Solving parity games, very slowly

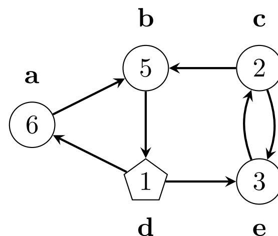
Tom van Dijk

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

Abstract. Parity games are conceptually easy to understand and might just be solvable in polynomial time! So far, no polynomial time solution has been discovered. Surely this fact attracts the attention of algorithm aficionados. Quasi-polynomial time solutions have been found in the recent decade, along with proofs that certain families of parity game algorithms have a quasi-polynomial time lower bound. Since parity games are in the intersection of NP and co-NP, even UP and co-UP, surely they admit a polynomial-time solution? The quest is therefore not finished, the question remains open: can we solve parity games in polynomial time? Or can we not, and would that imply that parity games separate P and NP? We focus our attention on algorithms that repeatedly partition parity games using attractors, extended with knowledge of tangles. Tangles are subgames that are won by one player, forcing the other player to escape the tangle. By repeatedly partitioning the game and obtaining new tangles from the partition, tangle learning algorithms solve parity games. Our journey so far has focused on designing various variations of tangle learning and subsequently exploring examples that maximally distract and delay these tangle learning variations.

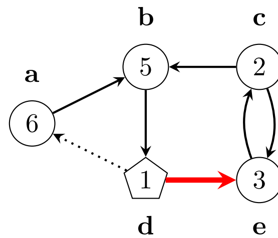
1 Introduction

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



Vertices controlled by player Odd are shaped ⬠ and vertices controlled by player Even are shaped ○. In this example, player Odd can make a decision at vertex **d**: play towards **a** and see a 6, or play towards **e** and avoid that 6? Similarly, player Even can make a decision at vertex **c** to either play to **b** or to **e**.

It is well known that parity games admit a positional strategy for the winner. That is, if player Even wins all plays in a parity game (or a part of a parity game), then player Even has a strategy for its winning vertices such that all cycles in the (sub)game are won by Even. Regardless of the choices by player Odd, every reachable cycle is won by Even. In the above example, it is actually player Odd who wins the entire parity game with the strategy $\mathbf{d} \to \mathbf{e}$. This looks as follows:



Now there are only two simple cycles in the graph: one of priorities 5, 1, 3 and 2, the other of priorities 2 and 3. Both are won by player Odd as both have an odd priority as their highest priority.

Parity games are pretty interesting in the area of model checking and verification. They are used for solving synthesis problems for ω-regular specifications, which is closely related to the model checking of ω-regular specifications, extensively described in [1] by Baier and Katoen. For model checking, non-deterministic Büchi automata suffice. For the synthesis problem of automatically constructing an implementation that satisfies the specification, we need a deterministic game and parity games are a popular choice.

Can we solve parity games in polynomial time? Solving parity games is known to lie in NP ∩ co-NP. It shares this status with a number of other path-forming graph problems, including mean payoff games and simple stochastic games [4,5,15]. Because parity games are in NP ∩ co-NP, it is widely speculated that a polynomial-time solution exists. Yet despite years of effort, no such solution has been found for the parity game problem or the related problems. In recent times, solutions have been found that have a "quasi-polynomial" upper bound, i.e., $\mathcal{O}(n^{\log n})$, which is above polynomial but below exponential [3,19]. A proof that parity games do not admit a polynomial-time solution would imply P $\neq$ NP, so it is unlikely that this could be obtained.

Our aim is to study features of hard parity games that slow down algorithms to their worst-case behavior. One such feature is the **tangle**, introduced in [6]. Tangles are roughly described as subgames where one player has a winning strategy for all plays confined to the tangle. Tangles play a fundamental role in parity game algorithms, but most algorithms are not explicitly aware of tangles and can explore the same tangles repeatedly, especially in the presence of *nested*

tangles [6]. The algorithms proposed in [6] solve parity games by explicitly computing tangles using attractor computation.

Another feature of hard parity games is that some vertices are **distractions**. We developed the concept of a distraction earlier in [6,8,11]. With a distraction we mean a vertex that a solver "assumes" to be good for one of the players, typically because the vertex has a high even or odd priority, but that must be avoided along some or all of the paths in order to win.

In this paper we give several examples of parity games with distractions. One particular example is the Two Counters parity game family [8], which is an exponential lower bound for many of the parity game algorithms implemented in Oink [7], such as Zielonka's recursive algorithm, priority promotion, tangle learning, the fixed point algorithms DFI and FPJ, and small progress measures [16]. This family also slows down the quasi-polynomial time progress measures algorithms [14,17] and quasi-polynomial variations on Zielonka's algorithm.

Every algorithm makes assumptions about the preference order between vertices, i.e., which vertices are good targets to play towards and which vertices are not. A fundamental difficulty in path-forming problems such as parity games is that it is not known what happens "after" a vertex is visited in a play, without investigating the rest of the parity game. This is especially difficult when many tangles need to be explored to determine if a vertex is safe to play towards or a distraction that leads to a losing game. For hard parity games, the decision that a vertex is a distraction assumes that certain other vertices are not distractions. When such a vertex is then found to be a distraction, earlier decisions are invalidated and earlier distractions need to be reevaluated. This leads to an exponential running time for many algorithms.

At the core of difficult to solve parity games are therefore **distracted tangles**, designed in a way that forces a parity game solving algorithm to find exponentially many tangles and "disarm" distractions exponentially often before a tangle is found that solves the game. An ongoing challenge is to find better mechanisms to deal with these distractions and avoid them. So far, we have worked on several different ideas and found weaknesses in the resulting algorithms that could be exploited to delay the algorithm repeatedly.

2 Preliminaries

Parity games

We formally define a parity game $\Game$ as a tuple $(V_{\bigcirc}, V_{\pentagon}, E, \mathsf{pr})$ where $V = V_{\bigcirc} \uplus V_{\pentagon}$ is a set of n vertices partitioned into disjoint sets $V_{\bigcirc}$ controlled by player *Even* and $V_{\pentagon}$ controlled by player *Odd*, and $E \subseteq V \times V$ is a left-total binary relation describing all edges. 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 $\alpha \in \{\circ, \pentagon\}$ to denote player $\circ$ or $\pentagon$ and $\overline{\alpha}$ for the opponent of α. Given some set of vertices U, we write U_α for all vertices in U controlled by player α.

We write $\mathsf{pr}(v)$ for the priority of a vertex v, $\mathsf{pr}(V)$ for the largest (highest) priority of a set of vertices V, and $\mathsf{pr}(⅁)$ for the largest priority in ⅁. Furthermore, we write $\mathsf{pr}^{-1}(p)$ for all vertices with the priority p. With $\mathsf{pr}^{-1}(\bigcirc)$ and $\mathsf{pr}^{-1}(⬠)$ we denote all vertices with an even or odd priority. Given some priority p, we write $\mathsf{parity}(p)$ to mean $\bigcirc$ if p is even, or $⬠$ if p is odd.

