(n)$, by $b(n) \triangleq \lceil \log_2 n \rceil$. We'll extend the definition to the rationals by defining the bit-length for $x \in \mathbb{Q}$ as the minimum number of bits needed to represent the numberator and denominator of some representation of x as a ratio of integers:

$$b(x) \triangleq \min_{\substack{p,q \in \mathbb{Z} \\ x = p/q}} (b(p) + b(q)).$$

We'll extend this notion of bit-length to matrices by defining the bit-length $b(M)$ of a matrix $M \in \mathbb{Q}^{m \times n}$ by $b(M) \triangleq \max_{i,j} b(M_{ij})$ and to vectors by defining $b(v) \triangleq \max_i b(v_i)$ for $v \in \mathbb{Q}^n$.

LinearFIXP circuits and LCPs. The goal of this proof is to find a tuple $(k_1, k_2, \ldots, k_d)$ such that the lemma holds. We utilize a lemma from [59] which asserts that every LinearFIXP circuit can be transformed in polynomial time into an LCP with bounded bit-length, such that solutions to the LCP capture exactly the fixpoints of the circuit.

For any LinearFIXP circuit $C : [0,1]^d \to [0,1]^d$ with gates $g_1, \ldots, g_m$ and constants $\zeta_1, \ldots, \zeta_q$, we'll define the size of C by

$$\text{size}(C) \triangleq d + m + \sum_{i=1}^{q} b(\zeta_i).$$

Lemma 66 ([59]). *Let $C : [0,1]^d \to [0,1]^d$ be a LinearFIXP circuit. We can produce in polynomial time an LCP defined by an $n \times n$ matrix M_C and n-dimensional vector $\boldsymbol{q}_C$ for some n with $d \leq n \leq \text{size}(C)$ such that there is a bijection between solutions of the LCP $(M_C, \boldsymbol{q}_C)$ and fixpoints of C. Moreover, to obtain a fixpoint x of C from a solution $\mathbf{y}$ to the LCP $(M_C, \boldsymbol{q}_C)$, we can just set $x = (\mathbf{y}_1, \ldots, \mathbf{y}_d)$. Furthermore, $b(M_C)$ and $b(\boldsymbol{q}_C)$ are both at most $O(n \times \text{size}(C))$.*

Crucially the construction interacts nicely with fixing inputs; if C' denotes a circuit where one of the inputs of C is fixed to be some number x, we can bound the bit-length of $M_{C'}$ and $\boldsymbol{q}_{C'}$ in terms of the bit-lengths of M_C, $\boldsymbol{q}_C$, and x.

Observation 67. *$b(M_{C'}) \leq b(M_C)$.*

Observation 68. *$b(\boldsymbol{q}_{C'}) \leq \max\{b(\boldsymbol{q}_C), b(M_C) + b(x)\} + 1$.*

In other words, the bit-length of $M_{C'}$ does not depend on x, and is in fact at most the bit-length of M_C, and the bit-length of $\boldsymbol{q}_{C'}$ is bounded by the worse of the bit-lengths of $\boldsymbol{q}_C$ or the sum of the bit-lengths of M_C and x plus an additional bit.

Bounding the bit-length of a solution of an LCP. We now prove two technical lemmas about the bit-length of any solution to an LCP. We begin with the following lemma regarding the bit-length of a matrix inverse.

Lemma 69. *Let $A \in \mathbb{Z}^{n\times n}$ be a square matrix of full rank, and let $B = \max_{ij} |A_{ij}|$ be the largest absolute value of an entry of A. Then $b(A^{-1}) \leq 2n \log B + n \log n$.*

Proof. We have $A^{-1} = \frac{1}{\det(A)} \operatorname{adj}(A)$. Each entry of $\operatorname{adj}(A)$ is a cofactor of A, each of which is a (possibly negated) determinant of an $(n-1) \times (n-1)$ submatrix of A. It is a well-known corollary of Hadamard's inequality that $|\det(A)| \leq B^n n^{n/2}$. The same bound obtains for each submatrix. We can write entry A^{-1}_{ij} as p/q where $p = \operatorname{adj}(A)_{ij}$ and $q = \det(A)$ are both integers. We have $b(p), b(q) \leq n \log B + (n/2) \log n$. The lemma follows immediately. □

We now use this to prove the following bound on the bit-length of a solution of an LCP.

Lemma 70. *Let $M \in \mathbb{Q}^{n\times n}$ and $\boldsymbol{q} \in \mathbb{Q}^n$. Let $\mathbf{y}$ be a solution to the LCP $(M, \boldsymbol{q})$. Then*

$$b(\mathbf{y}) \leq (5n+2)\log n + (4n+1)b(M) + n + b(\boldsymbol{q}).$$

Proof. We first note that if an LCP has a solution, then it has a vertex solution. Let $(\mathbf{y}, \mathbf{w})$ be such a vertex solution of the LCP and, as in Section 4.3, let $\alpha = \{i \mid \mathbf{y}_i > 0\}$, and let $A = A_\alpha$ be defined according to (3). We have that A is guaranteed to be invertible, and we have that $A\boldsymbol{x} = \boldsymbol{q}$, with $\mathbf{y}_i = \boldsymbol{x}_i$ for $i \in \alpha$ and $\mathbf{y}_i = 0$ for $i \notin \alpha$, so we have $b(\mathbf{y}) \leq b(\boldsymbol{x})$. Also note that we have $b(A) \leq b(M)$, since the entries in columns that take the value of e_i have constant bit-length.

We must transform A into an integer matrix in order to apply Lemma 69. Let ℓ denote the least common multiple of the denominators of the entries in A. Note that $\ell \leq n^2 2^{b(A)}$ and hence

$b(\ell) \leq b(A) + 2\log(n)$. Our matrix equation above can be rewritten as $\ell A\boldsymbol{x} = \ell\boldsymbol{q}$, where (ℓA) is an integer matrix. Hence we have $\boldsymbol{x} = (\ell A)^{-1}(\ell\boldsymbol{q})$.

If B denotes the maximum entry of ℓA, then $B \leq \ell 2^{b(A)}$. So by Lemma 69 every entry of $(\ell A)^{-1}$ can be represented with integer numerators and denominators bounded by

$$B^n n^{n/2} \leq (\ell 2^{b(A)})^n n^{n/2} \leq (n^2 2^{2b(A)})^n n^{n/2}.$$

From this bound we obtain

$$\begin{aligned} b((\ell A)^{-1}) &\leq 2\log\left((n^2 2^{2b(A)})^n n^{n/2}\right) \\ &= 2\left(n\left(2\log n + 2b(A)\right) + (n/2)\log n\right) \\ &\leq 5n\log n + 4nb(A). \end{aligned}$$

Each entry of $\mathbf{y}$ consists of the sum of n entries of $(\ell A)^{-1}$ each of which is multiplied by an entry of $\boldsymbol{q}$, followed at the end with a multiplication by ℓ. We get the following bound on the bit-length of $\mathbf{y}$.

$$\begin{aligned} b(\mathbf{y}) &\leq b((\ell A)^{-1}) + b(\boldsymbol{q}) + n + b(\ell) \\ &\leq 5n\log n + 4nb(A) + n + b(A) + 2\log n \\ &\leq (5n+2)\log n + (4n+1)b(A) + n + b(\boldsymbol{q}) \end{aligned}$$

□

Fixing the grid size. We shall fix the grid size iteratively, starting with k_d, and working downwards. At each step, we will have a space that is partially continuous and partially discrete. Specifically, after the iteration in which we fix k_i, the dimensions $j < i$ will allow any number in $[0,1]$, while the dimensions $j \geq i$ will only allow the points with denominator k_j. If I_k denotes the set of all rationals with denominator at most k, then we will denote this as

$$P(k_i, k_{i+1}, \ldots, k_d) = [0,1]^{i-1} \times I_{k_i} \times I_{k_{i+1}} \times \cdots \times I_{k_d}.$$

Moreover, after fixing k_i, we will have that the property required by Lemma 30 holds for all j-slices s with $j \geq i$. Specifically, that if x is a fixpoint of s according to f, then there exists a $p \in P(k_i, k_{i+1}, \ldots, k_d)$ that is a fixpoint of s according to $\mathcal{D}$.

We'll start by bounding the bit-length of a solution to the PL-Contraction problem computed one coordinate at a time. Given a PL-circuit C, we use Lemma 66 to produce an LCP defined by $M \in \mathbb{Q}^{n\times n}$ and $\boldsymbol{q} \in \mathbb{Q}^n$. Now let $x_1, \ldots, x_n$ be formal variables representing the inputs to our circuit C. We want to determine parameters $\kappa_1, \ldots, \kappa_d$ such that such that if we fix variables $x_{i+1}, ..., x_d$ to values with bit-lengths $b(x_{i+1}), \ldots, b(x_d)$ where $b(x_j) \leq \kappa_j$ for each $j \in \{i+1, \ldots, d\}$, then any fixpoints of C with respect to the free variables $x_1, \ldots, x_i$ will have bit-lengths $b(x_1), \ldots, b(x_i)$ with $b(x_j) \leq \kappa_j$ for $j \in [i]$.

We'll set

$$\kappa_i \triangleq (d-i+1)((5n+2)\log n + n + (4n+2)b(M) + 1) + b(\boldsymbol{q}).$$

Note that $\kappa_1 \geq \kappa_2 \geq \cdots \geq \kappa_d$.

To prove that these bounds suffice, we'll use induction on i, starting from $i = d$. First, we observe that by Lemma 70, we have $b(\mathbf{y}) \leq (5n+2)\log n + (4n+1)b(M) + b(\boldsymbol{q}) = \kappa_d - 1 \leq \kappa_d$ for any solution $\mathbf{y}$ to the LCP $(M, \boldsymbol{q})$. Moreover each fixpoint x of C corresponds to a solution $\mathbf{y}$ to the LCP by Lemma 66, so we have $b(x_1), \dots, b(x_d) \leq \kappa_d$ which implies the weaker claim that all fixpoints can be found by choosing $b(x_i) \leq \kappa_i$ for all $i \in [d]$, since $\kappa_i \geq \kappa_d$ for all $i \in [d]$.

Now we'll handle the inductive case. For $i = 1, \dots, d$, let $M^{(i)}, \boldsymbol{q}^{(i)}$ be the pair defining the LCP after x_{i+1} through x_d are fixed to values with bit-lengths bounded by $\kappa_{i+1}, \dots, \kappa_d$, respectively. This pair will of course depend on the values $x_{i+1}, \dots, x_d$, but since the bit-length of $M^{(i)}$ and $\boldsymbol{q}^{(i)}$ depend only on the bit-lengths of the fixed values, we can ignore the values of $x_{i+1}, \dots, x_d$ as long as we have bounds on their bit-lengths and as long as we restrict our attention to the bit-lengths of $M^{(i)}$ and $\boldsymbol{q}^{(i)}$ and not the values themselves.

