Avoiding distractions in parity games

Tom van Dijk

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

Abstract. We consider algorithms for parity games that use attractor decomposition, such as Zielonka's recursive algorithm, priority promotion, and tangle learning. In earlier work, we identified the Two Counters parity game family that requires exponential time for many algorithms, including attractor decomposition algorithms, and we identified the main mechanism that slows down parity game algorithms as so-called distractions. We observe a fundamentally different approach in avoiding distractions between algorithms that use attractor decomposition and algorithms that compute progress measures. We now propose an alternative attractor-based method to avoid distractions by applying the attractor decomposition recursively. We demonstrate that this algorithm solves the Two Counters games efficiently, but that a modification of the Two Counters method can also delay the recursive algorithm exponentially.

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, 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 game 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.

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 [5,6,16]. 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 [4,24]. A proof that parity games do not admit a polynomial-time solution would imply P ≠ NP.

Parity games also play a central role in several popular domains in theoretical computer science. Several problems in formal verification and synthesis can be reduced to the problem of solving parity games, as parity games are as expressive as the modal μ-calculus, capturing propositional logic with nested least and

greatest fixed point operators. We highlight in particular their application to the synthesis problem of linear temporal logic (LTL). In recent years, the SYNTCOMP competition track of LTL synthesis [17] has been won repeatedly by tools that translate a specification to a parity game and solve the parity game in order to produce a controller, such as Strix [23] and ltlsynt [25].

In this work, practical performance is not our primary concern. A number of current algorithms, including DFI [13], FPJ [22], priority promotion [2], and tangle learning [8] already solve practical parity games of millions of vertices in mere seconds [9,13], while generating these games requires several orders of magnitude more time. These practical parity games have very few distinct priorities relative to the size of the game. This is not surprising as many popular logics such as LTL and CTL* can be captured by μ-calculus formulae of at most two alternation levels [3,7]. Furthermore, what makes parity games difficult for algorithms like tangle learning is the presence of distractions that repeatedly "mislead" the solver, as discussed below. Parity games arising from practical applications do not appear to have these features, as even algorithms that are highly sensitive to them, like DFI and FPJ, are among the fastest solvers.

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 [8]. Tangles are roughly described as strongly connected 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 [8]. The algorithms proposed in [8] 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 [8,10,13]. 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 family [10], which is an exponential lower bound for many of the parity game algorithms implemented in Oink [9], such as Zielonka's recursive algorithm, priority promotion, tangle learning, the fixed point algorithms DFI and FPJ, and small progress measures [18]. This family also slows down the quasi-polynomial time progress measures algorithms [15,19] 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.

Since different algorithms use different methods to deal with the distractions, one of our aims is to design algorithms that combine different methods, so that the algorithm is less "vulnerable" to certain distractions. The eventual goal is of course to find an algorithm that runs in polynomial-time, or improve our understanding of how this is difficult if not impossible to achieve.

Contributions We observe that algorithms based on attractor decomposition, such as Zielonka's recursive algorithm, priority promotion, and tangle learning, recognize distractions by attracting distractions to regions of the opponent. Other parity game algorithms, such as those based on computing progress measures, use a fundamentally different mechanism to avoid distractions. Our main contribution is to extend tangle learning to avoid distractions in a way similar to how these other parity game algorithms avoid distractions. After applying attractor decomposition once to the parity game, we continue applying attractor decomposition *recursively*, further decomposing the game into ever smaller regions. We propose to call this algorithm **recursive tangle learning**. We propose two versions of this algorithm: one-sided recursive tangle learning uses only this alternative method to avoid distractions; whereas standard recursive tangle learning combines the two methods (by attracting and by recursion). We also show that the two new algorithms can both be tricked to run in exponential time, by designing parity games where distractions become distracting again.

2 Preliminaries

We formally define a parity game $\mathcal{D}$ as a tuple $(V_\bigcirc, V_\pentagon, E, \mathsf{pr})$ where $V = V_\bigcirc \cup V_\pentagon$ 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. 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 $\alpha \in \{\bigcirc, \pentagon\}$ to denote player $\bigcirc$ 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 $\mathsf{pr}(\mathcal{D})$ for the largest priority in $\mathcal{D}$. Furthermore, we write $\mathsf{pr}^{-1}(p)$ for all vertices with the priority p. With $\mathsf{pr}^{-1}(\bigcirc)$ and $\mathsf{pr}^{-1}(\pentagon)$ 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 $\pentagon$ if p is odd.

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 successive vertices. We denote with $\inf(\pi)$ the 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* $\sigma : U \to U$ ($U \subseteq V$) 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 when the domain is restricted to U_α. We write $\text{Plays}(v)$ for the set of plays starting at vertex v. We write $\text{Plays}(v, \sigma)$ for all plays from v consistent with σ, and $\text{Plays}(V, \sigma)$ for $\{\pi \in \text{Plays}(v, \sigma) \mid v \in V\}$.

A basic result for parity games is that they are memoryless determined [14], i.e., each vertex is either winning for player Even or for player Odd, and both players have a strategy for their winning vertices. Player α wins vertex v if they have a strategy σ such that every $\pi \in \text{Plays}(v, \sigma)$ 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 $\partial[\sigma]$, 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 player α has a strategy to stay in D and win all plays in D and D is α-closed.

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 vertices A, the attractor to A for player α represents those vertices from which player α can force a visit of A. We write $Attr_\alpha^\partial(A)$ 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 $A = Attr_\alpha^\partial(A)$. The attractor also yields an *attractor strategy* by selecting a vertex in Z for every α-vertex v when v is added to Z, and by selecting a vertex in Z for all α-vertices in A that can play to 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 $Attr_\alpha^U(A)$ restricted to a set of vertices U, 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.

