Tree Polymatrix Games are PPAD-hard

Argyrios Deligkas
Royal Holloway University of London, UK
Argyrios.Deligkas@rhul.ac.uk

John Fearnley
University of Liverpool, UK
John.Fearnley@liverpool.ac.uk

Rahul Savani
University of Liverpool, UK
Rahul.Savani@liverpool.ac.uk

Abstract

We prove that it is PPAD-hard to compute a Nash equilibrium in a tree polymatrix game with twenty actions per player. This is the first PPAD hardness result for a game with a constant number of actions per player where the interaction graph is acyclic. Along the way we show PPAD-hardness for finding an ϵ-fixed point of a 2D-LinearFIXP instance, when ϵ is any constant less than $(\sqrt{2}-1)/2 \approx 0.2071$. This lifts the hardness regime from polynomially small approximations in k-dimensions to constant approximations in two-dimensions, and our constant is substantial when compared to the trivial upper bound of 0.5.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness; Exact and approximate computation of equilibria

Keywords and phrases Nash Equilibria, Polymatrix Games, PPAD, Brouwer Fixed Points

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

A *polymatrix game* is a succinctly represented many-player game. The players are represented by vertices in an *interaction graph*, where each edge of the graph specifies a two-player game that is to be played by the adjacent vertices. Each player picks a *pure strategy*, or *action*, and then plays that action in all of the edge-games that they are involved with. They then receive the *sum* of the payoffs from each of those games. A *Nash equilibrium* prescribes a mixed strategy to each player, with the property that no player has an incentive to unilaterally deviate from their assigned strategy.

Constant-action polymatrix games have played a central role in the study of equilibrium computation. The classical PPAD-hardness result for finding Nash equilibria in bimatrix games [4] uses constant-action polymatrix games as an intermediate step in the reduction [4,5]. Rubinstein later showed that there exists a constant $\epsilon > 0$ such that computing an ϵ-approximate Nash equilibrium in two-action bipartite polymatrix games is PPAD-hard [15], which was the first result of its kind to give hardness for constant ϵ.

These hardness results create polymatrix games whose interaction graphs contain cycles. This has lead researchers to study *acyclic* polymatrix games, with the hope of finding tractable cases. Kearns, Littman, and Singh claimed to produce a polynomial-time algorithm for finding a Nash equilibrium in a two-action tree *graphical game* [11], where graphical games are a slight generalization of polymatrix games. However, their algorithm does not work, which was pointed out by Elkind, Goldberg, and Goldberg [9], who also showed that the natural fix gives an exponential-time algorithm.

Elkind, Goldberg, and Goldberg also show that a Nash equilibrium can be found in polynomial time for two-action graphical games whose interaction graphs contain only paths and cycles. They also show that finding a Nash equilibrium is PPAD-hard when the interaction graph has pathwidth at most four, but there appears to be some issues with their approach (see Appendix A). Later work of Barman, Ligett, and Piliouras [1] provided a QPTAS for constant-action tree polymatrix games, and then Ortiz and Irfan [13] gave an FPTAS for this case. All three papers, [1,9,13], leave as a main open problem the question of whether it is possible to find a Nash equilibrium in a tree polymatrix game in polynomial time.

Our contribution. In this work we show that finding a Nash equilibrium in twenty-action tree polymatrix games is PPAD-hard. Combined with the known PPAD containment of polymatrix games [5], this implies that the problem is PPAD-complete. This is the first hardness result for polymatrix (or graphical) games in which the interaction graph is acyclic, and decisively closes the open question raised by prior work: tree polymatrix games cannot be solved in polynomial time unless PPAD is equal to P.

Our reduction produces a particularly simple class of interaction graphs: all of our games are played on *caterpillar* graphs (see Figure 3) which consist of a single path with small one-vertex branches affixed to every node. These graphs have pathwidth 1, so we obtain a stark contrast with prior work: two-action path polymatrix games can be solved in polynomial time [9], but twenty-action pathwidth-1-caterpillar polymatrix games are PPAD-hard.

Our approach is founded upon Mehta's proof that 2D-LinearFIXP is PPAD-hard [12]. We show that her reduction can be implemented by a synchronous arithmetic circuit with *constant width*. We then embed the constant-width circuit into a caterpillar polymatrix game, where each player in the game is responsible for simulating all gates at a particular level of the circuit. This differs from previous hardness results [5,15], where each player is responsible for simulating exactly one gate from the circuit.

Along the way, we also substantially strengthen Mehta's hardness result for LinearFIXP.

She showed PPAD-hardness for finding an exact fixed point of a `2D-LinearFIXP` instance, and an ϵ-fixed point of a `kD-LinearFIXP` instance, where ϵ is polynomially small. We show PPAD-hardness for finding an ϵ-fixed point of a `2D-LinearFIXP` instance when ϵ is any constant less than $(\sqrt{2}-1)/2 \approx 0.2071$. So we have lifted the hardness regime from polynomially small approximations in k-dimensions to constant approximations in two-dimensions, and our constant is substantial when compared to the trivial upper bound of 0.5.

Related work. The class PPAD was defined by Papadimitriou [14]. Years later, Daskalakis, Goldberg, and Papadimitriou (DGP) [5] proved PPAD-hardness for graphical games and 3-player normal form games. Chen, Deng, and Teng (CDT) [4] extended this result to 2-player games and proved that there is no FPTAS for the problem unless PPAD = P. The observations made by CDT imply that DGP's result also holds for polymatrix games with constantly-many actions (but with cycles in the interaction graph) for an exponentially small ϵ. More recently, Rubinstein [16] showed that there exists a *constant* $\epsilon > 0$ such that computing an $\epsilon - NE$ in binary-action bipartite polymatrix games is PPAD-hard (again with cycles in the interaction graph).

Etessami and Yiannakakis [10] defined the classes FIXP and LinearFIXP and they proved that LinearFIXP = PPAD. Mehta [12] strengthened these results by proving that two-dimensional LinearFIXP equals PPAD, building on the result of Chen and Deng who proved that 2D-discrete Brouwer is PPAD-hard [3].

On the positive side, Cai and Daskalakis [2], proved that NE can be efficiently found in polymatrix games where every 2-player game is zero-sum. Ortiz and Irfan [13] and Deligkas, Fearnley, and Savani [7] produced QPTASs for polymatrix games of bounded treewidth (in addition to the FPTAS of [13] for tree polymatrix games mentioned above). For general polymatrix games, the only positive result to date is a polynomial-time algorithm to compute a $(\frac{1}{2}+\delta)$-NE [8]. Finally, an empirical study on algorithms for exact and approximate NE in polymatrix games can be found in [6].

2 Preliminaries

Polymatrix games. An n-player *polymatrix game* is defined by an undirected *interaction graph* $G = (V, E)$ with n vertices, where each vertex represents a player, and the edges of the graph specify which players interact with each other. Each player in the game has m actions, and each edge $(v, u) \in E$ of the graph is associated with two $m \times m$ matrices $A^{v,u}$ and $A^{u,v}$ which specify a bimatrix game that is to be played between the two players, where $A^{v,u}$ specifies the payoffs to player v from their interaction with player u.

Each player in the game selects a single action, and then plays that action in *all* of the bimatrix games with their neighbours in the graph. Their payoff is the *sum* of the payoffs that they obtain from each of the individual bimatrix games.

A *mixed strategy* for player i is a probability distribution over the m actions of that player, a *strategy profile* is a vector $\mathbf{s} = (s_1, s_2, \dots, s_n)$ where s_i is a mixed strategy for player i. The *vector of expected payoffs* for player i under strategy profile $\mathbf{s}$ is $\mathbf{p}_i(\mathbf{s}) := \sum_{(i,j)\in E} A^{i,j} s_j$. The *expected payoff* to player i under $\mathbf{s}$ is $s_i \cdot \mathbf{p}_i(\mathbf{s})$. A strategy profile is a *mixed Nash equilibrium* if $s_i \cdot \mathbf{p}_i(\mathbf{s}) = \max \mathbf{p}_i(\mathbf{s})$ for all i, which means that no player can unilaterally change their strategy in order to obtain a higher expected payoff. In this paper we are interested in the problem of computing a Nash equilibrium of a *tree* polymatrix game, which is a polymatrix game in which the interaction graph is a tree.

Arithmetic circuits. For the purposes of this paper, each gate in an arithmetic circuit will operate only on values that lie in the range $[0, 1]$. In our construction, we will use four specific gates, called *constant introduction* denoted by c, *bounded addition* denoted by $+^b$, *bounded subtraction* denoted by $-^b$, and *bounded multiplication by a constant* denoted by $*^b c$. These gates are formally defined as follows.

- c is a gate with no inputs that outputs some fixed constant $c \in [0,1]$.
- Given inputs $x, y \in [0,1]$ the gate $x +^b y := \min(x+y, 1)$.
- Given inputs $x, y \in [0,1]$ the gate $x -^b y := \max(x-y, 0)$.
- Given an input $x \in [0,1]$, and a constant $c \geq 0$, the gate $x *^b c := \min(x * c, 1)$.

These gates perform their operation, but also clip the output value so that it lies in the range $[0,1]$. Note that the constant c in the $*^b c$ gate is specified as part of the gate. Multiplication of two inputs is not allowed.

