Attracting Tangles to Solve Parity Games

Tom van Dijk*

Formal Models and Verification
Johannes Kepler University, Linz
`tom.vandijk@jku.at`

Abstract Parity games have important practical applications in formal verification and synthesis, especially to solve the model-checking problem of the modal mu-calculus. They are also interesting from the theory perspective, because they are widely believed to admit a polynomial solution, but so far no such algorithm is known.
We propose a new algorithm to solve parity games based on learning tangles, which are strongly connected subgraphs for which one player has a strategy to win all cycles in the subgraph. We argue that tangles play a fundamental role in the prominent parity game solving algorithms. We show that tangle learning is competitive in practice and the fastest solver for large random games.

1 Introduction

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

Parity games are interesting both for their practical applications and for complexity theoretic reasons. Their study has been motivated by their relation to many problems in formal verification and synthesis that can be reduced to the problem of solving parity games, as parity games capture the expressive power of nested least and greatest fixpoint operators [11]. In particular, deciding the winner of a parity game is polynomial-time equivalent to checking non-emptiness of non-deterministic parity tree automata [21], and to the explicit model-checking problem of the modal μ-calculus [9,15,20].

Parity games are interesting in complexity theory, as the problem of determining the winner of a parity game is known to lie in UP $\cap$ co-UP [16], which is contained in NP$\cap$co-NP [9]. This problem is therefore unlikely to be NP-complete and it is widely believed that a polynomial solution exists. Despite much effort, such an algorithm has not been found yet.

The main contribution of this paper is based on the notion of a *tangle*. A tangle is a strongly connected subgraph of a parity game for which one of

* The author is supported by the FWF, NFN Grant S11408-N23 (RiSE)

the players has a strategy to win all cycles in the subgraph. We propose this notion and its relation to dominions and cycles in a parity game. Tangles are related to snares [10] and quasi-dominions [3], with the critical difference that tangles are strongly connected, whereas snares and quasi-dominions may be unconnected as well as contain vertices that are not in any cycles. We argue that tangles play a fundamental role in various parity game algorithms, in particular in priority promotion [3,5], Zielonka's recursive algorithm [25], strategy improvement [10,11,24], small progress measures [17], and in the recently proposed quasi-polynomial time progress measures [6,12].

The core insight of this paper is that tangles can be used to attract sets of vertices at once, since the losing player is forced to escape a tangle. This leads to a novel algorithm to solve parity games called *tangle learning*, which is based on searching for tangles along a top-down α-maximal decomposition of the parity game. New tangles are then attracted in the next decomposition. This naturally leads to learning nested tangles and, eventually, finding dominions. We prove that tangle learning solves parity games and present several extensions to the core algorithm, including *alternating* tangle learning, where the two players take turns maximally searching for tangles in their regions, and *on-the-fly* tangle learning, where newly learned tangles immediately refine the decomposition.

We relate the complexity of tangle learning to the number of learned tangles before finding a dominion, which is related to how often the solver is distracted by paths to higher winning priorities that are not suitable strategies.

We evaluate tangle learning in a comparison based on the parity game solver Oink [7], using the benchmarks of Keiren [19] as well as random parity games of various sizes. We compare tangle learning to priority promotion [3,5] and to Zielonka's recursive algorithm [25] as implemented in Oink.

2 Preliminaries

Parity games are two-player turn-based infinite-duration games over a finite directed graph $G = (V, E)$, where every vertex belongs to exactly one of two players called player *Even* and player *Odd*, and where every vertex is assigned a natural number called the *priority*. Starting from some initial vertex, a play of both players is an infinite path in G where the owner of each vertex determines the next move. The winner of such an infinite play is determined by the parity of the highest priority that occurs infinitely often along the play.

More formally, a parity game $\Game$ is a tuple $(V_\Diamond, V_\Box, E, \mathsf{pr})$ where $V = V_\Diamond \cup V_\Box$ is a set of vertices partitioned into the sets $V_\Diamond$ controlled by player *Even* and $V_\Box$ controlled by player *Odd*, and $E \subseteq V \times V$ is a left-total binary relation describing all moves, i.e., every vertex has at least one successor. We also write $E(u)$ for all successors of u and $u \to v$ for $v \in E(u)$. The function $\mathsf{pr}\colon V \to \{0, 1, \ldots, d\}$ assigns to each vertex a *priority*, where d is the highest priority in the game.

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

A *strategy* $\sigma\colon V \to V$ is a partial function that assigns to each vertex in its domain a single successor in E, i.e., $\sigma \subseteq E$. We refer to a strategy of player α to restrict the domain of σ to V_α. In the remainder, all strategies σ are of a player α. We write $\text{Plays}(v, \sigma)$ for the set of plays from v consistent with σ, and $\text{Plays}(V, \sigma)$ for $\{\, \pi \in \text{Plays}(v, \sigma) \mid v \in V \,\}$.

A fundamental result for parity games is that they are memoryless determined [8], 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 all plays in $\text{Plays}(v, \sigma)$ are winning for player α.

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

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

Informally, 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 also obtain a strategy σ for player α, starting with an empty strategy, by selecting a successor in Z when we attract vertices of player α and when the backward search finds a successor in Z for the α-vertices in A. We call a set of vertices A α-maximal if $A = Attr_\alpha^{\eth}(A)$.

A *dominion* D is a set of vertices for which player α has a strategy σ such that all plays consistent with σ stay in D and are winning for player α. We also write a *p-dominion* for a dominion where p is the highest priority encountered infinitely often in plays consistent with σ, i.e., $p := \max\{\, \mathsf{pr}(\inf(\pi)) \mid \pi \in \text{Plays}(D, \sigma) \,\}$.