Attractor decomposition Attractors are typically used to attract to a set $A := \mathsf{pr}^{-1}(\mathsf{pr}(\partial))$, 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

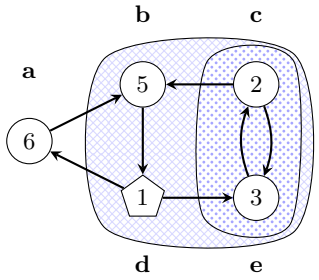


Fig. 1: A parity game with two tangles

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. By definition, only vertices in A can be open, as all vertices in $Z \setminus A$ are attracted.

Typically 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*.

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 $\partial[U, \sigma]$, and $\partial[U, \sigma]$ is strongly connected.

Tangles have some important properties. All vertices of the winner have a strategy to stay inside the tangle, while the opponent *must* escape; otherwise they lose. Due to the strongly connected nature of a tangle, the opponent is free to choose any escape. If a tangle is closed, i.e., the opponent cannot escape, 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 Figure 1. Player Odd wins the strategy $\{\mathbf{d} \to \mathbf{e}\}$. Player Even either loses in a cycle with highest priority 5 or in a cycle with highest priority 3. The dominion $\{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d} \to \mathbf{e}, \mathbf{e}\}$ (i.e., vertices $\{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}\}$ with ∂-strategy $\{\mathbf{d} \to \mathbf{e}\}$) contains the tangle $\{\mathbf{b}, \mathbf{c}, \mathbf{d} \to \mathbf{e}, \mathbf{e}\}$ and the tangle $\{\mathbf{c}, \mathbf{e}\}$.

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 $Attr_\alpha^\partial(A, T)$ to attract vertices in ∂ 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

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

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

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

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 $Attr_\alpha^{\supset}(A, T, \leq p)$ 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. This is important for the algorithms introduced in this paper.

3 Tangle learning

In earlier work, we presented the tangle learning algorithm [8]. We recall the basic algorithm here.

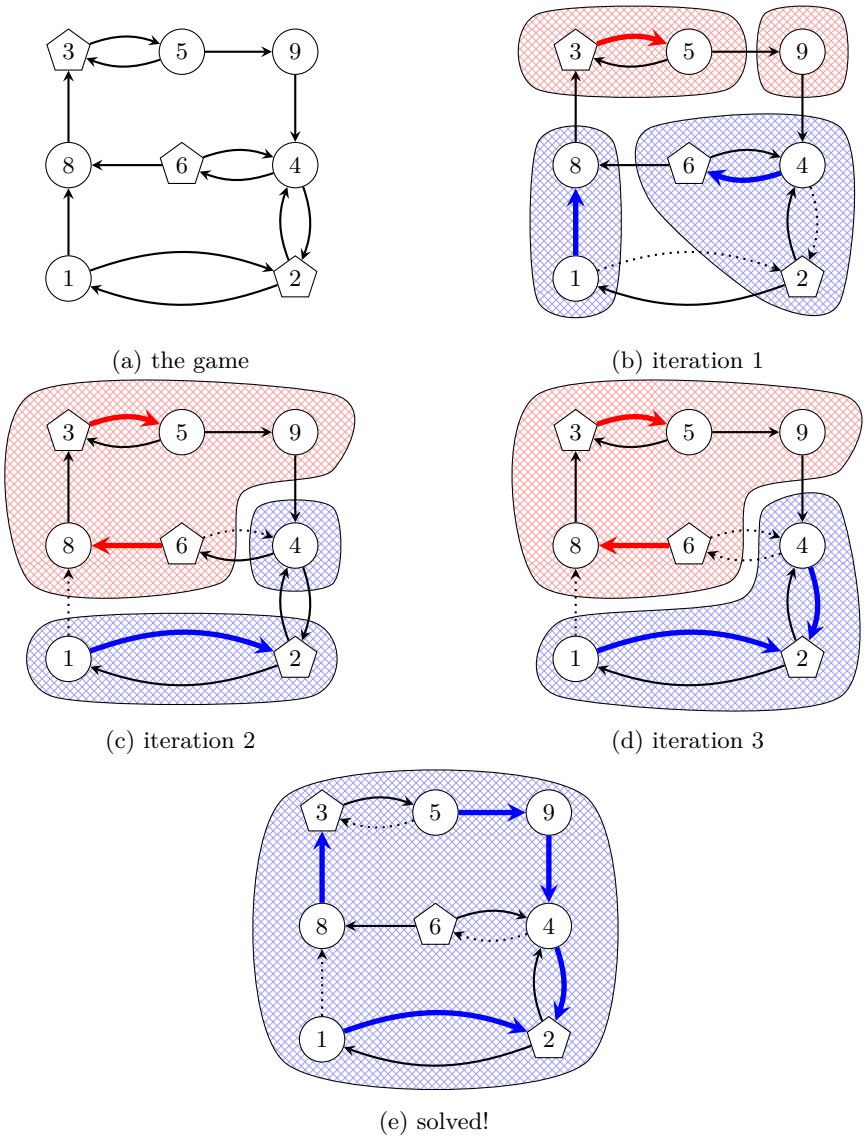
Core idea Several algorithms to solve parity games, in particular priority promotion and Zielonka's recursive algorithm, essentially decompose the parity game into regions using attractor computation. These regions are then refined in a systematic way, ultimately yielding a partition between the winning regions of both players. In [8], we introduced an algorithm that extracts from these regions precisely the information that allows attractor decomposition algorithms to refine the regions, namely the tangles.

By explicitly searching for these tangles, we find an algorithm that repeatedly computes a new attractor decomposition, using the obtained tangles to improve the decomposition. Each iteration of tangle learning decomposes the game into regions using the tangle attractor, starting with the highest priority. If a region is locally closed, i.e., player α can stay in the region and player $\overline{\alpha}$ cannot escape to the remaining subgame, then the region contains new tangles. The new tangles are obtained by finding the strongly connected components of the region, restricted to the attractor strategy of player α.