We will build arithmetic circuits that compute functions of the form $[0,1]^d \to [0,1]^d$. A circuit $C = (I, G)$ consists of a set $I = \{\mathtt{in}_1, \mathtt{in}_2, \ldots, \mathtt{in}_d\}$ containing d input nodes, and a set $G = \{g_1, g_2, \ldots, g_k\}$ containing k gates. Each gate g_i has a type from the set $\{c, +^b, -^b, *^b c\}$, and if the gate has one or more inputs, these are taken from the set $I \cup G$. The connectivity structure of the gates is required to be a directed acyclic graph.

The *depth* of a gate, denoted by $d(g)$ is the length of the longest path from that gate to an input. We will build *synchronous* circuits, meaning that all gates of the form $g_x = g_y +^b g_z$ satisfy $d(g_x) = 1 + d(g_y) = 1 + d(g_z)$, and likewise for gates of the form $g_x = g_y -^b g_z$. There are no restrictions on c-gates, or $*^b c$-gates.

The *width* of a particular level i of the circuit is defined to be $w(i) = |\{g_j \,:\, d(g_j) = i\}|$, which is the number of gates at that level. The *width* of a circuit is defined to be $w(C) = \max_i w(i)$, which is the maximum width taken over all the levels of the circuit.

Straight line programs. A convenient way of specifying an arithmetic circuit is to write down a straight line program (SLP) [10].

SLP 1 Example

```
x ← 0.5
z ← x +^b in1
x ← x *^b 0.5
out1 ← z +^b x
```

SLP 2 `if` and `for` example

```
x ← in1 *^b 1
for i in {1, 2, ..., 10} do
    if i is even then
        x ← x +^b 0.1
    end
end
out1 ← x *^b 1
```

Each line of an SLP consists of a statement of the form `v` $\leftarrow$ `op`, where `v` is a *variable*, and `op` consists of exactly one arithmetic operation from the set set $\{c, +^b, -^b, *^b c\}$. The inputs to the gate can be any variable that is defined before the line, or one of the inputs to the circuit. We permit variables to be used on the left hand side in more than one line, which effectively means that we allow variables to be overwritten.

It is easy to turn an SLP into a circuit. Each line is turned into a gate, and if variable `v` is used as the input to gate g, then we set the corresponding input of g to be the gate g' that corresponds to the line that most recently assigned a value to `v`. SLP 1 above specifies a circuit with four gates, and the output of the circuit will be $0.75 +^b \mathtt{in}_1$.

For the sake of brevity, we also allow `if` statements and `for` loops in our SLPs. These two pieces of syntax can be thought of as macros that help us specify a straight line program concisely. The arguments to an `if` statement or a `for` loop must be constants that do not depend on the value of any gate in the circuit. When we turn an SLP into a circuit, we unroll every `for` loop the specified number of times, and we resolve every `if` statement by deleting the block if the condition does not hold. So the example in SLP 2 produces a circuit with seven gates: two gates correspond to the lines `x` $\leftarrow$ `in1` $*^b$ `1` and `out1` $\leftarrow$ `x` $*^b$ `1`, while there are five gates corresponding to the line `x` $\leftarrow$ `x` $+^b$ `0.1`, since there are five copies of the line remaining after we unroll the loop and resolve the `if` statements. The output of the resulting circuit will be $0.5 +^b \mathtt{in}_1$.

Liveness of variables and circuit width. Our ultimate goal will be to build circuits that have small width. To do this, we can keep track of the number of variables that are *live* at any one time in our SLPs. A variable `v` is live at line i of an SLP if both of the following conditions are met.

- There exists a line with index $j \leq i$ that assigns a value to `v`.
- There exists a line with index $k \geq i$ that uses the value assigned to `v` as an argument.

The number of variables that are live at line i is denoted by $\text{live}(i)$, and the number of variables *used by* an SLP is defined to be $\max_i \text{live}(i)$, which is the maximum number of variables that are live at any point in the SLP. The following is proved in Appendix B.

▶ **Lemma 1.** *An SLP that uses w variables can be transformed into a polynomial-size synchronous circuit of width w.*

3 Hardness of 2D-Brouwer

In this section, we consider the following problem. It is a variant of two-dimensional Brouwer that uses only our restricted set of bounded gates.

▶ **Definition 2** (`2D-Brouwer`). *Given an arithmetic circuit $F : [0,1]^2 \to [0,1]^2$ using gates from the set $\{c, +^b, -^b, *^b\ c\}$, find $x \in [0,1]^2$ such that $F(x) = x$.*

As a starting point for our reduction, we will show that this problem is `PPAD`-hard. Our proof will follow the work of Mehta [12], who showed that the closely related `2D-LinearFIXP` problem is `PPAD`-hard. There are two differences between `2D-Brouwer` and `2D-LinearFIXP`.

- In `2D-LinearFIXP`, all internal gates of the circuit take and return values from $\mathbb{R}$ rather than $[0,1]$.
- `2D-LinearFIXP` takes a circuit that uses gates from the set $\{c, +, -, *c, \max, \min\}$, where none of these gates bound their outputs to be in $[0,1]$.

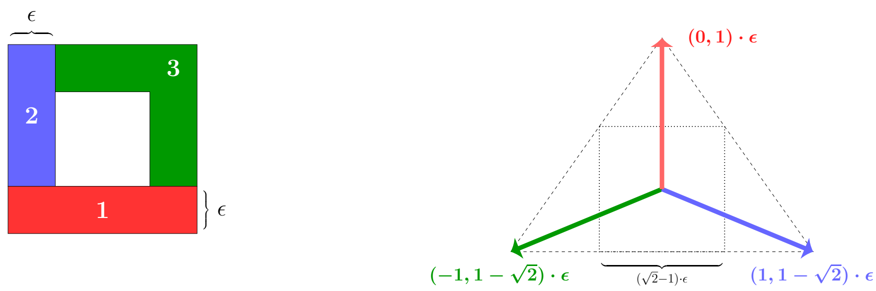
In this section, we present an altered version of Mehta's reduction, which will show that finding an ϵ-solution to `2D-Brouwer` is `PPAD`-hard for a constant ϵ.

Discrete Brouwer. The starting point for Mehta's reduction is the two-dimensional discrete Brouwer problem, which is known to be `PPAD`-hard [3]. This problem is defined over a discretization of the unit square $[0,1]^2$ into a grid of points $G = \{0, 1/2^n, 2/2^n, \dots, (2^n - 1)/2^n\}^2$. The input to the problem is a Boolean circuit $C : G \to \{1,2,3\}$ the assigns one of three colors to each point. The coloring will respect the following boundary conditions.

- We have $C(0,i) = 1$ for all $i \geq 0$.
- We have $C(i,0) = 2$ for all $i > 0$.
- We have $C(\frac{2^n-1}{2^n}, i) = C(i, \frac{2^n-1}{2^n}) = 3$ for all $i > 0$.

These conditions can be enforced syntactically by modifying the circuit. The problem is to find a grid square that is *trichromatic*, meaning that all three colors appear on one of the four points that define the square.

▶ **Definition 3** (`DiscreteBrouwer`). *Given a Boolean circuit $C : \{0,1\}^n \times \{0,1\}^n \to \{1,2,3\}$ that satisfies the boundary conditions, find a point $x, y \in \{0,1\}^n$ such that, for each color $i \in \{1,2,3\}$, there exists a point (x', y') with $C(x', y') = i$ where $x' \in \{x, x+1\}$ and $y' \in \{y, y+1\}$.*



(a) Our stronger boundary conditions. **(b)** The mapping from colors to vectors.

Figure 1 Reducing ϵ-`ThickDisBrouwer` to `2D-Brouwer`.

Our first deviation from Mehta's reduction is to insist on the following stronger boundary condition, which is shown in Figure 1a.

- We have $C(i, j) = 1$ for all i, and for all $j \leq \epsilon$.
- We have $C(i, j) = 2$ for all $j > \epsilon$, and for all $i \leq \epsilon$.
- We have $C(i, j) = C(j, i) = 3$ for all $i > \epsilon$, and all $j \geq 1 - \epsilon$.

The original boundary conditions placed constraints only on the outermost grid points, while these conditions place constraints on a border of width ϵ. We call this modified problem ϵ-`ThickDisBrouwer`, which is the same as `DiscreteBrouwer`, except that the function is syntactically required to satisfy the new boundary conditions.

It is not difficult to produce a polynomial time reduction from `DiscreteBrouwer` to ϵ-`ThickDisBrouwer`. It suffices to increase the number of points in the grid, and then to embed the original `DiscreteBrouwer` instance into the $[\epsilon, 1-\epsilon]^2$ square in the middle of the instance. The proof of the following lemma can be found in Appendix C.

▶ **Lemma 4.** *`DiscreteBrouwer` can be reduced in polynomial time to ϵ-`ThickDisBrouwer`.*

Embedding the grid in $[0, 1]^2$. We now reduce ϵ-`ThickDisBrouwer` to `2D-Brouwer`. One of the keys steps of the reduction is to map points from the continuous space $[0, 1]^2$ to the discrete grid G. Specifically, given a point $x \in [0, 1]$, we would like to determine the n bits that define the integer $\lfloor x \cdot 2^n \rfloor$.

Mehta showed that this mapping from continuous points to discrete points can be done by a linear arithmetic circuit. Here we give a slightly different formulation that uses only gates from the set $\{c, +^b, -^b, *^b c\}$. Let L be a fixed constant that will be defined later.