A *play* $\pi = v_0 v_1 \dots$ is an infinite sequence of vertices consistent with E, i.e., $v_i \to v_{i+1}$ for all successive vertices. We denote with $\inf(\pi)$ the set of vertices 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. Similarly, for any cycle in the game, we say that player Even wins the cycle if the highest priority along the cycle is even; or player Odd if the highest priority is odd.

A (positional) *strategy* $\sigma\colon V \to V$ 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 to V_α. We write $\mathrm{Plays}(v)$ for the set of plays starting at vertex v. We write $\mathrm{Plays}(v,\sigma)$ for all plays from v consistent with σ, 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 [13], i.e., each vertex is either winning for player Even or for player Odd, and both players have a positional strategy for their winning vertices. Player α wins vertex v if they have a strategy σ such that every $\pi \in \mathrm{Plays}(v,\sigma)$ is winning for player α. That is, player α has a strategy σ such that they win all cycles reachable from v in the induced game $⅁[\sigma]$, i.e., the game where vertices in the domain of σ only have an edge to the chosen successor in σ.

For some set of vertices U, we write $E(U)$ to denote $\{v \in E(u) \mid u \in U\}$. We call a set of vertices U *α-closed* if $(\forall u \in U_\alpha . E(u) \cap U \neq \emptyset) \wedge (\forall u \in U_{\overline{\alpha}} . E(u) \subseteq U)$, i.e., player α can stay in U and player $\overline{\alpha}$ cannot leave U. A set of vertices $D \subseteq V$ is called a dominion of player α if D is α-closed and player α wins all plays in D.

Solving a parity game means computing the winner of each vertex (assuming perfect play) and the strategy of each player to win these vertices.

Attractor computation

Several algorithms for solving parity games employ *attractor computation*. Given a set of *target vertices* A, the attractor to A for player α represents those vertices from which player α can force a visit of A. We write $\mathit{Attr}^{⅁}_{\alpha}(A)$ to attract vertices in ⅁ to A as player α, i.e., the least fixed point of

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

That is, starting with $Z = \emptyset$, we evaluate the above expression updating Z until we reach a fixed point. In practice, 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 $A = \mathit{Attr}^{⅁}_{\alpha}(A)$. The attractor also yields an *attractor strategy* for player α as follows: when attracting a vertex v controlled by player α to Z via an edge to a vertex $u \in Z$, then pick $v \to u$ as the strategy for v; if a

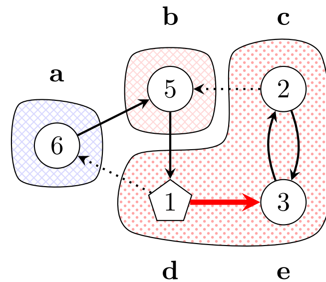
vertex $v \in A$ has an edge to a vertex $u \in Z$, then pick $v \to u$ as the strategy for v. It may be that some α-vertices in A do not get a strategy, i.e., when they cannot stay in Z. In the remainder of this work, we typically use A for the *target* set of vertices and Z for the *attractor* set. We also write $Attr_\alpha^U(A)$ when we restrict the α-attractor to attract from a set of vertices U only, i.e., the least fixed point of

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

Typically, $A \subseteq U$. The idea is that we only attract vertices from the set U towards the target set A, and that the vertices of player $\overline{\alpha}$ may only escape to vertices in the set U. Practical implementations of attractor computation often only count the number of edges of player $\overline{\alpha}$ that are not yet attracted, which improved the computational complexity.

Attractor decomposition

Attractors are typically used to attract to a set $A := \mathsf{pr}^{-1}(\mathsf{pr}(\circlearrowleft))$, i.e., the vertices with the highest priority in the game. By repeatedly computing this attractor and removing it from the game, the game is decomposed into so-called *regions*. We call the vertices in A the *top vertices* of region Z. We identify a set of *open top vertices* $O := \{v \in A_\alpha \mid E(v) \cap Z = \emptyset\} \cup \{v \in A_{\overline{\alpha}} \mid E(v) \not\subseteq Z\}$, i.e., all top vertices controlled by α that cannot stay in Z and all top vertices controlled by $\overline{\alpha}$ that can leave Z. That is, the open top vertices are exactly those that are not attracted to their own region. See for example the following decomposition:



Parity game algorithms usually prefer high priorities over low priorities, so we begin by computing the *Even* attractor to vertex **a**. We try to attract vertices of player Even that can play to **a** and vertices of player Odd that must play to **a**. In this case, no vertices are attracted, since vertex **d** still has the option of playing to **e**. We then attract for *Odd* to the next highest vertex, vertex **b**. Again, player Even can still play from **c** to **e** so vertex **c** is not attracted to **b** and there is no other vertex with an edge to **b**. Finally, we attract to vertex **e**, which has the highest priority in the remaining (unpartitioned) game with priority 3, and now vertex **c** has no other edge and must play to **e**. Vertex **d** is also attracted because it is controlled by player Odd.

We are interested in whether a region is closed w.r.t. a local subgame rather than the entire game, i.e., whether player $\overline{\alpha}$ can escape from A to this local subgame. We distinguish these by calling one *locally closed* and the other *globally closed*. The regions of vertices **a** and **b** are both open, as the top vertices are not attracted to their region, but attracted to the "lower" regions. Vertex **a** plays to region **b**, while vertex **b** plays to region **e**. Vertex **e** is attracted to its own region, therefore the region is locally closed. Since the opponent, player Even, can still leave the region by playing to vertex **b**, the region is not globally closed.

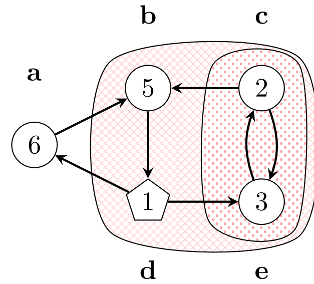
Computing attractors also computes attractor strategies. In the above example, player Odd has the strategy $\mathbf{d} \to \mathbf{e}$ inside region **e**.

Tangles

A *tangle* is a pair (U, σ), where $U \subseteq V$ is a nonempty set of vertices, $\sigma \colon U_\alpha \to U$ is a strategy for all vertices of player α, such that player α wins all cycles in the induced subgame $\Game[U, \sigma]$, and $\Game[U, \sigma]$ is non-trivially strongly connected.

Tangles have some important properties. All vertices of the winner have a strategy to stay inside the tangle, and the opponent *must* escape to avoid losing. As a tangle is strongly connected when fixing the strategy for the winner, the loser is free to choose any escape. If a tangle is closed, i.e., there are no escapes, then the tangle is a dominion for player α. Furthermore, the highest priority in the tangle is of player α.