If a tangle has no escapes, then it is a dominion. When dominions are found, they are maximized with another attractor for player α on the rest of the game,

```
1  def search(R, T):
2      Y ← ∅
       /* while remaining set R is nonempty...                    */
3      while R ≠ ∅ :
           /* obtain highest priority and player                  */
4          p ← pr(R), α ← parity(pr(R))
           /* tangle-attract to highest-priority vertices         */
5          A ← {v ∈ R | pr(v) = p}
6          Z, σ ← Attr_α^R(A, T)
           /* compute open top vertices                           */
7          O ← {v ∈ A_α | E(v) ∩ Z = ∅} ∪ {v ∈ A_ᾱ | E(v) ∩ R ⊈ Z}
           /* compute tangles in Z if locally closed              */
8          if O = ∅ : Y ← Y ∪ sccs(Z, σ)
           /* remove region Z from R                              */
9          R ← R \ Z
10     return Y
11 def tl(↻):
       /* initialize sets                                         */
12     W_○ ← ∅, W_⬠ ← ∅, R ← V, T ← ∅
       /* until the entire game is solved...                      */
13     while R ≠ ∅ :
           /* run one iteration to search for tangles             */
14         Y ← search(R, T)
           /* add new open tangles to T                           */
15         T ← T ∪ {(U, σ) ∈ Y | E(U_ᾱ) ⊈ U}
           /* collect any new dominions                           */
16         D ← {(U, σ) ∈ Y | E(U_ᾱ) ⊆ U}
17         if D ≠ ∅ :
               /* maximize the dominions                          */
18             W_○ ← W_○ ∪ Attr_○^↻(⋃ D_○, T)
19             W_⬠ ← W_⬠ ∪ Attr_⬠^↻(⋃ D_⬠, T)
               /* compute the remaining game                      */
20             R ← R \ (W_○ ∪ W_⬠)
       /* return winning regions and strategies                   */
21     return W_○, W_⬠
```

Algorithm 1: The basic tangle learning algorithm [8].



(a) the game

(b) iteration 1

(c) iteration 2

(d) iteration 3

(e) solved!

Fig. 2: Solving an example game with standard tangle learning.

and removed from the game. This procedure is simply repeated until there is nothing left to solve.

See Algorithm 1. Each iteration of `search` decomposes the game into regions and new tangles are stored in the set Y. The `search` method is repeatedly called; after each call to `search`, we separate dominions from other tangles. If any dominion is found, it is maximized (line 18) and removed from the game. We omit from Algorithm 1 the computation of winning strategies, but they are easily obtained from the tangle strategy of the dominion and the attractor strategy when maximizing the dominion.

We prove that this algorithm solves parity games in [8]. 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.

Example We illustrate how the algorithm works using an example game in Figure 2. Recall that to determine whether a region is open, we only need to consider the top vertices.

Iteration 1. We decompose the game into regions for the first time, and do not yet know tangles. See Figure 2b. 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 2c. 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 2d. 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 2e.

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

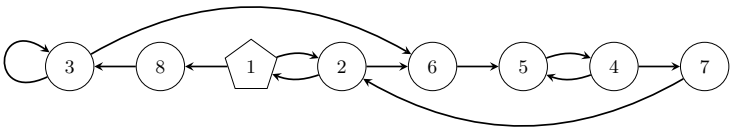


Fig. 3: Example of a game where the vertex with priority 6 is a distraction (do not play from 2 to 6) but the entire game is won by player Even.

Exponential lower bound: The Tale of Two Counters We found in the example of Figure 2 that vertex 8 was a distraction. 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 $\overline{\alpha}$.

Not every distraction is won by the opponent. See for example Figure 3. Also the distractions 8 and 6 in Figure 2 were eventually 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. If distractions repeatedly become distracting again, solvers can be delayed up to exponential lower bounds.

In [10], 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 4. 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 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

