Improving Parity Game Solvers with Justifications

Ruben Lapauw *, Maurice Bruynooghe **, and Marc Denecker *** †

KU Leuven, Dept. of Computer Science, B-3001 Leuven, Belgium

Abstract. Parity games are infinite two-player games played on node-weighted directed graphs. Formal verification problems such as verifying and synthesizing automata, bounded model checking of LTL, CTL*, propositional μ-calculus, ... reduce to problems over parity games. The core problem of parity game solving is deciding the winner of some (or all) nodes in a parity game. In this paper, we improve several parity game solvers by using a justification graph. Experimental evaluation shows our algorithms improve upon the state-of-the-art.

1 Introduction

Parity games are infinite two player games played on node-weighted directed graphs without leaves. Priorities, the weights of the nodes, are integers in an interval $[1, d]$ with d a parameter of the parity game. All nodes belong to one player, either Even or Odd. The players play by moving one token from node to node following the edges of the graph. The owner of the node on which the token lands, plays next. The winner of this infinite path, a play, is Even if the maximal priority that occurs infinitely often is even, otherwise Odd wins. A parity game solver determines for each node the winner and a winning strategy. Many problems over boolean equation systems [5, 13], μ-calculus [10, 20], nested fixpoints [11] and temporal logics such as LTL, CTL and CTL* [10] reduce to parity games.

The algorithm with the best known time complexity [4, 7] is quasi-polynomial in the number of weights. However, in practice, it is outperformed by several exponential algorithms, most notably: fixpoint induction [3], Zielonka's algorithm [21], multiple variants of strategy-improvement [16], priority promotion [1,

* ruben.lapauw@cs.kuleuven.be, Supported by a IWT research grant
** maurice.bruynooghe@cs.kuleuven.be
*** marc.denecker@cs.kuleuven.be
† The final authenticated version is available at https://doi.org/10.1007/978-3-030-39322-9\protect_21

2] and tangle learning [18]. Currently, Zielonka's algorithm and priority promotion are considered the two fastest algorithms though there is experimental evidence that tangle learning is faster on large random graphs [18].

Our contribution is to extend these algorithms with a justification graph data structure. For fixpoint induction this allows us to efficiently reconstruct winning strategies; more importantly, it enables optimizations in the computation of a solution. Furthermore, we applied these optimisations for Zielonka's algorithm, and tangle learning. As our experiments show, adding justifications to these algorithms improve their performance. Overall, tangle learning extended with justifications performs best over the whole of our benchmark set.

In Section 2, the preliminaries, we formally define the parity game problem, introduce the μ-formula [20] that determines the winners of a parity game and a parity game solving fixpoint algorithm by Bruse et al. [3]. In Section 3, we introduce justifications, integrate them in the fixpoint algorithm and argue correctness is preserved. In Section 4 this technique is applied to Zielonka's algorithm and tangle learning. Next, in Section 5, we evaluate our three justification based implementations. We round up in Section 6.

2 Preliminaries

2.1 Parity games

A parity game is a two player game with players 0 (Even) and 1 (Odd). We use α to denote a player with $\alpha \in \{0, 1\}$ and use $\bar{\alpha}$ to denote the opponent of player α. Formally, a parity game is a tuple $PG = (V, E, V_0, V_1, Pr)$ consisting of a finite game graph (V, E) with outgoing edges for every node, a partition $\{V_0, V_1\}$ of nodes V and a priority function $Pr : V \to P$, with P an interval of integer priorities $\{1 \dots d\}$. In nodes of V_0, player Even plays, and in those of V_1, it is player Odd. Without loss of generality we assume d is even.

Given a set of nodes S, a player α, a priority i and a relation $\sim$ in the set $\{=, \neq, \leq, \geq, <, >, \equiv_2, \not\equiv_2\}$: $S|_\alpha$ denotes the set $S \cap V_\alpha$ and $S^{\sim i}$ denotes the set of nodes $\{v \in S | Pr(v) \sim i\}$. E.g., $V^{<i}$ is $\{v \in V | Pr(v) < i\}$ and $V^{\equiv_2 0}$ is the set of nodes with a priority which is equivalent – modulo 2 – with priority 0, i.e., the nodes with an even priority.

A play is an infinite sequence of nodes $\langle v_0 v_1 \dots v_n \dots \rangle$ where $\forall i \in \mathbb{N} : v_i \in V \wedge (v_i, v_{i+1}) \in E$. We use π to denote a play. Since every node has outgoing edges, there exist plays in every node. The winner of a play is the player with the parity of the highest priority that occurs infinitely often: $Winner(\pi) = \lim_{i \to +\infty} max \{Pr(v_j) | j \geq i\} \mod 2$.

A strategy for α is a partial function $\sigma_\alpha : V_\alpha \to V$ for which for every x in the domain of σ_α: $(x, \sigma_\alpha(x)) \in E$. A play π is consistent with σ_α if $v_{n+1} = \sigma_\alpha(v_n)$ for every v_n in the domain of σ_α. A strategy σ_α is winning in v if α is the winner of every play in v consistent with σ_α. A node v is won by α if there exists a strategy σ_α that is winning in v. The set of nodes won by α is denoted as $\mathcal{W}_\alpha$.

Strategies, as defined here, are *positional*: they do not depend on the history of the play. Furthermore, both players α have at least one strategy σ_α that is

winning for every node in $\mathcal{W}_\alpha$. Such strategies are *winning*. Most algorithms in this paper compute winning strategies.

2.2 Nested fixpoint iteration for parity games

A basic algorithm for computing the winning positions of a parity game uses a nested fixpoint computation. For a given set T, an operator $\Phi : 2^T \to 2^T$ that is monotone with respect to $\subseteq$ has a least and a greatest fixpoint, denoted $\mu S.\Phi(S)$ resp. $\nu S.\Phi(S)$. The least fixpoint is the limit of the ascending sequence $\emptyset, \Phi(\emptyset), \Phi^2(\emptyset), \ldots$, while the greatest fixpoint is the limit of the descending sequence $T, \Phi(T), \Phi^2(T), \ldots$ For a given operator $\Phi : \left(2^T\right)^2 \to 2^T$ that is monotone in both arguments, it is not difficult to show that the functions $\Psi_\mu : S_2 \mapsto \mu S_1.\Phi(S_2, S_1)$ and $\Psi_\nu : S_2 \mapsto \nu S_1.\Psi(S_2, S_1)$ are monotone functions. Hence, their least and greatest fixpoints are well-defined and can be computed in the standard way, leading to a nested fixpoint computation.

Let Φ be d-ary operator of 2^T that is monotone in all its arguments. Then $\nu S_d.\mu S_{d-1}. \ldots \nu S_2.\mu S_1.\Phi(S_d, \ldots, S_1)$ is a unique and well-defined set that can be computed by a nested least and greatest fixpoint computation. This algorithm was originally used by Emerson and Lei [6] for evaluating μ-calculus formulae in transition systems. For a finite set T, Φ is evaluated $|T|^d$ times by the algorithm in the worst case. An improved algorithm by Long et al. [15] reduced the number of evaluations to $|T|^{\lceil d/2 \rceil}$.

