A Parity Game Tale of Two Counters

Tom van Dijk
Formal Methods and Tools
University of Twente, Enschede
t.vandijk@utwente.nl

Parity games are simple infinite games played on finite graphs with a winning condition that is expressive enough to capture nested least and greatest fixpoints. Through their tight relationship to the modal mu-calculus, they are used in practice for the model-checking and synthesis problems of the mu-calculus and related temporal logics like LTL and CTL. Solving parity games is a compelling complexity theoretic problem, as the problem lies in the intersection of UP and co-UP and is believed to admit a polynomial-time solution, motivating researchers to either find such a solution or to find superpolynomial lower bounds for existing algorithms to improve the understanding of parity games.

We present a parameterized parity game called the Two Counters game, which provides an exponential lower bound for a wide range of attractor-based parity game solving algorithms. We are the first to provide an exponential lower bound to priority promotion with the delayed promotion policy, and the first to provide such a lower bound to tangle learning.

1 Introduction

Parity games are turn-based games played on a finite graph. Two players *Odd* and *Even* play an infinite game by moving a token along the edges of the graph. Each vertex is labeled with a natural number *priority* and the winner of the game is determined by the parity of the highest priority that is encountered infinitely often. Player Odd wins if this parity is odd; otherwise, player Even wins.

Parity games play a central role in several domains in theoretical computer science. Their study has been motivated by their relation to various problems in formal verification and synthesis that can be reduced to solving parity games, as parity games capture the expressive power of nested least and greatest fixpoint operators [15]. Deciding the winner of a parity game is polynomial-time equivalent to checking non-emptiness of non-deterministic parity tree automata [32], and to the explicit model-checking problem of the modal μ-calculus [13, 27, 31]. Synthesis problems ask for an implementation of a system that satisfies the desired properties and reactive synthesis tools like Strix use parity games to synthesize controllers satisfying LTL formulas [34]. Solving the related parity game either results in a correct implementation or in a counterexample demonstrating how an adversary can make the property fail.

Parity games are also interesting for complexity theory, as the problem of determining the winner of a parity game is known to lie in UP $\cap$ co-UP [28], which is contained in NP $\cap$ co-NP [13]. This problem is therefore unlikely to be NP-complete and it is widely believed that a polynomial solution exists.

Recent work proposes novel algorithms based on the notion of a tangle [9]. A tangle is a strongly connected subgame of a parity game for which one player has a strategy to win all cycles in the subgame. Tangles play a fundamental role in various parity game algorithms, but most algorithms are not explicitly aware of tangles and can explore the same tangles repeatedly [9]. The algorithms proposed in [9] solve parity games by explicitly computing tangles using attractor computation.

Tangles are related to snares [14] and quasi-dominions [4], with the critical difference that tangles are strongly connected whereas snares and quasi-dominions may be composed of multiple tangles. Thus it is an obvious question whether the subexponential counterexample of Friedmann [23] to Fearnley's snare-based algorithm [14] can be adapted for tangle learning. We show that this is possible and that the resulting counterexample is powerful enough to be a difficult lower bound to a wide range of algorithms.

We propose a parameterized parity game based on binary counters. The goal is to trick the algorithms to explore the progression of the counters, only solving the game when all bits are set. The critical ingredient to make these games difficult for attractor-based algorithms is to use two intertwined binary counters, one for each player, that progress together. We show empirically that these games are difficult for a wide range of algorithms, in particular for those based on attractor computation, such as priority promotion [4, 6] and its variations [2, 3], tangle learning [9], and the recursive algorithm by Zielonka [33, 37]. For these, we provide the exponential lower bound of $\Omega(2^{\sqrt{n}})$. We are the first to provide an exponential lower bound for the delayed promotion policy of priority promotion and for tangle learning.

2 Preliminaries

A parity game $\eth$ is a tuple $(V_\Diamond, V_\Box, E, \mathsf{pr})$ where $V = V_\Diamond \uplus V_\Box$ is a set of n vertices partitioned into the sets $V_\Diamond$ controlled by player *Even* and $V_\Box$ controlled by player *Odd*, and $E \subseteq V \times V$ is a left-total binary relation describing all moves, i.e., every vertex has at least one successor. We also write $E(u)$ for all successors of u and $u \to v$ for $v \in E(u)$. The function $\mathsf{pr}\colon V \to \{0, 1, \ldots, d\}$ assigns to each vertex a *priority*, where d is the highest priority in the game. We write $\alpha \in \{\Diamond, \Box\}$ to denote a player $\Diamond$ or $\Box$ and $\overline{\alpha}$ for the opponent of α. In visual representations of parity games, we use diamonds and boxes for vertices of the two players.

We write $\mathsf{pr}(V)$ for $\max\{\mathsf{pr}(v) \mid v \in V\}$ and similarly $\mathsf{pr}(\eth)$ for the highest priority in $\eth$. Furthermore, we write $\mathsf{pr}^{-1}(i)$ for all vertices with the priority i. A *play* $\pi = v_0 v_1 \ldots$ is an infinite sequence of vertices consistent with E, i.e., $v_i \to v_{i+1}$ for all $i \geq 0$. We denote with $\inf(\pi)$ the vertices in π that occur infinitely many times in π. Player Even wins a play π if $\mathsf{pr}(\inf(\pi))$ is even; player Odd wins if $\mathsf{pr}(\inf(\pi))$ is odd. We write $\mathrm{Plays}(v)$ to denote all plays starting at vertex v and $\mathrm{Plays}(V)$ for $\{\pi \in \mathrm{Plays}(v) \mid v \in V\}$.

A (positional) *strategy* $\sigma\colon V \nrightarrow V$ assigns to each vertex in its domain a single successor in E, i.e., $\sigma \subseteq E$. We refer to a strategy of player α to restrict the domain of σ to V_α. In the remainder, all strategies σ are of a player α. Given a strategy σ, we define the subgame induced by σ as $\eth[\sigma] := (V_\Diamond, V_\Box, E', \mathsf{pr})$ where $E' := \{uv \in E \mid u \notin \mathrm{dom}(\sigma) \vee \sigma(u) = v\}$. We write $\mathrm{Plays}(v, \sigma)$ for all plays from v consistent with σ, i.e., $\mathrm{Plays}_{\eth[\sigma]}(v)$, and $\mathrm{Plays}(V, \sigma)$ for $\{\pi \in \mathrm{Plays}(v, \sigma) \mid v \in V\}$.

A basic result for parity games is that they are memoryless determined [12], i.e., each vertex is either winning for player Even or for player Odd, and the players have a positional strategy for their winning vertices. Player α wins vertex v if there is a strategy σ such that player α wins all plays in $\mathrm{Plays}(v, \sigma)$.

A set of vertices U is called *closed* with respect to player α if all vertices that belong to player α have a successor in U and all vertices of player $\overline{\alpha}$ only have successors inside U. That is, player $\overline{\alpha}$ cannot leave U. If a set of vertices is closed in the full game $\eth$ then we also call the set *globally closed* and if it is closed in some subgame of $\eth$ then we call the set *locally closed* in that subgame.

A *dominion* D is a (globally) closed set of vertices for which player α has a strategy σ such that player α wins all plays in $\mathrm{Plays}(D, \sigma)$. That is, player α wins all cycles in the subgame $\eth[D, \sigma]$ induced by D and σ. We also write a *p-dominion* for a dominion where p is the highest priority encountered infinitely often in plays consistent with σ, i.e., $p := \max\{\mathsf{pr}(\inf(\pi)) \mid \pi \in \mathrm{Plays}(D, \sigma)\}$.