SLP 3 `ExtractBit(x, b)`

```
b ← 0.5
b ← x -^b b
b ← b *^b L
```

SLP 4 `ExtractBits(x, b_1, b_2, ..., b_n)`

```
for i in {1, 2, ..., n} do
    ExtractBit(x, b_i)
    y ← b_i *^b 0.5
    x ← x -^b y
    x ← x *^b 2
end
```

SLP 3 extracts the first bit of the number $x \in [0, 1]$. The first three lines of the program compute the value $b = (x -^b 0.5) *^b L$. There are three possibilities.

- If $x \leq 0.5$, then $b = 0$.
- If $x \geq 0.5 + 1/L$, then $b = 1$.

- If $0.5 < x < 0.5 + 1/L$, then b will be some number strictly between 0 and 1.

The first two cases correctly decode the first bit of x, and we call these cases *good decodes*. We will call the third case a *bad decode*, since the bit has not been decoded correctly.

SLP 4 extracts the first n bits of x, by extracting each bit in turn, starting with the first bit. The three lines after each extraction erase the current first bit of x, and then multiply x by two, which means that the next extraction will give us the next bit of x. If any of the bit decodes are bad, then this procedure will break, meaning that we only extract the first n bits of x in the case where all decodes are good. We say that x is *well-positioned* if the procedure succeeds, and *poorly-positioned* otherwise.

Multiple samples. The problem of poorly-positioned points is common in PPAD-hardness reductions. Indeed, observe that we cannot define an SLP that always correctly extracts the first n bits of x, since this would be a discontinuous function, and all gates in our arithmetic circuits compute continuous functions. As in previous works, this is resolved by taking multiple samples around a given point. Specifically, for the point $p \in [0,1]^2$, we sample k points $p_1, p_2, \ldots, p_k$ where $p_i = p + \left(\frac{i-1}{(k+1)\cdot 2^{n+1}}, \frac{i-1}{(k+1)\cdot 2^{n+1}}\right)$. Mehta proved that there exists a setting for L that ensures that there are at most two points that have poorly positioned coordinates. We have changed several details, and so we provide our own statement and proof here. The proof can be found in Appendix D.

▶ **Lemma 5.** *If $L = (k+2) \cdot 2^{n+1}$, then at most two of the points p_1 through p_k have poorly-positioned coordinates.*

Evaluating a Boolean circuit. Once we have decoded the bits for a well-positioned point, we have a sequence of 0/1 variables. It is easy to simulate a Boolean circuit on these values.

- The operator $\neg\, x$ can be simulated by $1 -^b x$.
- The operator $x \vee y$ can be simulated by $x +^b y$.
- The operator $x \wedge y$ can be simulated by applying De Morgan's laws and using $\vee$ and $\neg$.

Recall that C outputs one of three possible colors. We also assume, without loss of generality, that C gives its output as a *one-hot vector*. This means that there are three Boolean outputs $x_1, x_2, x_3 \in \{0,1\}^3$ of the circuit. The color 1 is represented by the vector $(1,0,0)$, the color 2 is represented as $(0,1,0)$, and color 3 is represented as $(0,0,1)$. If the simulation is applied to a point with well-positioned coordinates, then the circuit will output one of these three vectors, while if it is applied to a point with poorly positioned coordinates, then the circuit will output some value $x \in [0,1]^3$ that has no particular meaning.

The output. The key idea behind the reduction is that each color will be mapped to a displacement vector, as shown in Figure 1b. Here we again deviate from Mehta's reduction, by giving different vectors that will allow us to prove our approximation lower bound.

- Color 1 will be mapped to the vector $(0,1) \cdot \epsilon$.
- Color 2 will be mapped to the vector $(1, 1-\sqrt{2}) \cdot \epsilon$.
- Color 3 will be mapped to the vector $(-1, 1-\sqrt{2}) \cdot \epsilon$.

These are irrational coordinates, but in our proofs we argue that a suitably good rational approximation of these vectors will suffice. We average the displacements over the k different sampled points to get the final output of the circuit. Suppose that x_{ij} denotes output i from sampled point j. Our circuit will compute

$$\mathtt{disp}_x = \sum_{j=1}^{k} \frac{(x_{2j} - x_{3j}) \cdot \epsilon}{k}, \quad \mathtt{disp}_y = \sum_{j=1}^{k} \frac{\left(x_{1j} + (1-\sqrt{2})(x_{2j} + x_{3j})\right) \cdot \epsilon}{k}.$$

Finally, we specify $F : [0,1]^2 \to [0,1]^2$ to compute $F(x,y) = (x + \mathtt{disp}_x \cdot \epsilon, y + \mathtt{disp}_y \cdot \epsilon)$.

Completing the proof. To find an approximate fixed point of F, we must find a point where both $\mathtt{disp}_x$ and $\mathtt{disp}_y$ are close to zero. The dotted square in Figure 1b shows the set of displacements that satisfy $\|x - (0,0)\|_\infty \leq (\sqrt{2}-1)\cdot\epsilon$, which correspond to the displacements that would be $(\sqrt{2}-1)\cdot\epsilon$-fixed points.

The idea is that, if we do not sample points of all three colors, then we cannot produce a displacement that is strictly better than an $(\sqrt{2}-1)\cdot\epsilon$-fixed point. For example, if we only have points of colors 1 and 2, then the displacement will be some point on the dashed line between the red and blue vectors in Figure 1b. This line touches the box of $(\sqrt{2}-1)\cdot\epsilon$-fixed points, but does not enter it. It can be seen that the same property holds for the other pairs of colors: we specifically chose the displacement vectors in order to maximize the size of the inscribed square shown in Figure 1b.

The argument is complicated by the fact that two of our sampled points may have poorly positioned coordinates, which may drag the displacement towards $(0,0)$. However, this effect can be minimized by taking a large number of samples. We show show the following lemma.

▶ **Lemma 6.** *Let $\epsilon' < (\sqrt{2}-1)\cdot\epsilon$ be a constant. There is a sufficiently large constant k such that, if $\|x - F(x)\|_\infty < \epsilon'$, then x is contained in a trichromatic square.*

The proof of Lemma 6 can be found in Appendix E. Since ϵ can be fixed to be any constant strictly less than 0.5, we obtain the following.

▶ **Theorem 7.** *Given a `2D-Brouwer` instance, it is `PPAD`-hard to find a point $x \in [0,1]^2$ s.t. $\|x - F(x)\|_\infty < (\sqrt{2}-1)/2 \approx 0.2071$.*

Reducing `2D-Brouwer` to `2D-LinearFIXP` is easy, since the gates $\{c, +^b, -^b, *^b c\}$ can be simulated by the gates $\{c, +, -, *c, \max, \min\}$. This implies that it is `PPAD`-hard to find an ϵ-fixed point of a `2D-LinearFIXP` instance with $\epsilon < (\sqrt{2}-1)/2$.

It should be noted that an ϵ-approximate fixed point can be found in polynomial time if the function has a suitably small Lipschitz constant, by trying all points in a grid of width ϵ. We are able to obtain a lower bound for constant ϵ because our functions have exponentially large Lipschitz constants.

4 Hardness of `2D-Brouwer` with a constant width circuit

In our reduction from `2D-Brouwer` to tree polymatrix games, the number of actions in the game will be determined by the width of the circuit. This means that the hardness proof from the previous section is not a sufficient starting point, because it produces `2D-Brouwer` instances that have circuits with high width. In particular, the circuits will extract $2n$ bits from the two inputs, which means that the circuits will have width at least $2n$.

Since we desire a constant number of actions in our tree polymatrix game, we need to build a hardness proof for `2D-Brouwer` that produces a circuit with constant width. In this section we do exactly that, by reimplementing the reduction from the previous section using gadgets that keep the width small.

Bit packing. We adopt an idea of Elkind, Goldberg, and Goldberg [9], to store many bits in a single arithmetic value using a *packed* representation. Given bits $b_1, b_2, \ldots, b_k \in \{0,1\}$, the packed representation of these bits is the value $\text{packed}(b_1, b_2, \ldots, b_k) := \sum_{i=1}^{k} b_i/2^i$. We will show that the reduction from the previous section can be performed while keeping all Boolean values in a single variable that uses packed representation.

Working with packed variables. We build SLPs that work with this packed representation, two of which are shown below.

SLP 5 FirstBit(x, b) +0 variables

```
// Extract the first bit of x
   into b
b ← 0.5
b ← x -^b b
b ← b *^b L

// Remove the first bit of x
b ← b *^b 0.5
x ← x -^b b
x ← x *^b 2
b ← b *^b 2
```

SLP 6 Clear(I, x) +2 variables

```
x' ← x *^b 1
for i in {1,2,...,k} do
    b ← 0
    FirstBit(x', b)
    if i ∈ I then
        b ← b *^b 1/2^i
        x ← x -^b b
    end
end
```

The FirstBit SLP combines the ideas from SLPs 3 and 4 to extract the first bit from a value $x \in [0, 1]$. Repeatedly applying this SLP allows us to read out each bit of a value in sequence. The Clear SLP uses this to set some bits of a packed variable to zero. It takes as input a set of indices I, and a packed variable $x = \text{packed}(b_1, b_2, \ldots, b_k)$. At the end of the SLP we have $x = \text{packed}(b'_1, b'_2, \ldots, b'_k)$ where $b'_i = 0$ whenever $i \in I$, and $b'_i = b_i$ otherwise.