Solving a parity game amounts to computing the positions won by Even and Odd and a winning strategy for both. Walukiewicz [20] characterized the winning positions of Even with a fixpoint calculation $\nu S_d.\mu S_{d-1}. \ldots \mu S_1.\Phi(S_d, \ldots, S_1)$ with

$$\begin{aligned}\Phi(S_d, \ldots, S_1) = \{&v \in V_0 | \exists w : (v, w) \in E \wedge w \in S_i \text{ with } i = Pr(w)\} \cup \\ \{&v \in V_1 | \forall w : (v, w) \in E \Rightarrow w \in S_i \text{ with } i = Pr(w)\}\end{aligned}$$

Since the inner operator Φ is monotone in all of its arguments $S_d, \ldots, S_1$, the corresponding fixpoint computation has a unique and well-defined outcome which is also computed by Algorithm 1. Note that S_i is initialized with respectively V for even i (a greatest fixpoint), and $\emptyset$ for odd i (a least fixpoint).

Bruse et al. [3] observe that it suffices to store elements of priority i in S_i; indeed computing Φ only checks if $w \in S_i$ for $i = Pr(w)$, hence: $\Phi(S_d, \ldots, S_1) = \Phi(S_d^{=d}, \ldots, S_1^{=1})$. Omitting the irrelevant elements, all S_i can be combined in a single set $S = \cup_i S_i^{=i}$; finally, the operation $\Phi(S_d, \ldots, S_1)$ is to be replaced by:

$$\begin{aligned}\phi(S) = \{&v \in V_0 | \exists w : (v, w) \in E \wedge w \in S\} \cup \\ \{&v \in V_1 | \forall w : (v, w) \in E \Rightarrow w \in S\}\end{aligned}$$

A second optimization, developed by Long et al. [15], partially eliminates re-initializations. Instead of resetting all nodes of lower priority than i to their initial state ($S_R^{<i} \leftarrow V^{<i} \cap V^{\equiv_2 0}$), only nodes with a priority of opposite parity are reset (Algorithm 2, Lines 9 to 12): If i is even then the nodes with even

input: A parity game
output: S_d: nodes won by *Even*

Func *basic*$((V, E, V_0, V_1, Pr))$**:**
  for $i \in \{1 \dots d\}$ **do**
    if i *is even* **then** $S_i \leftarrow V$;
    else $S_i \leftarrow \emptyset$;
  do
    $S_\phi \leftarrow \Phi(S_d, \dots, S_2, S_1)$
    $i \leftarrow 1$
    while $S_i = S_\phi \wedge i \leq d$ **do**
      $i \leftarrow i + 1$
    if $i \leq d$ **then**
      $S_i \leftarrow S_\phi$
      for $j \in \{i - 1 \dots 1\}$ **do**
        if j *is even* **then** $S_j \leftarrow V$;
        else $S_j \leftarrow \emptyset$;
  until $i = d + 1$;
  return S_d

Algorithm 1: A basic fixpoint iteration algorithm

input: A parity game
output: S: the nodes won by *Even*

Func *time*$((V, E, V_0, V_1, Pr))$**:**
  $S \leftarrow \{v \in V | Pr(v)$ is even$\}$
  while $S \neq \phi(S)$ **do**
    $S \leftarrow next(S)$
  return S
Func $next(S)$**:**
  $S_\phi \leftarrow \phi(S)$
  $i \leftarrow min\,\{Pr(v) | v \in S \triangle S_\phi\}$
  if i is even **then**
    $S_R \leftarrow S \setminus V^{\equiv_2 1}$
  else
    $S_R \leftarrow S \cup V^{\equiv_2 0}$
  return $S^{>i} \cup S_\phi^{=i} \cup S_R^{\leq i}$

Algorithm 2: The time- and memory-efficient algorithm of Bruse et al. [3]. $\triangle$ is the symmetric set difference.

priority are kept and nodes with an odd priority are removed; analogously, if i is odd, odd nodes are kept and nodes with an even priority are added.

The first optimization reduces the memory use by a factor d, the second is, according to Seidl [17], sufficient to reduce the time complexity to $O(|E| \cdot n^{\lceil d/2 \rceil})$ with $n = |V|/d + 1$. These two optimizations result in Algorithm 2 which calculates the nodes won by Even as S.

The following example shows a simulation of Algorithm 2 for the small parity game in Figure 1.

Example 1. Initially, the nodes won by Even are estimated as $S = \{v_6, v_4\}$. In the first iteration is $S_\phi = \{v_3, v_7\}$ and $S \triangle S_\phi = \{v_3, v_4, v_6, v_7\}$. Node v_3 is the only node of priority $i = 3$. Even wins v_3 by playing to v_4 and S is updated to $\{v_6, v_4, v_3\}$. Five iterations later is S, the current estimate for Even, equal to $\{v_7, v_6, v_4, v_3, v_1\}$. In the next iteration is $S_\phi = \{v_7, v_4, v_3, v_1\}$: node v_6 is won by Odd by playing to v_9. The reset operation removes nodes v_3 and v_1 (and adds v_4). The set S is now $\{v_7, v_4\}$. In the final iterations, Even wins v_1 by playing to v_7, and wins v_3 by playing to v_4. Now, the estimate is $S = \{v_7, v_4, v_3, v_1\}$; as this is a fixpoint, this is the set of nodes won by Even.

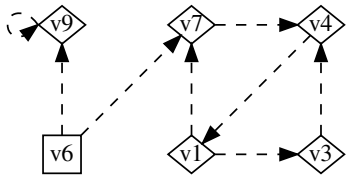


Fig. 1: A small parity game. The priority of node v_i is i. Even plays in diamond nodes and Odd in the square node v_6.

It would be useful if we could construct a winning strategy from the previous calculation. However, this is tricky. E.g., in the final steps we derived that Even wins v_1 using its move to v_7, suggesting that in node v_1 Even should play to v_7. This is wrong: Even must play to v_3 to win.

Bruse et al. [3] then extended Algorithm 2 to compute winning strategies (with domain $\mathcal{W}_\alpha|_\alpha$). The algorithm works by recording for every iteration, for every player, for every node of that player that is supposed to be won by that player in that iteration, a winning move for that node. After the run, for each player one global winning strategy, called the *eventually-positional winning strategy*, is extracted from this data structure. A disadvantage of this method is that its memory use is proportional to the running time. This leads to exponential memory consumption in terms of the parity games' size.

3 Speeding up fixpoint iteration with justifications

The first contribution of this paper is to propose an algorithm that maintains an improved datastructure called a justification graph, to store a single (partial) winning strategy during execution. Justification graphs as we use it here, were first introduced in Hou et al. [11] in the context of a semantic study of nested least and greatest fixpoint definitions.

Below, we view the set S in Algorithm 2 as the current estimate of the nodes won by Even, and $V \setminus S$ as the nodes won by Odd. If $v \in S$, we say that v is won by Even according to S (or Even is the winner of v according to S), and if $v \notin S$, we say that v is won by Odd according to S.

Definition 1 (Justification). *A pair (S, J) is a justification if S is a set of nodes, an estimation of nodes won by Even, and J is a sub-graph of the game graph (V, E), an extended candidate winning strategy, satisfying two constraints:*

i) Each node v won according to S by its owner (i.e., $v \in S|_0 \cup (V \setminus S)|_1$) has either no outgoing edges or one outgoing edge.
ii) Each node v won according to S by the opponent of its owner (i.e., $v \in S|_1 \cup (V \setminus S)|_0$) has either no outgoing edges or all outgoing edges of that node.