3 Tangles

Definition 1. *A* p-tangle *is a nonempty set of vertices* $U \subseteq V$ *with* $p = \mathsf{pr}(U)$*, for which player* $\alpha \equiv_2 p$ *has a strategy* $\sigma\colon U_\alpha \to U$*, such that the graph* (U, E')*, with* $E' := E \cap \big(\sigma \cup (U_{\overline{\alpha}} \times U)\big)$*, is strongly connected and player* α *wins all cycles in* (U, E')*.*

Informally, a tangle is a set of vertices for which player α has a strategy to win all cycles inside the tangle. Thus, player $\overline{\alpha}$ loses all plays that stay in U and is therefore forced to escape the tangle. The highest priority by which player α wins a play in (U, E') is p. We make several basic observations related to tangles.

1. A p-tangle from which player $\overline{\alpha}$ cannot leave is a p-dominion.
2. Every p-dominion contains one or more p-tangles.
3. Tangles may contain tangles of a lower priority.

Observation 1 follows by definition. Observation 2 follows from the fact that dominions won by player α with some strategy σ must contain strongly connected subgraphs where all cycles are won by player α and the highest winning priority is p. For observation 3, consider a p-tangle for which player $\overline{\alpha}$ has a strategy that avoids priority p while staying in the tangle. Then there is a p'-tangle with $p' < p$ in which player $\overline{\alpha}$ also loses.

We can in fact find a hierarchy of tangles in any dominion D with winning strategy σ by computing the set of winning priorities $\{\, \mathsf{pr}(\inf(\pi)) \mid \pi \in \text{Plays}(D, \sigma) \,\}$. There is a p-tangle in D for every p in this set. Tangles are thus a natural substructure of dominions.

See for example Figure 1. Player Odd wins this dominion with highest priority 5 and strategy $\{\, \mathbf{d} \to \mathbf{e} \,\}$. Player Even can also avoid priority 5 and then loses with priority 3. The 5-dominion $\{\, \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e} \,\}$ contains the 5-tangle $\{\, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e} \,\}$ and the 3-tangle $\{\, \mathbf{c}, \mathbf{e} \,\}$.

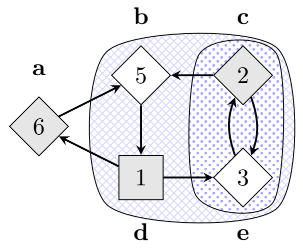

Figure 1. A 5-dominion with a 5-tangle and a 3-tangle

4 Solving by Learning Tangles

Since player $\overline{\alpha}$ must escape tangles won by player α, we can treat a tangle as an abstract vertex controlled by player $\overline{\alpha}$ that can be attracted by player α, thus attracting all vertices of the tangle. This section proposes the *tangle learning* algorithm, which searches for tangles along a top-down α-maximal decomposition of the game. We extend the attractor to attract all vertices in a tangle when player $\overline{\alpha}$ is forced to play from the tangle to the attracting set. After extracting new tangles from regions in the decomposition, we iteratively repeat the procedure until a dominion is found. We show that tangle learning solves parity games.

4.1 Attracting tangles

Given a tangle t, we denote its vertices simply by t and its witness strategy by $\sigma_T(t)$. We write $E_T(t)$ for the edges from $\overline{\alpha}$-vertices in the tangle to the rest of the game: $E_T(t) := \{\, v \mid u \to v \wedge u \in t \cap V_{\overline{\alpha}} \wedge v \in V \setminus t \,\}$. We write $T_\square$ for all tangles where $\mathsf{pr}(t)$ is odd (won by player Odd) and $T_\Diamond$ for all tangles where $\mathsf{pr}(t)$ is even. We write $TAttr_\alpha^{\circlearrowright,T}(A)$ to attract vertices in $\circlearrowright$ and vertices of tangles in T to A as player α, i.e.,

$$\mu Z \,.\, A \cup \{\, v \in V_\alpha \mid E(v) \cap Z \neq \emptyset \,\} \cup \{\, v \in V_{\overline{\alpha}} \mid E(v) \subseteq Z \,\}$$
$$\cup \{\, v \in t \mid t \in T_\alpha \wedge E_T(t) \neq \emptyset \wedge E_T(t) \subseteq Z \,\}$$

This approach is not the same as the subset construction. Indeed, we do not add the tangle itself but rather add all its vertices together. Notice that this attractor does not guarantee arrival in A, as player $\overline{\alpha}$ can stay in the added tangle, but then player $\overline{\alpha}$ loses.

```
1 def solve($\Game$):
2     $W_\Diamond \leftarrow \emptyset$, $W_\Box \leftarrow \emptyset$, $\sigma_\Diamond \leftarrow \emptyset$, $\sigma_\Box \leftarrow \emptyset$, $T \leftarrow \emptyset$
3     while $\Game \neq \emptyset$ :
4         $T, d \leftarrow$ search($\Game$, $T$)
5         $\alpha \leftarrow \mathsf{pr}(d) \bmod 2$
6         $D, \sigma \leftarrow Attr^{\Game}_{\alpha}(d)$
7         $W_\alpha \leftarrow W_\alpha \cup D$, $\sigma_\alpha \leftarrow \sigma_\alpha \cup \sigma_T(d) \cup \sigma$
8         $\Game \leftarrow \Game \setminus D$ , $T \leftarrow T \cap (\Game \setminus D)$
9     return $W_\Diamond, W_\Box, \sigma_\Diamond, \sigma_\Box$
```

Algorithm 1: The `solve` algorithm which computes the winning regions and winning strategies for both players of a given parity game.

To compute a witness strategy σ for player α, as with $Attr^{\Game}_{\alpha}$, we select a successor in Z when attracting single vertices of player α and when we find a successor in Z for the α-vertices in A. When we attract vertices of a tangle, we update σ for each tangle t sequentially, by updating σ with the strategy in $\sigma_T(t)$ of those α-vertices in the tangle for which we do not yet have a strategy in σ, i.e., $\{\, (u, v) \in \sigma_T(t) \mid u \notin \mathsf{dom}(\sigma) \,\}$. This is important since tangles can overlap.