Using Lemma 70 we know that the solution to the LCP has $b(x_i) \leq (5n+2)\log n + (4n+1)b(M^{(i)}) + b(\boldsymbol{q}^{(i)})$. Moreover, we obtained $M^{(i)}$ by repeatedly fixing inputs to C, so the repeated application of Observation 67 implies that $b(M^{(i)}) \leq b(M^{(i+1)}) \leq b(M)$. Moreover, Observation 68 gives us

$$b(\boldsymbol{q}^{(i)}) \leq \max\left\{b(\boldsymbol{q}^{(i+1)}), b(M^{(i+1)}) + b(x_{i+1})\right\} + 1.$$

By a simple argument, we can also show that $\kappa_{i+1} \geq b(\boldsymbol{q}^{(i+1)})$ so that the above bound can be simplified to

$$b(\boldsymbol{q}^{(i)}) \leq b(M^{(i)}) + b(x_{i+1}) + 1$$

at which point we can use our observation and the bound $b(x_{i+1}) \leq \kappa_{i+1}$ to obtain

$$b(\boldsymbol{q}^{(i)}) \leq b(M) + \kappa_{i+1} + 1.$$

Now we use the bound from Lemma 70 again to obtain

$$\begin{aligned} b(x_i) &\leq (5n+2)\log n + (4n+1)b(M^{(i)}) + b(\boldsymbol{q}^{(i)}) \\ &\leq (5n+2)\log n + (4n+1)b(M) + b(M) + \kappa_{i+1} + 1 \end{aligned}$$

and we now use the definition of κ_{i+1} to conclude

$$\begin{aligned} b(x_i) \leq\ & (5n+2)\log n + (4n+1)b(M) + b(M) \\ & + (d-(i+1)+1)((5n+2)\log n + n + (4n+2)b(M) + 1) + b(\boldsymbol{q}) + 1 \\ =\ & (d-i+1)((5n+2)\log n + (4n+2)b(M) + 1) + b(\boldsymbol{q}) \\ =\ & \kappa_i. \end{aligned}$$

We conclude that all solutions to the LCP $(M^{(i)}, \boldsymbol{q}^{(i)})$ have free coordinates with bit-length at most κ_i, and similarly for the fixpoints of C. Thus, every fixpoint of C with respect to the free variables can be found by choosing $x_1, \dots, x_i$ to have bit-lengths at most $\kappa_1, \dots, \kappa_i$, respectively, when the bit-lengths of $x_{i+1}, \dots, x_d$ are chosen to have bit-lengths at most $\kappa_{i+1}, \dots, \kappa_d$.

To conclude the proof of Lemma 30, we need to choose $k_1, \dots, k_d$ so that every i-slice with fixed coordinates in $P(k_1, \dots, k_d)$ has a fixpoint also on $P(k_1, \dots, k_d)$ when we map points $x \in P(k_1, \dots, k_d)$ to $[0,1]^d$ by

$$x \mapsto (x_1/k_1, \dots, x_d/k_d).$$

We set $k_i \triangleq 2^{\kappa_i}$. Now a point $x \in P(k_1, \dots, k_d)$ corresponds to a point $y \in [0,1]^d$ with $b(y_i) \leq 2\kappa_i$ for all $i \in [d]$ and any i-slice where the fixed coordinates satisfy the bit-length bounds will have fixpoints for the remaining variables with all coordinates satisfying the bit-length bounds.

Finally, we observe that for each $i \in [d]$, $b(k_i) = \kappa_i \leq \kappa_1 = O(\text{poly}(\text{size}(C)))$, so each of the k_i can be represented using polynomially many bits in the size of the circuit C.

F.2 Proof of Lemma 31

The proof of this lemma will make use of Lemmas 59 and 60, which are proved in Appendix E.1.

The statement of the proof says that we can assume that f is contracting with respect to some ℓ_p norm, and that we have an i-slice s and two points p, q in s satisfying the following conditions.

- $D_j(p) = D_j(q) = \mathsf{zero}$ for all $j < i$,
- $p_i = q_i + 1$, and
- $D_i(p) = \mathsf{down}$ and $D_i(q) = \mathsf{up}$.

To translate p and q from the grid to the $[0, 1]^n$ space, we must divide each component by the grid length in that dimension. Specifically, we define the point $a \in [0, 1]^n$ so that $a_i = p_i/k_i$ for all i, and the point $b \in [0, 1]^n$ such that $b_i = q_i/k_i$ for all i.

Lemma 59 states that if f is contracting with respect to an ℓ_p norm, then the restriction of f to the slice s is also contracting in that ℓ_p norm. Hence, f must have a fixpoint in the slice s. Let $x \in [0, 1]^n$ denote this fixpoint.

By definition we have that $(f(x) - x)_j = 0$ for all $j \leq i$, and we also have $(f(a) - a)_j = 0$ for all $j < i$ from the fact that p is a fixpoint of its i-slice. So we can apply Lemma 60 to x and a, and from this we get that $(x_i - a_i) \cdot (f(a)_i - a_i) > 0$. Since $D_i(p) = \mathsf{down}$, we have that $f(a)_i < a_i$, and hence $(f(a)_i - a_i) < 0$. Therefore, we can conclude that $(x_i - a_i) < 0$, which implies that $x_i < a_i$.

By the same reasoning we can apply Lemma 60 to x and b, and this gives us that $(x_i - b_i) \cdot (f(b)_i - b_i) > 0$. This time we have $D_i(q) = \mathsf{up}$, which implies that $f(b)_i > b_i$, and hence $(f(b)_i - b_i) > 0$. So we can conclude that $(x_i - b_i) > 0$, meaning that $x_i > b_i$.

Hence we have shown that $b_i < x_i < a_i$, and so the point x satisfies the conditions of the lemma. This completes the proof of Lemma 31.

F.3 Proof of Lemma 33

We must map every possible solution of the OPDC instance back to a solution of the PL-Contraction instance. We will enumerate all solution types for OPDC.

- In solutions of type (O1), we have a point $p \in P$ such that $D_i(p) = \mathsf{zero}$ for all i. This means that the point x, where $x_i = p_i/k_i$ for all i, satisfies $f(x) - x = 0$. Hence x is a solution of type (CM1).
- In solutions of type (OV1) we have i-slice s and two points $p, q \in P_s$ with $p \neq q$ such that $D_j(p) = D_j(q) = \mathsf{zero}$ for all $j \leq i$. Let a and b be the two corresponding points in $[0, 1]^n$, meaning that $a_i = p_i/k_i$ for all i, and $b_i = q_i/k_i$ for all i.

  We first first consider the contraction property in the $[0, 1]^i$ space defined by the slice s. Let $\|a - b\|_p^i$ denote the distance between a and b according to the ℓ_p norm restricted to the $[0, 1]^i$ space, and likewise let $\|f(a) - f(b)\|_p^i$ be the distance between $f(a)$ and $f(b)$ in that space. Since $D_j(p) = D_j(q) = \mathsf{zero}$ for all $j \leq i$, this means that we have $\|f(a) - f(b)\|_p^i = \|a - b\|_p^i$, which is a clear violation of contraction in the space $[0, 1]^i$.

  To see that a and b also violate contraction in $[0, 1]^d$, observe that $\|a - b\|_p^i = \|a - b\|_p$, because a and b lie in the same i-slice, meaning that $a_j = b_j$ for all $j > i$. Furthermore, we have $\|f(a) - f(b)\|_p \geq \|f(a) - f(b)\|_p^i$, since adding in extra dimensions can only increase the distance in an ℓ_p norm. Hence we have $\|f(a) - f(b)\|_p \geq \|a - b\|_p$, which is a violation of contraction, giving us a solution of type (CMV1).

- Violations of type (OV2) map directly to violations of type (CMV3), as discussed in the main body.
- Violations of type (OV3) give us a point p such that $p_i = 0$ and $D_i(p) = \mathsf{down}$, or $p_i = k_i$ and $D_i(p) = \mathsf{up}$. In both cases this means that $f(p) \notin [0,1]^d$, and so we have a violation of type (CMV2).

To see that the reduction is promise-preserving, it suffices to note that solutions of type (CM1) are only ever mapped on to solutions of type (O1). Hence, if the input problem is a contraction map, the resulting OPDC instance only has solutions of type (CM1).

G Proofs for Section 4.3: PLCP to EOPL and UEOPL

G.1 Background on Lemke's algorithm

The explanation of Lemke's algorithm in this section is taken from [32]. Recall the LCP problem from Definition 35, where given a $d \times d$ matrix M and d-dimensional vector q, we want to find $\mathbf{y}$ satisfying (1). That is ($\mathbf{w}$ is a place-holder variable),

$$\mathbf{w} = M\mathbf{y} + \boldsymbol{q}, \quad \mathbf{y} \geq 0, \quad \mathbf{w} \geq 0 \quad \text{and} \quad y_i w_i = 0, \ \forall i \in [d].$$

The problem is interesting only when $\boldsymbol{q} \not\geq 0$, since otherwise $\mathbf{y} = 0$ is a trivial solution. Let $\mathcal{Q}$ be the polyhedron in $2d$ dimensional space defined by the first three conditions; we will assume that $\mathcal{Q}$ is non-degenerate (just for simplicity of exposition; this will not matter for our reduction). Under this condition, any solution to (1) will be a vertex of $\mathcal{Q}$, since it must satisfy $2d$ equalities. Note that the set of solutions may be disconnected. The ingenious idea of Lemke was to introduce a new variable and consider the system:

$$\mathbf{w} = M\mathbf{y} + \boldsymbol{q} + z\mathbf{1}, \quad \mathbf{y} \geq 0, \quad \mathbf{w} \geq 0, \quad z \geq 0 \quad \text{and} \quad y_i w_i = 0, \ \forall i \in [d]. \tag{18}$$

The next lemma follows by construction of (18).

Lemma 71. *Given $(M, \boldsymbol{q})$, $(\mathbf{y}, \mathbf{w}, z)$ satisfies (18) with $z = 0$ iff $\mathbf{y}$ satisfies (1).*

Let $\mathcal{P}$ be the polyhedron in $2d+1$ dimensional space defined by the first four conditions of (18), i.e.,

$$\mathcal{P} = \{(\mathbf{y}, \mathbf{w}, z) \mid \mathbf{w} = M\mathbf{y} + \boldsymbol{q} + z\mathbf{1}, \quad \mathbf{y} \geq 0, \quad \mathbf{w} \geq 0, \quad z \geq 0\}; \tag{19}$$

for now, we will assume that $\mathcal{P}$ is *non-degenerate.*

Since any solution to (18) must still satisfy $2d$ equalities in $\mathcal{P}$, the set of solutions, say S, will be a subset of the one-skeleton of $\mathcal{P}$, i.e., it will consist of edges and vertices of $\mathcal{P}$. Any solution to the original system (1) must satisfy the additional condition $z = 0$ and hence will be a vertex of $\mathcal{P}$.