Tangles can be *nested*. See for example the tangles in our example game:



We find that there are two Odd tangles in this game. The larger tangle, $\{\mathbf{b}, \mathbf{c}, \mathbf{d} \to \mathbf{e}, \mathbf{e}\}$ contains the smaller tangle $\{\mathbf{c}, \mathbf{e}\}$. In fact, player Odd wins all games that stay in the smaller tangle, and player Even is forced to play from **c** to **b**. Player Odd also wins all games that stay in the larger tangle, where player Even has no escaping edges. In the remainder, we use this notation for tangles of the set of vertices V, including the chosen edge in σ for vertices controlled by the winner.

The above game has one more tangle. The tangle $\{\mathbf{a} \to \mathbf{b}, \mathbf{b} \to \mathbf{d}, \mathbf{d}\}$ is won by player Even. However, while games that stay inside this tangle are won by player Even, player Odd can simply choose to not stay inside the tangle.

In general, there can be multiple tangles for the same set of vertices U, with different strategies. Here, we are not interested in other tangles for the same

vertices U and the tangle learning algorithms will only find at most a single tangle for any set of vertices U.

Tangle attractors

Because the opponent $\overline{\alpha}$ must escape a tangle won by player α, we can extend attractor computation by attracting all vertices of a tangle simultaneously when the escapes lead to Z. We update the attractor strategy with the tangle strategy.

We extend attractor computation to attract tangles, writing $Attr_\alpha^{\Game}(A, T)$ to attract vertices in $\Game$ and vertices of tangles in the set of tangles T to the target set of vertices A as player α, i.e., the least fixed point of

$$\begin{aligned} Z := A &\cup \{v \in V_\alpha \mid E(v) \cap Z \neq \emptyset\} \cup \{v \in V_{\overline{\alpha}} \mid E(v) \subseteq Z\} \\ &\cup \{v \in U \mid (U, \sigma) \in T \wedge \mathsf{parity}(\mathsf{pr}(U)) = \alpha \wedge (E(U_{\overline{\alpha}}) \setminus U) \subseteq Z\} \end{aligned}$$

In the implementation, we make use of the observation that if a α-tangle is already partially attracted to higher α-regions, then the remainder of the tangle is also an α-tangle. We can therefore also attract tangles when some part of them is already in higher region(s) of player α.

We also write $Attr_\alpha^U(A, T)$ restricted to a set of vertices U of the game, i.e., the least fixed point of

$$\begin{aligned} Z := A &\cup \{v \in U_\alpha \mid E(v) \cap Z \neq \emptyset\} \cup \{v \in U_{\overline{\alpha}} \mid E(v) \cap U \neq \emptyset \wedge E(v) \cap U \subseteq Z\} \\ &\cup \{v \in W \mid (W, \sigma) \in T \wedge W \subseteq U \wedge \mathsf{parity}(\mathsf{pr}(W)) = \alpha \\ &\qquad \wedge E(W_{\overline{\alpha}}) \cap Z \neq \emptyset \wedge (E(W_{\overline{\alpha}}) \setminus W) \cap U \subseteq Z\} \end{aligned}$$

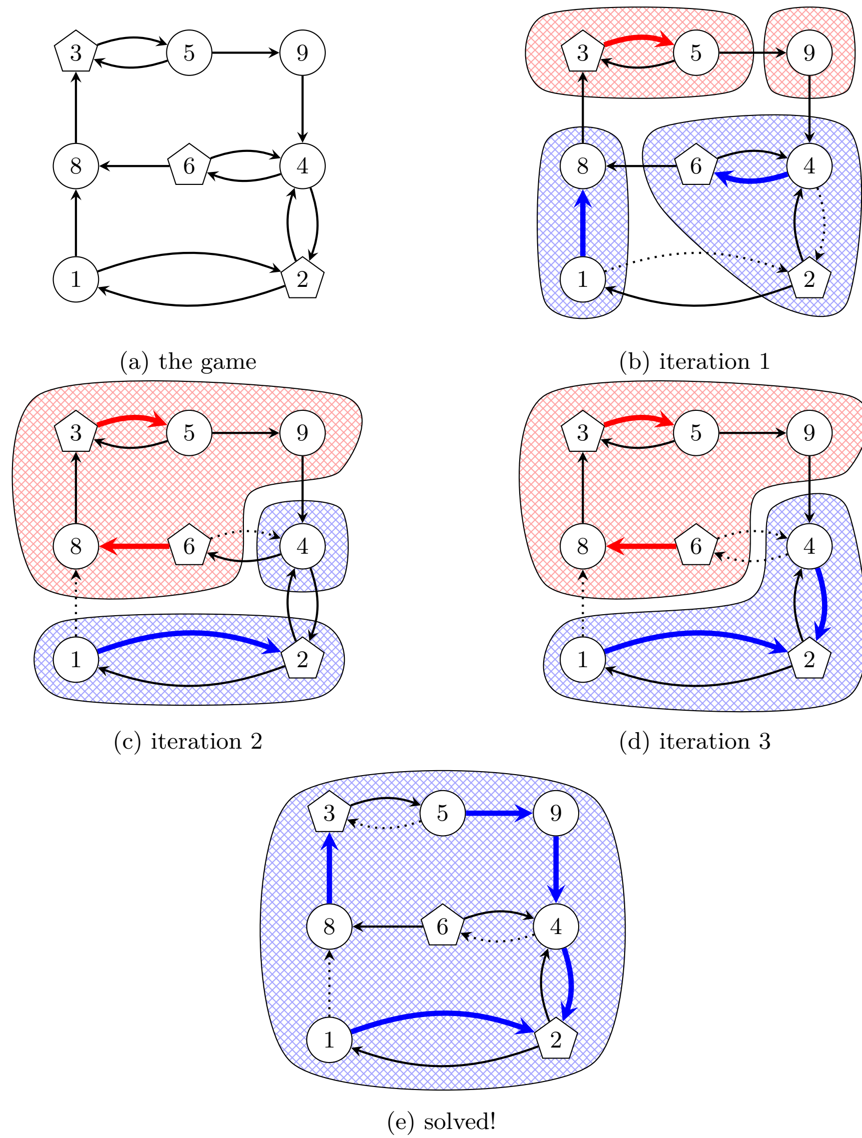
That is, the first line equals the definition of $Attr_\alpha^U(A)$. We furthermore add all vertices in some tangle $(W, \sigma) \in T$, where W is a subset of U, the winning player is player α, there is at least one escape from the tangle to Z, and there are no escapes to $U \setminus Z$.

Finally, we also write $Attr_\alpha^{\Game}(A, T, {\leq}p)$ and $Attr_\alpha^U(A, T, {\leq}p)$ to only attract vertices v where $\mathsf{pr}(v) \leq p$ and tangles (U, σ) where $\mathsf{pr}(U \setminus Z) \leq p$. This lets us properly attract to vertices that are not the highest priority in the game, without attracting vertices that have a higher priority.

In our example game, if we know that the tangle $\{\mathbf{c}, \mathbf{e}\}$ exists, then we can attract these vertices to region **b**, which also attracts vertex **d**. As a result, there are two regions: the open region **a** which attracts no other vertices, and the closed region **b** which attracts the rest of the game and is in fact globally closed, meaning that all plays inside the region are won by player Odd, and player Even is unable to leave the region.