In the following, we call a set of vertices A α-maximal if $A = TAttr^{\Game,T}_{\alpha}(A)$. Given a game $\Game$ and a set of vertices U, we write $\Game \cap U$ for the subgame $\Game'$ where $V' := V \cap U$ and $E' := E \cap (V' \times V')$. Given a set of tangles T and a set of vertices U, we write $T \cap U$ for all tangles with all vertices in U, i.e., $\{\, t \in T \mid t \subseteq U \,\}$, and we extend this notation to $T \cap \Game'$ for the tangles in the game $\Game'$, i.e., $T \cap V'$.

4.2 The `solve` algorithm

We solve parity games by iteratively searching and removing a dominion of the game, as in [3,18,22]. See Algorithm 1. The `search` algorithm (described below) is given a game and a set of tangles and returns an updated set of tangles and a tangle d that is a dominion. Since the dominion d is a tangle, we derive the winner α from the highest priority (line 5) and use standard attractor computation to compute a dominion D (line 6). We add the dominion to the winning region of player α (line 7). We also update the winning strategy of player α using the witness strategy of the tangle d plus the strategy σ obtained during attractor computation. To solve the remainder, we remove all solved vertices from the game and we remove all tangles that contain solved vertices (line 8).When the entire game is solved, we return the winning regions and winning strategies of both players (lines 9). Reusing the (pruned) set of tangles for the next `search` call is optional; if `search` is always called with an empty set of tangles, the “forgotten” tangles would be found again.

4.3 The `search` algorithm

The `search` algorithm is given in Algorithm 2. The algorithm iteratively computes a top-down decomposition of $\Game$ into sets of vertices called *regions* such that each

```
1  def search($\Game$, $T$):
2      while true :
3          $\mathsf{r} \leftarrow \emptyset$, $Y \leftarrow \emptyset$
4          while $\Game \setminus \mathsf{r} \neq \emptyset$ :
5              $\Game' \leftarrow \Game \setminus \mathsf{r}$, $T' \leftarrow T \cap (\Game \setminus \mathsf{r})$
6              $p \leftarrow \mathsf{pr}(\Game')$, $\alpha \leftarrow \mathsf{pr}(\Game') \bmod 2$
7              $Z, \sigma \leftarrow \mathit{TAttr}_{\alpha}^{\Game',T'}\big(\{\, v \in \Game' \mid \mathsf{pr}(v) = p \,\}\big)$
8              $A \leftarrow$ extract-tangles($Z$, $\sigma$)
9              if $\exists\, t \in A\colon E_T(t) = \emptyset$ : return $T \cup Y$, $t$
10             $\mathsf{r} \leftarrow \mathsf{r} \cup \big(Z \mapsto p\big)$, $Y \leftarrow Y \cup A$
11         $T \leftarrow T \cup Y$
```

Algorithm 2: The `search` algorithm which, given a game and a set of tangles, returns the updated set of tangles and a tangle that is a dominion.

region is α-maximal for the player α who wins the highest priority in the region. Each next region in the remaining subgame $\Game'$ is obtained by taking all vertices with the highest priority p in $\Game'$ and computing the tangle attractor set of these vertices for the player that wins that priority, i.e., player $\alpha \equiv_2 p$. As every next region has a lower priority, each region is associated with a unique priority p. We record the current region of each vertex in an auxiliary partial function $\mathsf{r}\colon V \to \{0, 1, \ldots, d\}$ called the region function. We record the new tangles found during each decomposition in the set Y.

In each iteration of the decomposition, we first obtain the current subgame $\Game'$ (line 5) and the top priority p in $\Game'$ (line 6). We compute the next region by attracting (with tangles) to the vertices of priority p in $\Game'$ (line 7). We use the procedure `extract-tangles` (described below) to obtain new tangles from the computed region (line 8). For each new tangle, we check if the set of outgoing edges to the full game $E_T(t)$ is empty. If $E_T(t)$ is empty, then we have a dominion and we terminate the procedure (line 9). If no dominions are found, then we add the new tangles to Y and update r (line 10). After fully decomposing the game into regions, we add all new tangles to T (line 11) and restart the procedure.

4.4 Extracting tangles from a region

To search for tangles in a given region A of player α with strategy σ, we first remove all vertices where player $\overline{\alpha}$ can play to lower regions (in $\Game'$) while player α is constrained to σ, i.e.,

$$\nu Z \,.\, A \cap \big(\{\, v \in V_{\overline{\alpha}} \mid E'(v) \subseteq Z \,\} \cup \{\, v \in V_{\alpha} \mid \sigma(v) \in Z \,\}\big)$$

This procedure can be implemented efficiently with a backward search, starting from all vertices of priority p that escape to lower regions. Since there can be multiple vertices of priority p, the reduced region may consist of multiple unconnected tangles. We compute all nontrivial bottom SCCs of the reduced region, restricted by the strategy σ. Every such SCC is a unique p-tangle.

4.5 Tangle learning solves parity games

We now prove properties of the proposed algorithm.

Lemma 1. *All regions recorded in* r *in Algorithm 2 are* α*-maximal in their subgame.*

Proof. This is vacuously true at the beginning of the search. Every region Z is α-maximal as Z is computed with *TAttr* (line 7). Therefore the lemma remains true when r is updated at line 10. New tangles are only added to T at line 11, after which r is reset to $\emptyset$. $\square$

Lemma 2. *All plays consistent with* σ *that stay in a region are won by player* α.

Proof. Based on how the attractor computes the region, we show that all cycles (consistent with σ) in the region are won by player α. Initially, Z only contains vertices with priority p; therefore, any cycles in Z are won by player α. We consider two cases: a) When attracting a single vertex v, any new cycles must involve vertices with priority p from the initial set A, since all other α-vertices in Z already have a strategy in Z and all other $\overline{\alpha}$-vertices in Z have only successors in Z, otherwise they would not be attracted to Z. Since p is the highest priority in the region, every new cycle is won by player α. b) When attracting vertices of a tangle, we set the strategy for all attracted vertices of player α to the witness strategy of the tangle. Any new cycles either involve vertices with priority p (as above) or are cycles inside the tangle that are won by player α. $\square$

Lemma 3. *Player* $\overline{\alpha}$ *can reach a vertex with the highest priority* p *from every vertex in the region, via a path in the region that is consistent with strategy* σ.