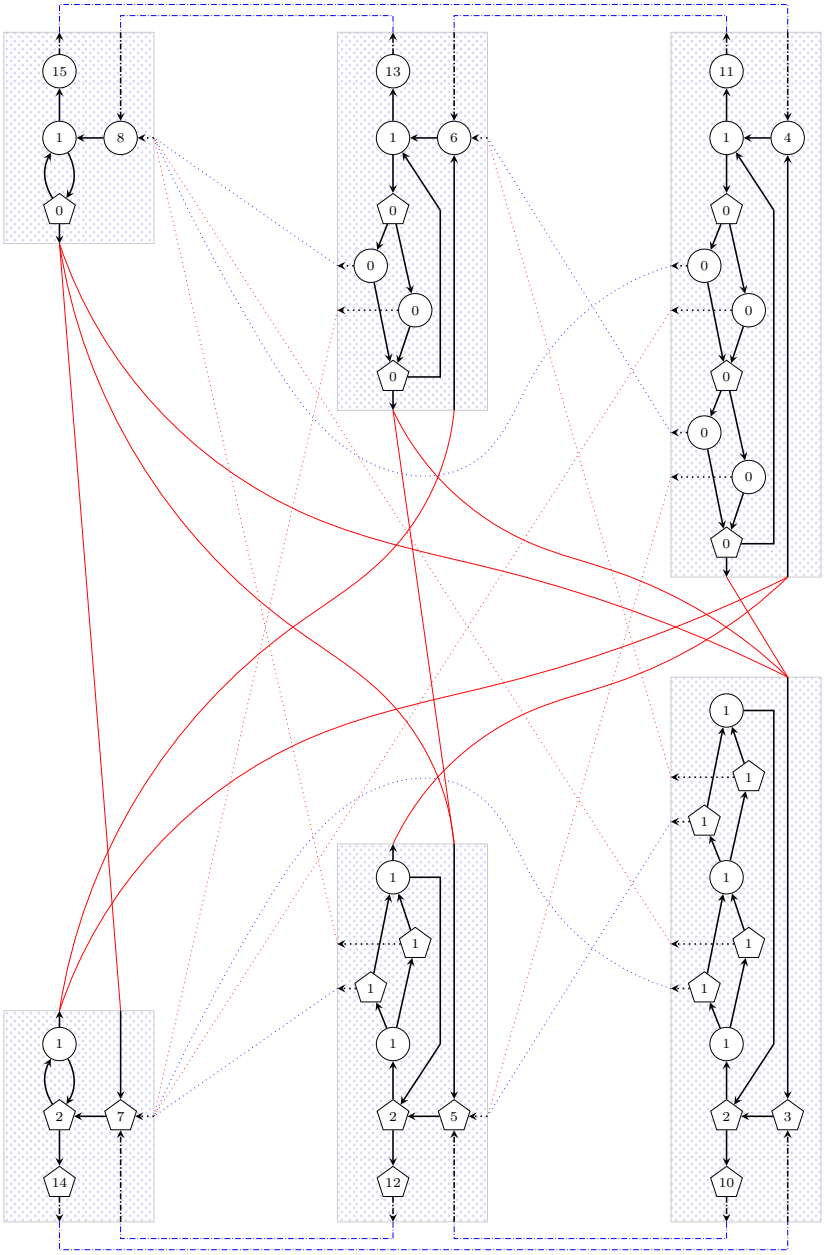


Fig. 4: The 3-bit Two Counters game [10].

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 4 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.

4 Recursive Tangle Learning

We now present a novel algorithm to solve parity games called **recursive tangle learning**.

Core idea Not all algorithms deal with distractions by attracting them to a region or dominion of the opponent. Most algorithms from the progress measures family [18] only view the parity game from the perspective of one of the players. These algorithms cannot neutralize distractions by attracting them to a region or dominion of the opponent. The small progress measures algorithm still requires exponential time to solve the Two Counters games, but this is due to repeatedly exploring the same tangle, not due to the distractions.

Instead, these algorithms slowly increase the values (measures) of vertices along good paths. That is, considering the even priorities, if some even-priority vertex u can reach a vertex v with some value a without encountering a vertex with a higher odd priority than both $\mathsf{pr}(u)$ and $\mathsf{pr}(v)$, then vertex u will obtain some value $b > a$. A distraction is limited in its ability to make progress, since the higher opponent-priority vertex or the opponent's dominion prevents the distraction from increasing in value. Thus, vertices that initially play towards the distraction eventually obtain a higher value and are preferred over the distraction.

This is a fundamentally different method; in fact the two methods are dual. The first method recognizes distractions because they are attracted to an opponent's region or dominion. This is how attractor-based algorithms reason. The second method avoids distractions because these vertices cannot increase in value. This is how algorithms computing progress measures reason.

A different perspective is that *distracted tangles* are "stuck" inside the regions of their distraction(s). For example, in the Two Counters game, the small $\{0, 1 \to 0\}$ tangle that attracts to 15 in the top left of Figure 4 is stuck either in region 3, in region 5, or in region 7, which are its distractions. Thus, rather than waiting until opponent tangles eventually "free" the distracted tangle by attracting these distractions, perhaps another technique could be used to find these tangles.

We now propose **recursive tangle learning**. Whenever a region is open, we assume that the open top vertices are distractions. We *avoid* these distractions by computing the *opponent's* attractor inside the region to the open top vertices.

```
def searchrec(U, R, T):
    Y ← ∅
    /* while top vertices remain...                                   */
    while R ∩ U ≠ ∅ :
        /* obtain highest priority and player                         */
        p ← pr(R ∩ U), α ← parity(pr(R ∩ U))
        /* tangle-attract to next top vertices                        */
        Z, σ ← Attr^R_α({v ∈ R ∩ U | pr(v) = p}, T, ≤p)
        /* compute open vertices                                      */
        O ← {v ∈ Z_α | E(v) ∩ Z = ∅} ∪ {v ∈ Z_ᾱ | E(v) ∩ R ⊈ Z}
        /* compute tangles in Z if locally closed                     */
        if O = ∅ : Y ← Y ∪ sccs(Z, σ)
        /* go recursive if not locally closed                         */
        else: Y ← Y ∪ searchrec(U, Z \ Attr^Z_α(O, T), T)
        /* remove region Z from R                                     */
        R ← R \ Z
    return Y
def rtl(ↀ):
    /* initialize sets                                                */
    W_○ ← ∅, W_⬠ ← ∅, R ← V, T ← ∅
    /* until the entire game is solved...                             */
    while R ≠ ∅ :
        /* run one iteration to search for tangles                    */
        Y ← searchrec(V,R,T)
        /* handle new tangles/dominions                               */
        T ← T ∪ {(U, σ) ∈ Y | E(U_ᾱ) ⊈ U}
        D ← {(U, σ) ∈ Y | E(U_ᾱ) ⊆ U}
        if D ≠ ∅ :
            W_○ ← W_○ ∪ Attr^ↀ_○(⋃ D_○, T)
            W_⬠ ← W_⬠ ∪ Attr^ↀ_⬠(⋃ D_⬠, T)
            R ← R \ (W_○ ∪ W_⬠)
    return W_○, W_⬠
def ortl(ↀ, α):
    /* initialize sets                                                */
    W_α ← ∅, R ← V, T ← ∅
    /* until no new tangles are found...                              */
    loop:
        /* run one iteration to search for tangles                    */
        Y ← searchrec(pr^{-1}(α),R,T)
        if Y = ∅ : return W_α
        /* handle new tangles/dominions                               */
        T ← T ∪ {(U, σ) ∈ Y | E(U_ᾱ) ⊈ U}
        D ← {(U, σ) ∈ Y | E(U_ᾱ) ⊆ U}
        if D ≠ ∅ :
            W_α ← W_α ∪ Attr^ↀ_α(⋃ D, T)
            R ← R \ W_α
    return W_α
```

Algorithm 2: The **recursive tangle learning** algorithm and the **one-sided** variant. The `searchrec` algorithm searches for new tangles given a set of target vertices U, a remaining subgame R, and a current set of tangles T.