Several algorithms for solving parity games employ *attractor computation*. Given a set of vertices A, the attractor of A for a player α contains exactly those vertices from which player α can ensure arrival in A. We write $\mathit{Attr}^{\eth}_{\alpha}(A)$ to attract vertices in $\eth$ to A as player α, i.e., the least fixpoint of

$$Z := A \cup \{v \in V_\alpha \mid E(v) \cap Z \neq \emptyset\} \cup \{v \in V_{\overline{\alpha}} \mid E(v) \subseteq Z\}$$

```
def zielonka($\Game$):
    if $\Game = \emptyset$ : return $\emptyset, \emptyset$                          solve the empty game
    $p \leftarrow \mathsf{pr}(\Game)$, $\alpha \leftarrow \mathsf{pr}(\Game) \bmod 2$
    $A \leftarrow \mathrm{Attr}^{\Game}_{\alpha}(\mathsf{pr}^{-1}(p))$              attract to highest priority
    $W_{\Diamond}, W_{\Box} \leftarrow$ zielonka($\Game \setminus A$)               compute remaining subgame
    $B \leftarrow \mathrm{Attr}^{\Game}_{\overline{\alpha}}(W_{\overline{\alpha}})$  attract to opponent region
    if $B = W_{\overline{\alpha}}$ :
        $W_{\alpha} \leftarrow W_{\alpha} \cup A$                                   $A$ is won by $\alpha$
    else:
        $W_{\Diamond}, W_{\Box} \leftarrow$ zielonka($\Game \setminus B$)           recompute remaining subgame
        $W_{\overline{\alpha}} \leftarrow W_{\overline{\alpha}} \cup B$             $B$ is won by $\overline{\alpha}$
    return $W_{\Diamond}, W_{\Box}$
```

Algorithm 1: Zielonka's recursive algorithm.

We compute the α-attractor of A with a backward search from A, initially setting $Z := A$ and iteratively adding α-vertices with a successor in Z and $\overline{\alpha}$-vertices with no successors outside Z. We call a set of vertices A α-maximal if $A = Attr^{\Game}_{\alpha}(A)$. The attractor also yields an "attractor strategy" by selecting a vertex in Z for every added α-vertex v when v is added to Z, and by selecting a vertex in Z for all α-vertices in A that do not yet have a strategy but can play to Z.

Attractors are often used to attract to a set $A := \mathsf{pr}^{-1}(\mathsf{pr}(\Game))$ for the player that wins the highest priority in $\Game$. By repeatedly computing this attractor and removing it from the game, the game is decomposed into *regions* each associated with the priority p and player $\alpha = p \bmod 2$. Each such region has the property that all plays that stay in the region are won by player α, and that player $\overline{\alpha}$ can leave the region to higher α-regions and, only via a vertex of priority p, to lower regions. The recursive algorithm [37] investigates whether (lower) α-regions attract vertices from higher $\overline{\alpha}$-regions. Priority promotion [4] merges locally closed regions with higher α-regions that the opponent can escape to. Tangle learning [9] computes tangles from locally closed regions which are used to improve the attractor-based decomposition.

3 Recursive Algorithm

The recursive algorithm by Zielonka [37] is a well-known algorithm to solve parity games with fast practical performance [10, 25]. Although it requires exponential time in the worst-case, tight bounds are known for various classes of games [26]. Very recently, Parys has proposed a version of the recursive algorithm that runs in quasi-polynomial time [35].

The recursive algorithm is based on attractor computation. See Algorithm 1. We omit computing the winning strategy. At each step, given the current subgame $\Game$, the algorithm computes the attractor A to the highest vertices in $\Game$ and recursively solves the subgame $\Game \setminus A$. If the opponent $\overline{\alpha}$ can attract vertices in A to $W_{\overline{\alpha}}$, then $\overline{\alpha}$ wins $B := \mathrm{Attr}^{\Game}_{\overline{\alpha}}(W_{\overline{\alpha}})$. The vertices attracted to $W_{\overline{\alpha}}$ are vertices of priority p that are now estimated to be won by player $\overline{\alpha}$ plus vertices of priority $< p$ that player $\overline{\alpha}$ attracts to $W_{\overline{\alpha}}$ via vertices of priority p. The algorithm then computes the remaining subgame $\Game \setminus B$ recursively and returns the winning regions. If player $\overline{\alpha}$ does not attract any vertices to $W_{\overline{\alpha}}$, then no second recursion is needed. The winning strategies are trivially obtained as the attractor strategy plus any successor in the set $W_{\alpha} \cup A$ for α-vertices with priority p. One can also view the recursive algorithm as decomposing the game into α-maximal regions and refining these regions starting with the lowest region.

```
def search-dominion($\Game$):
    $\mathsf{r} \leftarrow V \mapsto \bot$                                          all vertices to $\bot$
    $p \leftarrow \mathsf{pr}(\Game)$                                        start at highest priority
    while True :
        $\alpha \leftarrow p \bmod 2$                                                  current player
        $\text{Subgame} \leftarrow V \setminus \{v \mid \mathsf{r}(v) > p\}$                          current subgame
        $A \leftarrow \mathsf{r}^{-1}(p) \cup (\mathsf{pr}^{-1}(p) \cap \text{Subgame})$        current region/attractor target
        $Z \leftarrow \text{Attr}_{\alpha}^{\Game \cap \text{Subgame}}(A)$                                   attract to $A$
        $\text{Open} \leftarrow \{v \in Z \cap V_{\alpha} \mid E(v) \cap Z = \emptyset\}$              open $\alpha$-vertices
        $\text{Esc} \leftarrow E(Z \cap V_{\overline{\alpha}}) \setminus Z$                            escape targets for $\overline{\alpha}$
        if $\text{Open} \neq \emptyset \vee (\text{Esc} \cap \text{Subgame}) \neq \emptyset$ :           is the region open?
            $\mathsf{r} \leftarrow \mathsf{r}[Z \mapsto p]$                                     record the region
            $p \leftarrow \mathsf{pr}(\text{Subgame} \setminus Z)$             continue with next highest priority
        elif $\text{Esc} \neq \emptyset$ :                                               locally closed?
            $p \leftarrow \min\{\mathsf{r}(v) \mid v \in \text{Esc}\}$                     set $p$ to lowest escape
            $\mathsf{r} \leftarrow \mathsf{r}[Z \mapsto p][\{v \mid \mathsf{r}(v) < p\} \mapsto \bot]$           merge, reset lower than $p$
        else:                                                         globally closed?
            $Z \leftarrow Attr_{\alpha}^{\Game}(Z)$                                  maximize the dominion
            return $\alpha, Z$                                      dominion of player $\alpha$!
def prioprom($\Game$):
    $W_{\Diamond} \leftarrow \emptyset, W_{\Box} \leftarrow \emptyset$                                         initialize sets
    while $\Game \neq \emptyset$ :
        $\alpha, D \leftarrow$ search-dominion($\Game$)                          compute the next dominion
        $W_{\alpha} \leftarrow W_{\alpha} \cup D$                  add dominion to winning region of player $\alpha$
        $\Game \leftarrow \Game \setminus D$                                          remove dominion from game
    return $W_{\Diamond}, W_{\Box}$
```

Algorithm 2: The priority promotion algorithm.

4 Priority Promotion

Priority promotion was proposed in [4, 6] and improved in [2, 3]. Like the recursive algorithm, priority promotion computes the top-down α-maximal decomposition of the game into regions. These regions have the property that all plays that stay in the region are won by player α. Regions are *locally closed* when all vertices of player α have a successor in the region and no vertices of player $\overline{\alpha}$ can play to lower regions. Because all regions are α-maximal, player $\overline{\alpha}$ can only escape to "higher" regions of player α. Locally closed regions are merged with the lowest higher region to which player $\overline{\alpha}$ can escape, after which the decomposition of the game is refined by attracting to this merged region and recomputing the lower regions. This is called promoting, as the lower region is "promoted" to the higher region.