3 Tangle learning

In earlier work, we presented the tangle learning algorithm [6]. We recall the basic algorithm here.



(a) the game (b) iteration 1

(c) iteration 2 (d) iteration 3

(e) solved!

Fig. 1: Solving an example game with standard tangle learning.

The core idea is to decompose the parity game into regions using tangle-attractor computation, then learn new tangles from the (locally) closed regions. We repeat this process of decomposition and extracting tangles until a region is globally closed, yielding a closed tangle. This closed tangle is now definitively won, as are also all vertices that are attracted from the rest of the game to this closed tangle. We remove this winning subgame and continue the process with the remaining unsolved vertices.

We learn new tangles from locally closed regions by performing SCC decomposition. Recall from the definition of tangles that a tangle is strongly connected if we restrict vertices to the chosen strategy of the winner. We simply apply Tarjan's algorithm starting in the top vertex to find the new tangle. When applying Tarjan's algorithm, we only follow the edges of the strategy for vertices of the winner.

We prove that this algorithm solves parity games in [6]. The argument is straightforward. We can prove that the lowest region in the attractor decomposition always contains a *new* tangle. Since the number of tangles in a parity game is finite, eventually a dominion must be found, therefore eventually the algorithm solves the entire game. For any set of vertices U, the algorithm will learn at most one tangle (U, σ).

Example 1. We illustrate how the algorithm works using an example game. Recall that to determine whether a region is open, we only need to consider the top vertices. See Figure 1. In this example, we refer to vertices by their priorities. In general, vertices do not need to have unique priorities. However, w.l.o.g., we can assume that each vertex is assigned a unique priority.

Iteration 1 We decompose the game into regions for the first time, and we do not know any tangles yet. See Figure 1b. The first highest vertex is 9. As player Even can still play from 5 to 3, no vertices are attracted to 9. Region 9 is open due to the edge from 9 to 4. The next highest vertex in the remaining subgame is 8. Player Even attracts 1 towards 8 and no other vertices are attracted: vertices 2 and 6 can still go to 4. The region is open, as player Even cannot stay in the region from 8. The next highest vertex is 6. We attract vertices 4 and 2 to the region; notice that player Odd cannot escape from 2 to 1, since it is in the higher region 8. Region 6 is closed, so we now learn new tangles. When running Tarjan's algorithm on the region, with player Even's vertices constricted to the strategy, we find the SCC $\{\mathbf{6}, \mathbf{4} \to \mathbf{6}\}$. This is a tangle with a single escape from 6 to 8. Thus the tangle would be attracted to region 8 in the next iteration. The next highest vertex is 5. We attract 3 to this region. The region is closed, as it is the lowest region in the game. We learn the tangle $\{\mathbf{5}, \mathbf{3} \to \mathbf{5}\}$, which will be attracted to region 9.

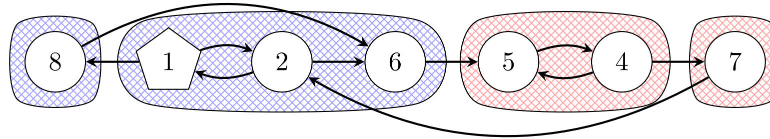
Iteration 2 We decompose the game again, and we now have the two tangles $\{\mathbf{6}, \mathbf{4} \to \mathbf{6}\}$ and $\{\mathbf{5}, \mathbf{3} \to \mathbf{5}\}$. See Figure 1c. Due to tangle $\{\mathbf{5}, \mathbf{3} \to \mathbf{5}\}$, vertices 8 and 6 are now attracted to region 9. Apparently playing towards 8 is not productive for player Even. Both regions 9 and 4 are open, but the lowest region 2 is closed and here we find a new tangle $\{\mathbf{2}, \mathbf{1} \to \mathbf{2}\}$.

Iteration 3 We decompose the game again. See Figure 1d. Now the lowest region 4 is closed and the tangle $\{\mathbf{4} \to \mathbf{2}, \mathbf{2}, \mathbf{1} \to \mathbf{2}\}$ is closed in the entire game, i.e., it is a dominion. We can attract from the rest of the game toward the dominion and now find that the entire game is won by player Even as in Figure 1e.

In this example, vertex 8 is a **distraction** for player Even. Even wins vertex 8, but accomplishes this by avoiding to play towards it. Ultimately, vertex 9 is a distraction for player Odd.

4 Recursive Tangle Learning

We can extend the basic tangle learning algorithm presented above with recursive decomposition of open regions [9]. The core idea is that when a region in tangle learning is open, there may still be a tangle inside the region that is distracted by the top vertex. See the following example:



In this example, region 6 is open but contains the tangle $\{\mathbf{1}, \mathbf{2} \to \mathbf{1}\}$ which would be attracted to region 8. Standard tangle learning would only be able to find this tangle after vertex 6 is attracted to region 7 after learning the tangle $\{\mathbf{4}, \mathbf{5}\}$.

Whenever a region is open, we first compute the attractor for the other player to the top vertex, with escapes restricted to the open region. For example, if the open region is of the Even player, then we compute the tangle attractor for Odd to the top vertex, attracting any odd vertex from the region that can play to the top vertex, and any even vertex that cannot stay in the region. Any vertices that can stay in the region thus avoid the distraction.

We then recursively decompose the remainder of the open region, learn tangles from closed regions, further recursively decompose open regions, etc.

We can even run a variation of this algorithm that only considers vertices with an even or with an odd priority, since this mechanism is sufficient to solve parity games, without explicitly attracting distractions. In the following, we call this one-sided version `ortl`.

Example 2. We illustrate how `ortl` works using the example game in Figure 2. We show the algorithm for player Even and for player Odd.

Iteration even-1. We decompose the game with only even-priority vertices as targets. See Figure 2a. We do not attract vertex 9 to region 6, as we only attract vertices and tangles with at most priority 6. This time, we only learn