Now S turns out to have some nice properties. Any point of S is *fully labeled* in the sense that for each i, $y_i = 0$ or $w_i = 0$. We will say that a point of S *has duplicate label* i if $y_i = 0$ and $w_i = 0$ are both satisfied at this point. Clearly, such a point will be a vertex of $\mathcal{P}$ and it will have only one duplicate label. Since there are exactly two ways of relaxing this duplicate label, this vertex must have exactly two edges of S incident at it. Clearly, a solution to the original system (i.e., satisfying $z = 0$) will be a vertex of $\mathcal{P}$ that does not have a duplicate label. On relaxing $z = 0$, we get the unique edge of S incident at this vertex.

As a result of these observations, we can conclude that every vertex of S with $z > 0$ has degree two within S, while a vertex with $z = 0$ has degree one. Thus, S consists of paths and cycles.

Of these paths, Lemke's algorithm explores a special one. An unbounded edge of S such that the vertex of $\mathcal{P}$ it is incident on has $z > 0$ is called a *ray*. Among the rays, one is special – the one on which $\mathbf{y} = 0$. This is called the *primary ray* and the rest are called *secondary rays*. Now Lemke's algorithm explores, via pivoting, the path starting with the primary ray. This path must end either in a vertex satisfying $z = 0$, i.e., a solution to the original system, or a secondary ray. In the latter case, the algorithm is unsuccessful in finding a solution to the original system; in particular, the original system may not have a solution. We give the full pseudo-code for Lemke's algorithm in Table 1.

Table 1: Lemke's Complementary Pivot Algorithm

```
If $\boldsymbol{q} \geq 0$ then Return $\mathbf{y} \leftarrow \mathbf{0}$
$\mathbf{y} \leftarrow 0, z \leftarrow |\min_{i\in[d]} q_i|, \mathbf{w} = \boldsymbol{q} + z\mathbf{1}$
$i \leftarrow$ duplicate label at vertex $(\mathbf{y}, \mathbf{w}, z)$ in $\mathcal{P}$. $flag \leftarrow 1$
While $z > 0$ do
 If $flag = 1$ then set $(\mathbf{y}', \mathbf{w}', z') \leftarrow$ vertex obtained by relaxing $y_i = 0$ at $(\mathbf{y}, \mathbf{w}, z)$ in $\mathcal{P}$
 Else set $(\mathbf{y}', \mathbf{w}', z') \leftarrow$ vertex obtained by relaxing $w_i = 0$ at $(\mathbf{y}, \mathbf{w}, z)$ in $\mathcal{P}$
 If $z > 0$ then
  $i \leftarrow$ duplicate label at $(\mathbf{y}', \mathbf{w}', z')$
  If $v_i > 0$ and $v'_i = 0$ then $flag \leftarrow 1$. Else $flag \leftarrow 0$
  $(\mathbf{y}, \mathbf{w}, z) \leftarrow (\mathbf{y}', \mathbf{w}', z')$
End While
Return $\mathbf{y}$
```

G.2 Reduction from P-LCP with (PV1) violations to EndOfPotentialLine

It is well known that if matrix M is a P-matrix (P-LCP), then z strictly decreases on the path traced by Lemke's algorithm [14]. Furthermore, by a result of Todd [76, Section 5], paths traced by complementary pivot rule can be locally oriented. Based on these two facts, we derive a polynomial-time reduction from P-LCP to EndOfPotentialLine first, and then from P-LCP to UniqueEOPL.

Let $\mathcal{I} = (M, \boldsymbol{q})$ be a given P-LCP instance, and let $\mathcal{L}$ be the length of the bit representation of M and $\boldsymbol{q}$. We will reduce $\mathcal{I}$ to an EndOfPotentialLine instance $\mathcal{E}$ in time poly($\mathcal{L}$). According to Definition 9, the instance $\mathcal{E}$ is defined by its vertex set vert, and procedures S (successor), P (predecessor) and V (potential). Next we define each of these.

As discussed in Section G.1 the linear constraints of (18) on which Lemke's algorithm operates forms a polyhedron $\mathcal{P}$ given in (19). We assume that $\mathcal{P}$ is non-degenerate. This is without loss of generality since, a typical way to ensure this is by perturbing $\boldsymbol{q}$ so that configurations of solution vertices remain unchanged [14], and since M is unchanged if $\mathcal{I}$ was a P-LCP instance then it remains so.

Lemke's algorithm traces a path on feasible points of (18) which is on 1-skeleton of $\mathcal{P}$ starting at $(\mathbf{y}^0, \mathbf{w}^0, z^0)$, where:

$$\mathbf{y}^0 = 0, \qquad z^0 = |\min_{i\in[d]} q_i|, \qquad \mathbf{w}^0 = \boldsymbol{q} + z^0\mathbf{1} \tag{20}$$

We want to capture vertex solutions of (18) as vertices in EndOfPotentialLine instance $\mathcal{E}$. To differentiate we will sometimes call the latter, *configurations*. Vertex solutions of (18) are exactly the vertices of polyhedron $\mathcal{P}$ with either $y_i = 0$ or $w_i = 0$ for each $i \in [d]$. Vertices of (18) with $z = 0$ are our final solutions (Lemma 71). While each of its *non-solution* vertex has a duplicate label. Thus, a vertex of this path can be uniquely identified by which of $y_i = 0$ and $w_i = 0$ hold for each i and its duplicate label. This gives us a representation for vertices in the EndOfPotentialLine instance $\mathcal{E}$.

EndOfPotentialLine Instance $\mathcal{E}$.

- Vertex set $\mathsf{vert} = \{0,1\}^n$ where $n = 2d$.
- Procedures S and P as defined in Tables 4 and 6 respectively
- Potential function $V : \mathsf{vert} \to \{0, 1, \ldots, 2^m - 1\}$ defined in Table 5 for $m = \lceil ln(2\Delta^3)\rceil$, where

$$\Delta = (n! \cdot I_{max}^{2d+1}) + 1$$

  and $I_{max} = \max\{\max_{i,j\in[d]} M(i,j),\ \max_{i\in[d]} |q_i|\}$.

For any vertex $\boldsymbol{u} \in \mathsf{vert}$, the first d bits of $\boldsymbol{u}$ represent which of the two inequalities, namely $y_i \geq 0$ and $w_i \geq 0$, is tight for each $i \in [d]$.

$$\forall i \in [d], \quad u_i = 0 \Rightarrow y_i = 0, \quad \text{and} \quad u_i = 1 \Rightarrow w_i = 0$$

A valid setting of the second set of d bits, namely u_{d+1} through u_{2d}, will have at most one non-zero bit – if none is one then $z = 0$, otherwise the location of one bit indicates the duplicate label. Thus, there are many invalid configurations, namely those with more than one non-zero bit in the second set of d bits. These are dummies that we will handle separately, and we define a procedure IsValid to identify non-dummy vertices in Table 2. To go between "valid" vertices of $\mathcal{E}$ and corresponding vertices of the Lemke polytope $\mathcal{P}$ of LCP $\mathcal{I}$, we define procedures EtoI and ItoE in Table 3.

By construction of IsValid, EtoI and ItoE, the next lemma follows.

Table 2: Procedure IsValid($\boldsymbol{u}$)

```
**If** $\boldsymbol{u} = 0^n$ **then Return** 1
**Else** let $\tau = (u_{(d+1)} + \cdots + u_{2d})$
  **If** $\tau > 1$ **then Return** 0
  Let $S \leftarrow \emptyset$. % set of tight inequalities.
  **If** $\tau = 0$ **then** $S = S \cup \{z = 0\}$.
  **Else**
    Set $l \leftarrow$ index of the non-zero coordinate in vector $(u_{(d+1)}, \ldots, u_{2d})$.
    Set $S = \{y_l = 0, w_l = 0\}$.
  **For** each $i$ from 1 to $d$ **do**
    **If** $u_i = 0$ **then** $S = S \cup \{y_i = 0\}$, **Else** $S = S \cup \{w_i = 0\}$
  Let $A$ be a matrix formed by l.h.s. of equalities $M\mathbf{y} - \mathbf{w} + \mathbf{1}z = -\boldsymbol{q}$ and that of set $S$
  Let $\mathbf{b}$ be the corresponding r.h.s., namely $\mathbf{b} = [-\boldsymbol{q}; \mathbf{0}_{(d+1)\times 1}]$.
  Let $(\mathbf{y}', \mathbf{w}', z') \leftarrow \mathbf{b} * A^{-1}$
  **If** $(\mathbf{y}', \mathbf{w}', z') \in \mathcal{P}$ **then Return** 1, **Else Return** 0
```

Table 3: Procedures ItoE($\boldsymbol{u}$) and EtoI($\mathbf{y}, \mathbf{w}, z$)

```
ItoE($\mathbf{y}, \mathbf{w}, z$)
  **If** $\exists i \in [d]$ s.t. $y_i * w_i \neq 0$ **then Return** $(\mathbf{0}_{(2d-2)\times 1}; 1; 1)$ % Invalid
  Set $\boldsymbol{u} \leftarrow \mathbf{0}_{2d\times 1}$. Let $DL = \{i \in [d] \mid y_i = 0$ and $w_i = 0\}$.
  **If** $|DL| > 1$ **then Return** $(\mathbf{0}_{(2d-2)\times 1}; 1; 1)$ %Invalid
  **If** $|DL| = 1$ **then** for $i \in DL$, set $u_{d+i} \leftarrow 1$
  **For** each $i \in [d]$ **If** $w_i = 0$ **then** set $u_i \leftarrow 1$
  **Return** $\boldsymbol{u}$

EtoI($\boldsymbol{u}$)
  **If** $\boldsymbol{u} = 0^n$ **then Return** $(\mathbf{0}_{d\times 1}, \boldsymbol{q} + (z^0 + 1)\mathbf{1}, z^0 + 1)$ % This case will never happen
  **If** IsValid($\boldsymbol{u}$)=0 **then Return** $\mathbf{0}_{(2d+1)\times 1}$
  Let $\tau = (u_{(d+1)} + \cdots + u_{2d})$
  Let $S \leftarrow \emptyset$. % set of tight inequalities.
  **If** $\tau = 0$ **then** $S = S \cup \{z = 0\}$.
  **Else**
    Set $l \leftarrow$ index of non-zero coordinate in vector $(u_{(d+1)}, \ldots, u_{2d})$.
    Set $S = \{y_l = 0, w_l = 0\}$.
  **For** each $i$ from 1 to $d$ **do**
    **If** $u_i = 0$ **then** $S = S \cup \{y_i = 0\}$, **Else** $S = S \cup \{w_i = 0\}$
  Let $A$ be a matrix formed by lhs of equalities $M\mathbf{y} - \mathbf{w} + \mathbf{1}z = -\boldsymbol{q}$ and that of set $S$
  Let $\mathbf{b}$ be the corresponding rhs, namely $\mathbf{b} = [-\boldsymbol{q}; \mathbf{0}_{(d+1)\times 1}]$.
  **Return** $\mathbf{b} * A^{-1}$
```

Lemma 72. *If IsValid*($\boldsymbol{u}$) = 1 *then* $\boldsymbol{u}$ = *ItoE*(*EtoI*($\boldsymbol{u}$)), *and the corresponding vertex* $(\mathbf{y}, \mathbf{w}, z)$ = *EtoI*($\boldsymbol{u}$) *of* $\mathcal{P}$ *is feasible in (18). If* $(\mathbf{y}, \mathbf{w}, z)$ *is a feasible vertex of (18) then* $\boldsymbol{u}$ = *ItoE*$(\mathbf{y}, \mathbf{w}, z)$ *is a valid configuration,* i.e., *IsValid*($\boldsymbol{u}$) = 1.