See Algorithm 2. Again we omit explicitly computing the winning strategies, but they are computed trivially by the attractors. The `search-dominion` algorithm is given a game and returns a dominion and the winner of this dominion. By repeatedly calling `search-dominion` and removing the dominion from $\Game$, priority promotion solves parity games. The `search-dominion` algorithm refines the regions until a dominion is found. The function r records the current region (priority p) of each vertex, or $\bot$ if the vertex is not in a region. After promoting a region to a higher priority p (lines 15–16) the algorithm resets all regions below p and then attracts to the merged region at lines 5–8.

Consider some region of priority 4 that is promoted to priority 16. By attracting to the combined region 16, priority promotion attracts vertices from regions $5 \ldots 15$ of player $\overline{\alpha}$ that are attracted to the lower region 4 in one step, while the recursive algorithm attracts from each affected region of $\overline{\alpha}$ separately. Another difference with the recursive algorithm is that priority promotion always promotes the highest locally closed region whereas the recursive algorithm starts refining with the lowest regions.

The original algorithm [4] resets all lower regions after a promotion. These lower regions might be the result of earlier promotions and these promotions are then often repeated. The PP+ extension [2] only resets lower regions of player $\overline{\alpha}$, since promoting regions of player α can only "break" regions of player $\overline{\alpha}$. The region restoration (RR) extension [3] further improves upon PP+ by only resetting a lower region if the earlier attractor strategy of the player for the vertices that are still in the original region now leaves this region to a higher region (of the opponent). The delayed promotion (DP) policy (also [2]) uses a heuristic to delay promotions. The algorithm records which regions are the result of a promotion and if another promotion might affect a merged region, then this promotion is delayed. The delayed promotions are instead performed on a copy of the decomposition. When no more normal promotions can be performed, the delayed promotions of one player are applied from this copy, where this player is the one that wins the highest merged region in the copy. The other delayed promotions are discarded and the algorithm continues with an empty record of merged regions.

5 Tangles and Tangle Learning

Earlier work introduced tangles as substructures of parity games [9]. Tangles are strongly connected subgames of a parity game for which one player has a strategy to win all cycles in the subgame. The losing player must therefore escape the tangle, so we extend attractor computation to simultaneously attract all vertices in a tangle when the losing player must escape to the attracting set. This leads to the *tangle learning* algorithm, which computes new tangles along the decomposition of the game, computed using the extended attractor. The algorithm solves parity games by finding tangles that are dominions.

Definition 1. *A* tangle *is a pair $T = (U, \sigma)$ where U is a nonempty set of vertices $U \subseteq V$ and $\sigma\colon U_\alpha \to U$ is a strategy for all vertices in U of player $\alpha := \mathsf{pr}(U) \bmod 2$, such that the subgame $\partial[U, \sigma]$ induced by the tangle is strongly connected and player α wins all cycles in $\partial[U, \sigma]$.*

We say that a tangle with the highest priority p is a p-tangle, and that it is an α-tangle. We often use T for just the set of vertices in the tangle, e.g., writing $v \in T$ if a vertex v is in the tangle. We write $E_T(T)$ for the successors outside T from $\overline{\alpha}$-vertices in T, $E_T(T) := \{\, v \in V \setminus T \mid u \to v \mid u \in T \cap V_{\overline{\alpha}} \,\}$.

We have several basic observations related to tangles [9]. A *closed* p-tangle, from which player $\overline{\alpha}$ cannot leave, is a p-dominion. Vice versa, every p-dominion contains at least one p-tangle. Furthermore, tangles are often composed of subtangles with lower priorities. We thus find a hierarchy of tangles in any dominion D with winning strategy σ by computing the set of winning priorities $\{\, \mathsf{pr}(\inf(\pi)) \mid \pi \in \mathrm{Plays}(D, \sigma) \,\}$. There is a p-tangle in D for every p in this set. Tangles are thus intuitive substructures of dominions. One can find all tangles in a dominion D by computing $\{\, \inf(\pi) \mid \pi \in \mathrm{Plays}(D, \sigma) \,\}$. See for example Figure 1. Player Odd wins with strategy $\{\, \mathbf{d} \to \mathbf{e} \,\}$. Player Even can avoid priority 5, but then loses with priority 3. The 5-dominion contains a 5-tangle and a 3-tangle.

As described in [9], tangles play a central role for various parity game solving algorithms, as they implicitly explore tangles and may explore the same tangles repeatedly, especially when tangles are nested. This motivates algorithms that explicitly target tangles, such as the "snare memoization" extension to strategy improvement [14] and the "tangle attractor" approach of tangle learning [9].

```
1  def search($\Game$, $\mathbf{T}$):
2      if $\Game = \emptyset$ : return $\emptyset$                                  no tangles in an empty game
3      $p \leftarrow \mathsf{pr}(\Game)$, $\alpha \leftarrow \mathsf{pr}(\Game) \bmod 2$              obtain current priority and player
4      $Z, \sigma \leftarrow \mathit{TAttr}_{\alpha}^{\Game,\mathbf{T}}(\mathsf{pr}^{-1}(p))$                    tangle-attract to highest vertices
5      $O \leftarrow \{v \in Z_\alpha \mid E(v) \cap Z = \emptyset\} \cup \{v \in Z_{\overline{\alpha}} \mid E(v) \nsubseteq Z\}$      compute open vertices
6      $Y \leftarrow$ if $O = \emptyset$ then bottom-sccs($Z, \sigma$) else $\emptyset$         compute tangles if closed
7      return $Y \cup$ search($\Game \setminus Z, \mathbf{T}$)                         find tangles in remaining game
8  def tangle-learning($\Game$):
9      $W_\Diamond \leftarrow \emptyset$, $W_\Box \leftarrow \emptyset$, $\mathbf{T} \leftarrow \emptyset$                                   initialize sets
10     while $\Game \neq \emptyset$ :
11         $Y \leftarrow$ search($\Game, \mathbf{T}$)                                      search for tangles
12         $\mathbf{T} \leftarrow \mathbf{T} \cup \{T \in Y \mid E_T(T) \neq \emptyset\}$                      add new open tangles to T
13         $D \leftarrow \{T \in Y \mid E_T(T) = \emptyset\}$                              obtain new dominions
14         if $D \neq \emptyset$ :
15             $W_\Diamond \leftarrow W_\Diamond \cup \mathit{TAttr}_{\Diamond}^{\Game,\mathbf{T}}(\bigcup D_\Diamond), W_\Box \leftarrow W_\Box \cup \mathit{TAttr}_{\Box}^{\Game,\mathbf{T}}(\bigcup D_\Box)$    extend dominions
16             $\Game \leftarrow \Game \setminus (W_\Diamond \cup W_\Box)$                              remaining game
17             $\mathbf{T} \leftarrow \mathbf{T} \cap \Game$                                     remaining tangles
18     return $W_\Diamond, W_\Box$
```

Algorithm 3: The tangle learning algorithm.