It first copies x to a fresh variable x'. The bits of x' are then read-out using FirstBit. Whenever a bit b_i with $i \in I$ is decoded from x', we subtract $b_i/2^i$ from x. If $b_i = 1$, then this sets the corresponding bit of x to zero, and if $b_i = 0$, then this leaves x unchanged.

We want to minimize the the width of the circuit that we produce, so we keep track of the number of *extra* variables used by our SLPs. For FirstBit, this is zero, while for Clear this is two, since that SLP uses the fresh variables x' and b.

Packing and unpacking bits. We implement two SLPs that manipulated packed variables. The Pack(x, y, S) operation allows us to extract bits from $y \in [0, 1]$, and store them in x, while the Unpack(x, y, S) operation allows us to extract bits from x to create a value $y \in [0, 1]$. This is formally specified in the following lemma, which is proved in Appendix F.

▶ **Lemma 8.** *Suppose that we are given* $\mathtt{x} = \text{packed}(b_1, b_2, \ldots, b_k)$*, a variable* $\mathtt{y} \in [0, 1]$*, and a sequence of indices* $S = \langle s_1, s_2, \ldots, s_j \rangle$*. Let* y_j *denote the* j*th bit of* y*. The following SLPs can be implemented using at most two extra variables.*

- *Pack(x, y, S) modifies* x *so that* $\mathtt{x} = \text{packed}(b'_1, b'_2, \ldots, b'_k)$ *where* $b'_i = y_j$ *whenever there exists an index* $s_j \in S$ *with* $s_j = i$*, and* $b'_i = b_i$ *otherwise.*
- *Unpack(x, y, S) modifies* y *so that* $\mathtt{y} = \mathtt{y} +^b \sum_{i=1}^{j} b_{s_i}/2^i$

Simulating a Boolean operations. As described in the previous section, the reduction only needs to simulate or- and not-gates. Given $\mathtt{x} = \text{packed}(b_1, b_2, \ldots, b_k)$, and three indices i_1, i_2, i_3, we implement two SLPs, which both modify x so that $\mathtt{x} = \text{packed}(b'_1, b'_2, \ldots, b'_k)$. SLP 7 implements $\mathtt{Or}(\mathtt{x}, \mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)$, which ensures that $b'_{i_3} = b_{i_1} \vee b_{i_2}$, and $b'_i = b_i$ for $i \neq i_3$. SLP 8 implements $\mathtt{Not}(\mathtt{x}, \mathtt{i}_1, \mathtt{i}_2)$, which ensures that $b'_{i_2} = \neg b_{i_1}$, and $b'_i = b_i$ for $i \neq i_2$.

These two SLPs simply unpack the input bits, perform the operation, and then pack the result into the output bit. The Or SLP uses the Unpack operation to set $\mathtt{a} = b_{i_1} +^b b_{i_2}$. Both SLPs use three extra variables: the fresh variable a is live throughout, and the pack and unpack operations use two extra variables. The variable b in the Not SLP is not live concurrently with a pack or unpack, and so does not increase the number of live variables. These two SLPs can be used to simulate a Boolean circuit using at most three extra variables.

SLP 7 Or(x, i_1, i_2, i_3) +3 variables

```
a ← 0
Unpack(x, a, ⟨i1⟩)
Unpack(x, a, ⟨i2⟩)
Pack(x, a, ⟨i3⟩)
```

SLP 8 Not(x, i_1, i_2) +3 variables

```
a ← 0
Unpack(x, a, ⟨i1⟩)
b ← 1
a ← b -^b a
Pack(x, a, ⟨i2⟩)
```

▶ **Lemma 9.** *Let C be a Boolean circuit with n inputs and k gates. Suppose that $x = \text{packed}(b_1, \dots, b_n)$, gives values for the inputs of the circuit. There is an SLP* `Simulate(C, x)` *that uses three extra variables, and modifies x so that $x = \text{packed}(b_1, \dots, b_n, b_{n+1}, \dots, b_{n+k})$, where b_{n+i} is the output of gate i of the circuit.*

Implementing the reduction. Finally, we can show that the circuit built in Theorem 7 can be implemented by an SLP that uses at most 8 variables. This SLP cycles through each sampled point in turn, computes the x and y displacements by simulating the Boolean circuit, and then adds the result to the output. The following theorem is proved in Appendix H

▶ **Theorem 10.** *Given a* `2D-Brouwer` *instance, it is* PPAD*-hard to find a point $x \in [0,1]^2$ with $\|x - F(x)\|_\infty < \frac{\sqrt{2}-1}{2}$ even for a synchronous circuit of width eight.*

5 Hardness for tree polymatrix games

Now we show that finding a Nash equilibrium of a tree polymatrix game is PPAD-hard. We reduce from the low-width `2D-Brouwer` problem, whose hardness was shown in Theorem 10. Throughout this section, we suppose that we have a `2D-Brouwer` instance defined by a synchronous arithmetic circuit F of width eight and depth n. The gates of this circuit will be indexed as $g_{i,j}$ where $1 \le i \le 8$ and $1 \le j \le n$, meaning that $g_{i,j}$ is the ith gate on level j.

Modifying the circuit. The first step of the reduction is to modify the circuit. First, we modify the circuit so that all gates operate on values in $[0, 0.1]$, rather than $[0, 1]$. We introduce the operators $+^b_{0.1}$, $-^b_{0.1}$, and $*^b_{0.1}$, which bound their outputs to be in $[0, 0.1]$. The following lemma, proved in Appendix I, states that we can rewrite our circuit using these new gates. The transformation simply divides all c-gates in the circuit by ten.

▶ **Lemma 11.** *Given an arithmetic circuit $F : [0,1]^2 \to [0,1]^2$ that uses gates from $\{c, +^b, -^b, *^b\}$, we can construct a circuit $F' : [0, 0.1]^2 \to [0, 0.1]^2$ that uses the gates from $\{c, +^b_{0.1}, -^b_{0.1}, *^b_{0.1}\}$, so that $F(x, y) = (x, y)$ if and only if $F'(x/10, y/10) = (x/10, y/10)$.*

Next we modify the structure of the circuit by connecting the two outputs of the circuit to its two inputs. Suppose, without loss of generality, that $g_{7,1}$ and $g_{8,1}$ are the inputs and that $g_{7,n}$ and $g_{8,n}$ are outputs. Note that the equality $x = y$ can be implemented using the gate $x = y *^b_{0.1} 1$. We add the following extra equalities, which are shown in Figure 2.

- We add gates $g_{9,n-1} = g_{7,n}$ and $g_{10,n-1} = g_{8,n}$.
- For each j in the range $2 \le j < n - 1$, we add $g_{9,j} = g_{9,j+1}$ and $g_{10,j} = g_{10,j+1}$.
- We modify $g_{7,1}$ so that $g_{7,1} = g_{9,2}$, and we modify $g_{8,1}$ so that $g_{8,1} = g_{10,2}$.

Note that these gates are backwards: they copy values from higher levels in the circuit to lower levels, and so the result is not a circuit, but a system of constraints defined by gates, with some structural properties. Firstly, each gate $g_{i,j}$ is only involved in constraints with

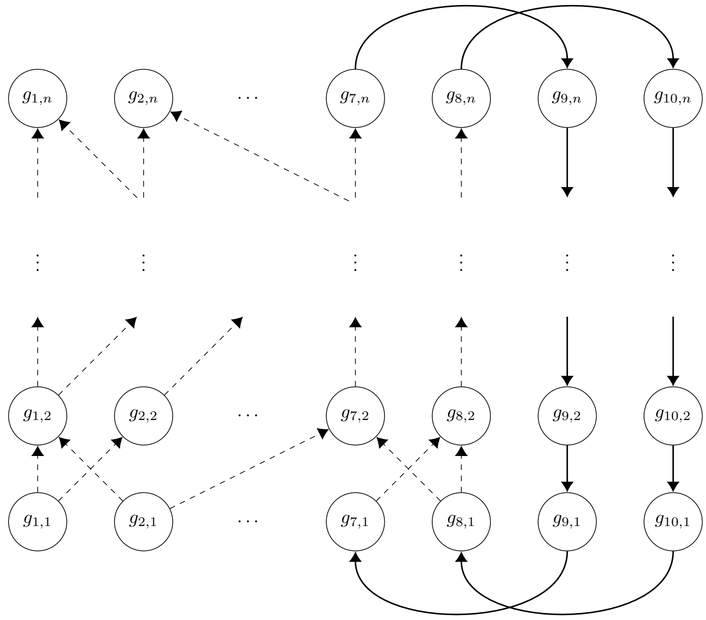


Figure 2 Extra equalities to introduce feedback of $g_{7,n}$ and $g_{8,n}$ to $g_{7,1}$ and $g_{8,1}$ respectively.

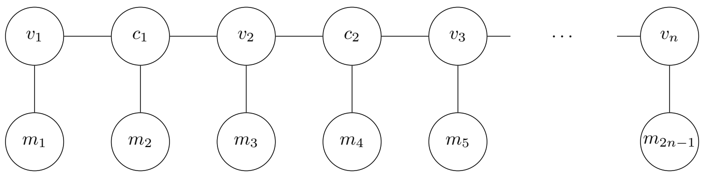


Figure 3 The structure of the polymatrix game.

gates of the form $g_{i',j+1}$ and $g_{i',j-1}$. Secondly, finding values for the gates that satisfy all of the constraints is PPAD-hard, since by construction such values would yield a fixed point of F.