Proof. The only thing that can go wrong is that the matrix A generated in IsValid and EtoI procedures are singular, or the set of double labels DL generated in ItoE has more than one elements. □

Table 4: Successor Procedure $S(\boldsymbol{u})$

```
**If** IsValid($\boldsymbol{u}$) $= 0$ **then Return** $\boldsymbol{u}$
**If** $\boldsymbol{u} = 0^n$ **then Return** ItoE($\mathbf{y}^0, \mathbf{w}^0, z^0$)
$\boldsymbol{x} = (\mathbf{y}, \mathbf{w}, z) \leftarrow$ EtoI($\boldsymbol{u}$)
**If** $z = 0$ **then**
  $\boldsymbol{x}^1 \leftarrow$ vertex obtained by relaxing $z = 0$ at $\boldsymbol{x}$ in $\mathcal{P}$.
  **If** Todd [76] prescribes edge from $\boldsymbol{x}$ to $\boldsymbol{x}^1$
   **then** set $\boldsymbol{x}' \leftarrow \boldsymbol{x}^1$. **Else Return** $\boldsymbol{u}$
**Else** set $l \leftarrow$ duplicate label at $\boldsymbol{x}$
  $\boldsymbol{x}^1 \leftarrow$ vertex obtained by relaxing $y_l = 0$ at $\boldsymbol{x}$ in $\mathcal{P}$
  $\boldsymbol{x}^2 \leftarrow$ vertex obtained by relaxing $w_l = 0$ at $\boldsymbol{x}$ in $\mathcal{P}$
  **If** Todd [76] prescribes edge from $\boldsymbol{x}$ to $\boldsymbol{x}^1$
   **then** $\boldsymbol{x}' = \boldsymbol{x}^1$
  **Else** $\boldsymbol{x}' = \boldsymbol{x}^2$
Let $\boldsymbol{x}'$ be $(\mathbf{y}', \mathbf{w}', z')$.
**If** $z > z'$ **then Return** ItoE($\boldsymbol{x}'$). **Else Return** $\boldsymbol{u}$.
```

Table 5: Potential Value $V(\boldsymbol{u})$

```
**If** IsValid($\boldsymbol{u}$) $= 0$
 **then Return** 0
**If** $\boldsymbol{u} = 0^n$
 **then Return** 0
$(\mathbf{y}, \mathbf{w}, z) \leftarrow$ EtoI($\boldsymbol{u}$)
**Return** $\lfloor \Delta^2 * (\Delta - z) \rfloor$
```

The main idea behind procedures S and P, given in Tables 4 and 6 respectively, is the following (also see Figure 5): Make dummy configurations in vert to point to themselves with cycles of length one, so that they can never be solutions or violations. The starting vertex $0^n \in$ vert points to the configuration that corresponds to the first vertex of the Lemke path, namely $\boldsymbol{u}^0 = \text{ItoE}(\mathbf{y}^0, \mathbf{w}^0, z^0)$. Precisely, $S(0^n) = \boldsymbol{u}^0$, $P(\boldsymbol{u}^0) = 0^n$ and $P(0^n) = 0^n$ (start of a path).

For the remaining cases, let $\boldsymbol{u} \in$ vert have corresponding representation $\boldsymbol{x} = (\mathbf{y}, \mathbf{w}, z) \in \mathcal{P}$, and suppose $\boldsymbol{x}$ has a duplicate label. As one traverses a Lemke path for a P-LCP, the value of z monotonically decreases []. So, for $S(\boldsymbol{u})$ we compute the adjacent vertex $\boldsymbol{x}' = (\mathbf{y}', \mathbf{w}', z')$ of $\boldsymbol{x}$ on Lemke path such that the edge goes from $\boldsymbol{x}$ to $\boldsymbol{x}'$, and if the $z' < z$, as expected, then we point $S(\boldsymbol{u})$ to configuration of $\boldsymbol{x}'$ namely ItoE($\boldsymbol{x}'$). Otherwise, we let $S(\boldsymbol{u}) = \boldsymbol{u}$. Similarly, for $P(\boldsymbol{u})$, we find $\boldsymbol{x}'$ such that edge is from $\boldsymbol{x}'$ to $\boldsymbol{x}$, and then we let $P(\boldsymbol{u})$ be ItoE($\boldsymbol{x}'$) if $z' > z$ as expected, otherwise $P(\boldsymbol{u}) = \boldsymbol{u}$.

For the case when $\boldsymbol{x}$ does not have a duplicate label, then we have $z = 0$. This is handled separately since such a vertex has exactly one incident edge on the Lemke path, namely the one obtained by relaxing $z = 0$. According to the direction of this edge, we do similar process as before. For example, if the edge goes from $\boldsymbol{x}$ to $\boldsymbol{x}'$, then, if $z' < z$, we set $S(\boldsymbol{u}) = \text{ItoE}(\boldsymbol{x}')$ else $S(\boldsymbol{u}) = \boldsymbol{u}$, and we always set $P(\boldsymbol{u}) = \boldsymbol{u}$. In case the edge goes from $\boldsymbol{x}'$ to $\boldsymbol{x}$, we always set $S(\boldsymbol{u}) = \boldsymbol{u}$, and we set $P(\boldsymbol{u})$ depending on whether or not $z' > z$.

The potential function V, formally defined in Table 5, gives a value of zero to dummy vertices and the starting vertex 0^n. To all other vertices, essentially it is $((z^0 - z) * \Delta^2) + 1$. Since value of z starts at z^0 and keeps decreasing on the Lemke path this value will keep increasing starting from zero at the starting vertex 0^n. Multiplication by Δ^2 will ensure that if $z_1 > z_2$ then the corresponding potential values will differ by at least one. This is because, since z_1 and z_2 are coordinates of two vertices of polytope $\mathcal{P}$, their maximum value is Δ and their denominator is also bounded above by Δ. Hence $z_1 - z_2 \leq 1/\Delta^2$ (Lemma 74).

To show correctness of the reduction we need to show two things: (i) All the procedures are well-defined and polynomial time. (ii) We can construct a solution of $\mathcal{I}$ from a solution of $\mathcal{E}$ in polynomial time.

Table 6: Predecessor Procedure $P(\boldsymbol{u})$

```
**If** IsValid($\boldsymbol{u}$) = 0 **then Return** $\boldsymbol{u}$
**If** $\boldsymbol{u} = 0^n$ **then Return** $\boldsymbol{u}$
$(\mathbf{y}, \mathbf{w}, z) \leftarrow$ EtoI($\boldsymbol{u}$)
**If** $(\mathbf{y}, \mathbf{w}, z) = (\mathbf{y}^0, \mathbf{w}^0, z^0)$ **then Return** $0^n$
**If** $z = 0$ **then**
  $\boldsymbol{x}^1 \leftarrow$ vertex obtained by relaxing $z = 0$ at $\boldsymbol{x}$ in $\mathcal{P}$.
  **If** Todd [76] prescribes edge from $\boldsymbol{x}^1$ to $\boldsymbol{x}$ **then** set $\boldsymbol{x}' \leftarrow \boldsymbol{x}^1$. **Else Return** $\boldsymbol{u}$
**Else**
  $l \leftarrow$ duplicate label at $\boldsymbol{x}$
  $\boldsymbol{x}^1 \leftarrow$ vertex obtained by relaxing $y_l = 0$ at $\boldsymbol{x}$ in $\mathcal{P}$
  $\boldsymbol{x}^2 \leftarrow$ vertex obtained by relaxing $w_l = 0$ at $\boldsymbol{x}$ in $\mathcal{P}$
  **If** Todd [76] prescribes edge from $\boldsymbol{x}^1$ to $\boldsymbol{x}$ **then** $\boldsymbol{x}' = \boldsymbol{x}^1$ **Else** $\boldsymbol{x}' = \boldsymbol{x}^2$
Let $\boldsymbol{x}'$ be $(\mathbf{y}', \mathbf{w}', z')$. **If** $z < z'$ **then Return** ItoE($\boldsymbol{x}'$). **Else Return** $\boldsymbol{u}$.
```

Lemma 73. *Functions P, S and V of instance $\mathcal{E}$ are well defined, making $\mathcal{E}$ a valid* EndOfPotentialLine *instance.*

Proof. Since all three procedures are polynomial-time in $\mathcal{L}$, they can be defined by poly($\mathcal{L}$)-sized Boolean circuits. Furthermore, for any $\boldsymbol{u} \in \mathsf{vert}$, we have that $S(\boldsymbol{u}), P(\boldsymbol{u}) \in \mathsf{vert}$. For V, since the value of $z \in [0,\ \Delta - 1]$, we have $0 \le \Delta^2(\Delta - z) \le \Delta^3$. Therefore, $V(\boldsymbol{u})$ is an integer that is at most $2 \cdot \Delta^3$ and hence is in set $\{0, \ldots, 2^m - 1\}$. ☐

There are two possible types of solutions of an EndOfPotentialLine instance (see Definition 9). One indicates the beginning or end of a line (R1), and the other is a vertex with locally optimal potential (R2). First we show that (R2) never arises. For this, we need the next lemma, which shows that potential differences in two adjacent configurations adheres to differences in the value of z at corresponding vertices.

Lemma 74. *Let $\boldsymbol{u} \neq \boldsymbol{u}'$ be two valid configurations, i.e., IsValid$(\boldsymbol{u})$ = IsValid$(\boldsymbol{u}')$ = 1, and let $(\mathbf{y}, \mathbf{w}, z)$ and $(\mathbf{y}', \mathbf{w}', z')$ be the corresponding vertices in $\mathcal{P}$. Then the following holds:*

(i) $V(\boldsymbol{u}) = V(\boldsymbol{u}')$ *iff* $z = z'$.

(ii) $V(\boldsymbol{u}) > V(\boldsymbol{u}')$ *iff* $z < z'$.

Proof. Among the valid configurations all except $\mathbf{0}$ has positive V value. Therefore, wlog let $\boldsymbol{u}, \boldsymbol{u}' \neq \mathbf{0}$. For these we have $V(\boldsymbol{u}) = \lfloor \Delta^2 \cdot (\Delta - z) \rfloor$, and $V(\boldsymbol{u}') = \lfloor \Delta^2 \cdot (\Delta - z') \rfloor$.

