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; Theory of computation → Exact and approximate computation of equilibria

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

Digital Object Identifier 10.4230/LIPIcs.ICALP.2020.38

Category Track A: Algorithms, Complexity and Games

Related Version A full version of this paper, with all proofs included, is available [8], https://arxiv.org/abs/2002.12119.

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 [16], 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* [12], 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 [10], 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 appear to be some issues with their approach. Later work of Barman, Ligett, and Piliouras [1] provided a QPTAS for constant-action tree polymatrix games, and then Ortiz and Irfan [14] gave an FPTAS for this case. All three papers, [1, 10, 14], 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 [10], 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 [13]. 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, 16], 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 [15]. 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 [17] 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 [11] defined the classes FIXP and LinearFIXP and they proved that LinearFIXP = PPAD. Mehta [13] 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 [14] and Deligkas, Fearnley, and Savani [7] produced QPTASs for polymatrix games of bounded treewidth (in addition to the FPTAS of [14] 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 [9]. 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 $\mathbf{s} = (s_1, s_2, \ldots, s_n)$ 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) [11].

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 $+^b$ $\texttt{in}_1$.

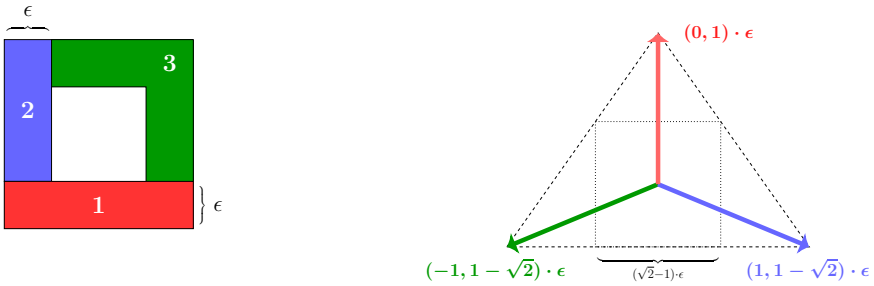
For the sake of brevity, we also allow `if` statements and `for` loops in our SLPs. These two pieces of syntax can be thought of as macros that help us specify a straight line program concisely. The arguments to an `if` statement or a `for` loop must be constants that do not depend on the value of any 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 $+^b$ $\texttt{in}_1$.

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.

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



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

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

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 [13], 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 $G = \{0, 1/2^n, 2/2^n, \ldots, (2^n - 1)/2^n\}^2$. 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 $y' \in \{y, y+1\}$.*

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.

▶ **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, $b_1, b_2, \ldots, b_n$).

```
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 here.

▶ **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

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

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

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

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 [10], 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.

▶ **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 `Or(x, i1, i2, i3)`, 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 `Not(x, i1, i2)`, 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 $\texttt{Or}(\texttt{x}, \texttt{i}_1, \texttt{i}_2, \texttt{i}_3)$ +3 variables.

```
a ← 0
Unpack(x, a, ⟨i_1⟩)
Unpack(x, a, ⟨i_2⟩)
Pack(x, a, ⟨i_3⟩)
```

SLP 8 $\texttt{Not}(\texttt{x}, \texttt{i}_1, \texttt{i}_2)$ +3 variables.