Any vertices that can stay in the region thus avoid the distraction. We then recursively perform tangle learning inside the remaining region, i.e., we recursively decompose the open region. This is somewhat similar to how progress measures algorithms function: after playing towards some vertex, the α-priority vertices that can reach that vertex gain a higher value and will then be played towards and if there are tangles, the value rises until the opponent must play towards some other high value vertex. Thus, instead of playing to the original top vertex, player α actually prefers to play to α-priority vertices inside the region. This is mimicked by the recursive decomposition.

See Algorithm 2 for the algorithm. We introduce an extra parameter U to `searchrec` which is the set of potential target vertices for the decomposition into regions. We use this to instantiate different recursive tangle learning algorithms. Now line 3 checks if any potential target vertices remain in region R. If so, we find the next priority and player (line 4) and attract to the target vertices in R with that priority, but we only attract vertices and tangles with at most priority p (line 5). This ensures that if the region is locally closed, then the highest vertices are of priority p which is essential for the property that all plays in the region are won by player α. The other difference is that if the region is open, then we remove the $\overline{\alpha}$-attractor inside Z towards O and recursively search for tangles in the pruned region.

The `rtl` algorithm is almost exactly the same as `tl`, except we now use the `searchrec` method with all vertices as potential target vertices. Essentially the `rtl` algorithm uses both the recursion and opponent attraction mechanisms to recognize or avoid distractions.

The `ortl` algorithm implements **one-sided recursive tangle learning**. Given a player α, `ortl` invokes `searchrec` with only α-priority vertices. As a result, all regions have α-priority top vertices, and all tangles will be for player α, including dominions. The algorithm computes the winning region of player α, which also implies the winning region for player $\overline{\alpha}$ as $W_{\overline{\alpha}} := V \setminus W_\alpha$. A winning strategy for player $\overline{\alpha}$ can be computed as well. For vertices that are attracted to a remaining (open) region, the $\overline{\alpha}$-attractor strategy to the open top vertices is the winning strategy for player $\overline{\alpha}$. For vertices that are not attracted, any edge that avoids the regions of player α is a winning strategy for player $\overline{\alpha}$.

Correctness We sketch why the two algorithms correctly solve parity games.

Correctness of `rtl` is trivial, as it is the same as correctness of `tl`. The difference between `tl` and `rtl` is that in addition to the tangles learned in the primary decomposition, the recursive decomposition may lead to additional new tangles. Thus, `rtl` will still learn a new tangle every iteration, from the lowest region in the primary decomposition, and therefore eventually terminate by finding the dominions.

Correctness of `ortl` is slightly more involved. We can no longer argue that the lowest region is always closed and yields a new tangle. Instead, we consider where vertices are in the recursive decomposition. For example, a vertex could be attracted to region 10, subregion 6. We call this the *position* of a vertex. For example, position 10-6 is higher than 10-4, but lower than 12-2. The argument is

that if there exists a tangle for player α that causes its vertices to reach a higher region, then *some new tangle* will be learned.

- If all vertices of the tangle are attracted to its top vertex, with the only escapes to higher regions, then that tangle will be learned: in the recursive decomposition, vertices of that tangle cannot escape to the distractions or to lower regions.
- If not all vertices are attracted to the top vertices yet, then there must exist another *lower* tangle (lower priority) that can be learned. This lower tangle would also improve the position of some vertices.
- Therefore, if there exists a tangle that can improve the position of its vertices, then some new tangle will be learned.

This also holds for dominions: as long as there is a dominion for player α in the remaining game, new tangles will be learned. As a result, `ortl` will only terminate when all vertices are in their highest position and the entire winning region of player α is computed. This then also proves correctness of the winning strategy for player $\overline{\alpha}$. As long as player $\overline{\alpha}$ avoids α-attractors and chooses $\overline{\alpha}$-attractor strategies along the recursive decomposition, player α has no new tangles, i.e., no winning cycles, and therefore the remaining game is won by player $\overline{\alpha}$ using that strategy.

Example We illustrate how `ortl` works using the example game in Figure 5. We show the algorithm for player Even and for player Odd.