Proof. The proof is based on how the attractor computes the region. This property is trivially true for the initial set of vertices with priority p. We consider again two cases: a) When attracting a single vertex v, v is either an α-vertex with a strategy to play to Z, or an $\overline{\alpha}$-vertex whose successors are all in Z. Therefore, the property holds for v. b) Tangles are strongly connected w.r.t. their witness strategy. Therefore player $\overline{\alpha}$ can reach every vertex of the tangle and since the tangle is attracted to Z, at least one $\overline{\alpha}$-vertex can play to Z. Therefore, the property holds for all attracted vertices of a tangle. $\square$

Lemma 4. *For each new tangle* t*, all successors of* t *are in higher* α*-regions.*

Proof. For every bottom SCC B (computed in `extract-tangles`), $E'(v) \subseteq B$ for all $\overline{\alpha}$-vertices $v \in B$, otherwise player $\overline{\alpha}$ could leave B and B would not be a bottom SCC. Recall that $E'(v)$ is restricted to edges in the subgame $\partial' = \partial \setminus \mathsf{r}$. Therefore $E(v) \subseteq \mathsf{dom}(\mathsf{r}) \cup B$ in the full game for all $\overline{\alpha}$-vertices $v \in B$. Recall that $E_T(t)$ for a tangle t refers to all successors for player $\overline{\alpha}$ that leave the tangle. Hence, $E_T(t) \subseteq \mathsf{dom}(\mathsf{r})$ for every tangle $t := B$. Due to Lemma 1, no $\overline{\alpha}$-vertex in B can escape to a higher $\overline{\alpha}$-region. Thus $E_T(t)$ only contains vertices from higher α-regions when the new tangle is found by `extract-tangles`. $\square$

Lemma 5. *Every nontrivial bottom SCC B of the reduced region restricted by witness strategy σ is a unique p-tangle.*

Proof. All α-vertices v in B have a strategy $\sigma(v) \in B$, since B is a bottom SCC when restricted by σ. B is strongly connected by definition. Per Lemma 2, player α wins all plays consistent with σ in the region and therefore also in B. Thus, B is a tangle. Per Lemma 3, player $\overline{\alpha}$ can always reach a vertex of priority p, therefore any bottom SCC must include a vertex of priority p. Since p is the highest priority in the subgame, B is a p-tangle. Furthermore, the tangle must be unique. If the tangle was found before, then per Lemma 1 and Lemma 4, it would have been attracted to a higher α-region. □

Lemma 6. *The lowest region in the decomposition always contains a tangle.*

Proof. The lowest region is always nonempty after reduction in `extract-tangles`, as there are no lower regions. Furthermore, this region contains nontrivial bottom SCCs, since every vertex must have a successor in the region due to Lemma 1. □

Lemma 7. *A tangle t is a dominion if and only if $E_T(t) = \emptyset$*

Proof. If the tangle is a dominion, then player $\overline{\alpha}$ cannot leave it, therefore $E_T(t) = \emptyset$. If $E_T(t) = \emptyset$, then player $\overline{\alpha}$ cannot leave the tangle and since all plays consistent with σ in the tangle are won by player α, the tangle is a dominion. □

Lemma 8. *$E_T(t) = \emptyset$ for every tangle t found in the highest region of player α.*

Proof. Per Lemma 4, $E_T(t) \subseteq \{\, v \in \mathsf{dom}(\mathsf{r}) \mid \mathsf{r}(v) \equiv_2 p \,\}$ when the tangle is found. There are no higher regions of player α, therefore $E_T(t) = \emptyset$. □

Lemma 9. *The `search` algorithm terminates by finding a dominion.*

Proof. There is always a highest region of one of the players that is not empty. If a tangle is found in this region, then it is a dominion (Lemma 7, Lemma 8) and Algorithm 2 terminates (line 9). If no tangle is found in this region, then the opponent can escape to a lower region. Thus, if no dominion is found in a highest region, then there is a lower region that contains a tangle (Lemma 6) that must be unique (Lemma 5). As there are only finitely many unique tangles, eventually a dominion must be found. □

Lemma 10. *The `solve` algorithm solves parity games.*

Proof. Every invocation of `search` returns a dominion of the game (Lemma 9). The α-attractor of a dominion won by player α is also a dominion of player α. Thus all vertices in D are won by player α. The winning strategy is derived as the witness strategy of d with the strategy obtained by the attractor at line 6. At the end of `solve` every vertex of the game is either in $W_\diamond$ or $W_\square$. □

4.6 Variations of tangle learning

We propose three different variations of tangle learning that can be combined.

The first variation is *alternating tangle learning*, where players take turns to maximally learn tangles, i.e., in a turn of player Even, we only search for tangles in regions of player Even, until no more tangles are found. Then we search only for tangles in regions of player Odd, until no more tangles are found. When changing players, the last decomposition can be reused.

The second variation is *on-the-fly tangle learning*, where new tangles immediately refine the decomposition. When new tangles are found, the decomposition procedure is reset to the highest region that attracts one of the new tangles, such that all regions in the top-down decomposition remain α-maximal. This is the region with priority $p := \max\{ \min\{ \mathsf{r}(v) \mid v \in E_T(t) \} \mid t \in A \}$.

A third variation skips the reduction step in `extract-tangles` and only extracts tangles from regions where none of the vertices of priority p can escape to lower regions. This still terminates finding a dominion, as Lemma 6 still applies, i.e., we always extract tangles from the lowest region. Similar variations are also conceivable, such as only learning tangles from the lowest region.