If in J, the node v has outgoing edges, we call v *justified* in J, otherwise we call v *unjustified* in J. We denote the set of nodes that are unjustified in J as $U(J)$. A justification is *complete* if every node is justified.

Definition 2 (Winning justification). *A justification (S, J) is winning if all connected nodes are won by the same player according to S and if every infinite path in J starting in a node v is a play won by the winner of v according to S.*

Note that paths in winning justifications are fragments of winning strategies (for Even when it is a path of nodes in S and for Odd when it is a path of nodes outside S).

input: A parity game $G = (V, E, V_0, V_1, Pr)$
output: S: nodes won by Even, J: a justification

Func *compute_J(G)*:
  $S \leftarrow \{v \in V | Pr(v) \text{ is even}\}$
  $J \leftarrow \emptyset$
  while $U(J) \neq \emptyset$ **do**
    $(S, J) \leftarrow next(S, J)$
  return S, J
Func $strategy_\alpha(S, U)$:
  $J \leftarrow \emptyset$
  for $v \in U$ **do**
    if $v \in V_\alpha$ **then**
      $W_v \leftarrow \{w | (v, w) \in E \wedge w \in S\}$
    else
      $W_v \leftarrow \{w | (v, w) \in E \wedge w \notin S\}$
    if $W_v \neq \emptyset$ **then**
      $w \leftarrow$ **choose**(W_v)
      $J \leftarrow J \cup \{(v, w)\}$
    else
      $J \leftarrow J \cup \{(v, w') | (v, w') \in E\}$
  return J

Func *next*(S, J):
  $i \leftarrow min\{Pr(v) | v \in U(J)\}$
  $U \leftarrow U(J)^{=i}$
  $Upd \leftarrow (\phi(S) \triangle S) \cap U$
  if $Upd \neq \emptyset$ **then**
    $R \leftarrow Reaches(J, Upd)$
    if i is even **then**
      $S_R \leftarrow (S \setminus R^{\equiv_2 1}) \cap V^{<i}$
    else
      $S_R \leftarrow (S \cup R^{\equiv_2 0}) \cap V^{<i}$
    $J_t \leftarrow J \setminus (R \times V)$
    $J' \leftarrow J_t \cup strategy_0(S, Upd)$
    $S' \leftarrow S^{>i} \cup (S \triangle Upd) \cup S_R$
  else
    $J' \leftarrow J \cup strategy_0(S, U)$
    $S' \leftarrow S$
  return (S', J')

Algorithm 3: *compute_J* determines winning strategies for both players

Lemma 1. *Let (S, J) be a complete and winning justification, $\sigma_0 = \{v \mapsto w : (v, w) \in J, v \in S|_0\}$ and $\sigma_1 = \{v \mapsto w : (v, w) \in J, v \in (V \setminus S)|_1\}$. Then every play starting in a node $v \in S$ and consistent with σ_0 is an infinite path in J and every play starting in a node $v \notin S$ and consistent with σ_1 is an infinite path in J. Furthermore, S equals $\mathcal{W}_0$, the set of nodes won by Even, and $V \setminus S$ equals $\mathcal{W}_1$, the set of nodes won by Odd and, σ_0 and σ_1 are global winning strategies for respectively Even and Odd.*

Proof. Let v_0 be won by α according to S and let $\pi = v_0 v_1 v_2 \dots$ be an arbitrary play in v_0 consistent with σ_α. Since (S, J) is a complete justification, v_0 is justified in J. Let $v_0 v_1 v_2 \dots v_i$ be an initial segment of π that is a path in J. All connected nodes in J share the same winner according to S, hence, also v_i is won by α according to S. Since (S, J) is complete, if $v_i \in V_\alpha$, v_i has one child in J, namely $\sigma_\alpha(v_i)$. Since π is consistent with σ_α, this node is v_{i+1}. If on the other hand $v_i \in V_{\bar{\alpha}}$, then J contains all outgoing edges from v_i in E, including v_{i+1}. Either way, $v_1 \dots v_{i+1}$ is a path in J. Using induction on i, we conclude that the entire play π is an infinite path in J. Since (S, J) is winning, π is won by α. It follows that σ_α is a winning strategy for α in v_0. Hence, the elements of S belong to $\mathcal{W}_0$ and those of $V \setminus S$ to $\mathcal{W}_1$. We obtain $S = \mathcal{W}_0$ and $V \setminus S = \mathcal{W}_1$.

We now present an improved algorithm that makes use of justifications: Algorithm 3 extends Algorithm 2 with management of such justification graph J. It uses the auxiliaries $U(J)$, the set of unjustified nodes of J, and $Reaches(J, X)$,

the set of nodes that have a path in J to an element of X. It also uses the auxiliary procedure $strategy_\alpha(S, X)$ which chooses justifications for nodes $v \in X \subseteq U$ [1]. The parameter α of the procedure is the player that wins the set S; here this is always Even (Lines 12 and 15 of *next*); in some later algorithms, it can be Odd. For the player that owns an unjustified node $v \in X$, σ_α selects a winning move if one exists (Line 11/13 with Lines 15, 16 of $strategy_\alpha$); otherwise, it selects all moves for v in E (Line 18).

Initially, J contains no edges. Subsequent calls to *next* are made until all nodes are justified. At each iteration, the lowest priority i with non-empty set U of unjustified nodes is determined by Line 2. The subset Upd of nodes with a revised winner is determined $((\phi(S) \triangle S) \cap U)$. If Upd is not empty, the set R of nodes with a path to Upd is computed (Line 6). Such paths represent partial plays to leafs won according to S, but these paths are outdated now. Therefore, the winners of these nodes are reset to their initial values (Lines 7-10) and their justifications are removed (Line 11). It will be proven that nodes in R belong to $V^{<i}$ and that all nodes in R are won by $\alpha = i \bmod 2$ according to S. Thus, all nodes of R belong to S if i is even and to $V \setminus S$ if i is odd. Nodes in R need to be reset to their initial winner: if i is even, it suffices to remove the nodes of R of odd priority ($R^{\equiv_2 1}$) from S; if i is odd, it suffices to add nodes of R of even priority ($R^{\equiv_2 0}$) to S. The set S_R is computed as the result of this update to $S^{<i}$. After resetting the nodes in R, new justifications for the nodes in Upd are computed in Line 12. Finally, the updated set of nodes won by Even, S', is computed by applying the update Upd and by using the reset-update S_R (Line 13).

The other case, when Upd is empty, is simpler: no nodes at level i are revised, S is unchanged and J is extended with justifications for all nodes in U (Line 15).

When the algorithm is finished, the nodes in S are won by Even, non-elements of S are won by Odd and J is a complete justification. For both players, the strategy for a node is given by the unique edge in the justification of that node.

Example 2. We apply Algorithm 3 on the parity game of Figure 1. The first iteration selects a justification for node v_1: v_1 is won by Odd by playing to v_3. The justification J is then $\{(v_1, v_3)\}$. The algorithm then follows Example 1. By choosing[2] node v_7 as winning move for v_6, the algorithm reaches an equivalent state after six iterations: S is $\{v_7, v_6, v_4, v_3, v_1\}$ and the solid edges in Figure 2 express the current justification graph.

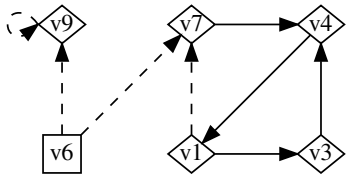


Fig. 2: The justification of Example 2 after 6 iterations .