The polymatrix game. The polymatrix game will contain three types of players.

- For each $i = 1, \dots, n$, we have a *variable* player v_i.
- For each $i = 1, \dots, n-1$, we have a *constraint* player c_i, who is connected to v_i and v_{i+1}.
- For each $i = 1, \dots, 2n-1$, we have a *mix* player m_i. If i is even, then m_i is connected to $c_{i/2}$. If i is odd, then m_i is connected to $v_{(i+1)/2}$.

The structure of this game is shown in Figure 3. Each player has twenty actions, which are divided into ten pairs, x_i and $\bar{x}_i$ for $i = 1, \dots, 10$.

Forcing mixing. The role of the mix players is to force the variable and constraint players to play specific mixed strategies: for every variable or constraint player j, we want $s_j(x_i) + s_j(\bar{x}_i) = 0.1$ for all i, which means that the same amount of probability is assigned to each pair of actions. To force this, each mix player plays a high-stakes hide-and-seek against their opponent, which is shown in Figure 4. This zero-sum game is defined by a 20×20 matrix Z and a constant M. The payoff Z_{ij} is defined as follows. If $i \in \{x_a, \bar{x}_a\}$ and $j \in \{x_a, \bar{x}_a\}$ for some a, then $Z_{ij} = M$. Otherwise, $Z_{ij} = 0$. For each i the player m_i plays

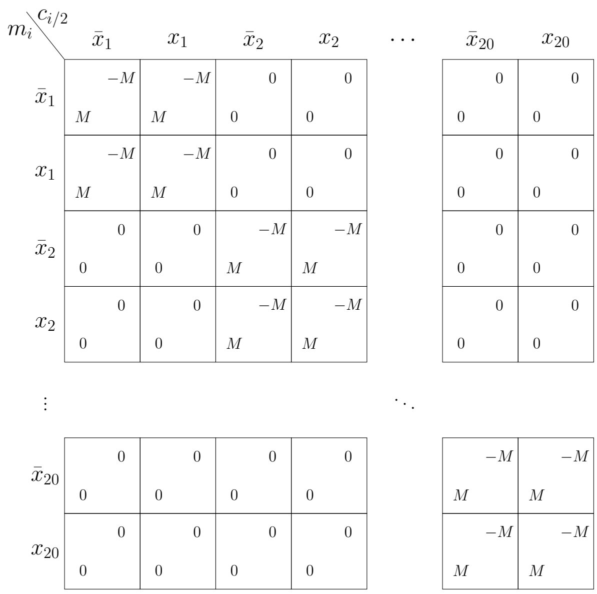


Figure 4 The hide and seek game that forces $c_{j/2}$ to play an appropriate mixed strategy. The same game is used to force $v_{(j-1)/2}$ mixes appropriately.

against player j, which is either a constraint player $c_{i'}$ or a variable player $v_{i'}$. We define the payoff matrix $A^{m_i,j} = Z$ and $G^{j,m_i} = -Z$. The following lemma, proved in Appendix J, shows that if M is suitably large, then the variable and constraint players must allocate probability 0.1 to each of the ten action pairs.

▶ **Lemma 12.** *Suppose that all payoffs in the games between variable and constraint players use payoffs in the range $[-P, P]$. If $M > 40 \cdot P$ then in every mixed Nash equilibrium $\mathbf{s}$, the action s_j of every variable and constraint player j satisfies $s_j(x_i) + s_j(\bar{x}_i) = 0.1$ for all i.*

Gate gadgets. We now define the payoffs for variable and constraint players. Actions x_i and $\bar{x}_i$ of variable player v_j will represent the output of gate $g_{i,j}$. Specifically, the probability that player v_j assigns to action x_i will be equal to the output of $g_{i,j}$. In this way, the strategy of variable player v_j will represent the output of every gate at level j of the circuit. The constraint player c_j enforces all constraints between the gates at level j and the gates at level $j+1$. To simulate each gate, we will embed one of the gate gadgets from Figure 5, which originated from the reduction of DGP [5], into the bimatrix games that involve c_j.

The idea is that, for the constraint player to be in equilibrium, the variable players must play x_i with probabilities that exactly simulate the original gate. Lemma 12 allows us to treat each gate independently: each pair of actions x_i and $\mathbf{s}_i$ must receive probability 0.1 in total, but the split of probability between x_i and $\mathbf{s}_i$ is determined by the gate gadgets.

Formally, we construct the payoff matrices A^{v_i,c_i} and $A^{c_i,v_{i+1}}$ for all $i < n$ by first setting each payoff to 0. Then, for each gate, we embed the corresponding gate gadget from Figure 5 into the matrices. For each gate $g_{a,j}$, we take the corresponding game from Figure 5, and embed it into the rows x_a and $\bar{x}_a$ of a constraint player's matrix. The diagrams specify specific actions of the constraint and variable players that should be modified.

For gates that originated in the circuit, the gadget is always embedded into the matrices $A^{v_{j-1},c_{j-1}}$ and A^{c_{j-1},v_j}, the synchronicity of the circuit ensures that the inputs for level j gates come from level $j-1$ gates. We have also added extra multiplication gates that

c_{j-1} \ v_{j-1}, v_j	$\bar{x}_b$	x_b	$\bar{x}_c$	x_c	$\bar{x}_a$	x_a
$\bar{x}_a$	0 / 0	1 / 0	0 / 0	1 / 0	0 / 0	0 / 1
x_a	0 / 0	0 / 0	0 / 0	0 / 0	0 / 1	1 / 0

$g_{a,j} = g_{b,j-1} +^b_{0.1} g_{c,j-1}$

$c_{j/j-1}$ \ $v_{j'}$, v_j	$\bar{x}_b$	x_b	$\bar{x}_c$	x_c
$\bar{x}_a$	0 / 0	c / 0	0 / 0	0 / 1
x_a	0 / 0	0 / 0	0 / 1	1 / 0

$g_{a,j} = g_{b,j'} *^b_{0.1} c$

c_{j-1} \ v_{j-1}, v_j	$\bar{x}_b$	x_b	$\bar{x}_c$	x_c	$\bar{x}_a$	x_a
$\bar{x}_a$	0 / 0	1 / 0	0 / 0	-1 / 0	0 / 0	0 / 1
x_a	0 / 0	0 / 0	0 / 0	0 / 0	0 / 1	1 / 0

$g_{a,j} = g_{b,j-1} -^b_{0.1} g_{c,j-1}$

c_j \ v_j	$\bar{x}_i$	x_i
$\bar{x}_i$	c / 0	c / 1
x_i	0 / 1	1 / 0

$g_{i,j} = c$

Figure 5 DGP polymatrix game gadgets.

copy values from the output of the circuit back to the input. These gates are of the form $g_{i,j} = g_{i',j+1}$, and are embedded into the matrices A^{v_j,c_j} and $A^{c_j,v_{j+1}}$.

The following lemma, proved in Appendix K, states that, in every Nash equilibrium, the strategies of the variable players exactly simulate the gates that have been embedded.

▶ **Lemma 13.** *In every mixed Nash equilibrium* $\mathbf{s}$ *of the game, the following are satisfied for each gate* $g_{i,j}$*.*

- *If* $g_{i,j} = c$, *then* $s_{v_j}(x_i) = c$.
- *If* $g_{i,j} = g_{i_1,j-1} +^b_{0.1} g_{i_2,j-1}$, *then* $s_{v_j}(x_i) = s_{v_{j-1}}(x_{i_1}) +^b_{0.1} s_{v_{j-1}}(x_{i_2})$.
- *If* $g_{i,j} = g_{i_1,j-1} -^b_{0.1} g_{i_2,j-1}$, *then* $s_{v_j}(x_i) = s_{v_{j-1}}(x_{i_1}) -^b_{0.1} s_{v_{j-1}}(x_{i_2})$.
- *If* $g_{i,j} = g_{i_1,j'} *^b_{0.1} c$, *then* $s_{v_j}(x_i) = s_{v_{j'}}(x_{i_1}) *^b_{0.1} c$.

Lemma 13 says that, in every Nash equilibrium of the game, the strategies of the variable players exactly simulate the gates, which by construction means that they give us a fixed point of the circuit F. Also note that it is straightforward to give a path decomposition for our interaction graph, where each node in the decomposition contains exactly two vertices from the game, meaning that the graph has pathwidth 1. So we have proved the following.

▶ **Theorem 14.** *It is* PPAD*-hard to find a Nash equilibrium of a tree polymatrix game, even when all players have at most twenty actions and the interaction graph has pathwidth 1.*

6 Open questions

For polymatrix games, the main open question is to find the exact boundary between tractability and hardness. Twenty-action pathwidth-1 tree polymatrix games are hard, but two-action path polymatrix games can be solved in polynomial time [9]. What about two-action tree polymatrix games, or path-polymatrix games with more than two actions?

For `2D-Brouwer` and `2D-LinearFIXP`, the natural question is: for which ϵ is it hard to find an ϵ-fixed point? We have shown that it is hard for $\epsilon = 0.2071$, while the case for $\epsilon = 0.5$ is trivial, since the point $(0.5, 0.5)$ must always be a 0.5-fixed point. Closing the gap between these two numbers would be desirable.