Tangle learning is based on the tangle attractor, extending attractor computation to attract vertices of α-tangles where player $\overline{\alpha}$ must play to the attracting set, writing $\mathit{TAttr}_{\alpha}^{\Game,\mathbf{T}}(A)$ to attract vertices in subgame $\Game$ and vertices of α-tangles in the set $\mathbf{T}$ that are in subgame $\Game$ to A as player α, i.e., the least fixpoint of

$$\begin{aligned} Z := A &\cup \{\, v \in V_\alpha \mid E(v) \cap Z \neq \emptyset \,\} \cup \{\, v \in V_{\overline{\alpha}} \mid E(v) \subseteq Z \,\} \\ &\cup \{\, v \in T \mid T \in \mathbf{T} \wedge T \subseteq V \wedge \mathsf{pr}(T) \equiv_2 \alpha \wedge E_T(T) \subseteq Z \,\} \end{aligned}$$

See further [9]. The tangle learning algorithm (Algorithm 3) repeatedly decomposes the game with the tangle attractor. By computing the bottom strongly connected components of the regions, new tangles are obtained and added to the set of tangles $\mathbf{T}$ (line 12). Each iteration of this algorithm adds new tangles to this set, resulting in a different decomposition each time. Tangles without escapes, i.e., closed tangles, are dominions, which are then maximized using the attractor and removed from the game (lines 13–16). After removing the dominions, all tangles with solved vertices are removed from $\mathbf{T}$ (line 17). As before, winning strategies are obtained from the attractor. We omit explicitly computing the winning strategies, but we do need the strategy that the tangle attractor yields for computing the tangles at line 6 as explained

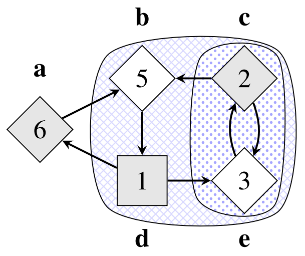


Figure 1: A 5-dominion with a 5-tangle and a 3-tangle

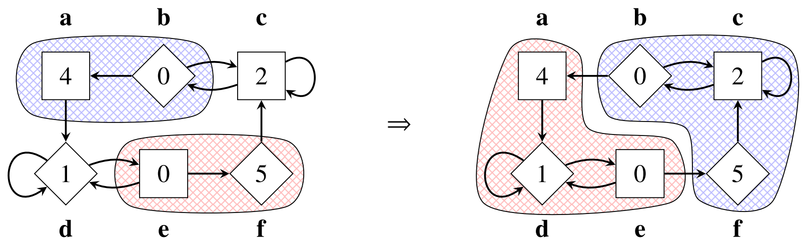


Figure 2: Vertex **a** is a distraction. The distraction is removed by the opponent's tangle $\{\mathbf{d}\}$. We can then learn tangle $\{\mathbf{b},\mathbf{c}\}$. Similarly, vertex **f** distracts tangle $\{\mathbf{d},\mathbf{e}\}$.

in [9]. A variation called "alternating tangle learning" is also described in [9], which alternates between exhaustively computing only tangles for player Odd or for player Even.

A closed region is essentially a collection of possibly unconnected tangles and vertices attracted to these tangles. Compared to tangle learning, priority promotion "attracts" all tangles in the region to the lowest region that would attract a tangle in the region, and then discards knowledge of the tangles.

6 Distractions

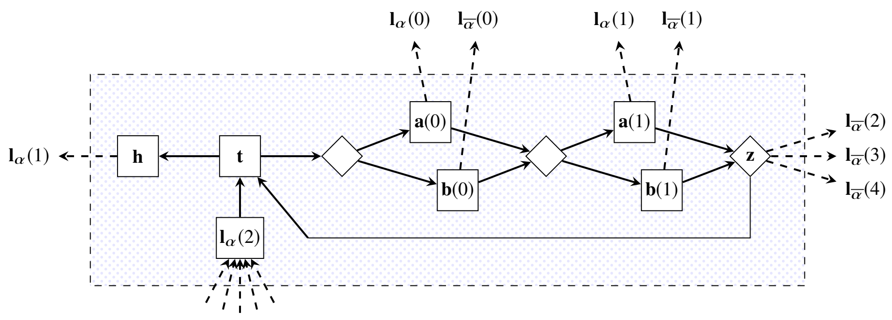
We introduce the notion of a distraction [9, 11] to further our understanding of parity game solving algorithms. A **distraction** for player α is a vertex v with an α-priority p, such that if player α always plays to reach v along paths of priorities $\leq p$, then player $\overline{\alpha}$ wins v and all vertices that reach v. That is, a distraction for α is a high value vertex v with an α-priority that player $\overline{\alpha}$ can win if player α always tries to visit it. This occurs when player $\overline{\alpha}$ can attract v to some vertex of a higher priority of player $\overline{\alpha}$, or when player $\overline{\alpha}$ can attract v to an $\overline{\alpha}$-dominion.

For attractor-based algorithms, the difficult distractions are those that are not immediately clear from the initial decomposition of the game, that is, when a distraction is attracted by the opponent *via a lower tangle*. Initially player α believes they should play to v, but after solving the lower subgame or finding the attracting tangle in the lower subgame, playing to v is believed to be good for the opponent. See e.g. Fig. 2. Vertices **a** and **f** are distractions. This is however not immediately clear.

We say that a distraction v distracts a tangle T, with $v \notin T$, if there is some *distracted* vertex $w \in T$, that is attracted by v but can also avoid v to play inside tangle T and player $\overline{\alpha}$ cannot escape T towards v. In order to learn the distracted tangle, the solver first needs to identify and avoid the distraction. As argued in [9], tangle learning identifies distractions by learning the $\overline{\alpha}$-tangle that attracts the distraction.

Zielonka's recursive algorithm and priority promotion also rely on this mechanism to avoid distractions. In priority promotion, whenever a merged region attracts the highest vertices from $\overline{\alpha}$-regions, then these vertices are distractions. In the recursive algorithm, the second recursive call is only performed if the opponent attracts from A, that is, when it discovers the attracted p-vertices in A are distractions. Thus, distractions make these algorithms slow, especially when lower distractions must be identified before *and* after removing a higher distraction. If there are no distractions in the game, then both the recursive algorithm and priority promotion require at most n recursive calls or promotions.

Algorithms that are based on play valuations such as progress measures and strategy iteration (see [10]) rely on a fundamentally different method to deal with distractions. They assign a higher value to vertices with α's priorities along the attractor paths towards vertices in A, and if a vertex in A is a distraction, then it does not increase in value, causing these algorithms to avoid it.



$\mathbf{l}_\alpha(0)$
$\mathbf{l}_{\overline{\alpha}}(0)$
$\mathbf{l}_\alpha(1)$
$\mathbf{l}_{\overline{\alpha}}(1)$
$\mathbf{l}_\alpha(1)$
h
t
$\mathbf{a}(0)$
$\mathbf{b}(0)$
$\mathbf{a}(1)$
$\mathbf{b}(1)$
z
$\mathbf{l}_{\overline{\alpha}}(2)$
$\mathbf{l}_{\overline{\alpha}}(3)$
$\mathbf{l}_{\overline{\alpha}}(4)$
$\mathbf{l}_\alpha(2)$

Figure 3: Bit 2 of a 5-bit Two Counters game. We use diamonds for vertices of player α and boxes for vertices of player $\overline{\alpha}$.

7 A Tale of Two Counters

The design of our parameterized parity game is *loosely* inspired upon earlier work by Friedmann [23], which provides an exponential lower bound for the non-oblivious strategy improvement algorithm [14]. That algorithm is a variation of strategy improvement that learns snares, which are related to tangles. However, attractor-based algorithms trivially solve these games and require a different approach.