5 Complexity

We establish a relation between the time complexity of tangle learning and the number of *learned* tangles.

Lemma 11. *Computing the top-down α-maximal decomposition of a parity game runs in time $O(dm + dn|T|)$ given a parity game with d priorities, n vertices and m edges, and a set of tangles T.*

Proof. The attractor $Attr_{\alpha}^{\eth}$ runs in time $O(n + m)$, if we record the number of remaining outgoing edges for each vertex [23]. The attractor $TAttr_{\alpha}^{\eth,T}$ runs in time $O(n + m + |T| + n|T|)$, if implemented in a similar style. As $m \geq n$, we simplify to $O(m + n|T|)$. Since the decomposition computes at most d regions, the decomposition runs in time $O(dm + dn|T|)$. □

Lemma 12. *Searching for tangles in the decomposition runs in time $O(dm)$.*

Proof. The `extract-tangles` procedure consists of a backward search, which runs in $O(n + m)$, and an SCC search based on Tarjan's algorithm, which also runs in $O(n + m)$. This procedure is performed at most d times (for each region). As $m \geq n$, we simplify to $O(dm)$. □

Lemma 13. *Tangle learning runs in time $O(dnm|T| + dn^2|T|^2)$ for a parity game with d priorities, n vertices, m edges, and $|T|$* learned *tangles.*

Proof. Given Lemma 11 and Lemma 12, each iteration in `search` runs in time $O(dm + dn|T|)$. The number of iterations is at most $|T|$, since we learn at least 1 tangle per iteration. Then `search` runs in time $O(dm|T| + dn|T|^2)$. Since each found dominion is then removed from the game, there are at most n calls to `search`. Thus tangle learning runs in time $O(dnm|T| + dn^2|T|^2)$. □

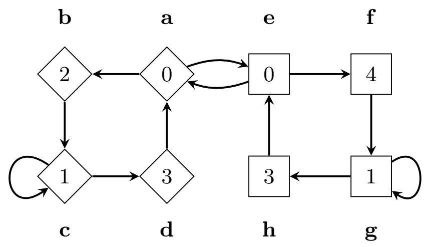

Figure 2. A parity game that requires several turns to find a dominion.

The complexity of tangle learning follows from the number of tangles that are learned before each dominion is found. Often not all tangles in a game need to be learned to solve the game, only certain tangles. Whether this number can be exponential in the worst case is an open question. These tangles often serve to remove *distractions* that prevent the other player from finding better tangles. This concept is illustrated by the example in Figure 2, which requires multiple turns before a dominion is found. The game contains 4 tangles: $\{\,\mathbf{c}\,\}$, $\{\,\mathbf{g}\,\}$ (a dominion), $\{\,\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d}\,\}$ and $\{\,\mathbf{a},\mathbf{e}\,\}$. The vertices $\{\,\mathbf{e},\mathbf{f},\mathbf{g},\mathbf{h}\,\}$ do not form a tangle, since the opponent wins the loop of vertex $\mathbf{g}$. The tangle $\{\,\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d}\,\}$ is a dominion in the remaining game after $Attr_{\square}^{\supset}(\{\,\mathbf{g}\,\})$ has been removed.

The tangle $\{\,\mathbf{g}\,\}$ is not found at first, as player Odd is distracted by $\mathbf{h}$, i.e., prefers to play from $\mathbf{g}$ to $\mathbf{h}$. Thus vertex $\mathbf{h}$ must first be attracted by the opponent. This occurs when player Even learns the tangle $\{\,\mathbf{a},\mathbf{e}\,\}$, which is then attracted to $\mathbf{f}$, which then attracts $\mathbf{h}$. However, the tangle $\{\,\mathbf{a},\mathbf{e}\,\}$ is blocked, as player Even is distracted by $\mathbf{b}$. Vertex $\mathbf{b}$ is attracted by player Odd when they learn the tangle $\{\,\mathbf{c}\,\}$, which is attracted to $\mathbf{d}$, which then attracts $\mathbf{b}$. So player Odd must learn tangle $\{\,\mathbf{c}\,\}$ so player Even can learn tangle $\{\,\mathbf{a},\mathbf{e}\,\}$, which player Even must learn so player Odd can learn tangle $\{\,\mathbf{g}\,\}$ and win the dominion $\{\,\mathbf{e},\mathbf{f},\mathbf{g},\mathbf{h}\,\}$, after which player Odd also learns $\{\,\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d}\,\}$ and wins the entire game.

One can also understand the algorithm as the players learning that their opponent can now play from some vertex v via the learned tangle to a higher vertex w that is won by the opponent. In the example, we first learn that $\mathbf{b}$ actually leads to $\mathbf{d}$ via the learned tangle $\{\,\mathbf{c}\,\}$. Now $\mathbf{b}$ is no longer safe for player Even. However, player Even can now play from both $\mathbf{d}$ and $\mathbf{h}$ via the learned 0-tangle $\{\,\mathbf{a},\mathbf{e}\,\}$ to $\mathbf{f}$, so $\mathbf{d}$ and $\mathbf{h}$ are no longer interesting for player Odd and vertex $\mathbf{b}$ is again safe for player Even.

6 Implementation

We implement four variations of tangle learning in the parity game solver Oink [7]. Oink is a modern implementation of parity game algorithms written in C++. Oink implements priority promotion [3], Zielonka's recursive algorithm [25], strategy improvement [11], small progress measures [17], and quasi-polynomial time progress measures [12]. Oink also implements self-loop solving and winner-controlled winning cycle detection, as proposed in [23]. The implementation is publicly available via `https://www.github.com/trolando/oink`.