Note that since both z and z' are coordinates of vertices of $\mathcal{P}$, whose description has highest coefficient of $\max\{\max_{i,j \in [d]} M(i,j), \max_{i \in [d]} |q_i|\}$, and therefore their numerator and denominator both are bounded above by Δ. Therefore, if $z < z'$ then we have

$$z' - z \ge \frac{1}{\Delta^2} \Rightarrow ((\Delta - z) - (\Delta - z')) \cdot \Delta^2 \ge 1 \Rightarrow V(\boldsymbol{u}) - V(\boldsymbol{u}') \ge 1.$$

For (i), if $z = z'$ then clearly $V(\boldsymbol{u}) = V(\boldsymbol{u}')$, and from the above argument it also follows that if $V(\boldsymbol{u}) = V(\boldsymbol{u}')$ then it can not be the case that $z \neq z'$. Similarly for (ii), if $V(\boldsymbol{u}) > V(\boldsymbol{u}')$ then clearly, $z' > z$, and from the above argument it follows that if $z' > z$ then it can not be the case that $V(\boldsymbol{u}') \ge V(\boldsymbol{u})$. ☐

Using the above lemma, we will next show that instance $\mathcal{E}$ has no local maximizer.

Lemma 75. *Let $\boldsymbol{u}, \boldsymbol{v} \in \mathsf{vert}$ s.t. $\boldsymbol{u} \neq \boldsymbol{v}$, $\boldsymbol{v} = S(\boldsymbol{u})$, and $\boldsymbol{u} = P(\boldsymbol{v})$. Then $V(\boldsymbol{u}) < V(\boldsymbol{v})$.*

Proof. Let $\boldsymbol{x} = (\mathbf{y}, \mathbf{w}, z)$ and $\boldsymbol{x}' = (\mathbf{y}', \mathbf{w}', z')$ be the vertices in polyhedron $\mathcal{P}$ corresponding to $\boldsymbol{u}$ and $\boldsymbol{v}$ respectively. From the construction of $\boldsymbol{v} = S(\boldsymbol{u})$ implies that $z' < z$. Therefore, using Lemma 74 it follows that $V(\boldsymbol{v}) < V(\boldsymbol{u})$. □

Due to Lemma 75 the only type of solutions available in $\mathcal{E}$ is (R1) where $S(P(\boldsymbol{u})) \neq \boldsymbol{u}$ and $P(S(\boldsymbol{u})) \neq \boldsymbol{u}$. Next two lemmas shows how to construct solution of P-LCP instance $\mathcal{I}$ or a (PV1) type violation (non-positive principle minor of matrix M) from these.

Lemma 76. *Let $\boldsymbol{u} \in \mathsf{vert}$, $\boldsymbol{u} \neq 0^n$. If $P(S(\boldsymbol{u})) \neq \boldsymbol{u}$ or $S(P(\boldsymbol{u})) \neq \boldsymbol{u}$, then $IsValid(\boldsymbol{u}) = 1$. Futhermore, for $(\mathbf{y}, \mathbf{w}, z) = EtoI(\boldsymbol{u})$ if $z = 0$, then $\mathbf{y}$ is a (Q1) type solution of* P-LCP *instance $\mathcal{I} = (M, \boldsymbol{q})$.*

Proof. By construction, if $\text{IsValid}(\boldsymbol{u}) = 0$, then $S(P(\boldsymbol{u})) = \boldsymbol{u}$ and $P(S(\boldsymbol{u})) = \boldsymbol{u}$, therefore $\text{IsValid}(\boldsymbol{u}) = 0$ when $\boldsymbol{u}$ has a predecessor or successor different from $\boldsymbol{u}$. Given this, from Lemma 72 we know that $(\mathbf{y}, \mathbf{w}, z)$ is a feasible vertex in (18). Therefore, if $z = 0$, then by Lemma 71 we have a solution of the LCP (1), *i.e.*, a type (Q1) solution of our P-LCP instance $\mathcal{I} = (M, \boldsymbol{q})$. □

Lemma 77. *Let $\boldsymbol{u} \in \mathsf{vert}$, $\boldsymbol{u} \neq 0^n$ such that $P(S(\boldsymbol{u})) \neq \boldsymbol{u}$ or $S(P(\boldsymbol{u})) \neq \boldsymbol{u}$, and let $\boldsymbol{x} = (\mathbf{y}, \mathbf{w}, z) = EtoI(\boldsymbol{u})$. If $z \neq 0$ then $\boldsymbol{x}$ has a duplicate label, say l. And for directions σ_1 and σ_2 obtained by relaxing $y_l = 0$ and $w_l = 0$ respectively at $\boldsymbol{x}$, we have $\sigma_1(z) \cdot \sigma_2(z) \geq 0$, where $\sigma_i(z)$ is the coordinate corresponding to z.*

Proof. From Lemma 76 we know that $\text{IsValid}(\boldsymbol{u}) = 1$, and therefore from Lemma 72, $\boldsymbol{x}$ is a feasible vertex in (18). From the last line of Tables 4 and 6 observe that $S(\boldsymbol{u})$ points to the configuration of vertex next to $\boldsymbol{x}$ on Lemke's path only if it has lower z value otherwise it gives back $\boldsymbol{u}$, and similarly $P(\boldsymbol{u})$ points to the previous only if value of z increases.

First consider the case when $P(S(\boldsymbol{u})) \neq \boldsymbol{u}$. Let $\boldsymbol{v} = S(\boldsymbol{u})$ and corresponding vertex in $\mathcal{P}$ be $(\mathbf{y}', \mathbf{w}', z') = \text{EtoI}(\boldsymbol{v})$. If $\boldsymbol{v} \neq \boldsymbol{u}$, then from the above observation we know that $z' > z$, and in that case again by construction of P we will have $P(\boldsymbol{v}) = \boldsymbol{u}$, contradicting $P(S(\boldsymbol{u})) \neq \boldsymbol{u}$. Therefore, it must be the case that $\boldsymbol{v} = \boldsymbol{u}$. Since $z \neq 0$ this happens only when the next vertex on Lemke path after $\boldsymbol{x}$ has higher value of z (by above observation). As a consequence of $\boldsymbol{v} = \boldsymbol{u}$, we also have $P(\boldsymbol{u}) \neq \boldsymbol{u}$. By construction of P this implies for $(\mathbf{y}'', \mathbf{w}'', z'') = \text{EtoI}(P(\boldsymbol{u}))$, $z'' > z$. Putting both together we get increase in z when we relax $y_l = 0$ as well as when we relax $w_l = 0$ at $\boldsymbol{x}$.

For the second case $S(P(\boldsymbol{u})) \neq \boldsymbol{u}$ similar argument gives that value of z decreases when we relax $y_l = 0$ as well as when we relax $w_l = 0$ at $\boldsymbol{x}$. The proof follows. □

Finally, we are ready to prove our main result of this section using Lemmas 75, 76 and 77. Together with Lemma 77, we will use the fact that on Lemke path z monotonically decreases if M is a P-matrix or else we get a (PV1) type witness that M is not a P-matrix [14].

Theorem 78. *There is a polynomial-time promise-preserving reduction from* P-LCP *with (PV1) violations to* EndOfPotentialLine.

Proof. Given an instance of $\mathcal{I} = (M, \boldsymbol{q})$ of P-LCP, where $M \in \mathbb{R}^{d \times d}$ and $\boldsymbol{q} \in \mathbb{R}^{d \times 1}$ reduce it to an instance $\mathcal{E}$ of EndOfPotentialLine as described above with vertex set $\mathsf{vert} = \{0, 1\}^{2d}$ and procedures S, P and V as given in Table 4, 6, and 5 respectively.

Among solutions of EndOfPotentialLine instance $\mathcal{E}$, there is no local potential maximizer, i.e., $\boldsymbol{u} \neq \boldsymbol{v}$ such that $\boldsymbol{v} = S(\boldsymbol{u})$, $\boldsymbol{u} = P(\boldsymbol{v})$ and $V(\boldsymbol{u}) > V(\boldsymbol{v})$ due to Lemma 75. We get a solution

$\boldsymbol{u} \neq 0$ such that either $S(P(\boldsymbol{u})) \neq \boldsymbol{u}$ or $P(S(\boldsymbol{u})) \neq \boldsymbol{u}$, then by Lemma 76 it is valid configuration and has a corresponding vertex $\boldsymbol{x} = (\mathbf{y}, \mathbf{w}, z)$ in $\mathcal{P}$. Again by Lemma 76 if $z = 0$ then $\mathbf{y}$ is a (Q1) type solution of our P-LCP instance $\mathcal{I}$. On the other hand, if $z > 0$ then from Lemma 77 we get that on both the two adjacent edges to $\boldsymbol{x}$ on Lemke path the value of z either increases or deceases. This gives us a minor of M which is non-positive [14], i.e., a (Q2) type solution of the P-LCP instance $\mathcal{I}$ with (PV1) violation.

The reduction is promise preserving because if the LCP instance is promised to be P-LCP then z monotonically decreases along the Lemke's path, and all feasible complementary vertices are on this path. Therefore, the corresponding EndOfPotentialLine instance will have exactly one path ending in a solution where the corresponding vertex $\boldsymbol{x} = (\mathbf{y}, \mathbf{w}, z)$ of the LCP has $z = 0$ mapping to the P-LCP solution. □

G.3 Reduction from P-LCP with (PV2) violations to UniqueEOPL

Next we show that the above construction also implies P-LCP with (PV2) violations is in UniqueEOPL, and thereby also in EndOfPotentialLine. We start with a simple well-known lemma that turns two solutions of an LCP $(M, \boldsymbol{q})$ into a non-zero sign-reversing vector for M, i.e. a (PV2) violation.

Lemma 79. *[14, Theorem 3.3.7] If for some d-dimensional vector $\boldsymbol{q}'$, LCP $(M, \boldsymbol{q}')$ has more than one solution then there exists a sign-reversing vector $\boldsymbol{x}$ w.r.t. M, i.e., $x_i(Mx)_i \leq 0, \quad \forall i \in [d]$.*

Proof. Let $(\mathbf{y}', \mathbf{w}')$ and $(\mathbf{y}^*, \mathbf{w}^*)$ be two distinct solutions of the LCP defined by $(M, \boldsymbol{q}')$. That is $\mathbf{y}' \neq \mathbf{y}^*$. Then,

$$\mathbf{w}^* = M\mathbf{y}^* + \boldsymbol{q}' \ \text{ and } \ \mathbf{w}' = M\mathbf{y}' + \boldsymbol{q}' \Rightarrow (\mathbf{w}^* - \mathbf{w}') = M(\mathbf{y}^* - \mathbf{y}')$$

Furthermore, for each $i \in [d]$ we have $w_i^* y_i^* = 0, w_i' y_i' = 0, w_i^* y_i' \geq 0$ and $w_i' y_i^* \geq 0$. This together with $(\mathbf{w}^* - \mathbf{w}')_i = (M(\mathbf{y}^* - \mathbf{y}'))_i$ gives,

$$\forall i \in [d], \quad (\mathbf{y}^* - \mathbf{y}')_i (\mathbf{w}^* - \mathbf{w}')_i \leq 0 \quad \Rightarrow \quad (\mathbf{y}^* - \mathbf{y}')_i (M(\mathbf{y}^* - \mathbf{y}'))_i \leq 0$$

Thus, $\boldsymbol{x} = \mathbf{y}^* - \mathbf{y}'$ is our desired vector. Note that $\boldsymbol{x} \neq 0$ since $\mathbf{y}' \neq \mathbf{y}^*$. □

UniqueEOPL has four types of solutions. Out of these, (UV1) is ruled out by Lemma 75. Next we show that any "extra" end of lines as well as (UV3) type solutions map to a (PV2) violation.