The next iteration shows the crucial difference between the previous and the new algorithms. Node v_6 is the unjustified node with the lowest priority; it is won by Odd by playing to v_9. The previous algorithm resets nodes v_1

[1] To make $strategy_\alpha$ deterministic, **choose** can return, e.g., the smallest element.

[2] Choosing v_9 as winning move for v_6 never resets node v_6, avoiding the difficulties of Example 1.

and v_3. However, since nodes v_1 and v_3 do not depend on v_6 they are not reset in Algorithm 3. The new justification consists of $S = \{v_7, v_4, v_3, v_1\}$ and $J = \{(v_1, v_3), (v_3, v_4), (v_4, v_1), (v_7, v_4), (v_6, v_9)\}$. The final iteration adds, for node v_9, the following justification to J: (v_9, v_9). The fixpoint has been reached and the final justifaction graph can be projected to a winning strategy. Note that, according to J, in node v_1 Even must play to v_3.

3.1 Correctness

The goal of the algorithm is to find a complete winning justification. We prove the loop invariant that (S, J) is a wining justification at the end of every iteration and that J grows monotonically to a complete justification w.r.t a finite order. To bring this about we prove invariance for three additional properties: the justification (S, J) in Algorithm 3 is coherent, dominating and default.

Definition 3 (Coherent). *A justification (S, J) is* coherent *if two connected nodes in J are won by the same player: $\forall (v, w) \in J : v \in S \Leftrightarrow w \in S$.*

Definition 4 (Default). *The default winner of a node is Even if its priority is even and Odd if its priority is odd. A justification (S, J) is* default *if every unjustified node is won by the default winner of that node according to S : $\forall v \in U(J) : v \in S \Leftrightarrow Pr(v) \equiv_2 0$.*

Definition 5 (Dominating). *A sub-graph J of a parity game is* dominating *if every leaf of J (every $v \in U(J)$) has a priority strictly larger than the priority of nodes on paths to that leaf. Formally: $\forall w \in U(J), \forall v \in Reaches(J, \{w\}) : Pr(v) < Pr(w)$.*

Adding a justification for an unjustified node v can introduce a cycle. If so, dominance of J ensures that the priority of v is the highest priority in the cycle and hence this priority determines the winner of this cycle. Moreover, if (S, J) is default, then the winner of that new cycle is the winner according to S.

The following invariant of Algorithm 3 proves correctness:

Invariant 1 *The pair (S, J) for each iteration of the while-loop is a coherent, dominating, default and winning justification.*

In the proof, we use the notation $J(v) = \{w \mid (v, w) \in J\}$, and likewise $E(v) = \{w \mid (v, w) \in E\}$. Recall that (S, J) is a justification iff for every node $v \in V$, $J(v) = \emptyset$ or v's owner wins in v according to S and $J(v)$ is a singleton subset of $E(v)$ or v's owner looses in v according to S and $J(v) = E(v)$.

Proof. We prove this invariant by induction.

The initial pair $(S, J) = (\{v \in V \mid Pr(v) \text{ is even }\}, \emptyset)$ is, by construction, a winning, coherent, dominating and default justification.

We prove in the induction step that if the justification (S, J) is winning, coherent, dominating and default then so is $(S', J') = next(S, J)$:

Justification: This invariant is at risk at a node v when a justification is set to v, or when v is added or removed from S since this modifies the assigned winner of v. For unjustified nodes v in $U(J)^{=i}$, the calls to $strategy_0$ in Lines 12 and 15 create a correct justification for v for its winner according to $\phi(S)$. As for updates of S, this occurs for nodes v in Upd and in the reset nodes in R. For $v \in Upd$, the call to $strategy_0$ creates a correct justification for v while for reset nodes $v \in R$, the justification of v is removed in Line 11 which also preserves the invariant. As such (S', J') is a justification.

Coherent: It is easy to check that the edges returned by $strategy_0$ are coherent with $\phi(S)$. Thus the edges added to J will be coherent since $S' \cap U$ is exactly $\phi(S) \cap U$. We still need to prove that edges (v, w) in J preserved in J' remain coherent. This follows from the fact if the edge (v, w) is preserved in J, then also the winners of v and w according to S must be preserved. Indeed, whenever v or w is assigned a different winner, v ends up in R and its edge to w is removed.

Default: Initially, all nodes are unjustified and won, according to S, by their default winner. When the winner of an unjustified node changes according to S, then the node becomes justified. When a node becomes unjustified, the winner is reset to the default winner.

Dominating: If Upd is non-empty, then first for nodes v in $Reaches(J, Upd)$, the justification is removed and each becomes an unreachable leaf of J' (edges from and to v are removed). This preserves the invariant. Then for every node v in Upd, justifying edges (v, w) are created where w is won by the opposite player according to S. Since (S, J) is coherent and default, all reachable unjustified nodes from w are won by the opposite player, hence have a priority different than i and hence, strictly larger than i (i is the least priority with unjustified nodes). Thus, the newly reachable unjustified nodes from v, and from nodes that can reach v (which by dominance of J have priority $< i$), have priority $> i$. Hence, dominance is preserved. If Upd is empty, all nodes v with priority i are assigned a justification, and all newly reachable unjustified nodes from v and from nodes that could reach v have strictly higher priority than i. Again, dominance is preserved.

Winning: The first condition, all connected nodes have the same winner according to S', follows from coherence. The second condition remains to be proven: Let π be an infinite path of J' consisting of nodes won by α according to S'. It must be proven that α is the winner of π.

Assume π has an index j such that the tail $\langle v_j, v_{j+1}, \dots \rangle$ is a path in J. Since (S, J) is a winning justification, α is the winner of the tail, and hence of π itself.

Assume π does not have a tail preserved from J. Then π passes infinitely often over new edges $(v, w) \in J' \setminus J$, where $v \in U(J)$ and v has priority i. In the play π there exist two types of nodes: nodes unjustified in J, of priority i, and nodes justified in J. The justified nodes have a path in J along π to the next unjustified node which has priority i. Since J is dominating, it follows that these justified nodes have a priority less than i. So the highest priority of the play that occurs infinitely often is i and π is won by player i mod 2. From the proof of dominating, it follows that truly new infinite paths are only possible when Upd

is empty. Since Upd is empty, player i mod 2 is, according to S', the winner of all nodes on π. Thus player α wins the play π. Therefore, (S', J') is a winning justification.

To ensure termination, we define the size of a justification and argue that the size decreases in every cycle until the justification is complete.

Definition 6 (Justification size). *The size of a justification, denoted* $size(J)$*, is a* d*-tuple* $(a_d, a_{d-1}, \ldots, a_1)$ *where each* a_i *counts the number of unjustified nodes of priority* i*. Justification sizes are ordered lexicographically:* $(a_d, a_{d-1}, \ldots, a_1) < (b_d, b_{d-1}, \ldots, b_1)$ *iff* $\exists i : a_i < b_i \wedge \forall j > i : a_j = b_j$.

Theorem 1 (Total correctness). *Algorithm 3 terminates with a complete winning justification* (S, J).

Proof. If Upd is empty, the algorithm strictly reduces the number of unjustified nodes of priority i to zero. Otherwise, if Upd is not empty, the algorithm strictly reduces the number of unjustified nodes at priority i (and may increase the number of unjustified nodes at strictly lower levels). Either way, the size of J strictly decreases at each iteration. Hence, the algorithm terminates.