The parity game consists of two binary counters. Each kth bit of the counter is a structure that contains 2^k many tangles. A bit in the counter of player α is set by attracting a distraction for player $\overline{\alpha}$ called a “low” vertex via a lower α-tangle to a “high” vertex. We use two counters, one for each player, that are intertwined and progress in turns. First the counter of player Even increases, then of player Odd, etc. The bits are connected to the higher bits such that this tangle is no longer “attracting” when the higher bits of the opponent change, letting the opponent escape the tangle, thus the tangle no longer attracts the distraction and the bit is reset. So when bit x of player α is set, then all bits below x of player $\overline{\alpha}$ are reset. Furthermore, if lower bits *of the opponent* are not set, then the tangle cannot be found because the player is distracted by these distractions. Bit x of player Odd is distracted by bits $\leq x$ of player Even, while bit x of player Even is distracted by bits $<x$ of player Odd. The result is the following example progression:

Even	**Odd**	**Event**
000	000	Initial state
001	000	Set Even 3
001	001	Set Odd 3
011	000	Set Even 2 (reset Odd 3)
010	010	Set Odd 2 (reset Even 3)
011	010	Set Even 3
011	011	Set Odd 3
111	000	Set Even 1 (reset Odd 2, 3)
100	100	Set Odd 1 (reset Even 2, 3)
101	100	Set Even 3
101	101	Set Odd 3
111	100	Set Even 2 (reset Odd 3)
110	110	Set Odd 2 (reset Even 3)
111	110	Set Even 3
111	111	Set Odd 3

Notice that the state of the counters after every Odd bit, colored blue, tracks the progression of the counters, while after every Even bit the counters are in an intermediate state.

Player	**Bit**	**l**	**h**	**Edge z** $\rightarrow$ $\mathbf{i}_{\overline{\alpha}}(b)$?
Odd	bit 0	8	15	yes
Even	bit 0	7	14	no
Odd	bit 1	6	13	yes
Even	bit 1	5	12	no
Odd	bit 2	4	11	yes
Even	bit 2	3	10	no

Table 1: Instantiation of a Two Counters game with $N = 3$, where player Even starts.

See Fig. 3 for bit 2 of some player α in a 5-bit counter. The highest bit is bit 0 and the lowest bit is bit 4. Vertex **l** is the "low" vertex and has a high priority of player $\overline{\alpha}$'s parity. This vertex is a distraction for the opponent if the bit is not set. Vertex **h** is the "high" vertex and has a high priority of player α's parity. Vertices **h** and **l** of higher bits have higher priorities. Vertex **h** has an edge to the input of the next higher bit, except vertex **h** of the highest bit has an edge to the input of the lowest bit. When all bits are set, the algorithms can find the winning dominion by connecting all bits. Vertex **t** is the "tangle" vertex and has priority 2 for even bits and priority 1 for odd bits. All other vertices have the priority $\mathsf{pr}(\mathbf{t}) - 1$.

To find a tangle, player α must force their opponent to play from **t** to **z** such that the opponent can only escape to **h** or to higher regions of player α. Each tangle in the bit uses a different path from **t** to **z**, depending on the state of the higher bits. As the two counters progress together, either $\mathbf{l}_{\alpha}(0)$ is good for α and $\mathbf{l}_{\overline{\alpha}}(0)$ is bad for α, or vice versa. In Fig. 3, if bit 0 is set and bit 1 is unset, then the path via $\mathbf{a}(0)$ and $\mathbf{b}(1)$ forces the opponent to stay in the tangle, whereas they could play to their own region via $\mathbf{b}(0)$ and $\mathbf{a}(1)$. Furthermore, player α must choose to play from **z** to **t**. When lower $\overline{\alpha}$-bits are not set, the $\mathbf{l}_{\overline{\alpha}}$ vertices are in a region of player α higher than **t** and are thus more attractive to play to, preventing player α from learning the tangle. Finally, we have an asymmetry, since one player starts counting first having the lowest priority **l** and **h** vertices. We only have the edge from vertex **z** to the matching vertex $\mathbf{l}_{\overline{\alpha}}$ for the bits of the second player. This forces the second player to wait until the first player sets the matching bit.

See Table 1 for an example with 3 bits, and Fig. 4 for the full 3-bit Two Counters game. Since player Even has the lower priorities, player Even first sets bit 2, followed by player Odd. When all bits are set, both players have a dominion consisting of their entire counter via the blue edges.

The number of vertices for a Two Counters game with N bits for both players is $3N^2 + 5N$ and the number of edges is $7N^2 + 4N$.

8 Empirical evaluation

To assess the runtime complexity of solving N-bit Two Counters games with different algorithms, we solve them with these algorithms and obtain a relevant statistic that is indicative of the runtime. In this section, we support the claim that the games are exponential for these solvers based on this statistic. Although this is a weaker basis than a machine-checkable complexity proof, such a proof is more difficult to produce and to understand, while the exponential behavior is clear from the empirical data. However, in Section 9, we do take a closer look at how these algorithms solve the Two Counters games, and we sketch out an inductive proof that the algorithms require exponentially many steps for higher N.

To solve the games we use the implementation of the algorithms in the parity game solver Oink [10], but we have also confirmed the results with implementations in PGSolver [25]. We use the following command line to obtain the results:

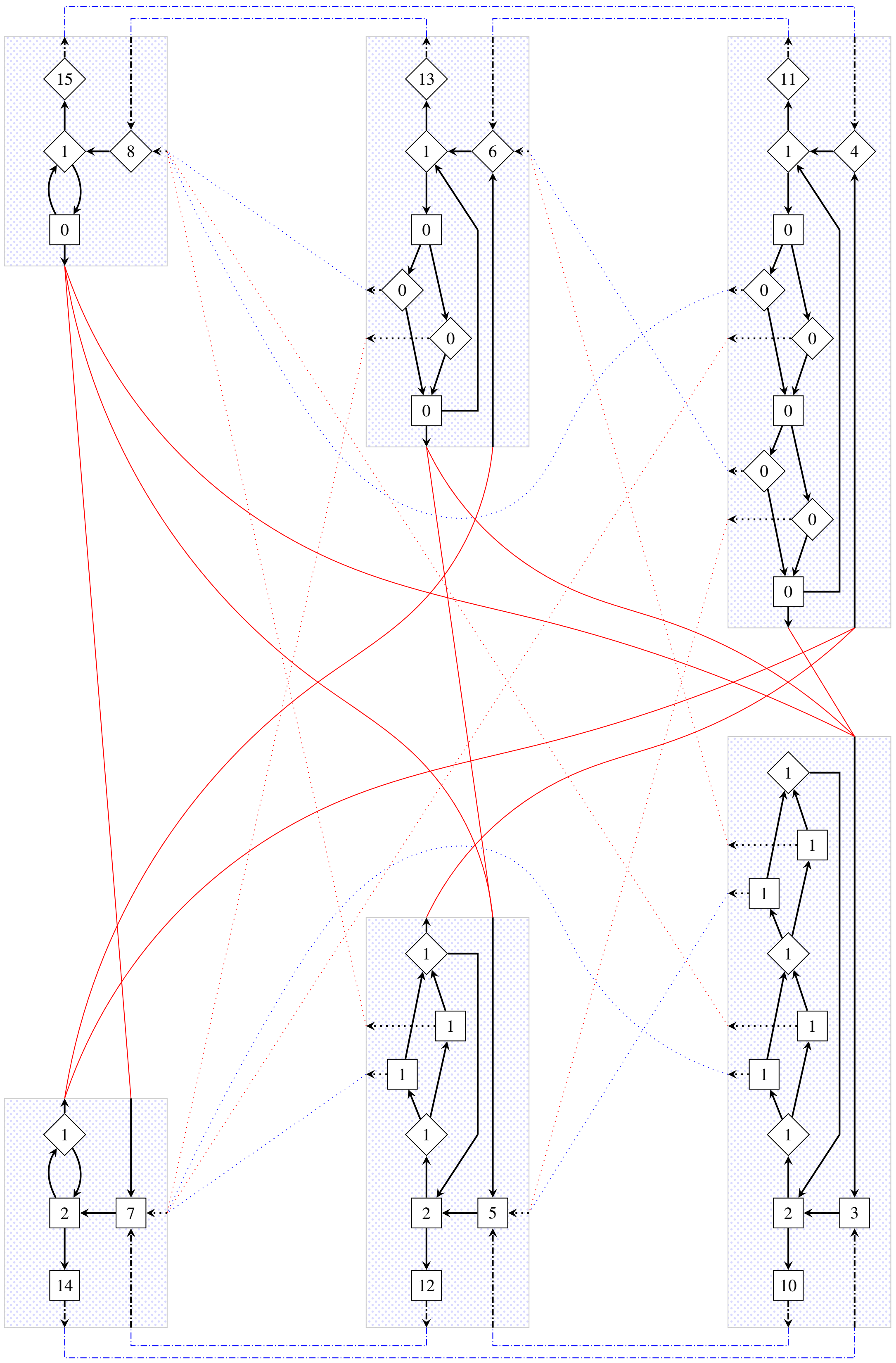


Figure 4: The 3-bit Two Counters game.