We implement the following variations of tangle learning: standard tangle learning (`tl`), alternating tangle learning (`atl`), on-the-fly tangle learning (`otftl`) and on-the-fly alternating tangle learning (`otfatl`). The implementation mainly differs from the presented algorithm in the following ways. We combine the **`solve`** and **`search`** algorithms in one loop. We remember the highest region that attracts a new tangle and reset the decomposition to that region instead of recomputing the full decomposition. In **`extract-tangles`**, we do not compute bottom SCCs for the highest region of a player, instead we return the entire reduced region as a single dominion (see also Lemma 8).

7 Empirical evaluation

The goal of the empirical evaluation is to study tangle learning and its variations on real-world examples and random games. Due to space limitations, we do not report in detail on crafted benchmark families (generated by PGSolver [13]), except that none of these games is difficult in runtime or number of tangles.

We use the parity game benchmarks from model checking and equivalence checking proposed by Keiren [19] that are publicly available online. These are 313 model checking and 216 equivalence checking games. We also consider random games, in part because the literature on parity games tends to favor studying the behavior of algorithms on random games. We include two classes of self-loop-free random games generated by PGSolver [13] with a fixed number of vertices:

– fully random games (**`randomgame`** `N N 1 N x`)
  $N \in \{ 1000, 2000, 4000, 7000 \}$
– large low out-degree random games (**`randomgame`** `N N 1 2 x`)
  $N \in \{ 10000, 20000, 40000, 70000, 100000, 200000, 400000, 700000, 1000000 \}$

We generate 20 games for each parameter N, in total 80 fully random games and 180 low out-degree games. All random games have N vertices and up to N distinct priorities. We include low out-degree games, since algorithms may behave differently on games where all vertices have few available moves, as also suggested in [3]. In fact, as we see in the evaluation, fully random games appear trivial to solve, whereas games with few moves per vertex are more challenging. Furthermore, the fully random games have fewer vertices but require more disk space (40 MB per compressed file for $N = 7000$) than large low out-degree games (11 MB per compressed file for $N = 1000000$).

We compare four variations of tangle learning to the implementations of Zielonka's recursive algorithm (optimized version of Oink) and of priority promotion (implemented in Oink by the authors of [3]). The motivation for this choice is that [7] shows that these are the fastest parity game solving algorithms.

In the following, we also use *cactus plots* to compare the algorithms. Cactus plots show that an algorithm solved X input games within Y seconds individually.

Solver	**MC&EC**	**Random**	**Random**	**(large)**
	time	time	time	timeouts
pp	503	21	12770	6
zlk	576	21	23119	13
otfatl	808	21	2281	0
atl	817	21	2404	0
otftl	825	21	2238	0
tl	825	21	2312	0

Table 1. Runtimes in sec. and number of timeouts (20 minutes) of the solvers Zielonka (`zlk`), priority promotion (`pp`), and tangle learning (`tl`, `atl`, `otftl`, `otfatl`).

All experimental scripts and log files are available online via `https://www.github.com/trolando/tl-experiments`. The experiments were performed on a cluster of Dell PowerEdge M610 servers with two Xeon E5520 processors and 24 GB internal memory each. The tools were compiled with gcc 5.4.0.

7.1 Overall results

Table 1 shows the cumulative runtimes of the six algorithms. For the runs that timed out, we simply used the timeout value of 1200 seconds, but this underestimates the actual runtime.

7.2 Model checking and equivalence checking games

See Figure 3 for the cactus plot of the six solvers on model checking and equivalence checking games. This graph suggests that for most games, tangle learning is only slightly slower than the other algorithms. The tangle learning algorithms require at most 2× as much time for 12 of the 529 games. There is no significant difference between the four variations of tangle learning.

The 529 games have on average 1.86 million vertices and 5.85 million edges, and at most 40.6 million vertices and 167.5 million edges. All equivalence checking games have 2 priorities, so every tangle is a dominion. The model checking games have 2 to 4 priorities. Tangle learning learns non-dominion tangles for only 30 games, and more than 1 tangle only for the 22 games that check the `infinitely_often_read_write` property. The most extreme case is 1,572,864 tangles for a game with 19,550,209 vertices. These are all 0-tangles that are then attracted to become part of 2-dominions.

That priority promotion and Zielonka's algorithm perform well is no surprise. See also Sec. 8.4. Solving these parity games requires few iterations for all algorithms, but tangle learning spends more time learning and attracting individual tangles, which the other algorithms do not do. Zielonka requires at most 27 iterations, priority promotion at most 28 queries and 9 promotions. Alternating tangle learning requires at most 2 turns. We conclude that these games are not complex and that their difficulty is related to their sheer size.

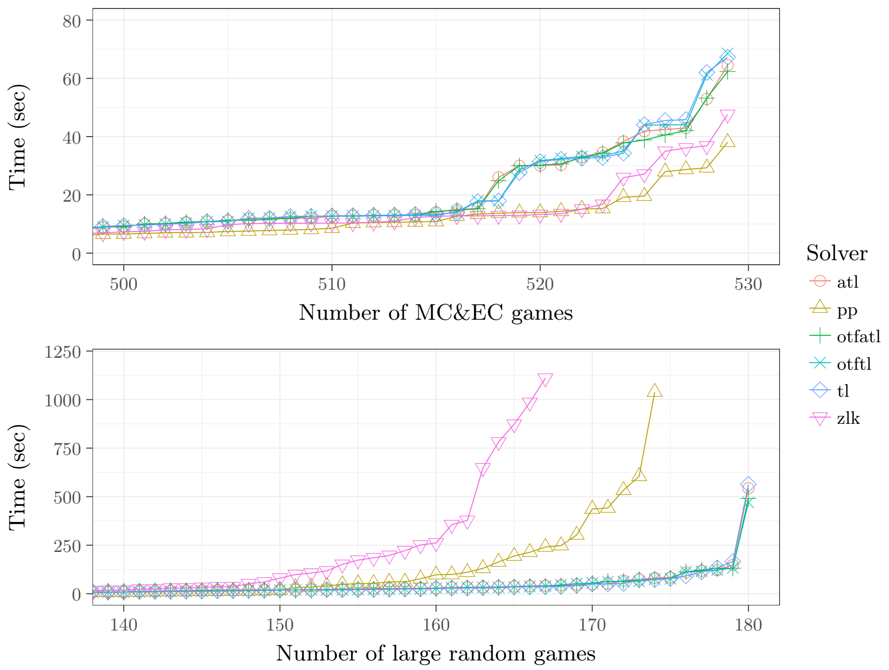

Figure 3. Cactus plots of the solvers Zielonka (`zlk`), priority promotion (`pp`) and tangle learning (`tl`, `atl`, `otftl`, `otfatl`). The plot shows how many MC&EC games (top) or large random games (bottom) are (individually) solved within the given time.