Lemma 80. *Given either of the following, we can construct two distinct solutions of LCP $(M, \boldsymbol{q}')$ for some $\boldsymbol{q}'$:*

(a) $\boldsymbol{u} \in \mathsf{vert}$ is a (U1) or (UV2) type solution of instance $\mathcal{E}$ such that corresponding vertex $\boldsymbol{x} = (\mathbf{y}, \mathbf{w}, z) = EtoI(\boldsymbol{u})$ has $z^* > 0$.

(b) $\boldsymbol{u}, \boldsymbol{v} \in \mathsf{vert}$ forms a (UV3) type solution of instance $\mathcal{E}$.

Proof. The common idea to go from (a) or (b) to two solutions of some LCP with matrix M is to create two (or more) solutions of (18) with same z value. Suppose $(\mathbf{y}^*, \mathbf{w}^*, a)$ and $(\mathbf{y}', \mathbf{w}', a)$ with $\mathbf{y} \neq \mathbf{y}'$ are feasible in (18) for some $a \in \mathbb{R}$, then clearly for $\boldsymbol{q}' = \boldsymbol{q} + a$, $(\mathbf{y}^*, \mathbf{w}^*)$ and $(\mathbf{y}', \mathbf{w}')$ are solutions of LCP (1) with matrix M and vector $\boldsymbol{q}'$.

For (a), let $\boldsymbol{x}^* = (\mathbf{y}^*, \mathbf{w}^*, z^*) = \mathrm{EtoI}(\boldsymbol{u})$ with $z^* > 0$, and let l be the duplicate label at vertex $\boldsymbol{x}^*$ in (Q1). Then from Lemma 77 we know that for directions σ_1 and σ_2 obtained by relaxing $y_l = 0$ and $w_l = 0$ respectively at $\boldsymbol{x}^*$, we have $\sigma_1(z) * \sigma_2(z) \geq 0$, where $\sigma_i(z)$ is the coordinate corresponding to z. Suppose $\sigma_1(z), \sigma_2(z) < 0$, and for $i = 1, 2$, let z_i be the value of z at the vertex adjacent to $\boldsymbol{x}^*$ in direction of σ_i; set $z_i = -\infty$ if no vertex encountered in direction σ_i. Let $\epsilon > 0$ be small enough so that $\epsilon < (z^* - z_i),\ i = 1, 2$, and consider the points $\boldsymbol{x}^i = (\boldsymbol{x}^* + \frac{\epsilon}{|\sigma_i(z)|}\sigma_i)$ on the edge corresponding to σ_i adjacent to $\boldsymbol{x}^*$. It is easy to check that by choice of ϵ, both $\boldsymbol{x}^1$ and $\boldsymbol{x}^2$ are feasible. We will next show that these are solutions of an LCP defined by $(M, \boldsymbol{q}')$ for some $\boldsymbol{q}'$.

Note that by construction the z coordinate at both $\boldsymbol{x}^1$ and $\boldsymbol{x}^2$ is $(z^* - \epsilon)$, giving us desired two solutions of (18) with the same z value. Similar, argument holds when $\sigma_1(z), \sigma_2(z) > 0$ where the corresponding z value is $(z^* + \epsilon)$. If either $\sigma_1(z)$ or $\sigma_2(z)$ is zero, then z remains unchanged on the entire corresponding edge.

For (b), $\boldsymbol{x}^* = (\mathbf{y}^*, \mathbf{w}^*, z^*) = \text{EtoI}(\boldsymbol{u})$ and $\boldsymbol{x}' = (\mathbf{y}', \mathbf{w}', z') = \text{EtoI}(\boldsymbol{v})$. If $V(\boldsymbol{u}) = V(\boldsymbol{v})$, then clearly $z^* = z'$ (Lemma 74) and we get the desired two points feasible in (18) with the same z value. If $V(\boldsymbol{u}) < V(\boldsymbol{v}) < V(S(\boldsymbol{u}))$, then there exists a point on the edge joining $\boldsymbol{x}^*$ with $\text{EtoI}(S(\boldsymbol{x}^*))$ with the same z value as z'. ☐

Now we are ready to show our main result of P-LCP with PV2 in UniqueEOPL using Lemmas 75, 76, 79 and 80

Theorem 81. *There is a polynomial-time promise-preserving reduction from* P-LCP *with (PV2) violations to* UniqueEOPL*, and thereby also to* EndOfPotentialLine.

Proof. Given an instance of $\mathcal{I} = (M, \boldsymbol{q})$ of P-LCP, where $M \in \mathbb{R}^{d \times d}$ and $\boldsymbol{q} \in \mathbb{R}^{d \times 1}$ reduce it to an instance $\mathcal{E}$ of UniqueEOPL as described above with vertex set $\mathsf{vert} = \{0, 1\}^{2d}$ and procedures S, P and V as given in Table 4, 6, and 5 respectively.

Lemma 75 rules out UV1 violation in $\mathcal{E}$. If we get U1 solution or UV2 violation $\boldsymbol{u}$ of $\mathcal{E}$, then corresponding vertex $\boldsymbol{x} = (\mathbf{y}, \mathbf{w}, z)$ is feasible in (18) by Lemma 76. Furthermore, if $z = 0$ then $\mathbf{y}$ is a $(Q1)$ type solution of our P-LCP instance $\mathcal{I}$. On the other hand if $z > 0$, then by Lemmas 80 and 79 we can construct a PV2 violation of our P-LCP instance $\mathcal{I}$. Similarly, Lemmas 80 and 79 also map any UV3 violation of UniqueEOPL instance $\mathcal{E}$ to a PV2 violation of $\mathcal{I}$.

By construction, if $\mathcal{I}$ is a promise P-LCP instance, then instance $\mathcal{E}$ of UniqueEOPL will have exactly one U1 solution corresponding to the unique solution of the $\mathcal{I}$. ☐

H Proofs for Section 5.1: Algorithms for Contraction Maps

In this section, we provide an exact algorithm for solving PL-Contraction, i.e., we either return a rational fixpoint of polynomial bit-length or a pair of points that prove (indirectly) that the given function is not a contraction map. which is guaranteed to have a rational fixpoint of polynomial bit-length or two points that prove (indirectly) that the given function is not contracting. Then we extend this algorithm to find an approximate fixpoint of general contraction maps for which there may not be an exact solution of polynomial bit length. In both cases, the problems solved by our algorithm are not promise problems and we always return either a solution or a violation. Our algorithms work for any ℓ_p norm with $p \in \mathbb{N}$, and are polynomial for constant dimension d. These are the first such algorithms for $p \neq 2$. Such algorithms were so far only known for the ℓ_2 and ℓ_∞ norms [43, 70, 71][5]

H.1 Overview: algorithm to find a fixed-point of PL-Contraction

The algorithm does a nested binary search using Lemmas 59 and 60 to find fixpoints of slices with increasing numbers of free coordinates. We illustrate the algorithm in two dimensions in Figure 6.

[5]Our approach does not cover the ℓ_∞ norm, as that would require more work and not give a new result.

The algorithm is recursive. To find the eventual fixpoint in d dimensions we fix a single coordinate s_1, find the unique $(d-1)$-dimensional fixpoint of $f_{|s}$, the $(d-1)$-dimensional contraction map obtained by fixing the first coordinate of the input to f to be s_1. Let x the unique fixpoint of $f_{|s}$ where $x_1 = s_1$. If $f(x_1) > s_1$, then the d-dimensional fixpoint x^* of f has $x_1^* > s_1$, and if $f(x_1) < s_1$, then $x_1^* < s_1$ (Lemma 60). We can thus do a binary search for the value of x_1^*. Once we've found x_1^*, we can recursively find the $(d-1)$-dimensional fixpoint of $f_{|s}$ where $s_1 = x_1$. The resulting solution will be the d-dimensional fixpoint. At each step in the recursive procedure, we do a binary search for the value of one coordinate of the fixpoint at the slice determined by all the coordinates already fixed. For piecewise-linear functions, we know that all fixpoints are rational with bounded bit-length (as discussed in Section F.1), so we can find each coordinate exactly.

If at any step in recursion our binary search finds two $k-1$ dimensional fixpoints on slices that are adjacent, differing only in the kth coordinate and by a small enough amount , we can return these points, which witness the failure of f to be a contraction map. These points correspond to a solution of type (CMV3) to the PL-Contraction problem. The proof that f is not a contraction is indirect, and uses the fact that the discretized grid implicitly searched by the algorithm will contain every fixpoint of f. Since we maintain the invariant that our two pivots bound the coordinate we're searching over from above and below when f is a contraction map, such a pair of points gives proof that f is not contracting.

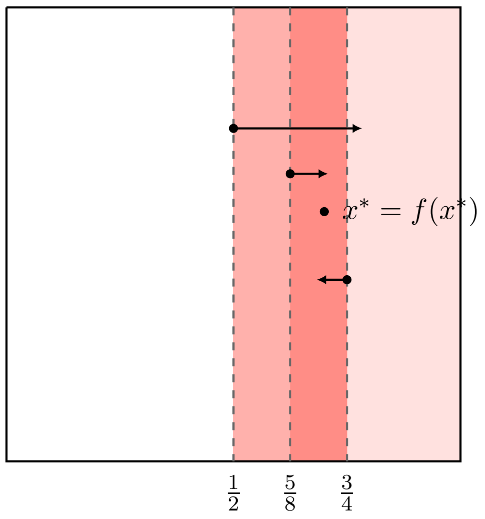


Figure 6: An illustration of the algorithm to find a fixpoint of a piecewise-linear contraction map in two dimensions. The algorithm begins by finding a fixpoint along the slice with $x_1 = 1/2$. The fixpoint along that slice points to the right, so we next find a fixpoint along the slice with $x_1 = 3/4$. The fixpoint along that slice points to the left, so we find the fixpoint along $x_1 = 5/8$. We successively find fixpoints of one-dimensional slices, and then use those to do a binary search for the two-dimensional fixpoint. The red regions are the successive regions considered by the binary search, where each successive step in the binary search results in a darker region.

Using this algorithm we obtain the following theorem.

Theorem 82. *Given a* LinearFIXP *circuit C purporting to encode a contraction map $f : [0,1]^d \to [0,1]^d$ with respect to any ℓ_p norm, there is an algorithm to find a fixpoint of f or return a pair of points witnessing that f is not a contraction map in time that is polynomial in size(C) and exponential in d.*

The full details of the algorithm can be found in Appendix H.3.