bits	#vertices	**ZLK** calls	**PP** promotions	**DP** promotions	**RR, RRDP** promotions	**TL, ATL** tangles
1	8	8	2	2	2	2
2	22	21	9	7	6	6
3	42	45	23	18	14	14
4	68	91	52	43	30	30
5	100	181	112	97	62	62
6	138	359	235	210	126	126
7	182	713	485	442	254	254
8	232	1,419	990	913	510	510
9	288	2,829	2,006	1,863	1,022	1,022
10	350	5,647	4,045	3,772	2,046	2,046
15	750	180,249	130,961	122,742	65,534	65,534
20	1,300	5,767,203	4,194,108	3,931,927	2,097,150	2,097,150

Table 2: Solving the Two Counters games with different algorithms. We report the relevant statistic that represents the runtime for each algorithm.

```
for s in zlk pp dp rr rrdp tl atl; do
    for i in 1 2 3 4 5 6 7 8 9 10 15 20; do
        ./tc $i | ./oink --$s | grep "solved with"; done; done
```

In Table 2, we report the following statistic for each solver:

- the number of recursive calls for Zielonka's recursive algorithm (ZLK)
- the number of promotions for priority promotion (PP, DP, RR, RRDP)
- the number of tangles for (alternating) tangle learning (TL, ATL)

Table 2 shows that the relevant statistic doubles with each higher number of bits, which supports that the runtime is exponential in the number of bits. The number of non-dominion tangles for ATL and TL is the same as the number of promotions for RR and RRDP, namely $2 \times (2^N - 1)$ tangles to set all bits. Since the number of vertices and edges increases quadratically, we conclude that the Two Counters games provide an exponential lower bound to these algorithms, namely $\Omega(2^{\sqrt{n}})$. We also investigated whether inflation and compression [10] had any effect and this was not the case.

In Section 9, we take a closer look at how these algorithms solve the Two Counters games, and sketch an inductive proof that the algorithms require exponentially many steps for higher N.

9 Analysis

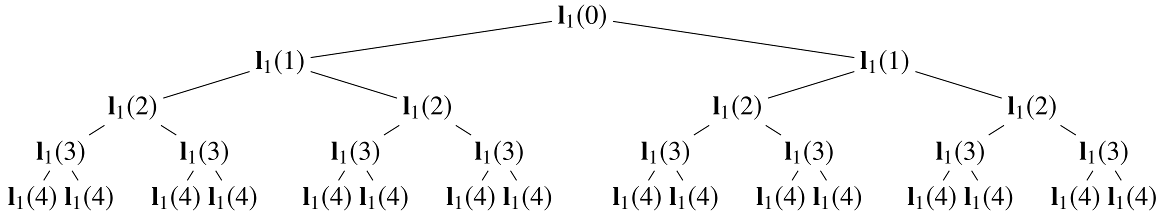
We study how the three algorithms solve a Two Counters game by looking at the timeline of events obtained from the trace output of the solver. We then *sketch a proof* arguing that any N-bit Two Counters game requires 2^N many steps for the three algorithms.

9.1 Recursive algorithm

As argued in Section 6, to make the recursive algorithm take exponential time, we need distractions such that after a "higher" distraction is attracted, all "lower" distractions must be attracted again. First, we use the trace output of Oink to obtain a timeline of identified distractions using the command line `./tc 5 | ./oink --zlk -t -t | grep "distraction" | grep "Odd"`:

4, 3, 4, 2, 4, 3, 4, 1, 4, 3, 4, 2, 4, 3, 4, 0, 4, 3, 4, 2, 4, 3, 4, 1, 4, 3, 4, 2, 4, 3, 4

We can also draw this timeline as a tree, which is an abstraction of the recursive call graph:


$\mathbf{l}_1(0)$
$\mathbf{l}_1(1)$ $\mathbf{l}_1(1)$
$\mathbf{l}_1(2)$ $\mathbf{l}_1(2)$ $\mathbf{l}_1(2)$ $\mathbf{l}_1(2)$
$\mathbf{l}_1(3)$ $\mathbf{l}_1(3)$ $\mathbf{l}_1(3)$ $\mathbf{l}_1(3)$ $\mathbf{l}_1(3)$ $\mathbf{l}_1(3)$ $\mathbf{l}_1(3)$ $\mathbf{l}_1(3)$
$\mathbf{l}_1(4)$ $\mathbf{l}_1(4)$ $\mathbf{l}_1(4)$ $\mathbf{l}_1(4)$ $\mathbf{l}_1(4)$ $\mathbf{l}_1(4)$ $\mathbf{l}_1(4)$ $\mathbf{l}_1(4)$ $\mathbf{l}_1(4)$ $\mathbf{l}_1(4)$ $\mathbf{l}_1(4)$ $\mathbf{l}_1(4)$ $\mathbf{l}_1(4)$ $\mathbf{l}_1(4)$ $\mathbf{l}_1(4)$ $\mathbf{l}_1(4)$

From the above, it is at least clear that the algorithm behaves as expected for the 5-bit Two Counters game. We now sketch a proof that the recursive algorithm requires exponential time for all Two Counters games.

Lemma 1. *The recursive algorithm follows the progression of a binary counter when solving a Two Counters game.*

Proof. We follow the progression of the *odd* bits. In the initial recursive decomposition, each **h** vertex attracts no vertices and each **l** vertex attracts the connected vertices of lower bits and distracts connected **z** vertices. Since no **l** vertex is itself attracted, all distractions are unattracted and thus are the bits unset.

Because of its strict recursive structure, the recursive algorithm cannot attract and identify a higher distraction before all lower distractions.

After removing an *even* distraction, Even attracts the entire subgame except the tangle of the corresponding bit of Odd. The reason for this is that the even bit **l** is now good-for-Even, while odd bit **l** is not yet good-for-Odd, so all lower bits are trivially attracted to the even region. For example, in Fig. 4, after attracting the distraction with priority 7 (in bit 0 of Even), vertices 7 and 8 are good for Even, and all lower vertices are now attracted to the even region, except the tangle of the odd bit. Hence, after removing the corresponding *odd* distraction, the set B (line 6 of Alg. 1) consists exactly of the odd tangle, the attracted distraction and the directly connected lower vertices. That is, all the lower distractions are in the subgame that is recomputed and are distracting again. Thus, setting higher bits resets the lower bits.

We have established that the counter starts with all bits unset, that setting higher bits resets all lower bits, and that higher bits are not set before the lower bits are set. Therefore the recursive algorithm follows the progression of a binary counter. □

Notice that in the above proof, because of the strict recursive nature, we do not need to distract **z** to make the recursive algorithm wait until all lower bits are set before setting a higher bit. In fact, removing the edges from **z** vertices to lower distractions results in even more recursive calls. Priority promotion and tangle learning are not bound to a strict order like the recursive algorithm.