7.3 Random games

Table 1 shows no differences between the algorithms for the fully random games. Tangle learning learns no tangles except dominions for any of these games. This agrees with the intuition that the vast number of edges in these games lets attractor-based algorithms quickly attract large portions of the game.

See Fig. 3 for a cactus plot of the solvers on the larger random games. Only 167 games were solved within 20 minutes each by Zielonka's algorithm and only 174 games by priority promotion. See Table 2 for details of the slowest 10 random games for alternating tangle learning. There is a clear correlation between the runtime, the number of tangles and the number of turns. One game is particularly interesting, as it requires significantly more time than the other games.

The presence of one game that is much more difficult is a feature of using random games. It is likely that if we generated a new set of random games, we would obtain different results. This could be ameliorated by experimenting on thousands of random games and even then it is still a game of chance whether some of these random games are significantly more difficult than the others.

Time	543	148	121	118	110	83	81	73	68	52
Tangles	4,018	1,219	737	560	939	337	493	309	229	384
Turns	91	56	23	25	30	12	18	10	10	18
Size	1M	1M	700K	1M	700K	1M	1M	1M	1M	1M

Table 2. The 10 hardest random games for the `atl` algorithm, with time in seconds and size in number of vertices.

8 Tangles in other algorithms

We argue that tangles play a fundamental role in various other parity game solving algorithms. We refer to [7] for descriptions of these algorithms.

8.1 Small progress measures

The small progress measures algorithm [17] iteratively performs local updates to vertices until a fixed point is reached. Each vertex is equipped with some measure that records a statistic of the best game either player knows that they can play from that vertex so far. By updating measures based on the successors, they essentially play the game backwards. When they can no longer perform updates, the final measures indicate the winning player of each vertex.

The measures in small progress measures record how often each even priority is encountered along the most optimal play (so far) until a higher priority is encountered. As argued in [7] and [14], player Even tries to visit vertices with even priorities as often as possible, while prioritizing plays with more higher even priorities. This often resets progress for lower priorities. Player Odd has the opposite goal, i.e., player Odd prefers to play to a lower even priority to avoid a higher even priority, even if the lower priority is visited infinitely often. When the measures record a play from some vertex that visits more vertices with some even priority than exist in the game, this indicates that player Even can force player Odd into a cycle, unless they concede and play to a higher even priority. A mechanism called cap-and-carryover [7] ensures via slowly rising measures that the opponent is forced to play to a higher even priority.

We argue that when small progress measures finds cycles of some priority p, this is due to the presence of a p-tangle, namely precisely those vertices whose measures increase beyond the number of vertices with priority p, since these measures can only increase so far in the presence of cycles out of which the opponent cannot escape except by playing to vertices with a higher even priority.

One can now understand small progress measures as follows. The algorithm indirectly searches for tangles of player Even, and then searches for the best escape for player Odd by playing to the lowest higher even priority. If no such escape exists for a tangle, then the measures eventually rise to $\top$, indicating that player Even has a dominion. Whereas tangle learning is affected by *distractions*, small progress measures is driven by the dual notion of *aversions*, i.e., high even vertices that player Odd initially tries to avoid. The small progress measures algorithm tends to find tangles repeatedly, especially when they are nested.

8.2 Quasi-polynomial time progress measures

The quasi-polynomial time progress measures algorithm [12] is similar to small progress measures. This algorithm records the number of dominating even vertices along a play, i.e., such that every two such vertices are higher than all intermediate vertices. For example, in the path $1\underline{2}1 3 1\underline{4}2 3\underline{2}1 5\underline{6}3\underline{2}1\underline{2}$, all vertices are dominated by each pair of underlined vertices of even priority. Higher even vertices are preferred, even if this (partially) resets progress on lower priorities.

Tangles play a similar role as with small progress measures. The presence of a tangle lets the value iteration procedure increase the measure up to the point where the other player "escapes" the tangle via a vertex outside of the tangle. This algorithm has a similar weakness to nested tangles, but it is less severe as progress on lower priorities is often retained. In fact, the lower bound game in [12], for which the quasi-polynomial time algorithm is slow, is precisely based on nested tangles and is easily solved by tangle learning.

8.3 Strategy improvement

As argued by Fearnley [10], tangles play a fundamental role in the behavior of strategy improvement. Fearnley writes that instead of viewing strategy improvement as a process that tries to increase valuations, one can view it as a process that tries to force "consistency with snares" [10, Sec. 6], i.e., as a process that searches for escapes from tangles.

8.4 Priority promotion

Priority promotion [3,5] computes a top-down α-maximal decomposition and identifies "closed α-regions", i.e., regions where the losing player cannot escape to lower regions. A closed α-region is essentially a collection of possibly unconnected tangles and vertices that are attracted to these tangles. Priority promotion then promotes the closed region to the lowest higher region that the losing player can play to, i.e., the lowest region that would attract one of the tangles in the region. Promoting is different from attracting, as tangles in a region can be promoted to a priority that they are not attracted to. Furthermore, priority promotion has no mechanism to remember tangles, so the same tangle can be discovered many times. This is somewhat ameliorated in extensions such as region recovery [2] and delayed promotion [1], which aim to decrease how often regions are recomputed.