H.2 Overview: algorithm to find an approximate fixed-point of Contraction

Here we generalize our algorithm to find an approximate fixpoint of an arbitrary function given by an arithmetic circuit, i.e., our algorithm solves Contraction, which is specified by a circuit f that represents the contraction map,[6] a p-norm, and ε. Again, let d denote the dimension of the problem, i.e. the number of inputs (and outputs) of f. Let x^* denote the unique exact fixpoint for the contraction map f. We seek an approximate fixpoint, i.e., a point for which $\|f(x) - x\|_p \leq \varepsilon$.

We do the same recursive binary search as in the algorithm above, but at each step of the algorithm instead of finding an exact fixpoint, we will only find an approximate fixpoint of $f_{|s}$. The difficulty in this case will come from the fact that Lemma 60 does not apply to approximate fixpoints. Consider the example illustrated in Figure 7. In this example, y is the unique fixpoint of the slice restriction along the gray dashed line. By Lemma 60, $(f(y)_1 - y_1)(x^*_1 - y_1) \geq 0$ so if we find y, we can observe that $f(y)_1 > y_1$ and recurse on the right side of the figure, in the region labeled $\mathcal{R}$. If we try to use the same algorithm but where we only find approximate fixopints at each step, we'll run into trouble. In this case, if we found z instead of y, we would observe that $f(z)_1 < z_1$ and conclude that $x^*_1 < z_1$, which is incorrect. As a result, we would limit our search to the region labeled $\mathcal{L}$, and wouldn't be able to find x^*.

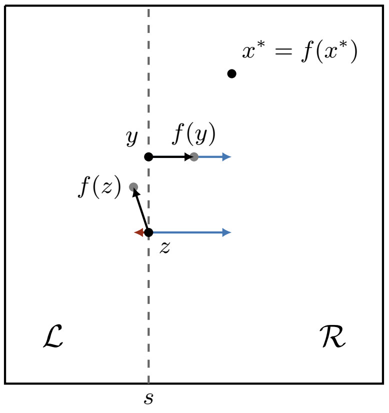


Figure 7: A step in the recursive binary search. Here, x^* is the fixpoint for the original function, y is the fixpoint for the slice restriction $f_{|s}$ along the dashed gray line, and z is an approximate fixpoint to the slice restriction.

When looking for an approximate fixpoint, we'll have to choose a different precision ε_i for each level of the recursion so that either the point x returned by the ith recursive call to our algorithm satisfies $|f(x)_i - x_i| > \varepsilon_i$ and we can rely on it for pivoting in the binary search, or $|f(x)_i - x_i| \leq \varepsilon_i$ and we can return x as an approximate fixpoint to the recursive call one level up. Each different ℓ_p norm will require a different choice of $(\varepsilon_i)_{i=1}^d$.

Using this idea we are able to obtain the following results:

Theorem 83. *For a contraction map $f : [0,1]^d \to [0,1]^d$ with respect to the ℓ_1 norm, there is an algorithm to compute a point $v \in [0,1]^d$ such that $\|f(v) - v\|_1 < \varepsilon$ or report a violation of contraction in time $O(d^d \log(1/\varepsilon))$.*

[6]The algorithm works even if f is given as an arbitrary black-box, as long as it is guaranteed to be a contraction map.

Theorem 84. *For a contraction map $f : [0,1]^d \to [0,1]^d$ under $\|\cdot\|_p$ for $2 \le p < \infty$, there is an algorithm to compute a point $v \in [0,1]^d$ such that $\|f(v) - v\|_p < \varepsilon$ or report a violation of contraction in time $O(p^{d^2} \log^d(1/\varepsilon) \log^d(pd))$.*

The full details of the algorithms can be found in Appendix H.4.

H.3 Details: finding a fixed-point of PL-Contraction

Suppose f is piecewise-linear; it follows that the coordinates of the unique fixpoints of $f_{|s}$ will be rational numbers with bounded denominators. Consider the values of κ_i's computed in Section F.1. The analysis in the section tells us that if we consider an i-slice s where s_j is a rational with bit-length at most κ_j for $j \in \{i+1, \dots, d\}$, the unique fixed-point of $f_{|s}$ will have coordinates with bit-lengths of at most $\kappa_1, \dots, \kappa_d$ Furthermore, κ_1 is the largest among all κ_i's and is bounded by polynomial in the size of the circuit C representing the given PL-Contraction instance.

Let $\mathsf{Slice}_d(2^\kappa)$ be the set of slices with fixed coordinates having denominators at most 2^{κ_i} in the ith coordinate:

$$\mathsf{Slice}_d(2^\kappa) \iff s \in \mathsf{Slice}_d \text{ and } s_i \in \{0/2^{\kappa_i}, 1/k_i, \dots, 2^{\kappa_i}/2^{\kappa_i}\}, \ \forall i \in \text{fixed}(s).$$

We will design an algorithm assuming an upper bound of 2^{κ_i} on the ith coordinate denominators of fixpoints for any slice restriction $f_{|s}$ where $s \in \mathsf{Slice}_d(2^\kappa)$.

Analysis. Given a $k < d$ and a k-slice $s \in \mathsf{Slice}_d(2^\kappa)$, we know from Lemma 59 that the restricted function $f_{|s}$ is also contracting. We will show that FindFP, the function given in Algorithm 1, computes a fixpoint of $f_{|s}$. Since the algorithm is recursive, we will prove its correctness by induction. The next lemma establishes the base case of induction and follows by design of the algorithm. It is equivalent to finding a fixpoint of a one dimensional function.

Lemma 85. *For any 1-slice s,* FindFP(s) *returns the unique fixpoint of $f_{|s}$ if it doesn't throw an error.*

Now for the inductive step, assuming FindFP can compute a fixpoint of any k-slice restriction of a contraction map, we will show that it can compute one for any $(k+1)$-slice of the contraction map.

Lemma 86. *Fix any k-slice $s \in \mathsf{Slice}_d(2^\kappa)$ and let $t = (*, \dots, *, t_k, s_{k+1}, \dots, s_d)$ for some $t_k \in \{0/2^{\kappa_k}, 1/2^{\kappa_k}, \dots, 2^{\kappa_k}/2^{\kappa_k}\}$. If* FindFP$(t)$ *returns the unique fixpoint of the restricted function $f_{|t}$, then* FindFP(s) *returns the unique fixpoint of the function $f_{|s}$ if it doesn't throw an error.*

Proof. Since f is a contraction map, so is $f_{|s}$ due to Lemma 59. Let $v =$ FindFP(t) be the value returned by the algorithm when input the slice t. Now, if $t_k = 0$ then $f(v)_k \ge 0 = t_k = v_k$ and if $t_k = 1$ then $f(v)_k \le 1 = t_k = v_k$. If either is an equality then v is the unique fixpoint of $f_{|s}$ as well and the lemma follows.

Otherwise, we know that the kth coordinate of the unique fixpoint of $f_{|s}$, which we'll call t_k^*, is between $t_k^{(\ell)} = 0$ and $t_k^{(h)} = 1$. Note that when t_k is set to t_k^*, the vector $v =$ FindFP(t) which is the unique fixpoint of $f_{|t}$ will also be the unique fixpoint of $f_{|s}$. Thus, it suffices to show that t_k will eventually be set to t_k^* during the execution of the algorithm.

The while loop of Algorithm 1 does a binary search between $t_k^{(h)}$ and $t_k^{(\ell)}$ to find t_k^*, while keeping track of the fixpoints of $f_{|t}$. We first observe that after line 17 of each execution of the loop in Algorithm 1, v is the unique fixpoint of $f_{|t}$. Therefore, whenever $f(v)_k = t_k$ is satisfied, we will return the unique fixpoint of $f_{|s}$ and the lemma follows.

Algorithm 1 Algorithm for PL-Contraction

```
1: Input: A $k$-slice $s \in \mathsf{Slice}_d(2^\kappa)$ for some $k \le d$.
2: Output: The unique fixpoint of $s$, i.e., a point $y$ such that $f_{|s}(y) = y_{|s}$ and $y = y_{|s}$.
3: function FindFP($s$)
4:     Let $k = |\mathrm{free}(s)|$.
5:     if $k = 0$ then return $s$
6:     end if
7:     Set $k \leftarrow k - 1$. Set $t^{(\ell)} \leftarrow s$, $t^{(h)} \leftarrow s$.
8:     Set $t^{(\ell)}_k \leftarrow 0$, $t^{(h)}_k \leftarrow 1$.
9:     Set $v^{(\ell)} \leftarrow$ FindFP($t^{(\ell)}$), and $v^{(h)} \leftarrow$ FindFP($t^{(h)}$).
10:    if $f(v^{(\ell)})_k = t^{(\ell)}_k$ then return $v^{(\ell)}$.
11:    end if
12:    if $f(v^{(h)})_k = t^{(h)}_k$ then return $v^{(h)}$.
13:    end if
14:    Set $t \leftarrow s$
15:    while $t^{(h)}_k - t^{(\ell)}_k > \frac{1}{2^{(\kappa_k - 1)}}$ do
16:        Set $t_k \leftarrow \frac{(t^{(h)}_k + t^{(\ell)}_k)}{2}$.
17:        Set $v \leftarrow$ FindFP($t$).
18:        if $f(v)_k = t_k$ then return $v$.
19:        end if
20:        if $f(v)_k > t_k$ then Set $t^{(\ell)}_k \leftarrow t_k$
21:        else Set $t^{(h)}_k \leftarrow t_k$.
22:        end if
23:    end while
24:    Set $t_k \leftarrow$ the unique number in $(t^{(\ell)}_k, t^{(h)}_k)$ with denominator at most $2^{\kappa_k}$.
25:    Set $v^* \leftarrow$ FindFP($t$).
26:    if $f(v^*)_k - v^*_k = 0$ then return FindFP($t$).
27:    else throw error: "The pair $\left(v^{(\ell)}, v^{(h)}\right)$ is a solution of type (CMV3)."
28:    end if
29: end function
```

The binary search maintains the invariant that if we let $v^{(\ell)} =$ FindFP$(t^{(\ell)})$ and $v^{(h)} =$ FindFP$(t^{(h)})$ we have $f(v^{(\ell)})_k > v^{(\ell)}_k$, and $f(v^{(h)})_k < v^{(h)}_k$. By Lemma 60, this invariant ensures that t^*_k satisfies

$$t^{(\ell)}_k < t^*_k < t^{(h)}_k$$

at all times. Therefore, at some point in the binary search either one of the endpoints is t^*_k and we return the desired fixpoint or we end the binary search with $t^{(\ell)}_k$ and $t^{(h)}_k$ such that $(t^{(h)}_k - t^{(\ell)}_k) \le 1/2^{\kappa_k - 1}$ and $t^*_k \in [t^{(\ell)}_k, t^{(h)}_k]$. By the assumption we know that t^*_k is a rational number with denominator at most 2^{κ_k}. Since there can be at most one such number in $(t^{(\ell)}_k, t^{(h)}_k)$, t^*_k can be uniquely identified. Let t^* denote the slice t after setting $t_k \leftarrow t^*_k$. The second to last line of Algorithm 1 will then return the unique fixpoint of $f_{|t^*}$, which will be the unique fixpoint of $f_{|s}$ is a contraction map. The only way this can fail to happen is if the point returned by FindFP(t^*) is not actually a k-dimensional fixpoint, in which case the algorithm will throw an error. ☐