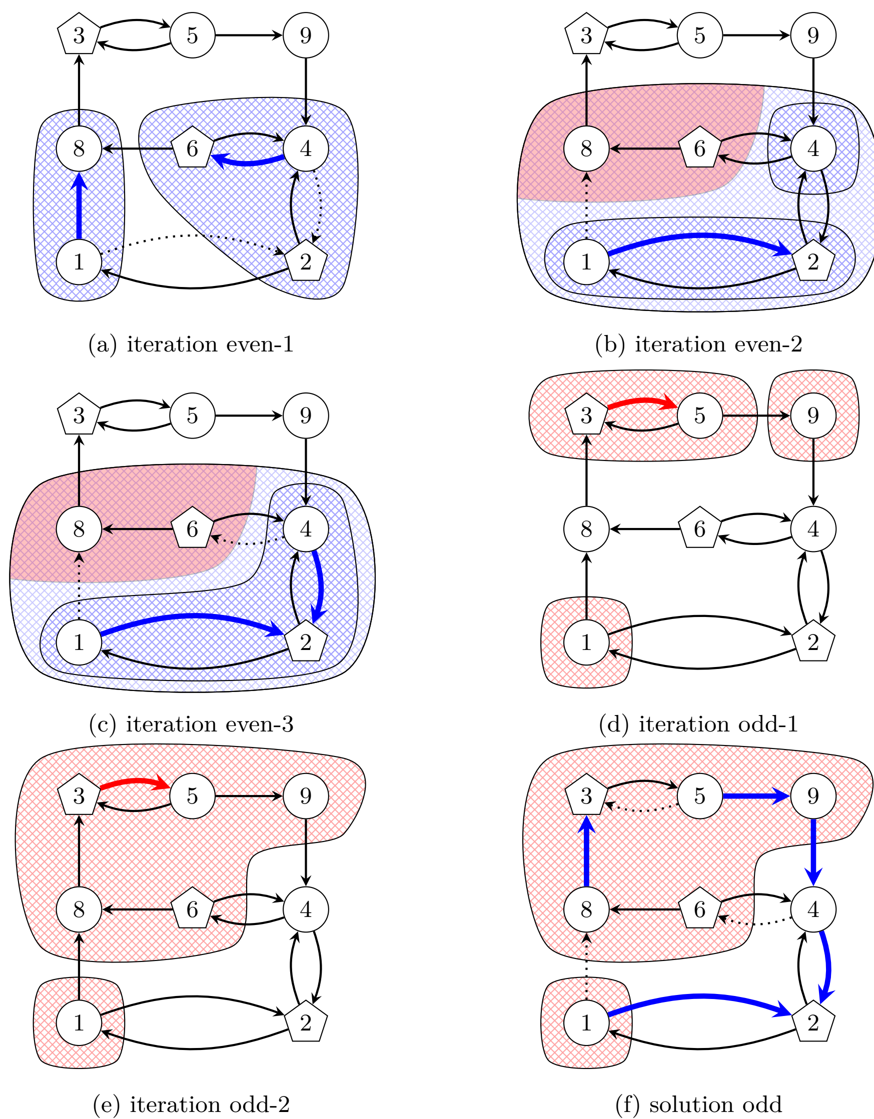


Fig. 2: Solving an example game with one-sided recursive tangle learning.

tangle $\{\mathbf{6}, \mathbf{4} \to \mathbf{6}\}$. Region 8 is open, and the attractor for player Odd to 8 includes vertex 1, which has no escapes inside region 8.

Iteration even-2. See Figure 2b. Now the new tangle is attracted to region 8 and there is no tangle for player Odd to neutralize the distraction, as would be the case with normal tangle learning. Since region 8 is open, we attract for player Odd to the open top vertex 8. Vertex 4 can still stay in the region by playing towards 2 and vertex 1 can do the same. Thus, we recursively decompose the subgame of vertices 4, 2 and 1. Subregion 2 is closed and we learn the tangle $\{\mathbf{2}, \mathbf{1} \to \mathbf{2}\}$.

Iteration even-3. See Figure 2c. As before, but now subregion 4 is locally closed and we obtain a dominion. Maximizing the dominion results in the entire game won by player Even.

Iteration odd-1. We attract to vertices with an odd priority. See Figure 2d. Region 5 is closed and we obtain tangle $\{\mathbf{5}, \mathbf{3} \to \mathbf{5}\}$.

Iteration odd-2. Region 9 is now formed as before and is open. See Figure 2e. If we attract for player Even towards the open top vertex 9, all vertices are attracted, since player Odd cannot avoid top vertex 9 while staying inside the region. No new tangles are learned, so we are done. The winning region of player Odd is empty, and the strategy for player Even is obtained by removing all edges from Even-controlled vertices that were avoided in the attractors of the Odd player and by choosing the Even-attractor strategy towards open top vertices of player Odd: the edge $1 \to 8$ is discarded, the edge $4 \to 6$ is discarded, and the edges $5 \to 9$ and $8 \to 3$ are selected. See Figure 2f for the result.

5 Distractions

Informally, a distraction is a vertex that at first appears to be good for a player, but which must be avoided for one of two reasons: either all plays to the vertex eventually reach a higher priority vertex of the opponent's parity, or all plays to the vertex eventually reach a winning region of the opponent. That is, a vertex v is a distraction if there exist tangles won by player $\overline{\alpha}$ such that

- v is attracted to a region of player $\overline{\alpha}$, or
- v is attracted to a dominion of player $\overline{\alpha}$.

If a vertex is a distraction, that does not mean that the opponent wins the vertex. In the example at the beginning of Section 4, vertex 6 was a distraction, but was still won by player Even.

Distractions delay parity game solvers since conclusions in earlier steps of the algorithm can depend on the distracting vertex being won by the player. When this assumption turns out to be false, all work based on this assumption is invalid. For example, a tangle learning algorithm might have learned many tangles with escapes to the distraction. If playing to the distraction is actually good for the opponent, then these tangles are all useless. If distractions repeatedly become distracting again, solvers can be delayed up to exponential lower bounds.

We say that a distraction v *distracts a tangle*, if that tangle contains some *distracted vertex* that is attracted to v, but that can also avoid playing to v and instead play inside the tangle. Vertex v typically has a higher priority than the top vertex of the tangle. In order to learn the distracted tangle, the solver first needs to "remove" the distraction. As argued in [6], tangle learning removes distractions by learning the opponent's tangle that attracts the distraction.

Zielonka's recursive algorithm and priority promotion also rely on this mechanism to remove distractions. This is most explicit in Zielonka's recursive algorithm as described in [7], as the second recursive call is only done if the opponent attracts from the current player's region, which is precisely when there is a distraction. Thus, distractions make the recursive algorithm slow, especially when the game is such that lower distractions must be removed in both recursive subgames of a higher distraction, i.e., in the first subgame to remove the higher distraction and in the second subgame after the distraction has been removed. If there are no distractions in the game, then a second recursive subgame is never solved and the recursive algorithm would run in at most n recursive calls.

The recursive tangle learning algorithm uses a different mechanism to disarm distractions. Vertices distract in tangle learning by being top vertices of open regions, with the distracted tangles (partially) inside these regions. If part of a tangle is distracted by a higher vertex, then there exists tangles that attract the rest of the distracted tangle to the distraction. After this happens, the distracted tangle can be found by avoiding the distraction using the recursive strategy. Thus, instead of learning the tangle that attracts the distraction, one-sided recursive tangle learning can disarm distractions by learning tangles that extend the region of the distraction until the distracted tangle is entirely inside, and then learn the distracted tangle.

Every parity game algorithm has some notion of distractions and must therefore have some method of eventually avoiding distractions. Algorithms in the *progress measures* family solve parity games by equipping every vertex with a monotonically increasing value which represents a preference ordering between vertices. If we can force a play from some vertex v with an even priority p to some vertex u with some even priority q via a path where all odd vertices are below $\max(p, q)$, then we assign v a value that is strictly higher than the value of u. Thus, the values of the vertices along good paths eventually overtake those of the distractions, since distractions must either eventually reach higher odd priority vertices or some odd dominion, curtailing their ability to keep increasing in value. This is how all algorithms in the progress measures family avoid distractions.