References

1. Siddharth Barman, Katrina Ligett, and Georgios Piliouras. Approximating Nash equilibria in tree polymatrix games. In *Proc. of SAGT*, pages 285–296, 2015.
2. Yang Cai and Constantinos Daskalakis. On minmax theorems for multiplayer games. In *Proc. of SODA*, pages 217–234, 2011.
3. Xi Chen and Xiaotie Deng. On the complexity of 2D discrete fixed point problem. *Theoretical Computer Science*, 410(44):4448–4456, 2009.
4. Xi Chen, Xiaotie Deng, and Shang-Hua Teng. Settling the complexity of computing two-player Nash equilibria. *Journal of the ACM*, 56(3):14:1–14:57, 2009.
5. Constantinos Daskalakis, Paul W. Goldberg, and Christos H. Papadimitriou. The complexity of computing a Nash equilibrium. *SIAM Journal on Computing*, 39(1):195–259, 2009.
6. Argyrios Deligkas, John Fearnley, Tobenna Peter Igwe, and Rahul Savani. An empirical study on computing equilibria in polymatrix games. In *Proc. of AAMAS*, pages 186–195, 2016.
7. Argyrios Deligkas, John Fearnley, and Rahul Savani. Computing constrained approximate equilibria in polymatrix games. In *Proc. of SAGT*, pages 93–105, 2017.
8. Argyrios Deligkas, John Fearnley, Rahul Savani, and Paul G. Spirakis. Computing approximate Nash equilibria in polymatrix games. *Algorithmica*, 77(2):487–514, 2017.
9. Edith Elkind, Leslie Ann Goldberg, and Paul W. Goldberg. Nash equilibria in graphical games on trees revisited. In *Proc. of EC*, pages 100–109, 2006.
10. Kousha Etessami and Mihalis Yannakakis. On the complexity of Nash equilibria and other fixed points. *SIAM Journal on Computing*, 39(6):2531–2597, 2010.
11. Michael L. Littman, Michael J. Kearns, and Satinder P. Singh. An efficient, exact algorithm for solving tree-structured graphical games. In *Proc. of NIPS*, pages 817–823. MIT Press, 2001.
12. Ruta Mehta. Constant rank two-player games are PPAD-hard. *SIAM J. Comput.*, 47(5):1858–1887, 2018.
13. Luis E. Ortiz and Mohammad Tanvir Irfan. Tractable algorithms for approximate Nash equilibria in generalized graphical games with tree structure. In *Proc. of AAAI*, pages 635–641, 2017.
14. Christos H. Papadimitriou. On the complexity of the parity argument and other inefficient proofs of existence. *J. Comput. Syst. Sci.*, 48(3):498–532, 1994.
15. Aviad Rubinstein. Settling the complexity of computing approximate two-player Nash equilibria. In *Proc. of FOCS*, pages 258–265, 2016.
16. Aviad Rubinstein. Inapproximability of Nash equilibrium. *SIAM J. Comput.*, 47(3):917–959, 2018.

A An issue with the lower bound in [9]

This section refers to the result in [9], which purports to show that finding a Nash equilibrium in a graphical game of pathwidth four is PPAD-hard. Like this paper, their proof reduces from discrete Brouwer, but unlike this paper and other work [4, 5, 12, 15], the proof attempts to carry out the reduction entirely using Boolean values. In other words, there is no step (like Lemmas 4 and 5 in this paper), where the Boolean outputs of the circuit are converted to arithmetic values. In all reductions of this type, this is carried out by averaging over multiple copies of the circuit, with the understanding that some of the circuits may give nonsensical outputs.

It is difficult to see how a reduction that avoids this step could work. This is because the expected payoff for a player in a polymatrix game is a continuous function of the other player's strategies. But attempting to reduce directly from a Boolean circuit would produce a function that is discontinuous.

It seems very likely that the proof in [9] can be repaired by including an explicit averaging step, and it this may still result in a graph that has bounded pathwidth, though it is less clear that the pathwidth would still be four. On the other hand, our work makes this less pressing, since the repaired result would still be subsumed by our lower bound for polymatrix games with pathwidth one.

B Proof of Lemma 1

Proof. The idea is to make each level of the circuit correspond to a line of the SLP. We assume that all `for` loops have been unrolled, and that all `if` statements have been resolved. Suppose that the resulting SLP has k lines, and furthermore assume that at each line of the SLP, we have an indexed list $v_1, v_2, \dots, v_l$ of the variables that are live on each line, where of course we have $l \leq w$.

We will build a circuit with $k \cdot w$ gates, and will index those gates as $g_{i,j}$, where $1 \leq i \leq k$ is a line, and $1 \leq j \leq w$ is a variable. The idea is that the gate $g_{i,j}$ will compute the value of the jth live variable on line i. The gate $g_{i,j}$ will be constructed as follows.

- If there are fewer than j variables live at line k of the SLP, then $g_{i,j}$ is a dummy c-gate.
- If line i of the SLP is $v_j \leftarrow \texttt{op}$, then we define $g_{i,j} = \texttt{op}$. If `op` uses a variable `x` as an input, then by definition, this variable must be live on line $i-1$, and so we find the index j' for `x` on line $i-1$, and we substitute $g_{i-1,j'}$ for `x` in `op`. We do this for both arguments in the case where `op` is $+^b$ or $-^b$.
- If line i of the SLP does not assign a value to v_j, then by definition, the variable must be live on line $i-1$. As before, let j' be the index of this variable on line $i-1$. We define $g_{i,j} = g_{i-1,j'} *^b 1$.

It is not difficult to see that this circuit exactly simulates the SLP. Moreover, by construction, we have $d(g_{i,j}) = i$. Hence, each level of the circuit has width exactly w, and so the overall width of the circuit is w.

◀

C Proof of Lemma 4

Proof. Suppose that we are given a `DiscreteBrouwer` instance defined by a circuit C over the grid $G_n = \{0, 1/2^n, 2/2^n, \dots, (2^n-1)/2^n\}^2$. Let n' be an integer such that $2^n/2^{n'} < (1-2\epsilon)$. We will build an ϵ-`ThickDisBrouwer` instance defined by a circuit C' over the grid $G_{n'} = \{0, 1/2^{n'}, 2/2^{n'}, \dots, (2^{n'}-1)/2^{n'}\}^2$. We will embed the original instance in the center of the new instance, where the point $(x_0, y_0) = (0.5 - 2^{n-1}/2^{n'}, 0.5 - 2^{n-1}/2^{n'})$ in G' will correspond to the point $(0, 0)$ in G. We use the following procedure to determine the color of a point $(x, y) \in G_{n'}$.

1. If $0 \leq x - x_0 \leq 2^n$ and $0 \leq y - y_0 \leq 2^n$, then $C'(x, y) = C(x - x_0, y - y_0)$.
2. Otherwise, if $x - x_0 < 0$, then $C(x, y) = 1$.
3. Otherwise, if $y - y_0 \leq 0$, then $C(x, y) = 2$.
4. Otherwise, $C(x, y) = 3$.

Observe that

$$x_0 = 0.5 - \frac{2^{n-1}}{2^{n'}} > 0.5 - \frac{(1 - 2\epsilon)}{2} = \epsilon,$$

where the second inequality used the definition of n'. Moreover

$$x_0 + 2^n = 0.5 + \frac{2^{n-1}}{2^{n'}} < 0.5 + \frac{(1 - 2\epsilon)}{2} = 1 - \epsilon,$$

where again the second inequality used the definition of n'. The same inequalities hold for y_0. Hence, the first step of our procedure perfectly embeds the original instance into the new instance, while the other steps ensure that the ϵ-`ThickDisBrouwer` boundary conditions hold.

Points in the boundary cannot be solutions, because the boundary constraints ensure that at least one of the three colors will be missing. Hence, every solution of C' on G' must also be a solution of C on G. ◀

D Proof of Lemma 5

Proof. Observe that SLP 3 produces a bad decode if and only if x is in the range $[0.5, 0.5 + 1/L)$. Since SLP 4 extracts n bits, multiplying x by two each time, it follows that one of the decodes will fail if

$$x \in I(a) = \left[\frac{a}{2^n}, \frac{a}{2^n} + \frac{1}{L}\right),$$

for some integer a.

Hence, the point $p_i = (p_i^1, p_i^2)$ has a poorly-positioned coordinate if there is some integer a such that $p_i^1 \in I(a)$, or $p_i^2 \in I(a)$. For a fixed dimension $j \in \{1, 2\}$, we have two properties.

- There cannot be two points p_i and $p_{i'}$ such that p_i^j and $p_{i'}^j$ both lie in the same interval $I(a)$. This is because the width of the interval is

$$\frac{1}{L} = \frac{1}{(k+2) \cdot 2^{n+1}} < \frac{1}{(k+1) \cdot 2^{n+1}},$$

  where the final term is the defined difference between p_i^j and p_{i+1}^j.
- There cannot be two distinct indices a and a' such that $p_i^j \in I(a)$ and $p_{i'}^j \in I(a')$. This is because the distance between p_1^j and p_k^j is at most

$$k \cdot \frac{1}{(k+1) \cdot 2^{n+1}} < \frac{1}{2^{n+1}},$$

  whereas the distance between any two consecutive intervals $I(a)$ and $I(a+1)$ is at least

$$\frac{a+1}{2^n} - \left(\frac{a}{2^n} + \frac{1}{(k+2) \cdot 2^{n+1}}\right) = \frac{1}{2^n} - \frac{1}{(k+2) \cdot 2^{n+1}} > \frac{1}{2^{n+1}}.$$

From these two facts, it follows that there is at most one point that has a poorly-positioned coordinate in dimension j, so there can be at most two points that have poorly positioned coordinates. ◀

E Proof of Lemma 6

Proof. We argue that if $\|x - F(x)\|_\infty < \epsilon'/2$, then there exist three indices i_1, i_2, and i_3 such that p_{i_j} has well-positioned coordinates, and that the lower-left corner of the square containing p_{i_j} has color j.