Furthermore, the algorithm terminates when no unjustified nodes exist. At this point, (S, J) is complete.

No improvements in time complexity are expected:

Theorem 2 (Time complexity). *Algorithm has time complexity* $O\left(|E| \cdot n^{\lceil d/2 \rceil}\right)$ *with* $n = |V|/d + 1$.

Proof omitted

4 Integrating justifications in other algorithms

Currently, three algorithms are considered as state-of-the-art: Zielonka's algorithm [21], priority promotion [1, 2] and tangle learning [19]. While studying related work the applicability of justifications to these algorithms was noticed: while all algorithms calculate a winning strategy, only region recovery, a priority promotion variant, and tangle learning use this strategy to improve the algorithm. In this section we sketch how to extend Zielonka's algorithm and tangle learning with justifications. While justifications do not improve the time complexity, one can expect a performance improvement. This is confirmed by the experimental evaluation. In the future work section we argue that applying justifications to priority promotion is an improvement of the region recovery variant.

Zielonka's algorithm Zielonka's algorithm determines the winners by recursively decomposing a game G in a smaller sub-game G' and first solving the sub-game. Once the winners of G' are determined, it returns to the nodes not in G' and, if needed, another sub-game is created and solved.

The algorithm is based on the notion of the *attracted node set* of a set S, denoted $Attr_\alpha(S)$. Informally, this is the set of all nodes from which α can force a play to S. Formally, this set can be computed using a least fixpoint computation:

$$\begin{aligned} Attr_\alpha(V, S) = \mu A.S \cup \{v \in V_\alpha | \exists w : (v, w) \in E \wedge w \in A\} \\ \cup \{v \in V_{\bar{\alpha}} | \forall w : (v, w) \in E \Rightarrow w \in A\} \end{aligned}$$

In addition, Zielonka's algorithm computes a corresponding α-strategy $\sigma_{Attr} : A|_\alpha \setminus S \to A$ for all attracted nodes which shows how any play starting in a node $v \in A$ eventually ends in S.

For correctness, Zielonka's algorithm depends on two properties of $Attr_\alpha$: (i) if all nodes of S can be won by α then all attracted nodes are won by α and (ii) if all nodes in S have priority $p \equiv_2 \alpha$ and all attracted nodes have a lower (or equal) priority then all moves from S to $Attr_\alpha(S)$ are won by α.

For a game G and a set of nodes A we define $G \setminus A$ as the removal of A from all parts of G: the nodes, and edges, owners, and priorities.

In the original algorithm, Algorithm 4, empty games are immediately solved. For non-empty games, the maximal priority p is calculated. Then the algorithm calculates the set A together with a strategy σ_A for all nodes attracted to the *heads*, i.e., nodes with the maximal priority, $V^{=p}$. A play π starting in a head and consistent with σ_A has two options: either π stays in A and forms an infinite play with highest priority p, won by α, or π escapes to an unattracted node in $V \setminus A$. To finally determine the winner the sub-game $G \setminus A$ is recursively solved to determine W'_0 and W'_1 (Line 7). If all nodes in this sub-game are also won by α, then, in this case, all nodes of this game are won by α. All that remains is finding a strategy for the heads since the moves for these nodes are explicitly not included in σ_A (Line 10). In the other case, some nodes are won by the opponent: heads may be attracted or forced to $W'_{\bar{\alpha}}$ and consequently attract more nodes of A (Line 14). These nodes are attracted to a set B and removed to create a second sub-game $G \setminus B$. The solution of this sub-game together with nodes B won by $\bar{\alpha}$ and the corresponding strategy σ_B form the final solution of G.

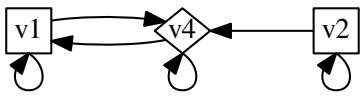


Fig. 3: A small parity game. Nodes v_i have priority i. Even plays in v_4. Odd plays in v_1 and v_2.

One weakness of the algorithm is that it resets all attracted nodes if a single node is attracted to the opponent. For sufficient complex game this results in an increased number of solved sub-games:

Example 3. We simulate Algorithm 4 for the parity game in Figure 3. Initially we solve the game with all three nodes. v_4 does not attract. This node is removed and the sub-game with nodes v_1, v_2 is solved. v_2 does not attract v_1; it is removed to solve the sub-game with one node, v_1. v_1 is the last node, the empty sub-game is solved trivially.

For the sub-game with only v_1, no nodes are won by $W_{\bar{\alpha}}$, v_1 is won by Odd by playing to itself. For the sub-game with v_1 and v_2, v_1 is won by $W_{\bar{\alpha}}$, v_1 does not attract v_2 thus the sub-game with only v_2 is solved. The steps of solving this game are skipped, v_2 is won by Even. For the sub-game with v_1 and v_2, v_1 is won by Odd, v_2 is won by Even. In the sub-game of v_1, v_2 and v_4, $W_{\bar{\alpha}}$ contains v_1. v_2 and v_4 are not attracted to v_1 thus the sub-game with v_2 and v_4 is solved. v_4 does not attract v_2, the sub-game with only v_2 is solved again. For the sub-game with v_2 and v_4, v_4 wins by playing to itself, v_2 is won by *Even* with either move. The game is solved: v_1 is won by Odd, v_2 and v_4 are won by Even.

This example shows that the sub-game with only v_2 is solved multiple times with the same results. Moreover, there was no reason to solve the sub-game with v_1 and v_2. Using justifications, we can do better by further partitioning $A \setminus B$ with the help of σ_A: the nodes that depended on nodes in B and safe nodes. Only the former nodes need to be recalculated in the new sub-game.

Algorithm 5 integrates a justification graph. The essential difference between the two algorithms is Line 19. To make this line useful the algorithm needs to be reformulated. First, the strategy variables σ_0, σ_1 are extended and merged into a single justification graph J which allows to easily calculate reachability. Second, the recursive tail-call in Algorithm 4 at Line 15 is transformed into a loop which is interrupted when $W'_{\bar{\alpha}} = \emptyset$. The iterative representation allows for a stateful algorithm to recover and modify previously calculated information.

After applying these two changes, the difference between the original algorithm and the justification variant is a single line: if R is overestimated as $W_\alpha^{\neq p}$ on Line 19 the behaviour of the algorithm reverts to the original Zielonka algorithm. In the justification variant of the algorithm the algorithm must, after attracting for the opponent (Line 18), create a new sub-game. It first calculates the set of nodes R that depended on nodes that changed winner (Line 19). Subsequently Line 20 fixes the justification J : it removes the justifications for all nodes in R and adds the justifications for nodes attracted for $\bar{\alpha}$. Then it removes these nodes from the set W_α, as the properties of *Attr* have been lost for nodes in R, while adding the attracted nodes to $W_{\bar{\alpha}}$. Now the loop is ready to restart with first attracting reset nodes that have a different way of playing to W_α and then solving the sub-game of nodes that are still not attracted.