The recursive tangle learning algorithm shares some intuition of this mechanism of progress measures algorithms. Initially, the algorithm tries to reach the distracting top vertex. Then the algorithm discards that top vertex and instead attracts to the vertices inside the region, preferring to play to these vertices rather than the original top vertex. Thus, the algorithm explores the decomposition when these vertices would have a higher value than the original top vertex.

As another example, consider the game in Figure 3. Vertices **a** and **f** seem appealing to player Even and player Odd respectively. However, if player Even

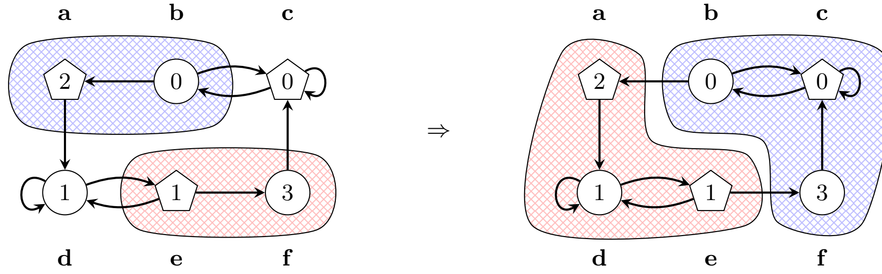


Fig. 3: A game with distractions for both players.

plays from **b** to **a**, then player Odd has a strategy to win the whole game. Similarly, if player Odd plays from **e** to **f**, then player Even has a strategy to win the whole game. A tangle learning algorithm can either learn the tangle $\{\mathbf{d}, \mathbf{e} \to \mathbf{d}\}$ and attract **a** to region **f**, or it can learn the tangle $\{\mathbf{c}\}$ which is attracted to region **a**, and then learn the distracted tangle $\{\mathbf{b} \to \mathbf{c}, \mathbf{c}\}$ in the recursive decomposition.

6 Slowly Solving Parity Games

It is not trivial to find examples of parity games that can delay parity game algorithms to their worst-case complexity.

For some classes of parity games this is easier than for other classes. For progress measures algorithms, simple cycles are already sufficient, as the algorithm slowly lifts the values of vertices until they reach the highest value. It is already more involved with strategy improvement algorithms, as evidenced by a whole range of difficult parity games for strategy improvement algorithms, recent examples being [12] and [10].

Difficult games for strategy improvement algorithms are typically trivial for algorithms that use attractor computation, as the difficult games for strategy improvement rely on the fact that improvements along paths occur step by step, since usually only direct successors are considered, instead of entire paths or sets of vertices as with attractor based algorithms.

For tangle learning algorithms specifically, we have found that a good counterexample must fulfill a number of requirements. The following list is not exhaustive:

1. The game should be a single strongly connected component, since SCC decomposition can be applied to solve the parity game one bottom SCC at a time.
2. It should take exponentially long to learn the next dominion for either player. Usually a difficult game has one dominion for each player.

3. There must be distractions that are detected and then reset to become distractions again exponentially often. That is, if distractions, after being found out, do not become distracting again, or only polynomially often, then the tangle learning algorithm would solve the game in polynomial time.
4. It should take exponentially long to learn the next tangle to the highest region in the game. If this always takes polynomial time, then an algorithm could simply maximize the highest region in polynomial time, and then recursively solve inside the region.
5. It should take exponentially long to learn the tangle that attracts the highest distraction in the game. Fairly obviously, if this always takes polynomial time, then all distractions are removed after polynomially many steps, starting with the highest distraction, and then the next highest distraction, etc.
6. From these points it follows that the structure must be such that after a high priority distraction is avoided/attracted, lower priority distractions should somehow be reactivated. There are multiple ways to do this, which is demonstrated with the difficult games below.
7. Some variations of tangle learning (not necessarily known in the literature) have various heuristics to determine whether a vertex is likely good or bad. They invariably either make this decision too quickly, leading to slow games where good vertices are exponentially often declared bad, or they make this decision too slowly, leading to slow games where bad vertices are exponentially often declared good. There are also variations that prefer “good paths” where player Even (or Odd) dominates the path often, and in that case we simply have a distractingly nice long path that eventually leads to a difficult to untangle higher region of the opponent, while the dominion is small and uninteresting, and always the last choice of the algorithm. If an algorithm is designed to simply ignore the highest vertices in the game, assuming that they are distractions, then a difficult game would merely be one where those vertices are not distractions.

We can see some of these principles at work in the counterexample for the `ortl` algorithm, the one-sided recursive tangle learning algorithm, which is unable to use opponent attractors to remove distractions. See Figure 4.

We have two versions: one with and one without the vertices 18 and 19. The mechanism is the same; the difference is that the game is an Even dominion without vertices 18 and 19, and an Odd dominion with vertices 18 and 19. The idea is that whenever a distraction is “defeated” by learning a tangle that escapes the distraction, that distraction is attracted to the higher region and becomes distracting again.

1. In the first iteration, we learn tangle $\{\mathbf{8} \to \mathbf{0}, \mathbf{0}\}$, which will be attracted towards 12 next. This will also attract 11 and 10.
2. In the second iteration, the recursive decomposition of region 12 contains vertices $12, 6, 8, 0, 11, 10$. We have subregions 12-10 (containing 10, 8) and 12-6 (containing 6, 0). We learn tangle $\{\mathbf{6} \to \mathbf{0}, \mathbf{0}\}$ in the subregion 12-6.
3. In the third iteration, we learn tangle $\{\mathbf{8} \to \mathbf{0}, \mathbf{6} \to \mathbf{0}, \mathbf{0}\}$ in the subregion 12-10-8. This tangle attracts towards region 14.

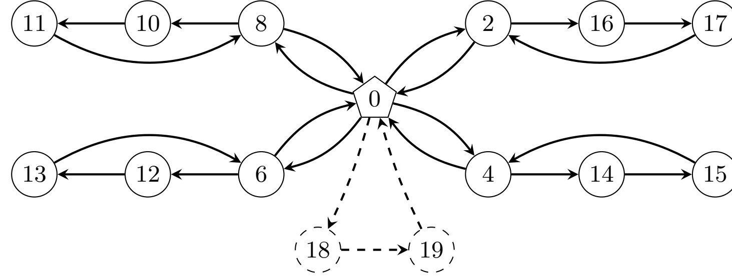


Fig. 4: A exponentially hard parity game for player Even in `ortl`. Vertices 18 and 19 are optional and convert the game from an Even dominion to an Odd dominion.