Suppose for the sake of contradiction that this is not true. Then there must be a color that is missing, and there are two cases to consider.

1. First suppose that color 1 is missing. Since there are at most two points with poorly-positioned coordinates, we know that we have at least $k-2$ points j for which $x_{2j} = 1$ or $x_{3j} = 1$. Hence we have

$$\mathtt{disp}_y \leq \left(\frac{(1-\sqrt{2})(k-2)}{k} + \frac{2}{k} \right) \cdot \epsilon,$$

   where the $2/k$ term comes from the fact that the poorly positioned points can maximize $\mathtt{disp}_y$ by fixing $x_{1j} = 1$ and $x_{2j} = x_{3j} = 0$, and thus can contribute at most $2 \cdot \epsilon/k$ to the sum.

   As k tends to infinity, the right-hand side converges to $(1-\sqrt{2}) \cdot \epsilon$. Since $\epsilon' < \epsilon$, we can choose a sufficiently large constant k such that $\mathtt{disp}_y < (1-\sqrt{2}) \cdot \epsilon'$. Now, observing that $1-\sqrt{2}$ is negative, we get the following

$$\|x - F(x)\|_\infty > \left| (1-\sqrt{2}) \cdot \epsilon' \right| = (\sqrt{2}-1) \cdot \epsilon',$$

   giving our contradiction.

2. Now suppose that one of colors 2 or 3 is missing. We will consider the case where color 3 is missing, as the other case is symmetric. As before, since there are at most two points with poorly-positioned coordinates, we know that we have at least $k-2$ points j for which $x_{1j} = 1$ or $x_{2j} = 1$. One of the two following cases applies.

   a. At least $(\sqrt{2}-1) \cdot k - 2$ well-positioned points satisfy $x_{2j} = 1$. If this is the case, then we have

$$\mathtt{disp}_x \geq \left(\frac{(\sqrt{2}-1) \cdot k - 2}{k} - \frac{2}{k} \right) \cdot \epsilon,$$

      where we have used the fact that there are no well positioned points with color 3, and the fact that the poorly-positioned points cannot reduce the sum by more than $\frac{2 \cdot \epsilon}{k}$. As k tends to infinity, the right-hand side tends to $(\sqrt{2}-1) \cdot \epsilon$, so there is a sufficiently large constant k such that $\mathtt{disp}_x > (\sqrt{2}-1) \cdot \epsilon'$, and so $\|x - F(X)\|_\infty > (\sqrt{2}-1) \cdot \epsilon'$.

   b. At least $k - (\sqrt{2}-1) \cdot k$ well-positioned points satisfy $x_{1j} = 1$. In this case we have

$$\begin{aligned}
\mathtt{disp}_y &\geq \sum_{j=1}^{k} \left(\frac{x_{1j} - (\sqrt{2}-1)x_{2j}}{k} - \frac{2}{k} \right) \cdot \epsilon \\
&\geq \left(\frac{\Big(k - (\sqrt{2}-1) \cdot k\Big) - \Big((\sqrt{2}-1)(\sqrt{2}-1) \cdot k\Big)}{k} - \frac{2}{k} \right) \cdot \epsilon \\
&= \left(\frac{(\sqrt{2}-1) \cdot k}{k} - \frac{2}{k} \right) \cdot \epsilon.
\end{aligned}$$

      The first line of this inequality uses the fact that we have no well-positioned points with color 3, and that the poorly-positioned points can reduce the sum by at most $\frac{2 \cdot \epsilon}{k}$.

The second line substitutes the bounds that we have for x_{1j} and x_{2j}. The third line uses the fact that $\sqrt{2}-1$ is a solution of the equation $x = 1 - x - x^2$.

As in the other two cases, this means that we can choose a sufficiently large constant k such that $\|x - F(X)\|_\infty > (\sqrt{2}-1) \cdot \epsilon'$.

Next we observe that the arguments given above all continue to hold if we substitute a sufficiently precise rational approximation $\sqrt{2}$ in our displacement vector calculation. This is because all three arguments prove that some expression converges to $(\sqrt{2}-1)\cdot\epsilon > (\sqrt{2}-1)\cdot\epsilon'$, thus we can replace $\sqrt{2}$ with any suitably close rational that ensures that the expressions converge to $(x-1)\cdot\epsilon > (\sqrt{2}-1)\cdot\epsilon'$ for some x.
So far we have shown that there exist three well-positioned points p_{i_1}, p_{i_2}, and p_{i_3} that have three distinct colors. To see that x is contained within a trichromatic square, it suffices to observe that $\|p_k - p_1\|_\infty \le 1/2^k$, which means that all three points must be contained in squares that are adjacent to the square containing x.

◀

F Proof of Lemma 8

We construct SLPs for both of the operations.

Packing bits. The `Pack` operation is implemented by the following SLP.

SLP 9 `Pack(x,y,S)` +2 variables

```
Clear(S, x)
y' ← y *^b 1
for i in {1,2,...,j} do
    b ← 0
    FirstBit(y', b)
    x ← b *^b 1/2^{s_i}
end
```

SLP 9 implements the pack operation. It begins by clearing the bits referenced by the sequence S. It then copies `y` to `y'`, and destructively extracts the first j bits of `y'`. These bits are then stored at the correct index in `x` by the final line of the `for` loop. In total, this SLP uses two additional variables `y'` and `b`. Two extra variables are used by `Clear`, but these stop being live after the first line, before `y'` and `b` become live.

Unpacking bits. The `Unpack` operation is implemented by the following SLP.

SLP 10 `Unpack(x,y,S)` +2 variables

```
x' ← x *^b 1
for i in {1,2,...,k} do
    b ← 0
    FirstBit(x', b)
    if i = s_j for some j then
        b ← b *^b 1/2^{s_j}
        y ← y +^b b
    end
end
```

SLP 10 implements the unpacking operation. It first copies `x` to `x'`, and then destructively extracts the first k bits of `x'`. Whenever a bit referred to by S is extracted from `x'`, it is first multiplied by $\frac{1}{2^{s_j}}$, which puts it at the correct position, and is then added to `y`. This SLP uses the two additional variables `x'` and `b`.

G Proof of Lemma 9

Simulating a Boolean circuit. Let $\langle g_{n+1}, g_{n+2}, \dots, g_{n+k}\rangle$ be the gates of the circuit, and suppose, without loss of generality, that the gates have been topologically ordered. The following SLP will simulate the circuit C.

SLP 11 $\texttt{Simulate}(\texttt{C}, \texttt{x})$ +3 variables

```
for i in {n+1, n+2, ..., n+k} do
    if g_i = g_j1 ∨ g_j2 then
        Or(x, i, j1, j2)
    end
    if g_i = ¬g_j then
        Not(x, i, j)
    end
end
```

Assuming that the first n bits of x already contain the packed inputs of the circuit, SLP 11 implements the operation $\texttt{Simulate}(\texttt{C}, \texttt{x})$ that computes the output of each gate. This simply iterates through and simulates each gate. The SLP introduce no new variables, and so it uses three additional live variables in total, which come from the `Or` and `Not` operations.

H Proof of Theorem 10

Dealing with the output. Recall that our Boolean circuit will output three bits, and that these bits determine which displacement vector is added to the output of the arithmetic circuit. We now build an SLP that does this conversion. It implements $\texttt{AddVector}(\texttt{x}, \texttt{i}, \texttt{out}_\texttt{x}, \texttt{out}_\texttt{y}, \texttt{k}, \texttt{d}_\texttt{x}, \texttt{d}_\texttt{y})$, where $x = \text{packed}(b_1, b_2, \dots, b_n)$, $i \le n$ is an index, $\texttt{out}_x$ and $\texttt{out}_y$ are variables, k is an integer, and $d_x, d_y \in [-1, 1]$. After this procedure, we should have $\texttt{out}_x = \texttt{out}_x + d_x \cdot b_i/k$, and $\texttt{out}_y = \texttt{out}_y + d_y \cdot b_i/k$. SLP 12 does this operation. It uses three extra variables in total: the fresh variable `a` is live throughout, and the two unpack operations use two extra variables.

SLP 12 $\texttt{AddVector}(\texttt{x}, \texttt{i}, \texttt{out}_\texttt{x}, \texttt{out}_\texttt{y}, \texttt{d}_\texttt{x}, \texttt{d}_\texttt{y}, \texttt{k})$ +3 variables

```
// Add d_x · b_i to out_x
a ← 0
Unpack(x, a, ⟨i⟩)
a ← |d_x|/k *^b a
out_x ← out_x +^b a     // Use -^b if d_x < 0

// Add d_y · b_i to out_y
a ← 0
Unpack(x, a, ⟨i⟩)
a ← |d_y|/k *^b a
out_y ← out_y +^b a     // Use -^b if d_y < 0
```

Implementing the reduction. Finally, we can implement the reduction from `DiscreteBrouwer` to `2D-Brouwer`. We will assume that we have been given a Boolean circuit C that takes $2n$ inputs, where the first n input bits correspond to the x coordinate, and the second n input bits correspond to the y coordinate. Recall that we have required that C gives its output as a one-hot vector. We assume that the three output bits of C are indexed $n+k-2$, $n+k-1$, and $n+k$, corresponding to colors 1, 2, and 3, respectively.