9.2 Priority promotion

In priority promotion, a bit is set when the region of tangle vertex **t** is closed, promotes to a higher region and attracts the distraction **l**. When solving the 3-bit TC game with the delayed promotion policy, the command line `./tc 3 | ./oink --dp -t -t | egrep "promoted|delayed"` gives us the following sequence of promotions:

	Promotion	**Bit**	**Counter state (DP)**		**Delayed**	**Recovered**
			even	odd	(in DP/RRDP)	(in RR/RRDP)
1	2 to 6	Even 2	001	000		
2	1 to 5	Odd 2	001	001		
3	2 to 8	Even 1	011	000	yes	
4	1 to 7	Odd 1	010	010		
5	2 to 8	Even 2	011	000	yes	
6	1 to 7	Odd 1	011	010		yes
7	1 to 7	Odd 2	011	011		
8	2 to 14	Even 0	111	000	yes	
9	1 to 15	Odd 0	000	100		
10	2 to 14	Even 0	100	100		yes
11	2 to 6	Even 2	101	100		
12	1 to 5	Odd 2	101	101		
13	2 to 14	Even 1	111	100	yes	
14	1 to 15	Odd 1	000	110		
15	2 to 14	Even 0	100	110		yes
16	2 to 14	Even 1	110	110		yes
17	2 to 14	Even 2	111	110		
18	1 to 15	Odd 2	111	111		

We record the state of the two counters after each promotion with the DP solver. The "recovered" promotions do not occur in the RR and RRDP algorithms, as their regions are recovered. All delayed promotions are immediately applied, because after delaying the promotion, no normal promotion is available. Thus the delayed promotion policy is not useful here. Region recovery is however useful, as was already clear from Table 2. Both PP and DP (based on PP+) perform additional promotions that are not necessary with region recovery. For the 3-bit TC game, PP requires 23 promotions, DP requires 18 promotions, RR requires 14 promotions and RRDP requires 14 promotions.

Lemma 2. *Priority promotion follows the progression of a binary counter when solving a Two Counters game.*

Proof. Similar to the proof of Lemma 1. Initially no distractions are attracted by their opponent, so the counters start with 0. Whenever a higher bit is reset, the PP and PP+ policies reset all lower bits of the other player and the RR algorithm will not be able to recover because the earlier strategy leaves the remaining subgame. Due to the interleaving, setting odd bits resets lower even bits and vice versa. Therefore higher bits reset lower bits. Notice that the region of a distraction can never be closed. Therefore as long as a distraction is not attracted, all vertices **z** from the higher bits are distracted and cannot be part of a tangle until the distraction is attracted. Thus high bits must wait for lower bits. From this follows Lemma 2. □

9.3 Tangle learning

We already explained in Section 7 why tangle learning follows the progression of the counters, as the counters are designed to cause tangle learning to learn all tangles in all bits. We see that it works as expected via the command line `./tc 5 | ./oink --tl -t -t -t | grep "new tangle"` which yields precisely the expected tangles: Even 4, Odd 4, Even 3, Odd 3, Even 4, Odd 4, Even 2, Odd 2, Even 4, Odd 4, Even 3, Odd 3, Even 4, Odd 4, Even 1, Odd 1, Even 4, Odd 4, Even 3, Odd 3, Even 4, . . .

9.4 Other algorithms

We also considered the fixpoint algorithms [7, 11, 36], the small progress measures algorithm [29], and the two quasi-polynomial algorithms, ordered progress measures [8, 16] and succinct progress measures [30].

The fixpoint algorithms require exponential time as they are essentially myopic versions of the recursive algorithm, based on a one-step attractor instead of full attractors. The other algorithms are certainly slow and appear to require lifting vertices across the entire available range, but this is not due to the design of the Two Counters games. Rather, these algorithms tend to be extremely slow even for trivial games like winning self-loops. The exception is in fact strategy iteration [15], which is the only algorithm that can quickly and in polynomial time solve the Two Counters games.

The Two Counters games are difficult for attractor-based algorithms for the simple reason that player α prefers to play to the distractions of the opponent's counters as these have the highest value for the player. As soon as the player decides to play from **z** to **t**, the tangle is learned. Algorithms using progress measures and the strategy iteration algorithm assign a higher value to vertex **t** as soon as there is a path to **t** from **z**, because vertices with α's priority along the path receive a progressively higher value. Thus the players are not distracted very long by the distractions.

10 Discussion

Since solving parity games is known to lie in UP$\cap$co-UP, it is widely believed to admit a polynomial solution. Researchers have been studying this problem for several decades. For some time, the focus was on variations of strategy iteration algorithms, where a suitable *improvement rule* could lead to a polynomial solution. Friedmann et al. provided superpolynomial or exponential lower bounds for many of such rules [1, 18, 19, 21, 22, 23, 24], as well as Fearnley and Savani recently [17].

For a long time, the main attractor-based algorithm was the recursive algorithm by McNaughton [33] and Zielonka [37], for which Friedmann also showed an exponential lower bound [20] and which Gazda and Willemse [26] later improved upon. These lower bounds resist techniques like inflation, compression and SCC decomposition. As the lower bound of Friedmann can be defeated by memoization, since the number of distinct subgames is polynomial, Benerecetti et al. proposed a more resilient lower bound [5]. Their priority promotion algorithm [4] solves [5, 20, 26] in polynomial time, although this requires inflation for [26]. For priority promotion, two exponential lower bounds have been presented [2, 6]. However for variations of priority promotion, in particular for DP [5], no superpolynomial lower bound has been published. Tangle learning [9] solves all these lower bound examples in polynomial time.

We presented the parameterized Two Counters game and showed that this provides an exponential lower bound for the recursive algorithm, for (all variations of) priority promotion, in particular with the delayed promotion policy, and for tangle learning. An N-bit Two Counters game has size $n = 3N^2 + 5N$ and requires at least $2 \times (2^N - 1)$ steps, thus providing a lower bound of $\Omega(2^{\sqrt{n}})$.

The Two Counters game is available online via `https://www.github.com/trolando/oink`.

Acknowledgements

We thank Veronika Loitzenbauer for the helpful discussions on these and other difficult parity games. We also thank Marcin Jurdziński, Ranko Lazić, Laure Daviaud and Laurent Doyen for their hospitality and discussions. Finally, we thank Armin Biere for generously supporting this research.

References

[1] David Avis & Oliver Friedmann (2017): *An exponential lower bound for Cunningham's rule*. *Math. Program.* 161(1-2), pp. 271–305, doi:10.1007/s10107-016-1008-4.

[2] Massimo Benerecetti, Daniele Dell'Erba & Fabio Mogavero (2016): *A Delayed Promotion Policy for Parity Games*. In: *GandALF* 2016, *EPTCS* 226, pp. 30–45, doi:10.4204/EPTCS.226.3.

[3] Massimo Benerecetti, Daniele Dell'Erba & Fabio Mogavero (2016): *Improving Priority Promotion for Parity Games*. In: *HVC* 2016, *LNCS* 10028, pp. 117–133, doi:10.1007/978-3-319-49052-6_8.

[4] Massimo Benerecetti, Daniele Dell'Erba & Fabio Mogavero (2016): *Solving Parity Games via Priority Promotion*. In: *CAV* 2016, *LNCS* 9780, Springer, pp. 270–290, doi:10.1007/978-3-319-41540-6_15.

[5] Massimo Benerecetti, Daniele Dell'Erba & Fabio Mogavero (2017): *Robust Exponential Worst Cases for Divide-et-Impera Algorithms for Parity Games*. In: *GandALF*, *EPTCS* 256, pp. 121–135, doi:10.4204/EPTCS.256.9.