Priority promotion has a good practical performance for games where computing and attracting individual tangles is not necessary, e.g., when tangles are only attracted once and all tangles in a closed region are attracted to the same higher region, as is the case with the benchmark games of [19].

8.5 Zielonka's recursive algorithm

Zielonka's recursive algorithm [25] also computes a top-down α-maximal decomposition, but instead of attracting from lower regions to higher regions, the algorithm attracts from higher regions to tangles in the subgame. Essentially, the algorithm starts with the tangles in the lowest region and attracts from higher regions to these tangles. When vertices from a higher α-region are attracted to tangles of player $\overline{\alpha}$, progress for player α is reset. Zielonka's algorithm also has no mechanism to store tangles and games that are exponential for Zielonka's algorithm, such as in [4], are trivially solved by tangle learning.

9 Conclusions

We introduced the notion of a tangle as a subgraph of the game where one player knows how to win all cycles. We showed how tangles and nested tangles play a fundamental role in various parity game algorithms. These algorithms are not explicitly aware of tangles and can thus repeatedly explore the same tangles. We proposed a novel algorithm called tangle learning, which identifies tangles in a parity game and then uses these tangles to attract sets of vertices at once. The key insight is that tangles can be used with the attractor to form bigger (nested) tangles and, eventually, dominions. We evaluated tangle learning in a comparison with priority promotion and Zielonka's recursive algorithm and showed that tangle learning is competitive for model checking and equivalence checking games, and outperforms other solvers for large random games.

We repeat Fearnley's assertion [10] that "a thorough and complete understanding of how snares arise in a game is a necessary condition for devising a polynomial time algorithm for these games." Fearnley also formulated the challenge to give a clear formulation of how the structure of tangles in a given game affects the difficulty of solving it. We propose that a difficult game for tangle learning must be one that causes alternating tangle learning to have many turns before a dominion is found.

Acknowledgements

We thank the anonymous referees for their helpful comments, Jaco van de Pol for the use of the computer cluster, and Armin Biere for generously supporting this research.

References

1. Benerecetti, M., Dell'Erba, D., Mogavero, F.: A Delayed Promotion Policy for Parity Games. In: GandALF 2016. EPTCS, vol. 226, pp. 30–45 (2016)
2. Benerecetti, M., Dell'Erba, D., Mogavero, F.: Improving Priority Promotion for Parity Games. In: HVC 2016. LNCS, vol. 10028, pp. 117–133 (2016)
3. Benerecetti, M., Dell'Erba, D., Mogavero, F.: Solving Parity Games via Priority Promotion. In: CAV 2016. LNCS, vol. 9780, pp. 270–290. Springer (2016)
4. Benerecetti, M., Dell'Erba, D., Mogavero, F.: Robust exponential worst cases for divide-et-impera algorithms for parity games. In: GandALF. EPTCS, vol. 256, pp. 121–135 (2017)
5. Benerecetti, M., Dell'Erba, D., Mogavero, F.: Solving parity games via priority promotion. Formal Methods in System Design (2018)
6. Calude, C.S., Jain, S., Khoussainov, B., Li, W., Stephan, F.: Deciding parity games in quasipolynomial time. In: STOC. pp. 252–263. ACM (2017)
7. van Dijk, T.: Oink: an implementation and evaluation of modern parity game solvers. In: TACAS (2018), https://arxiv.org/pdf/1801.03859
8. Emerson, E.A., Jutla, C.S.: Tree automata, mu-calculus and determinacy (extended abstract). In: FOCS. pp. 368–377. IEEE Computer Society (1991)
9. Emerson, E.A., Jutla, C.S., Sistla, A.P.: On model checking for the mu-calculus and its fragments. Theor. Comput. Sci. 258(1-2), 491–522 (2001)
10. Fearnley, J.: Non-oblivious strategy improvement. In: LPAR (Dakar). LNCS, vol. 6355, pp. 212–230. Springer (2010)
11. Fearnley, J.: Efficient parallel strategy improvement for parity games. In: CAV (2). LNCS, vol. 10427, pp. 137–154. Springer (2017)
12. 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)
13. Friedmann, O., Lange, M.: Solving parity games in practice. In: ATVA. LNCS, vol. 5799, pp. 182–196. Springer (2009)
14. Gazda, M., Willemse, T.A.C.: Improvement in small progress measures. In: GandALF. EPTCS, vol. 193, pp. 158–171 (2015)
15. Grädel, E., Thomas, W., Wilke, T. (eds.): Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001], LNCS, vol. 2500. Springer (2002)
16. Jurdzinski, M.: Deciding the winner in parity games is in UP ∩ co-UP. Inf. Process. Lett. 68(3), 119–124 (1998)
17. Jurdzinski, M.: Small progress measures for solving parity games. In: STACS. LNCS, vol. 1770, pp. 290–301. Springer (2000)
18. Jurdzinski, M., Paterson, M., Zwick, U.: A deterministic subexponential algorithm for solving parity games. SIAM J. Comput. 38(4), 1519–1532 (2008)
19. Keiren, J.J.A.: Benchmarks for parity games. In: FSEN. LNCS, vol. 9392, pp. 127–142. Springer (2015)
20. Kozen, D.: Results on the propositional mu-calculus. Theor. Comput. Sci. 27, 333–354 (1983)
21. Kupferman, O., Vardi, M.Y.: Weak alternating automata and tree automata emptiness. In: STOC. pp. 224–233. ACM (1998)
22. Schewe, S.: Solving parity games in big steps. J. Comput. Syst. Sci. 84, 243–262 (2017)

23. Verver, M.: Practical improvements to parity game solving. Master's thesis, University of Twente (2013)
24. Vöge, J., Jurdzinski, M.: A discrete strategy improvement algorithm for solving parity games. In: CAV. LNCS, vol. 1855, pp. 202–215. Springer (2000)
25. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theor. Comput. Sci. 200(1-2), 135–183 (1998)