```
a ← 0
Unpack(x, a, ⟨i_1⟩)
b ← 1
a ← b -^b a
Pack(x, a, ⟨i_2⟩)
```

▶ **Lemma 9.** *Let C be a Boolean circuit with n inputs and k gates. Suppose that $x = \text{packed}(b_1, \ldots, b_n)$, 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 $x = \text{packed}(b_1, \ldots, b_n, b_{n+1}, \ldots, b_{n+k})$, 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.

▶ **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 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 $g_{8,1} = g_{10,2}$.

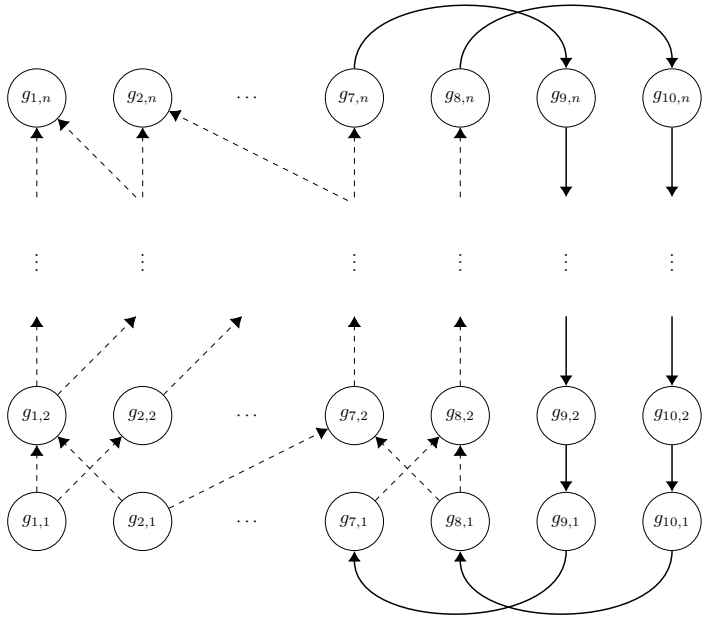


Figure 2 Extra equalities to introduce feedback of $g_{7,n}$ and $g_{8,n}$ to $g_{7,1}$ and $g_{8,1}$ respectively.

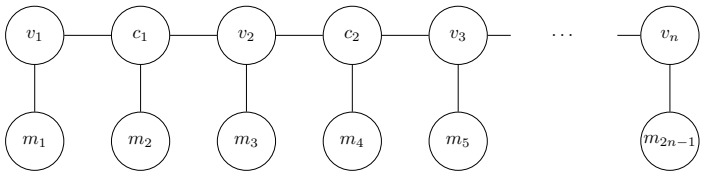


Figure 3 The structure of the polymatrix game.

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 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 $i = 1, \ldots, n$, we have a *variable* player v_i.
- For each $i = 1, \ldots, n-1$, we have a *constraint* player c_i, who is connected to v_i and v_{i+1}.
- For each $i = 1, \ldots, 2n-1$, 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 $i = 1, \ldots, 10$.

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

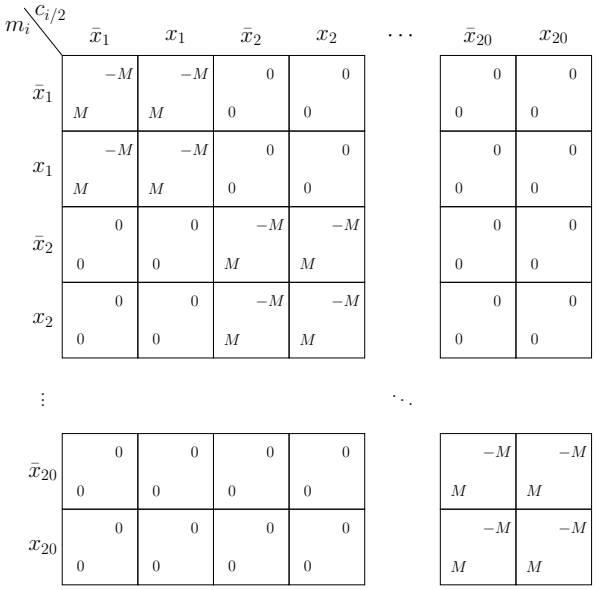


Figure 4 The hide and seek game that forces $c_{j/2}$ to play an appropriate mixed strategy. The same game is used to force $v_{(j-1)/2}$ mixes appropriately.

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 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 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.

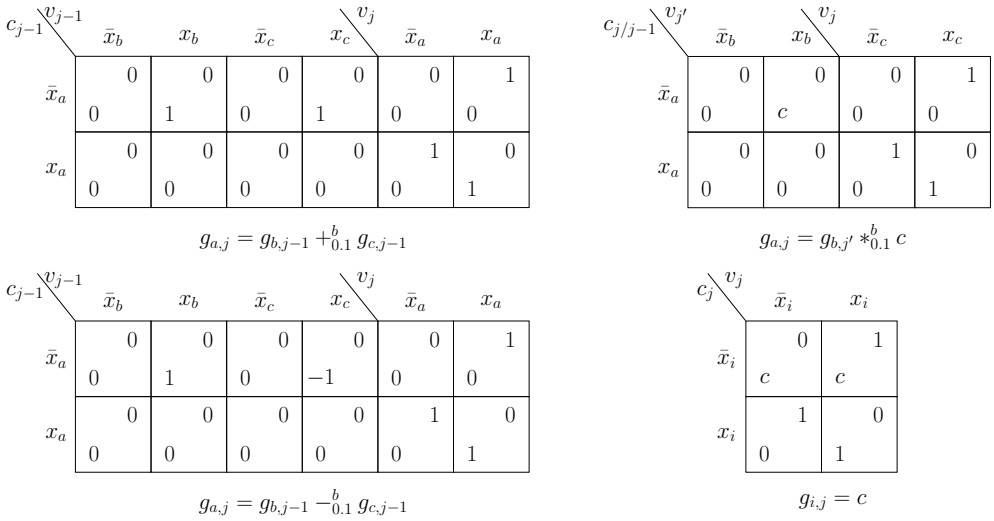


Figure 5 DGP polymatrix game gadgets.

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 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 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 [10]. 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. doi:10.1137/1.9781611973082.20.

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, and Rahul Savani. Tree polymatrix games are PPAD-hard, 2020. arXiv:2002.12119.

9 Argyrios Deligkas, John Fearnley, Rahul Savani, and Paul G. Spirakis. Computing approximate Nash equilibria in polymatrix games. *Algorithmica*, 77(2):487–514, 2017.

10 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.

11 Kousha Etessami and Mihalis Yannakakis. On the complexity of Nash equilibria and other fixed points. *SIAM Journal on Computing*, 39(6):2531–2597, 2010.

12 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.

13 Ruta Mehta. Constant rank two-player games are PPAD-hard. *SIAM J. Comput.*, 47(5):1858–1887, 2018.

14 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.

15 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.

16 Aviad Rubinstein. Settling the complexity of computing approximate two-player Nash equilibria. In *Proc. of FOCS*, pages 258–265, 2016.

17 Aviad Rubinstein. Inapproximability of Nash equilibrium. *SIAM J. Comput.*, 47(3):917–959, 2018.