As for correctness, similarly to Algorithm 3, the pair (S, J) with $S = W_0 \cup (V^{\equiv_2 0} \setminus W_1)$ remains a coherent, dominating, default and winning justification while the size of the justification increases every iteration of the while-loop at Line 8. We omit further details.

input: A parity game $G = (V, E, V_0, V_1, Pr)$
output: W_0, W_1: winning nodes, σ_0, σ_1: strategies

Func *zielonka*(G)**:**
  if $V = \emptyset$ **then**
    return $(\emptyset, \emptyset, \emptyset, \emptyset)$
  $p \leftarrow max\{Pr(v) | v \in V\}$
  $\alpha \leftarrow p \bmod 2$
  $A, \sigma_A \leftarrow Attr_\alpha(V^{<p}, V^{=p})$
  $W_0, W_1, \sigma_0, \sigma_1 \leftarrow zielonka(G \setminus A)$
  if $W_{\bar{\alpha}} = \emptyset$ **then**
    $W_\alpha \leftarrow W_\alpha \cup A$
    $\sigma_F \leftarrow strategy_\alpha(W_\alpha, V^{=p} \cap V|_\alpha)$
    $\sigma_\alpha \leftarrow \sigma_\alpha \cup \sigma_A \cup \sigma_F$
    return $W_0, W_1, \sigma_0, \sigma_1$
  else
    $B, \sigma_B \leftarrow Attr_{\bar{\alpha}}(V, W_{\bar{\alpha}})$
    $W'_0, W'_1, \sigma'_0, \sigma'_1 \leftarrow zielonka(G \setminus B)$
    $W'_{\bar{\alpha}} \leftarrow W'_{\bar{\alpha}} \cup B$
    $\sigma'_{\bar{\alpha}} \leftarrow \sigma'_{\bar{\alpha}} \cup \sigma_B$
    return $W'_0, W'_1, \sigma'_0, \sigma'_1$

Algorithm 4: Zielonka's algorithm

input: A parity game $G = (V, E, V_0, V_1, Pr)$
output: W_0, W_1: winning nodes, J: justification

Func *zielonka_j*(G)**:**
  if $V = \emptyset$ **then**
    return $(\emptyset, \emptyset, \emptyset)$
  $p \leftarrow max\{Pr(v) | v \in V\}$
  $\alpha \leftarrow p \bmod 2$
  $W_\alpha \leftarrow V^{=p}, W_{\bar{\alpha}} \leftarrow \emptyset$
  $R \leftarrow V^{<p}, J \leftarrow \emptyset$
  while *true* **do**
    $A, J_A \leftarrow Attr_\alpha(R, W_\alpha)$
    $W_\alpha \leftarrow W_\alpha \cup A$
    $G' \leftarrow (G \setminus W_\alpha) \setminus W_{\bar{\alpha}}$
    $W'_0, W'_1, J_Z \leftarrow zielonka\_j(G')$
    $(W_0, W_1) \leftarrow (W_0 \cup W'_0, W_1 \cup W'_1)$
    $J \leftarrow J \cup J_Z \cup J_A$
    if $W'_{\bar{\alpha}} = \emptyset$ **then**
      $J_F \leftarrow strategy_\alpha(W_\alpha, W_\alpha \cap V^{=p})$
      return $(W_0, W_1, J \cup J_F)$
    $B, J_B \leftarrow Attr_{\bar{\alpha}}(V, W_{\bar{\alpha}})$
    $R \leftarrow Reaches(J, B)$
    $J \leftarrow (J \setminus (R \times V)) \cup J_B$
    $W_\alpha \leftarrow W_\alpha \setminus R, W_{\bar{\alpha}} \leftarrow W_{\bar{\alpha}} \cup B$

Algorithm 5: Zielonka's algorithm with justifications

Tangle learning Tangle learning [18, 19] is a recent state-of-the-art algorithm based on deriving and attracting tangles. Tangles are sub-games won by a single player α with a strongly connected strategy:

Definition 7 (Tangles). *An α-tangle is a set of nodes $\tau \subset V$ for which there exists a tangle-strategy $\sigma : \tau|_\alpha \to \tau$ such that the graph (τ, E_τ) with $E_\tau = E \cap (\sigma \cup (\tau|_{\bar{\alpha}} \times \tau))$ has at least one edge, is strongly connected, and all cycles within (τ, E_τ) are won by α.*

A play starting in a node of an α-tangle can either stay within the tangle or $\bar{\alpha}$ can escape to a node in $Ext(\tau) = \{w | (v, w) \in E, v \in \tau|_{\bar{\alpha}}, w \notin \tau\}$. Thus if all escapes are won by α then all nodes in the α-tangle are won by α.

Integrating the tangles involves an extension to the attraction function used by Zielonka's algorithm. Recall that nodes are attracted for a player α if that player can force a play to a given set, rules (i) and (ii). For tangle learning the property is extended to either playing to the given set or α wins the infinite play. This property is satisfied for α-tangles: all nodes of an α-tangle are attracted if all escapes of the tangle are attracted, rule (iii).

Formally, given the set of tangles T (with the α-tangles of T denoted as $T|_\alpha$), the set of nodes V, the set of edges E, the subset of nodes attracted to S is:

$$
\begin{aligned}
&TAttr_\alpha(T, V, E, S) = \mu A.\ S \\
&\qquad \cup \{v \in V_\alpha | \exists w : (v, w) \in E \wedge w \in A\} && (i) \\
&\qquad \cup \{v \in V_{\bar{\alpha}} | \forall w : (v, w) \in E \Rightarrow w \in A\} && (ii) \\
&\qquad \cup \{v \in t \cap V | t \in T|_\alpha \wedge (Ext(t) \cap V) \neq \emptyset \wedge (Ext(t) \cap V) \subseteq A\} && (iii)
\end{aligned}
$$

The second part of the algorithm consists of learning tangles. They are found after attracting: Assume $A, \sigma_A = TAttr_\alpha(T, V, S)$. If a node $v \in S_\alpha$ can play within A or if a node $v \in S_{\bar{\alpha}}$ must play within A then the possibility of a cycle exists. Furthermore, if all nodes in S have the same priority $p \equiv_2 \alpha$ and all attracted nodes A have a lower priority then all cycles are won by α. The tangle learning algorithm will only attract nodes and tangles of a lower priority by invariant. Thus every non-trivial cycle will be an α-tangle. The problem then consists of determining the non-trivial strongly connected components in (A, E_A) with $E_A = E \cap (\sigma_A \cup (A_{\bar{\alpha}} \times A))$.

Tangle learning consists of first finding tangles and then, to progress, attracting them. If a tangle is used a second time, some work is saved. The more complex the tangle, the more work is saved. First a tangle learning algorithm without justifications is shown, then a new variant with justifications is discussed.

The original tangle learning algorithm (Algorithm 6). First, we explain some key variables: (i) tangles found so far: T, (ii) new tangles to be used in the next iteration, Y, and (iii) the region mappings: L. The solution of the parity game is accumulated in the output variables $W_0, W_1, \sigma_0, \sigma_1$.

The region mapping, $L : V \to P$, is a function that determines the priority to which every node is currently attracted. The notation $(L \setminus A) \cup (A \mapsto p)$ is used to map the value of the nodes in the set A with p. For a region mapping $L : V \to P$, a relation in the set $\{=, \neq, <, >, \equiv_2, \not\equiv_2\}$ and a priority $p \in P$ the notation $L^{-1}_{\sim p}$ denotes $\{v \in V | L(v) \sim p\}$.

The core loop of the algorithm runs from Lines 6 to 21: until all nodes have a region, the algorithm selects the highest priority among the nodes without a region, extends the set of nodes with this priority with all nodes attracted to them and computes all tangles in this set. The final winner can be assigned to such a tangle in case it has no externals (the for-loop at line 15) and the involved nodes will be removed from the game before the next iteration starts; other tangles are added to the set Y and all involved nodes are assigned region p. When all nodes have been assigned a region, at least one tangle has been found. The new tangles in Y are then "promoted" to the set T and the algorithm starts a new iteration with resetting L.