SLP 13 $\texttt{Reduction}(\texttt{in}_\texttt{x}, \texttt{in}_\texttt{y}, \texttt{out}_\texttt{x}, \texttt{out}_\texttt{y})$ +4 variables

```
out_x ← in_x
out_y ← in_y
for i in {1, 2, ..., k} do
    in_x ← in_x +^b 1/((k + 1) · 2^(n+1))
    in_y ← in_y +^b 1/((k + 1) · 2^(n+1))
    x ← 0
    Pack(x, in_x, ⟨1, 2, ..., n⟩)
    Pack(x, in_y, ⟨n + 1, n + 2, ..., 2n⟩)
    Simulate(C, x)
    AddVector(x, n+k-2, out_x, out_y, k,  0, 1)
    AddVector(x, n+k-1, out_x, out_y, k,  1,
     1-√2)
    AddVector(x, n+k  , out_x, out_y, k, -1,
     1-√2)
end
```

SLP 13 implements the reduction. The variables $\texttt{in}_x$ and $\texttt{in}_y$ hold the inputs to the circuit, while the variables $\texttt{out}_x$ and $\texttt{out}_y$ are the outputs. The SLP first copies the inputs to the outputs, and then modifies the outputs using the displacement vectors. Each iteration of the `for` loop computes the computes the displacement contributed by the point p_i (defined in the previous section). This involves decoding the first n bits of both $\texttt{in}_x$ and $\texttt{in}_y$, which can be done via the pack operation, simulating the circuit on the resulting bits, and then adding the correct displacement vectors to $\texttt{out}_x$ and $\texttt{out}_y$.

The correctness of this SLP follows from our correctness proof for Theorem 7, since all we have done in this section is reimplement while using a small number of live variables. In total, this SLP uses four extra variables. All of the macros use at most three extra variables, and the fresh variable x during these macros. Since $\texttt{in}_x$ $\texttt{in}_y$, $\texttt{out}_x$ and $\texttt{out}_y$ are all live throughout as well, this gives us 8 live variables in total.

I Proof of Lemma 11

Proof. The circuit F' consists of gates $g'_{i,j}$ for each $1 \leq i \leq 8$ and $1 \leq j \leq n$.

- If $g_{i,j} = c$, then $g'_{i,j} = c/10$.
- If $g_{i,j} = g_{a,b} +^b g_{x,y}$, then $g'_{i,j} = g'_{a,b} \; +^b_{0.1} \; g'_{x,y}$.
- If $g_{i,j} = g_{a,b} -^b g_{x,y}$, then $g'_{i,j} = g'_{a,b} \; -^b_{0.1} \; g'_{x,y}$.
- If $g_{i,j} = g_{a,b} *^b c$, then $g'_{i,j} = g'_{a,b} \; *^b_{0.1} \; c$.

Let $(x, y) \in [0, 1]^2$. It is not difficult to show by induction, that if we compute $F(x, y)$ and $F'(x/10, y/10)$, then $g'_{i,j} = g_{i,j}/10$ for all i and j. Hence, $F(x, y) = (x, y)$ if and only if $F'(x/10, y/10) = (x/10, y/10)$. ◀

J Proof of Lemma 12

Proof. For the sake of contradiction, suppose that there is a Nash equilibrium $\mathbf{s}$ in which there is some variable or constraint player j that fails to satisfy this equality. Let I be the subset of indices that maximize the expression $s_j(x_i) + s_j(\bar{x}_i)$, ie., I contains the pairs that player j plays with highest probability. Note that since player j does not play all pairs uniformly, I does not contain every index, so let J be the non-empty set of indices not in I.

Let m_k be the mix player who plays against player j. By construction, the actions x_i and $\bar{x}_i$ have payoff $(s_j(x_i) + s_j(\bar{x}_i)) \cdot M$ for m_k. Since $\mathbf{s}$ is a Nash equilibrium, m_k may only place probability on actions that are best responses, which means that he may only place probability on the actions x_i and $\bar{x}_i$ when $i \in I$.

Let i be an index that maximizes $s_{m_k}(x_i)+s_{m_k}(\bar{x}_i)$ for player m_k. By the above argument, we have $i \in I$. The actions x_i and $\bar{x}_i$ for player j give payoff at most

$$\begin{aligned} 2P - M \cdot (s_{m_k}(x_i) + s_{m_k}(\bar{x}_i)) &\leq 2P - M/10 \\ &< -2P. \end{aligned}$$

The first expression uses $2P$ as the maximum possible payoff that player j can obtain from the two other games in which he is involved. The first inequality uses the fact that i was the pair with maximal probability, and there are exactly 10 pairs. The second inequality uses the fact that $M/10 > 4P$.

On the other hand, let i' be an index in J. By the argument above, we have $s_{m_k}(x_{i'}) + s_{m_k}(\bar{x}_{i'}) = 0$. Hence, the payoff of actions $x_{i'}$ and $\bar{x}_{i'}$ to player j is at least $-2P$, since that is the lowest payoff that he can obtain from the other two games in which he is involved.

But now we have arrived at our contradiction. Player j places non-zero probability on at least one action x_i or $\bar{x}_i$ with $i \in I$ that is not a pure best response. Hence $\mathbf{s}$ cannot be a Nash equilibrium. ◀

K Proof of Lemma 13

Proof. We can actually prove this lemma for all four gates simultaneously. Let j' be the index constraint player into which the gate gadget is embedded. Observe that all four games for the four gate types have a similar structure: The payoffs for actions x_i and $\bar{x}_i$ for player

v_j are identical across all four games, and the payoff of action x_i for $c_{j'}$ are also identical; the only thing that differs between the gates is the payoff to player $c_{j'}$ for action $\bar{x}_i$. We describe these differences using a function f.

- For c-gates, we define $f(\mathbf{s}) = c$.
- For $+^b_{0.1}$-gates, we define $f(\mathbf{s}) = s_{v_{j-1}}(x_{i_1}) \ + \ s_{v_{j-1}}(x_{i_1})$.
- For $-^b_{0.1}$-gates, we define $f(\mathbf{s}) = s_{v_{j-1}}(x_{i_1}) \ - \ s_{v_{j-1}}(x_{i_1})$.
- For $*^b_{0.1}$-gates, we define $f(\mathbf{s}) = s_{v_{j'}}(x_{i_1}) \ * \ c$.

Observe that the payoff of action $\bar{x}_i$ to player $c_{j'}$ is $f(\mathbf{s})$. To prove the lemma, we must show that player v_j plays x_i with probability

$$\min(\max(f(\mathbf{s}), 0.1), 0).$$

There are three cases to consider.

- If $f(\mathbf{s}) \leq 0$, then we argue that $s_{v_j}(x_i) = 0$. Suppose for the sake of contradiction that player v_j places non-zero probability on action x_i. Then action x_i for player $c_{j'}$ will have payoff strictly greater than zero, whereas action $\bar{x}_i$ will have payoff $f(\mathbf{s}) \leq 0$. Hence, in equilibrium, $c_{j'}$ cannot play action $\bar{x}_i$. Lemma 12 then implies that player $c_{j'}$ must play x_i with probability 0.1. If $c_{j'}$ does this, then the payoff to v_j for x_i will be zero, and the payoff to v_j for $\bar{x}_i$ will be 0.1. This means that v_j places non-zero probability on an action that is not a best response, and so is a contradiction.
- If $f(\mathbf{s}) \geq 0.1$, then we argue that $s_{v_j}(x_i) = 0.1$. Suppose for the sake of contradiction with Lemma 12 that $s_{v_j}(\bar{x}_i) > 0$. Observe that the payoff to player $c_{j'}$ of action $\bar{x}_i$ is $f(\mathbf{s}) \geq 0.1$, whereas the payoff to player $c_{j'}$ of action x_i is $s_{v_j}(x_i) < 0.1$. So to be in equilibrium and consistent with Lemma 12, player $c_{j'}$ must place 0.1 probability on action $\bar{x}_i$, and 0 probability on action x_i. But this means that the payoff of action $\bar{x}_i$ to player v_j is zero, while the payoff of action x_i to player v_j is 0.1. Hence player v_j has placed non-zero probability on an action that is not a pure best response, and so we have our contradiction.
- If $0 < f(\mathbf{s}) < 0.1$, then we argue that $s_{v_j}(x_i) = f(\mathbf{s})$. We first prove that player $c_{j'}$ must play both x_i and $\bar{x}_i$ with positive probability.
  - If player $c_{j'}$ does not play $\bar{x}_i$ then player v_j will not play x_i, and player $c_{j'}$ will receive payoff 0, but in this scenario he could get $f(\mathbf{s}) > 0$ by playing $\bar{x}_i$ instead of his current strategy.
  - If player $c_{j'}$ does not play x_i then player v_j will not play $\bar{x}_i$. Player $c_{j'}$ will receive payoff $f(\mathbf{s})$ for playing $\bar{x}_i$, but in this scenario he could receive payoff $1 > f(\mathbf{s})$ for playing x_i instead.

  In order for player $c_{j'}$ to mix over x_i and $\bar{x}_i$ in equilibrium, their payoffs must be equal. This is only the case when $s_{v_j}(x_i) = f(\mathbf{s})$.

◀