[6] Massimo Benerecetti, Daniele Dell'Erba & Fabio Mogavero (2018): *Solving parity games via priority promotion*. *Formal Methods in System Design* 52(2), pp. 193–226, doi:10.1007/s10703-018-0315-1.

[7] Florian Bruse, Michael Falk & Martin Lange (2014): *The Fixpoint-Iteration Algorithm for Parity Games*. In: *GandALF*, *EPTCS* 161, pp. 116–130, doi:10.4204/EPTCS.161.12.

[8] Cristian S. Calude, Sanjay Jain, Bakhadyr Khoussainov, Wei Li & Frank Stephan (2017): *Deciding parity games in quasipolynomial time*. In: *STOC*, ACM, pp. 252–263, doi:10.1145/3055399.3055409.

[9] Tom van Dijk (2018): *Attracting Tangles to Solve Parity Games*. In: *CAV* (2), *LNCS* 10982, Springer, pp. 198–215, doi:10.1007/978-3-319-96142-2_14.

[10] Tom van Dijk (2018): *Oink: An Implementation and Evaluation of Modern Parity Game Solvers*. In: *TACAS* (1), *LNCS* 10805, Springer, pp. 291–308, doi:10.1007/978-3-319-89960-2_16.

[11] Tom van Dijk & Bob Rubbens (2019): *Simple Fixpoint Iteration to Solve Parity Games*. Accepted at GandALF 2019.

[12] E. Allen Emerson & Charanjit S. Jutla (1991): *Tree Automata, Mu-Calculus and Determinacy (Extended Abstract)*. In: *FOCS*, IEEE Computer Society, pp. 368–377, doi:10.1109/SFCS.1991.185392.

[13] E. Allen Emerson, Charanjit S. Jutla & A. Prasad Sistla (2001): *On model checking for the mu-calculus and its fragments*. *Theor. Comput. Sci.* 258(1-2), pp. 491–522, doi:10.1016/S0304-3975(00)00034-7.

[14] John Fearnley (2010): *Non-oblivious Strategy Improvement*. In: *LPAR* (*Dakar*), *LNCS* 6355, Springer, pp. 212–230, doi:10.1007/978-3-642-17511-4_13.

[15] John Fearnley (2017): *Efficient Parallel Strategy Improvement for Parity Games*. In: *CAV* (2), *LNCS* 10427, Springer, pp. 137–154, doi:10.1007/978-3-319-63390-9_8.

[16] John Fearnley, Sanjay Jain, Bart de Keijzer, Sven Schewe, Frank Stephan & Dominik Wojtczak (2019): *An ordered approach to solving parity games in quasi-polynomial time and quasi-linear space*. *STTT* 21(3), pp. 325–349, doi:10.1007/s10009-019-00509-3.

[17] John Fearnley & Rahul Savani (2016): *The Complexity of All-switches Strategy Improvement*. In: *SODA*, SIAM, pp. 130–139, doi:10.1137/1.9781611974331.ch10.

[18] Oliver Friedmann (2009): *An Exponential Lower Bound for the Parity Game Strategy Improvement Algorithm as We Know it*. In: *LICS*, IEEE Computer Society, pp. 145–156, doi:10.1109/LICS.2009.27.

[19] Oliver Friedmann (2011): *An Exponential Lower Bound for the Latest Deterministic Strategy Iteration Algorithms*. *Logical Methods in Computer Science* 7(3), doi:10.2168/LMCS-7(3:23)2011.

[20] Oliver Friedmann (2011): *Recursive algorithm for parity games requires exponential time*. *RAIRO - Theor. Inf. and Applic.* 45(4), pp. 449–457, doi:10.1051/ita/2011124.

[21] Oliver Friedmann (2011): *A Subexponential Lower Bound for Zadeh's Pivoting Rule for Solving Linear Programs and Games*. In: *IPCO*, *Lecture Notes in Computer Science* 6655, Springer, pp. 192–206, doi:10.1007/978-3-642-20807-2_16.

[22] Oliver Friedmann (2012): *A subexponential lower bound for the Least Recently Considered rule for solving linear programs and games*. In: *GAMES*.

[23] Oliver Friedmann (2013): *A superpolynomial lower bound for strategy iteration based on snare memorization*. *Discrete Applied Mathematics* 161(10-11), pp. 1317–1337, doi:10.1016/j.dam.2013.02.007.

[24] Oliver Friedmann, Thomas Dueholm Hansen & Uri Zwick (2011): *A subexponential lower bound for the Random Facet algorithm for Parity Games*. In: *SODA*, SIAM, pp. 202–216, doi:10.1137/1.9781611973082.19.

[25] Oliver Friedmann & Martin Lange (2009): *Solving Parity Games in Practice*. In: *ATVA*, *LNCS* 5799, Springer, pp. 182–196, doi:10.1007/978-3-642-04761-9_15.

[26] Maciej Gazda & Tim A. C. Willemse (2013): *Zielonka's Recursive Algorithm: dull, weak and solitaire games and tighter bounds*. In: *GandALF*, *EPTCS* 119, pp. 7–20, doi:10.4204/EPTCS.119.4.

[27] Erich Grädel, Wolfgang Thomas & Thomas Wilke, editors (2002): *Automata, Logics, and Infinite Games: A Guide to Current Research*. *LNCS* 2500, Springer, doi:10.1007/3-540-36387-4.

[28] Marcin Jurdzinski (1998): *Deciding the Winner in Parity Games is in UP ∩ co-UP*. *Inf. Process. Lett.* 68(3), pp. 119–124, doi:10.1016/S0020-0190(98)00150-1.

[29] Marcin Jurdzinski (2000): *Small Progress Measures for Solving Parity Games*. In: *STACS*, *LNCS* 1770, Springer, pp. 290–301, doi:10.1007/3-540-46541-3_24.

[30] Marcin Jurdzinski & Ranko Lazic (2017): *Succinct progress measures for solving parity games*. In: *LICS*, IEEE Computer Society, pp. 1–9, doi:10.1109/LICS.2017.8005092.

[31] Dexter Kozen (1983): *Results on the Propositional mu-Calculus*. *Theor. Comput. Sci.* 27, pp. 333–354, doi:10.1016/0304-3975(82)90125-6.

[32] Orna Kupferman & Moshe Y. Vardi (1998): *Weak Alternating Automata and Tree Automata Emptiness*. In: *STOC*, ACM, pp. 224–233, doi:10.1145/276698.276748.

[33] Robert McNaughton (1993): *Infinite Games Played on Finite Graphs*. *Ann. Pure Appl. Logic* 65(2), pp. 149–184, doi:10.1016/0168-0072(93)90036-D.

[34] Philipp J. Meyer, Salomon Sickert & Michael Luttenberger (2018): *Strix: Explicit Reactive Synthesis Strikes Back!* In: *CAV* (1), *Lecture Notes in Computer Science* 10981, Springer, pp. 578–586, doi:10.1007/978-3-319-96145-3_31.

[35] Pawel Parys (2019): *Parity Games: Zielonka's Algorithm in Quasi-Polynomial Time*. *CoRR* abs/1904.12446.

[36] Antonio Di Stasio, Aniello Murano, Giuseppe Perelli & Moshe Y. Vardi (2016): *Solving Parity Games Using an Automata-Based Algorithm*. In: *CIAA*, *LNCS* 9705, Springer, pp. 64–76, doi:10.1007/978-3-319-40946-7_6.

[37] Wieslaw Zielonka (1998): *Infinite Games on Finitely Coloured Graphs with Applications to Automata on Infinite Trees*. *Theor. Comput. Sci.* 200(1-2), pp. 135–183, doi:10.1016/S0304-3975(98)00009-7.