Similar to *Attr*, *TAttr* records how nodes are attracted as a strategy σ_A or as part of a justification J_A. The strategy for attracted nodes is unchanged while the strategy for attracted nodes of a tangle is determined by σ_t.

Given the strategy σ_A, the procedure *extract_tangles* returns the set of all promotable tangles within A. It obtains them by calculating all strongly connected components with, e.g., Tarjan's strongly connected component algorithm.

input: A parity game $\mathrm{G} = (V, E, V_0, V_1, Pr)$
output: W_0, W_1: winning nodes, σ_0, σ_1: strategies

1 **Func** *tangle_learning*(G)**:**
2 $\quad T \leftarrow \emptyset$
3 $\quad W_0 \leftarrow \emptyset, W_1 \leftarrow \emptyset, \sigma_0 \leftarrow \emptyset, \sigma_1 \leftarrow \emptyset$
4 $\quad$ **while** $V \neq \emptyset$ **do**
5 $\quad\quad L \leftarrow (V \mapsto \emptyset),\ Y \leftarrow \emptyset$
6 $\quad\quad$ **while** $L^{-1}_{=\emptyset} \neq \emptyset$ **do**
7 $\quad\quad\quad V' \leftarrow V \cap L^{-1}_{=\emptyset}$
8 $\quad\quad\quad E' \leftarrow E \cap (V' \times V')$
9 $\quad\quad\quad p \leftarrow max\,\{Pr(v) | v \in V'\}$
10 $\quad\quad\quad \alpha \leftarrow p \bmod 2$
11 $\quad\quad\quad A, \sigma_A \leftarrow TAttr_\alpha(T, V', E', V'^{=p})$
12 $\quad\quad\quad L \leftarrow L \cup (A \mapsto p)$
13 $\quad\quad\quad New \leftarrow extract\_tangles(A, \sigma_A)$
14 $\quad\quad\quad Dom \leftarrow \{t \in New | E_T(t) = \emptyset\}$
15 $\quad\quad\quad$ **for** $(t, \sigma_t) \in Dom$ **do**
16 $\quad\quad\quad\quad D, \sigma \leftarrow TAttr_\alpha(T, V, t)$
17 $\quad\quad\quad\quad W_\alpha \leftarrow W_\alpha \cup D$
18 $\quad\quad\quad\quad \sigma_\alpha \leftarrow \sigma_\alpha \cup \sigma$
19 $\quad\quad\quad\quad L \leftarrow L \setminus D$
20 $\quad\quad\quad\quad V \leftarrow V \setminus D$
21 $\quad\quad\quad Y \leftarrow Y \cup (New \setminus Dom)$
22 $\quad\quad T \leftarrow T \cup Y$
23 $\quad\quad$ **return** $W_0, W_1, \sigma_0, \sigma_1$

Algorithm 6: Tangle learning

input: A parity game $G = (V, E, V_0, V_1, Pr)$
output: W_0, W_1: winning nodes, J: justification

1 **Func** *tl_just(G)*:
2 $\quad L \leftarrow (V \mapsto \emptyset), T \leftarrow \emptyset$
3 $\quad W_0 \leftarrow \emptyset, W_1 \leftarrow \emptyset, J \leftarrow \emptyset$
4 $\quad$ **while** $V \neq \emptyset$ **do**
5 $\quad\quad Y \leftarrow \emptyset$
6 $\quad\quad$ **while** $impr(V, T, L) \neq \emptyset$ **do**
7 $\quad\quad\quad V' \leftarrow V \cap L^{-1}_{\leq p}$
8 $\quad\quad\quad E' \leftarrow E \cap (V' \times V')$
9 $\quad\quad\quad p \leftarrow max(impr(V, T, L))$
10 $\quad\quad\quad \alpha \leftarrow p \bmod 2$
11 $\quad\quad\quad H \leftarrow L^{-1}_{=p} \cup (L^{-1}_{=\emptyset} \cap V^{=p})$
12 $\quad\quad\quad A, J_A \leftarrow TAttr_\alpha(T, V', E', H)$
13 $\quad\quad\quad R \leftarrow Reaches(J, A \cap L^{-1}_{<p, \not\equiv_2 p})$
14 $\quad\quad\quad J \leftarrow (J \setminus ((R \cup A) \times V)) \cup J_A$
15 $\quad\quad\quad L \leftarrow (L \setminus R) \cup (R \mapsto \emptyset)$
16 $\quad\quad\quad L \leftarrow (L \setminus A) \cup (A \mapsto p)$
17 $\quad\quad\quad New \leftarrow extract\_tangles(A, J_A)$
18 $\quad\quad\quad Dom \leftarrow \{t \in New | E_T(t) = \emptyset\}$
19 $\quad\quad\quad$ **for** $(t, \sigma_t) \in Dom$ **do**
20 $\quad\quad\quad\quad D, J_D \leftarrow TAttr_\alpha(T, V, t)$
21 $\quad\quad\quad\quad W_\alpha \leftarrow W_\alpha \cup D$
22 $\quad\quad\quad\quad J_\alpha \leftarrow J_\alpha \cup J_D$
23 $\quad\quad\quad\quad L \leftarrow L \setminus D, V \leftarrow V \setminus D$
24 $\quad\quad\quad Y \leftarrow Y \cup (New \setminus Dom)$
25 $\quad\quad T \leftarrow T \cup Y$
26 $\quad\quad$ **return** W_0, W_1, J

Algorithm 7: Tangle learning improved with justifications

A tangle is promotable if the regions of all its externals are larger than the priority of the tangle. Unpromotable tangles are not returned by *extract_tangles*.

Like Zielonka's algorithm, tangle learning removes all information depending on faulty assumptions; indeed, it empties the region information L (Line 5). By adding justifications, we can do better.

Tangle learning with justifications (Algorithm 7). Like the previous algorithms, the changes are centred around the reset-phase. This variant makes full use of the region information L. This information is never fully reset; instead, *impr* determines outdated information and the justification J is used to selectively withdraw invalid information.

The function *impr* determines for which nodes and tangles the region L can be improved. If no such nodes exist, either the algorithm is finished or some

tangles must have been found which are not yet available for attraction; the inner while loop is exited, the new tangles are added to T and their region is reset to $\emptyset$ (Line 25). Otherwise we are interested in the nodes and tangles available for improvement that have the maximal priority p.

Line 9 identifies the maximal priority for which improvements are possible, Line 11 determines which nodes can be improved and Line 12 determines the attracting nodes. After resetting the nodes that become invalid (Lines 13 to 15), the improved region is assigned for all attracted nodes (Line 16). At this point, the set of nodes with region p is the same as in the non-justification algorithm (given the same set of tangles T). However, this does not guarantee that the set of extracted tangles will be the equal as it is possible that the chosen strategy σ_A and justification J differ.

As for correctness, similarly to Algorithm 3, the pair (S, J) with $S = W_0 \cup L^{\equiv_2 0} \cup (V^{\equiv_2 0} \setminus L^{\equiv_2 1} \setminus W_1)$ remains a coherent, dominating, default and winning justification while the size of the justification increases every iteration of the while-loop at Line 8. We omit further details.