We now address the case where the algorithm returns an error:

Lemma 87. *If* FindFP(t) *returns an error for k-slice $s \in \mathsf{Slice}_d(2^{\kappa_1}, 2^{\kappa_2}, \ldots, 2^{\kappa_d})$, the pair of points $(v^{(\ell)}, v^{(h)})$ indicated by the error witness that f is not a contraction map.*

Proof. The algorithm only returns an error when $t_k^{h(\ell)} - t_k^{(\ell)} \leq \frac{1}{2^{\kappa_k - 1}}$ and the point v^* returned by FindFP(t^*) is not a k-dimensional fixpoint, where t^* is the slice obtained by setting $t_k \leftarrow t^*$ in line 24. By induction we know that v^* must be a $(k-1)$-fixpoint, since the recursive call to the algorithm didn't throw an error. If f is a contraction map, then the fact that $v^{(\ell)}$ and $v^{(h)}$ are $(k-1)$-dimensional fixpoints of $f_{|s}$ with $f_{|s}(v^{(\ell)})_k - v_k^{(\ell)} > 0$ and $f_{|s}(v^{(h)})_k - v_k^{(h)} < 0$ together imply that the kth coordinate of the true fixpoint $f_{|s}$ will lie in $(v_k^{(\ell)}, v_k^{(h)})$ by Lemma 60. By Lemma 30, we know that any k-dimensional fixpoint of f has kth coordinate with bit-length at most κ_k. Thus, there is a unique value which the kth coordinate of the unique fixpoint of $f_{|s}$ can have, namely t_k^*. But v^* is not a fixpoint of $f_{|s}$ and so we must conclude that $f_{|s}$ is not a contraction map, which implies that f is not a contraction map. Thus, the pair $(v^{(\ell)}, v^{(h)})$ together witness that f is not a contraction map. □

Using Lemma 87 and applying induction using Lemma 85 as a base-case and Lemma 86 as an inductive step, the next theorem follows.

Theorem 88. FindFP$(*, *, \ldots, *)$ *returns the unique fixpoint of f, or a pair of points proving that f isn't a contraction map in time $O(L^d)$ where $L = \max_k L_k$ is polynomial in the size of the input instance.*

H.4 Details: finding an approximate fixed-point of Contraction

We now proceed to prove the correctness of our algorithm.

Analysis.

We will show that for any contraction map f with respect to an ℓ_p norm, and any $\varepsilon > 0$, if Algorithm 2 doesn't throw an error, then it returns an approximate fixpoint of f, i.e. a point $v \in [0,1]^d$ such that $\|f(v) - v\| \leq \varepsilon$. To do this, we'll show that for any $k < d$ and k-slice $s \in \mathsf{Slice}_d$ ApproxFindFP(s) will return a (s, ℓ_p, k)-approximate fixpoint (when it doesn't throw an error). Since Algorithm 2 is recursive, our proof will be by induction. The next lemma establishes the base case of the induction and follows by design of the algorithm.

Lemma 89. *For any* 1*-slice s,* ApproxFindFP(s) *returns a $(s, \ell_p, 1)$-approximate fixpoint when it doesn't throw an error.*

For the inductive step, we show that we can go from approximate fixpoints of $(k-1)$-slices to approximate fixpoints of k-slices.

Lemma 90. *Fix some k-slice $s \in \mathsf{Slice}_d$ and let $t = (*, \ldots, *, t_k, s_{k+1}, \ldots, s_d)$ for some $t_k \in [0,1]$. If* ApproxFindFP(t) *returns a $(t, \ell_p, k-1)$-approximate fixpoint v, then* ApproxFindFP(s) *returns an (s, ℓ_p, k)-approximate fixpoint when it doesn't throw an error.*

Proof. We observe that $f_{|s}$ is a contraction map by Lemma 59. We assume that $v =$ ApproxFindFP(t) is a $(t, \ell_p, k-1)$-approximate fixpoint of $f_{|t}$ for any value of $t_k \in [0,1]$.

We first observe that after the first recursive invocations of ApproxFindFP, as ApproxFindFP$(t^{(\ell)})$ and ApproxFindFP$(t^{(h)})$, if $\left|f(v^{(h)})_k - v_k^{(h)}\right| \leq \varepsilon_k(p,d)$ or $\left|f(v^{(\ell)})_k - v_k^{(\ell)}\right| \leq \varepsilon_k(p,d)$, we return $v^{(h)}$ or $v^{(\ell)}$, respectively, so the output of ApproxFindFP(s) satisfies the requirements of the lemma.

Moreover, in every subsequent call to ApproxFindFP(t), if the output v satisfies $|f(v)_k - v_k| \leq \varepsilon_k$, we return v, so we'll assume in what follows that the point v^* returned by the algorithm is from line 33.

We observe that v^* was the output of a call to ApproxFindFP with a $(k-1)$-slice t, so it is a $(t, \ell_p, k-1)$-approximate fixpoint and in order to get to the final line, we must have $|\Delta_k(v^*)| \leq \varepsilon_k(p, d)$. Thus, we have that v^* is a (t, ℓ_p, k)-approximate fixpoint. □

Applying induction using Lemma 89 as a base-case and Lemma 90 as an inductive step, we obtain the following lemma:

Lemma 91. *For a contraction map with respect to an ℓ_p norm for $p < \infty$, and the trivial slice $s = (*, *, \ldots, *)$,* ApproxFindFP(s) *returns a (s, ℓ_p, d)-approximate fixpoint when it doesn't throw an error.*

Lemma 92. *If Algorithm 2 throws an error, the pair (x, y) indicated by the error witnesses f not being a contraction map.*

Proof. Assume that the error was thrown in a call to ApproxFindFP(s) where s is a k-slice for some $k \leq d$. Then by inspection we can see that x and y satisfy $x_k < y_k$ and $y_k - x_k < \varepsilon_k(p, d)/2$. Moreover, x and y were returned by calls to ApproxFindFP with $(k-1)$-slices that we'll denote $t^{(x)}$ and $t^{(y)}$, respectively. It follows from the inductive argument above that x and y are $(t^{(x)}, \ell_p, k-1)$ and $(t^{(y)}, \ell_p, k-1)$-approximate fixpoints, respectively. Furthermore, to get to the line in which the error is thrown, we must have $\Delta_k(x) \geq \varepsilon_k(p, d)$ and $\Delta_k(y) \leq -\varepsilon_k(p, d)$. By Lemma 63, we immediately obtain the desired conclusion. □

Now applying Lemma 65 and analyzing the runtime of our algorithm, we obtain our final results.

Theorem 93. *For a contraction map with respect to the ℓ_1 norm,* ApproxFindFP$(*, *, \ldots, *)$ *returns a point $v \in [0, 1]^d$ such that $\|f(v) - v\|_1 < \varepsilon$ or reports a violation of contraction in time $O(d^d \log(1/\varepsilon))$.*

Proof. In the worst case, the ith recursive call to ApproxFindFP will terminate with $t_i^{(h)} - t_i^{(\ell)} < \varepsilon_i = \varepsilon/4^i$ so will require at most $O(\log(1/\varepsilon)i)$ iterations. The total runtime will thus be bounded by $O(d^d \log^d(1/\varepsilon))$. □

Theorem 94. *For a contraction map under $\|\cdot\|_p$ for $2 \leq p < \infty$,* ApproxFindFP$(*, *, \ldots, *)$ *returns a point $v \in [0, 1]^d$ such that $\|f(v) - v\|_p < \varepsilon$ or reports a violation of contraction in time $O(p^{d^2} \log^d(1/\varepsilon) \log^d(dp))$.*

Proof. In the worst case, the ith recursive call to ApproxFindFP will terminate with $t_i^{(h)} - t_i^{(\ell)} < \varepsilon_i = \varepsilon^{p^i} (dp)^{-2 \sum_{j=0}^{i} p^j}$ so will require $O(p^i \log(1/\varepsilon) \log(dp))$ iterations. The total runtime will thus be bounded by $O(p^{d^2} \log^d(1/\varepsilon) \log^d(dp))$. □

Algorithm 2 Algorithm for CONTRACTION, for a given f, ε, ℓ_p

```
Input: A $k$-slice $s \in \mathsf{Slice}_d$ for some $k \leq d$.
Output: An $(s, \ell_p, k)$-approximate fixpoint of $f_{|s}$.
function ApproxFindFP($s$)
    Let $k = |\mathrm{free}(s)|$.
    if $k = d$ then return $s$
    end if
    Set $k \leftarrow k - 1$. Set $t^{(\ell)} \leftarrow s$, $t^{(h)} \leftarrow s$.
    Set $t^{(\ell)}_k \leftarrow 0$, $t^{(h)}_k \leftarrow 1$.
    Set $v^{(\ell)} \leftarrow$ ApproxFindFP($t^{(\ell)}$), and $v^{(h)} \leftarrow$ ApproxFindFP($t^{(h)}$).
    if $\left|\Delta_k\left(v^{(\ell)}\right)\right| \leq \varepsilon_k(p, d)$ then return $v^{(\ell)}$.
    end if
    if $\left|\Delta_k\left(v^{(h)}\right)\right| \leq \varepsilon_k(p, d)$ then return $v^{(h)}$.
    end if
    Set $t \leftarrow s$
    while $t^{(h)}_k - t^{(\ell)}_k > \varepsilon_k(p, d)$ do
        Set $t_k \leftarrow \frac{(t^{(h)}_k + t^{(\ell)}_k)}{2}$.
        Set $v \leftarrow$ ApproxFindFP($t$).
        if $|\Delta_k(v)| \leq \varepsilon_k(p, d)$ then return $v$.
        end if
        if $f(v)_k > t_k$ then Set $t^{(\ell)}_k \leftarrow t_k$
        else Set $t^{(h)}_k \leftarrow t_k$.
        end if
    end while
    Set $t_k \leftarrow \frac{(t^{(h)}_k + t^{(\ell)}_k)}{2}$.
    Set $v^* \leftarrow$ ApproxFindFP($t$)
    if $|\Delta_k(v^*)| > \varepsilon_k(p, d)$ then
        if $\Delta_k(v^*) > \varepsilon_k(p, d)$ then
            throw error: "The pair $\left(v^*, v^{(h)}\right)$ witnesses $f$ not being a contraction map."
        else
            throw error: "The pair $\left(v^{(\ell)}, v^*\right)$ witnesses $f$ not being a contraction map."
        end if
    end if
    return $v^*$.
end function
```