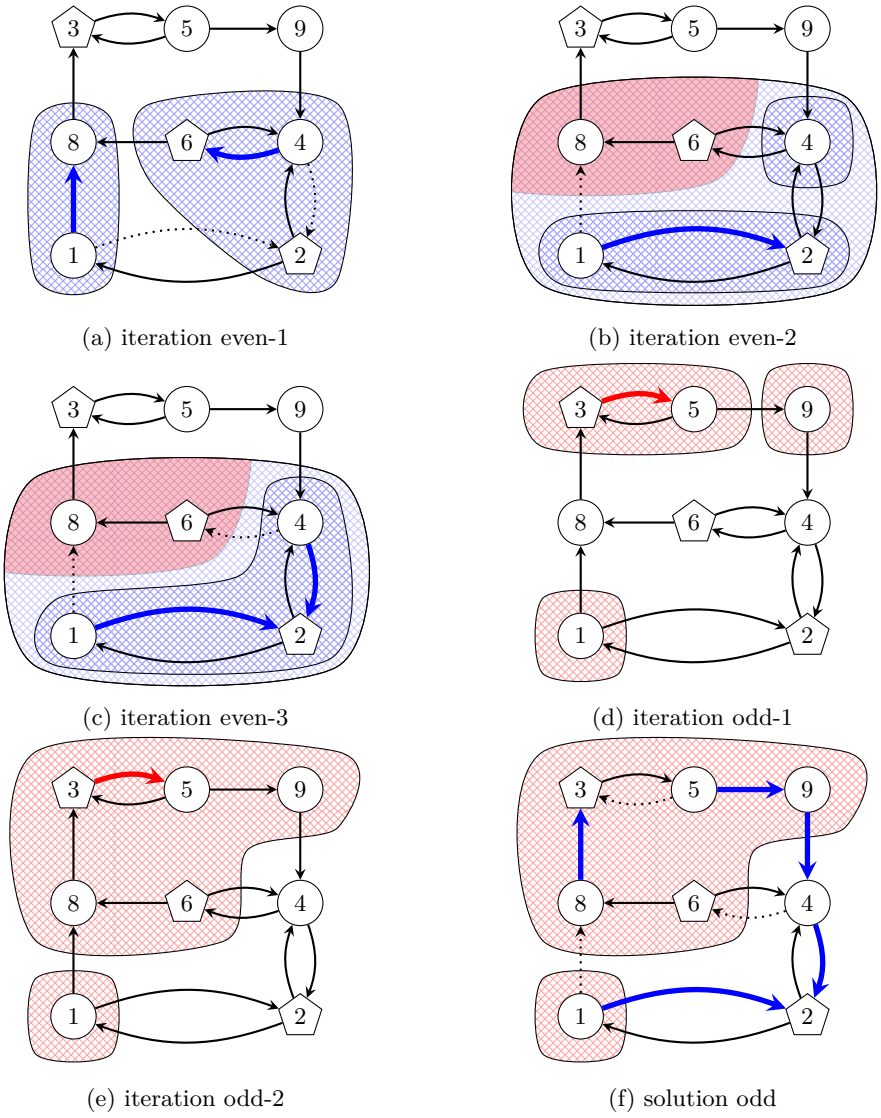
Iteration even-1. We decompose the game with only even-priority vertices as targets. See Figure 5a. We do not attract vertex 9 to region 6, as we only attract vertices and tangles with at most priority 6. This time, we only learn tangle $\{\mathbf{6}, \mathbf{4} \to \mathbf{6}\}$. Region 8 is open, and the attractor for player Odd to 8 includes vertex 1, which has no escapes inside region 8.

Iteration even-2. See Figure 5b. 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 5c. 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 5d. 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 5e. 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



(a) iteration even-1

(b) iteration even-2

(c) iteration even-3

(d) iteration odd-1

(e) iteration odd-2

(f) solution odd

Fig. 5: Solving an example game with one-sided recursive tangle learning.

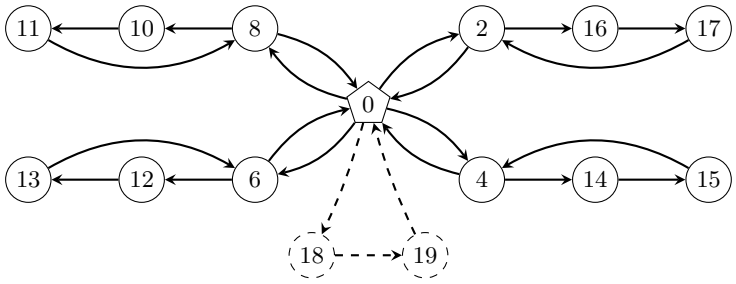


Fig. 6: A exponentially hard parity game for player Even in `ortl`. Vertices 18 and 19 are optional and convert the game from an Even dominion to an Odd dominion.

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 5f for the result.

Exponential lower bound There is a very simple exponential lower bound parity game for `ortl`. See Figure 6. 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.
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 $\{\mathbf{4} \to \mathbf{0}, \mathbf{0}\}$ in subregion 14-4.
5. In the fifth iteration, we learn $\{\mathbf{8} \to \mathbf{0}, \mathbf{4} \to \mathbf{0}, \mathbf{0}\}$ in subregion 14-10-8; then $\{\mathbf{6} \to \mathbf{0}, \mathbf{4} \to \mathbf{0}, \mathbf{0}\}$ in 14-12-6 in the sixth iteration and $\{\mathbf{8} \to \mathbf{0}, \mathbf{6} \to \mathbf{0}, \mathbf{4} \to \mathbf{0}, \mathbf{0}\}$ 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.

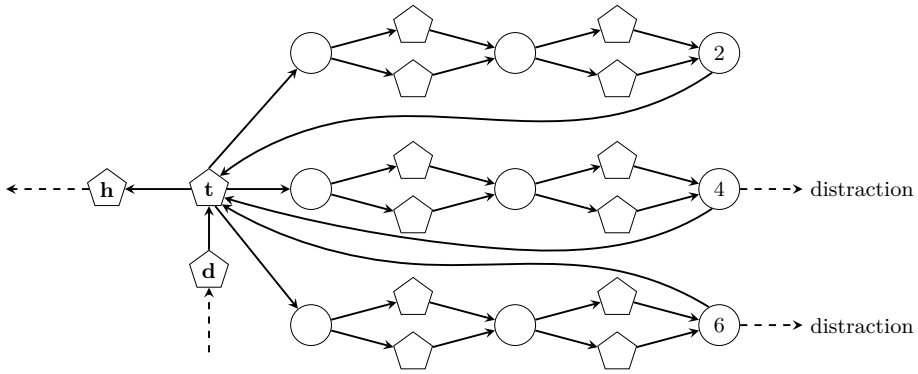


Fig. 7: Bit 2 of a 4-bit Two Counters game that delays `rtl` exponentially.

The `rtl` algorithm solves both exponential lower bounds of Figure 4 and Figure 6 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 4. 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 4, 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 4, 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 6, 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 7 for an example. For presentation, we omit the outgoing edges from the Odd-controlled vertices; they go to the higher bits like in Figure 4; the three chains are copies.