4. In the fourth iteration, region 14 has vertices $4, 8, 6, 0, 13, 12, 11, 10$. Vertices 10 and 12 are distracting again. We learn tangle $\{\mathbf{4} \rightarrow \mathbf{0}, \mathbf{0}\}$ in subregion 14-4.
5. In the fifth iteration, we learn $\{\mathbf{8} \rightarrow \mathbf{0}, \mathbf{4} \rightarrow \mathbf{0}, \mathbf{0}\}$ in subregion 14-10-8; then $\{\mathbf{6} \rightarrow \mathbf{0}, \mathbf{4} \rightarrow \mathbf{0}, \mathbf{0}\}$ in 14-12-6 in the sixth iteration and $\{\mathbf{8} \rightarrow \mathbf{0}, \mathbf{6} \rightarrow \mathbf{0}, \mathbf{4} \rightarrow \mathbf{0}, \mathbf{0}\}$ in 14-12-10-8 in the seventh iteration.
6. Similarly, we learn tangles in region 16 in iterations 8–15: $\{\mathbf{2}, \mathbf{0}\}$, $\{\mathbf{8}, \mathbf{2}, \mathbf{0}\}$, $\{\mathbf{6}, \mathbf{2}, \mathbf{0}\}$, $\{\mathbf{8}, \mathbf{6}, \mathbf{2}, \mathbf{0}\}$, $\{\mathbf{4}, \mathbf{2}, \mathbf{0}\}$, $\{\mathbf{8}, \mathbf{4}, \mathbf{2}, \mathbf{0}\}$, $\{\mathbf{6}, \mathbf{4}, \mathbf{2}, \mathbf{0}\}$, $\{\mathbf{8}, \mathbf{6}, \mathbf{4}, \mathbf{2}, \mathbf{0}\}$.

As is clear from the above example, the algorithm is tricked to play towards the same distractions over and over again. However, if we let our tangle learning algorithm attract to regions of the opponent, that is, vertices 11, 13, 15 and 17, the parity game is quickly solved.

To delay tangle learning algorithms that use attraction to the opponent's regions, such as `tl` and `rtl`, one needs a different structure. In [8], we presented a parity game called Two Counters which accomplishes this. A Two Counters game with parameter N has $2N^2 + 5N$ vertices, $2N$ distractions, and requires $2 \times (2^N - 1)$ iterations to solve. See Figure 5. There are three distractions for each player: vertices 3, 5, 7 distract player Odd and vertices 4, 6, 8 distract player Even. The rectangles represent bits of a binary counter; three bits for each player. A bit is *set* when a tangle is learned that attracts 3 to 10, 4 to 11, 5 to 12, etc.

Consider the highest bit of player Odd, which has vertices 8 and 15. This bit will be set when the tangle that attracts 8 to 15 is learned. This tangle is distracted via the solid red lines by vertices 3, 5, 7. Player Odd prefers to play towards those vertices rather than vertex 1, until the distractions are attracted by player Even. This happens when all three bits of player Even are set.

Initially, all 6 distractions are distracting and only the lowest Even bit is learned, as it is not distracted. This neutralizes distraction 3. In the second iteration, the lowest bit for Odd can be learned. This neutralizes distraction 4. In the third iteration, the second bit for Even is learned, which was only distracted

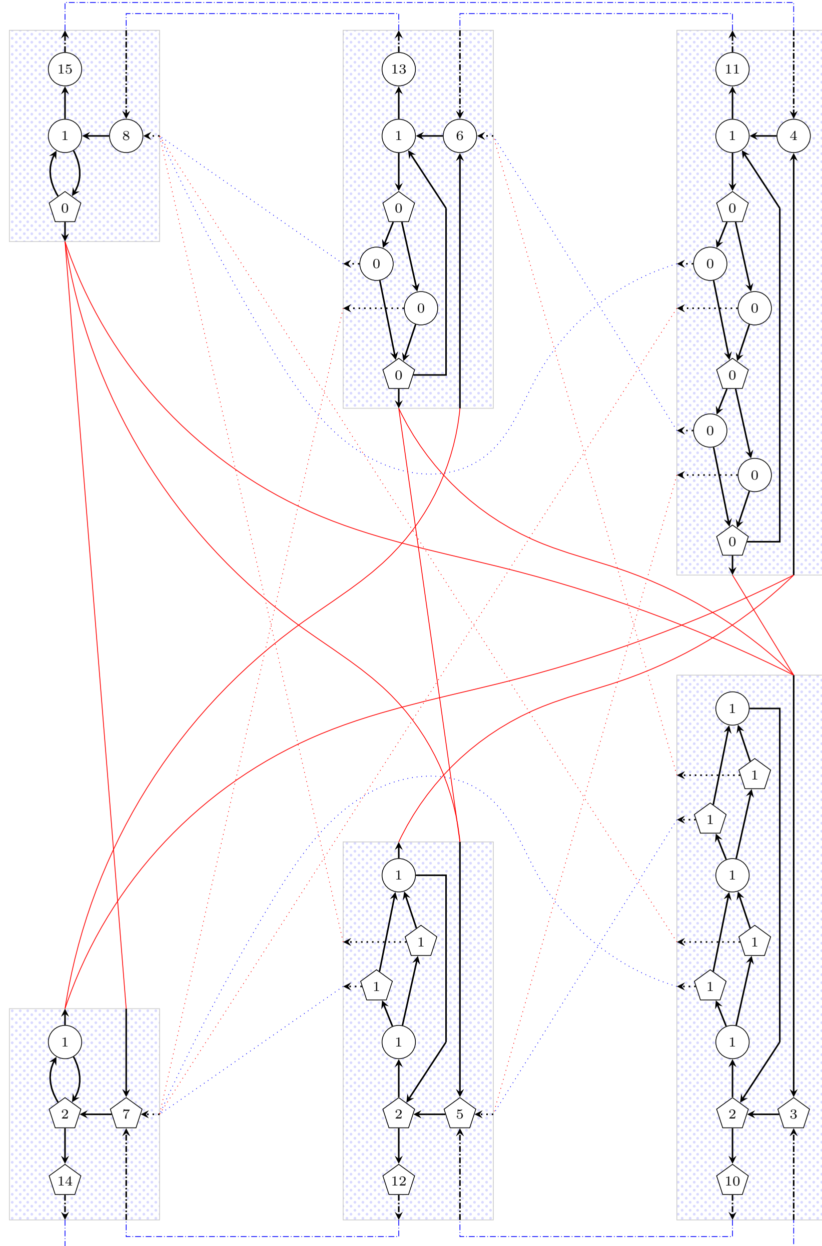


Fig. 5: The 3-bit Two Counters game [8].