Both exponential games can be found online[1]. Source code of the `ortl` and `rtl` algorithms is also available there.

5 Empirical evaluation

We present empirical results in this section. The algorithms are implemented in Oink[2]. The `ortl` algorithm is implemented in an interleaved fashion, i.e., one even iteration, then one odd iteration, etc, until the entire game is solved. We refer to Appendix A for a description of various optimizations.

[1] See generators `counter_ortl` and `tc+` at https://www.github.com/trolando/oink.
[2] Available online via https://www.github.com/trolando/oink

	equivalence checking		**model checking**		**reactive synthesis**	
count	216		313		288	
# priorities	2		1–4		3–9	
	mean	**max**	**mean**	**max**	**mean**	**max**
# vertices	3.3M	40.6M	866K	27.9M	2181	178K
# edges	10.1M	167.5M	2.9M	80.8M	20149	1.45M

Table 1: Statistics of the three benchmark sets used for the empirical evaluation: the number games per category, the number of priorities, vertices, and edges.

benchmark	`fpi`	`fpj`	`zlk`	`pp`	`tl`	`rtl`	`ortl`
synthesis	36.51	0.08	36.53	0.07	0.08	0.16	0.16
model checking	74.10	79.82	50.96	11.61	27.38	42.22	59.00
equiv checking	346.89	280.79	150.21	88.36	124.27	207.87	267.56

Table 2: Cumulative runtimes in sec. of the different parity game solvers for each benchmark set. Only includes the benchmarks that all solvers could solve.

5.1 Setup

The goal of the empirical evaluation is to study the performance of `ortl` and `rtl` on practical parity games. We compare its performance with other algorithms that have high practical performance, namely tangle learning (`tl`), priority promotion [1] (`pp`), Zielonka's recursive algorithm (`zlk`), fixed point iteration with freezing [13] (`fpi`) and fixed point iteration with justifications [22] (`fpj`). These are among the currently fastest parity game solving algorithms implemented in Oink [8,9,12,13]. All algorithms are run without preprocessing in Oink.

We use the parity game benchmarks from model checking and equivalence checking proposed by Keiren [21] that are publicly available online [20]. These are 313 model checking and 216 equivalence checking games. Furthermore, we consider the benchmarks obtained from the SYNTCOMP [17] synthesis competition, which were translated to parity games using Knor [12]. In total, there are 288 parity games for reactive synthesis. We do not consider random games. There is no good argument why *random* games would be a good representative for games derived from actual applications, and thus interesting for practical performance. We also do not present the performance on various artificial benchmarks for the same reason. The `rtl` and `ortl` algorithms solve all artificially hard games in polynomial time, except the games presented in the current paper. See Table 1 for some statistics of the three benchmark sets.

5.2 Results

Table 2 shows the cumulative runtimes of the seven algorithms across the three benchmark sets. The results clearly show that the `ortl` and `rtl` algorithms have

some overhead compared to `tl`; however, this overhead is limited. The higher times for `fpi` and `zlk` are due to small overhead from the parallel load balancer (although we only used 1 thread), which takes a fraction of a second.

For the synthesis benchmark, the runtimes for `rtl` and `ortl` are very close to `tl`, with only minor increases. The synthesis benchmarks are already solved in a fraction of a second by most algorithms. In model checking, although the `rtl` and `ortl` algorithms have higher runtimes compared to `tl`, they still perform competitively. The most significant overhead appears in the equivalence checking benchmark, but even here, the `rtl` and `ortl` algorithms demonstrate reasonable performance.

6 Discussion

In the above, we have studied two mechanisms to handle distractions in parity games: the known method of attracting to opponent's regions or dominions; the proposed new method of avoiding open top vertices and recursively partitioning regions, thus giving higher value to vertices inside the region. We proposed recursive tangle learning and one-sided recursive tangle learning. We showed that these two mechanisms are not sufficient to solve parity games in polynomial time, as we demonstrate with exponential lower bounds for both algorithms.

Overall, the new algorithms `rtl` and `ortl` are roughly as fast as other high-performance algorithms. They have the advantage of being less vulnerable to pathological parity games, although these do not occur in parity games derived from practical applications. Considering the vulnerability to pathological parity games, the algorithms `fpi` and `fpj` are the most vulnerable, followed by `zlk`, then `pp`, and `tl`. The `ortl` and `rtl` algorithms are the least vulnerable.

Future work concerns further ideas to detect or avoid distractions. One interesting direction is to study how the quasi-polynomial recursive algorithms, e.g. [24], avoid distractions. Another idea might be to find heuristics that identify vertices that are currently distracting (because they are top vertices of an open region) and that could be removed from the set of *target vertices* for recursive tangle learning.

Conceptually it is easy to imagine that any algorithm can be fooled to all too eagerly mark good vertices as bad and bad vertices as good, and each new decision resetting progress on earlier distractions. Thus, whether these ideas can lead to a true polynomial-time algorithm remains an open question.

Data availability statement

The software, benchmarks and analysed dataset are available as [11] and on Github as: https://github.com/trolando/rtl-experiments. In addition, the version of the algorithms studied in the current paper is tagged in the Github repository of Oink as: https://github.com/trolando/oink/tree/ISOLA24.