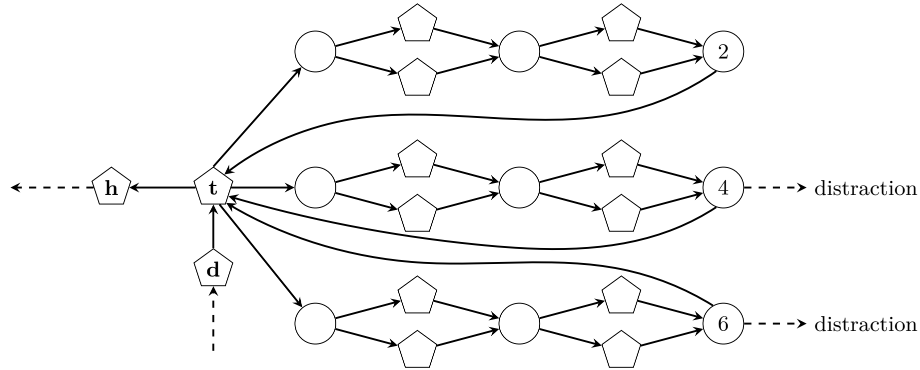


Fig. 6: Bit 2 of a 4-bit Two Counters game that delays `rtl` exponentially.

by 4. This neutralizes distraction 5. As a consequence, the tangle that attracted 4 is no longer attracted to 11. In the fourth iteration, the second bit for Odd is learned, which was only distracted by 3 and 5. This neutralizes distraction 6. As a consequence, the tangle that attracted 3 is no longer attracted to 10. So after these steps, distractions 3 and 4 are distracting again.

The Two Counters game of Figure 5 works for all parity game algorithms that neutralize distractions by attracting them to a region or dominion of the opponent. This includes all state-of-the-art algorithms for practical games: DFI, FPJ, priority promotion, tangle learning and Zielonka's recursive algorithm. They all purely rely on this mechanism and require exponential time to solve the Two Counters games. Various other algorithms, such as those computing progress measures, also require worst case time, but this is due to the tangles and not due to the mechanism to avoid distractions.

The `rtl` algorithm solves both games of Figure 4 and Figure 5 in polynomial time. To delay `rtl` exponentially often, we need to address both mechanisms. This can be accomplished by a hybrid version of both parity games. The global structure of the game is similar to Figure 5. Every distraction is designed to be attracted by a tangle of the opponent that requires exponentially many steps to be found by the recursive mechanism, and that simultaneously becomes unattractable when higher bits of the other counter are set. The former is accomplished by the global structure of Figure 5, the latter by the "selection chains" that give rise to exponentially many tangles depending on the configuration of higher bits of both players. On the game of Figure 5, both `ortl` and `rtl` would immediately ignore the distraction and set the highest bit, as only one vertex is distracted by several distractions. In Figure 4, each distraction distracts a different vertex. We can use this design to modify the bits of the Two Counters game to require many iterations to solve. See Figure 6 for an example. For presentation, we omit the outgoing edges from the Odd-controlled vertices; they go to the higher bits like in Figure 5; the three chains are copies.

7 Discussion

In this work, we have presented an overview of a few major tangle learning variations. We have explained the principles by which these tangle learning algorithms disarm or avoid distractions. We have seen that these algorithms can be delayed by suitable parity game designs and have discussed some of the underlying requirements for a parity game to be challenging.

In our current research, we study a number of different variations of tangle learning. For example, we can use progress measures instead of priorities to partition the parity game, essentially as if we are using tangle learning to accelerate progress measures. This way, tangle learning can run in quasipolynomial time.

We are considering different heuristics for disabling likely distractions, in particular we study how recent quasi-polynomial versions of Zielonka's algorithm, such as [18] and [2], can ignore distractions. We also have an alternative method to ignore distractions when the lowest region of a player is open, and are studying a method to delay this algorithm. It currently solves all known difficult parity games in polynomial time. Considering the effort required to find the difficult games for tangle learning and recursive tangle learning, finding a difficult parity game example for new variations is sometimes like finding a needle in a haystack.

Nevertheless, playing in haystacks and designing suitable needles is fun! There is always a chance to improve our understanding of what makes parity games difficult to solve, or to find promising mechanisms to defeat certain distractions. Our current understanding is that the nature of nested tangles interleaved with distractions makes parity games hard to solve. It may well be possible that there are in fact no polynomial time algorithms after all! The holy grail may be mere fiction, a mirage in a desert, an imaginary dot on the horizon that we may never reach. Perhaps in time, we will know the answer to this long standing open problem. Until then, we continue solving parity games, sometimes very slowly.

References

1. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
2. Benerecetti, M., Dell'Erba, D., Mogavero, F., Schewe, S., Wojtczak, D.: Priority promotion with Parysian flair. CoRR **abs/2105.01738** (2021)
3. Calude, C.S., Jain, S., Khoussainov, B., Li, W., Stephan, F.: Deciding parity games in quasipolynomial time. In: STOC. pp. 252–263. ACM (2017)
4. Chatterjee, K., Fijalkow, N.: A reduction from parity games to simple stochastic games. In: GandALF. EPTCS, vol. 54, pp. 74–86 (2011)
5. Condon, A.: The complexity of stochastic games. Inf. Comput. **96**(2), 203–224 (1992)
6. van Dijk, T.: Attracting tangles to solve parity games. In: CAV (2). LNCS, vol. 10982, pp. 198–215. Springer (2018)
7. van Dijk, T.: Oink: An implementation and evaluation of modern parity game solvers. In: TACAS (1). LNCS, vol. 10805, pp. 291–308. Springer (2018)
8. van Dijk, T.: A parity game tale of two counters. In: GandALF. EPTCS, vol. 305, pp. 107–122 (2019)
9. van Dijk, T.: Avoiding distractions in parity games (2024), accepted.

10. van Dijk, T., Loho, G., Maat, M.T.: The worst-case complexity of symmetric strategy improvement. In: CSL. LIPIcs, vol. 288, pp. 24:1–24:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2024)
11. van Dijk, T., Rubbens, B.: Simple fixpoint iteration to solve parity games. In: GandALF. EPTCS, vol. 305, pp. 123–139 (2019)
12. Disser, Y., Friedmann, O., Hopp, A.V.: An exponential lower bound for Zadeh's pivot rule. Math. Program. **199**(1), 865–936 (2023)
13. Emerson, E.A., Jutla, C.S.: Tree automata, mu-calculus and determinacy (extended abstract). In: FOCS. pp. 368–377. IEEE Computer Society (1991)
14. Fearnley, J., Jain, S., Schewe, S., Stephan, F., Wojtczak, D.: An ordered approach to solving parity games in quasi polynomial time and quasi linear space. In: SPIN. pp. 112–121. ACM (2017)
15. Halman, N.: Simple stochastic games, parity games, mean payoff games and discounted payoff games are all LP-type problems. Algorithmica **49**(1), 37–50 (2007)
16. Jurdzinski, M.: Small progress measures for solving parity games. In: STACS. LNCS, vol. 1770, pp. 290–301. Springer (2000)
17. Jurdzinski, M., Lazic, R.: Succinct progress measures for solving parity games. In: LICS. pp. 1–9. IEEE Computer Society (2017)
18. Lehtinen, K., Parys, P., Schewe, S., Wojtczak, D.: A recursive approach to solving parity games in quasipolynomial time. Log. Methods Comput. Sci. **18**(1) (2022)
19. Parys, P.: Parity games: Zielonka's algorithm in quasi-polynomial time. In: MFCS. LIPIcs, vol. 138, pp. 10:1–10:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)