References

1. Benerecetti, M., Dell'Erba, D., Mogavero, F.: Solving Parity Games via Priority Promotion. In: CAV 2016. LNCS, vol. 9780, pp. 270–290. Springer (2016)
2. Benerecetti, M., Dell'Erba, D., Mogavero, F.: Solving parity games via priority promotion. Formal Methods in System Design **52**(2), 193–226 (2018)
3. Berwanger, D., Grädel, E., Lenzi, G.: On the variable hierarchy of the modal μ-calculus. In: CSL. Lecture Notes in Computer Science, vol. 2471, pp. 352–366. Springer (2002)
4. Calude, C.S., Jain, S., Khoussainov, B., Li, W., Stephan, F.: Deciding parity games in quasipolynomial time. In: STOC. pp. 252–263. ACM (2017)
5. Chatterjee, K., Fijalkow, N.: A reduction from parity games to simple stochastic games. In: GandALF. EPTCS, vol. 54, pp. 74–86 (2011)
6. Condon, A.: The complexity of stochastic games. Inf. Comput. **96**(2), 203–224 (1992)
7. Dam, M.: CTL* and ECTL* as fragments of the modal mu-calculus. Theor. Comput. Sci. **126**(1), 77–96 (1994)
8. van Dijk, T.: Attracting tangles to solve parity games. In: CAV (2). LNCS, vol. 10982, pp. 198–215. Springer (2018)
9. van Dijk, T.: Oink: An implementation and evaluation of modern parity game solvers. In: TACAS (1). LNCS, vol. 10805, pp. 291–308. Springer (2018)
10. van Dijk, T.: A parity game tale of two counters. In: GandALF. EPTCS, vol. 305, pp. 107–122 (2019)
11. van Dijk, T.: RTL experiments (May 2024). https://doi.org/10.5281/zenodo.11265650
12. van Dijk, T., van Abbema, F., Tomov, N.: Knor: reactive synthesis using Oink. In: TACAS (1). Lecture Notes in Computer Science, vol. 14570, pp. 103–122. Springer (2024)
13. van Dijk, T., Rubbens, B.: Simple fixpoint iteration to solve parity games. In: GandALF. EPTCS, vol. 305, pp. 123–139 (2019)
14. Emerson, E.A., Jutla, C.S.: Tree automata, mu-calculus and determinacy (extended abstract). In: FOCS. pp. 368–377. IEEE Computer Society (1991)
15. 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)
16. 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)
17. Jacobs, S., Bloem, R., Colange, M., Faymonville, P., Finkbeiner, B., Khalimov, A., Klein, F., Luttenberger, M., Meyer, P.J., Michaud, T., Sakr, M., Sickert, S., Tentrup, L., Walker, A.: The 5th reactive synthesis competition (SYNTCOMP 2018): Benchmarks, participants & results. CoRR **abs/1904.07736** (2019)
18. Jurdzinski, M.: Small progress measures for solving parity games. In: STACS. LNCS, vol. 1770, pp. 290–301. Springer (2000)
19. Jurdzinski, M., Lazic, R.: Succinct progress measures for solving parity games. In: LICS. pp. 1–9. IEEE Computer Society (2017)
20. Keiren, J.: Parity games (2018). https://doi.org/10.6084/m9.figshare.6004130.v1
21. Keiren, J.J.A.: Benchmarks for parity games. In: FSEN. LNCS, vol. 9392, pp. 127–142. Springer (2015)
22. Lapauw, R., Bruynooghe, M., Denecker, M.: Improving parity game solvers with justifications. In: VMCAI. Lecture Notes in Computer Science, vol. 11990, pp. 449–470. Springer (2020)

23. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: Explicit reactive synthesis strikes back! In: CAV. Lecture Notes in Computer Science, vol. 10981, pp. 578–586. Springer (2018)
24. 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)
25. Renkin, F., Schlehuber, P., Duret-Lutz, A., Pommellet, A.: Improvements to ltlsynt. CoRR **abs/2201.05376** (2022), presented at SYNT 2021.

A Optimizations

We describe several optimizations for large practical games. These optimizations are all implemented in Oink[3]. Some of these optimizations were already described in the original paper on tangle learning [8].

1. After attracting to all vertices of the highest priority: if the next highest priority is of the same parity, we simply continue attracting.
2. To check if a region is closed, we only check the top vertices (that we attracted towards). We already know that attracted vertices of player α play to the region and that attracted vertices of player $\overline{\alpha}$ cannot escape to the remaining subgame.
3. If the highest region in the game is closed, then we immediately add it to the corresponding winning region and remove it from the game.
4. We do not need to check if the lowest region is open, as it is always closed.
5. When attracting vertices and tangles, we track the number of remaining edges towards the remaining game; when this number is 0, the vertex of player $\overline{\alpha}$ or the tangle can be attracted.
6. We only run Tarjan's algorithm starting from the top vertices. Tarjan's algorithm may find SCCs that are existing tangles; in the example of Figure 2b, if we had learned the tangle $\{2, 1 \to 2\}$ before and we attracted that tangle to region 6 instead of attracting 1 to 8, then region 6 would be closed and contain two SCCs, but only one new tangle. We can avoid duplicate tangles by only running Tarjan's algorithm starting from the top vertices.
7. When running Tarjan's algorithm, we immediately add dominions to a special queue so they can be processed afterwards, instead of separating dominions from other tangles later.
8. An optional optimization that improves the solver for some parity games, but delays it for other parity games: if a region is open, remove all open vertices O and iteratively remove all player $\overline{\alpha}$ controlled vertices with an edge to the removed vertices and all player α controlled vertices which have as the attractor strategy to play to a removed vertex. The remaining vertices, if any, now form a closed region from which new tangles can be obtained.
9. If a tangle overlaps with a winning region, we delete the tangle on-the-fly.

Most optimizations for `tl` also work for `rtl`. Optimization 5 does not fully work; we need to reset the number of remaining edges for the recursion. Optimization 8 is obviously replaced with the recursive decomposition. For `ortl`, additionally, optimization 4 no longer works, as the lowest region could still be open. We include these optimizations even though we are not primarily interested in the practical performance of the proposed algorithms. The algorithms are slower on practical games because they perform extra work that is not necessary. They are only faster on games that are designed to be hard for other algorithms.

[3] Available online via https://www.github.com/trolando/oink