Overall, the algorithm improves two facets of the attraction: first, unchanged regions are skipped and secondly even when extending the region, nodes already belonging to the region are not re-attracted.

5 Experimental results

In the literature, we selected three parity game solving projects:

- PGSolver [8, 9] (`github.com/tcsprojects/pgsolver`) is a collection of tools for solving parity games written in OCaml. We refer to its algorithms with `pgsolver-*`.
- Oink [18, 19] (`github.com/trolando/oink`) is a recent parity game solver suite written in C++. We refer to its algorithms with `oink-*`.
- pbespgsolver (`www.mcrl2.org`) is part of the mCRL2 toolset [5], centred around formal verification of automata. We refer to its algorithms with `pbes-*`.

Each project implements several algorithms, among them are fixpoint iteration (referred as `*-fp`, Zielonka's algorithm (referred as `*-zlk`), priority promotion (referred as `*-pp`) and tangle learning (referred as `*-tl`). For example, with `oink-tl` we refer to the tangle learning algorithm in the Oink solver. In total, 13 algorithms are benchmarked, 8 from the above three solvers and 5 algorithms we implemented. One of them, `oink-zlk-just`, is a modification of the implementation of Zielonka's algorithm in Oink. The fixpoint iteration algorithm is only implemented in PGSolver which, as Table 1 shows, has the worst performance of all three solvers. So it looks as a bad idea to modify that implementation. Therefore, we implemented our own version of the fixpoint algorithm (`prty-fp`) and an extension with justifications (`prty-fp-just`). As for priority promotion and tangle learning, we did not have the time to extend both with justifications. We selected the best performing one, tangle learning. However, extending the

Configuration	Both benchmarks (1297 instances)			
6GB	<1s	<10s	<100s	par2 (s)
prty-tl-just	976	1215	1289	4912
prty-tl	959	1207	1288	5729
oink-tl	947	1209	1277	7193
oink-zlk-just	947	1206	1260	10464
oink-pp	952	1188	1245	13295
oink-zlk	932	1186	1239	14471
pbes-pp	919	1141	1205	21406
pbes-zlk	860	1072	1140	34437
prty-fp-just	863	1094	1134	34866
pgsolver-pp	698	1034	1138	36453
pgsolver-zlk	625	910	1035	56824
prty-fp	590	780	845	92529
pgsolver-fp	532	743	807	100616

Table 1: Number of solved instances in less than 1, 10, and 100 seconds and par2 score (lower is better), sorted by the par2 score

Oink implementation was too involved: tangle learning is significantly more complex than Zielonka's algorithm. We started from scratch, implementing a baseline `prty-tl` and an extension with justifications (`prty-tl-just`). All `prty-*` solvers are implemented in Rust and are available at `bitbucket.org/krr/prty`.

In literature, we found information on two benchmarks. The parity game benchmark [14] and one with large random graphs used for benchmarking tangle learning [18]. These benchmarks[3] have been executed for the selected algorithms with a time-limit of 100 seconds and a memory limit of 6 GB on a PC with an Intel 'i7-4770' CPU. A summary of the results for these benchmarks is reported in Table 1: the number of games solved within 1 second, 10 seconds, and within the time-limit. The par2 score is also reported, it is the sum of the run-times of all solved games and a penalty of twice the time-limit (200 seconds) for each unsolved game.

This benchmark ranks the base algorithms as follows: tangle learning performs best, priority promotion performs better than Zielonka's algorithm and fixpoint iteration is the worst algorithm. The frameworks can also be ranked by comparing different implementations of the same algorithm: Oink performs best, then pbespgsolver second-best and PGSolver ranks last. More interesting is the effect of adding justifications to various algorithms. The table shows that each justification variant performs better than the corresponding base algorithm.

The fixpoint algorithm `prty-fp-just` outperforms `prty-fp` with 289 additional solved instances. As it ranks below the best algorithms, we do not analyse it in more detail but focus on the other two implementations with justifications.

[3] The benchmarks can be reproduced in the VMCAI 2020 virtual machine (https://doi.org/10.5281/zenodo.3533104) with the artifact at https://doi.org/10.5281/zenodo.3510292.

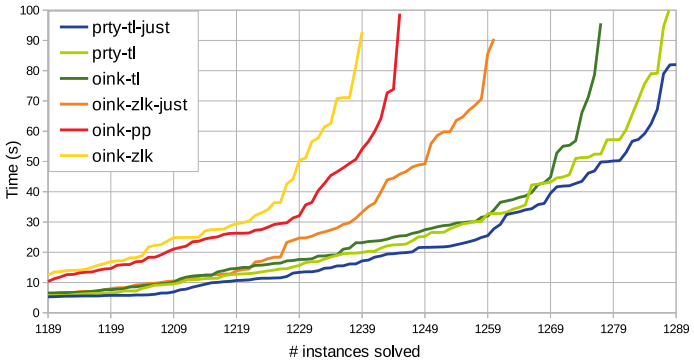
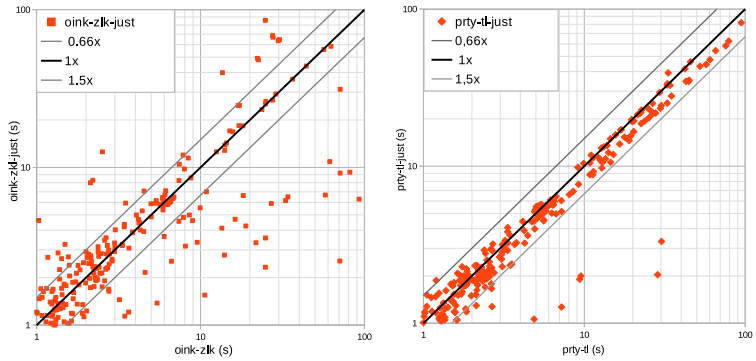


Fig. 4: Cactusplot showing runtime of the slowest 100 parity games for the top six algorithms

Figure 4 is a cactusplot of the 100 most time consuming instances; it zooms in on the top six algorithms. The figure shows that adding justifications results in a clear performance inprovement for Zielonka's algorithm (`oink-zlk` vs `oink-zlk-just`) and tangle learning (`prty-tl` vs `prty-tl-just`). Both the table and the figure show that our baseline tangle learning algorithm, `prty-tl`, performs better than `oink-tl`. Further analysis learned us that the cause is a solvable memory inefficiency in Oink. To measure the effect more in detail we considered the 1277 instance solved by both versions: `oink-tl` needs 2626 seconds to solve them while our baseline, `prty-tl`, only needs 2541 seconds.

Figure 5 zooms in on the differences in run-time for all instances solved by both versions of an algorithm. Below the black line are instances for which the justification version is faster; above the line those for which the justification version is slower. Figure 5a compares `oink-zlk` with `oink-zlk-just`. While there are examples with a large speed-up, there are also examples with a significant slowdown. We assume that most of the variability is caused by the explicit representation of the justification graph that uses hashing for direct access to the inverse of the justification function. Indeed, performance analysis showed that a major chunk of time is spent maintaining this data structure. Overall, there are more examples with a significant speed-up than with a significant slowdown and the overall time for solving the 1233 common examples is reduced from 2629 s to 2293 s. Considering that `oink-zlk-just` solves 21 extra instances, we conclude that adding justifications improves Zielonka's algorithm.

As the explicit justification graph used in Zielonka's algorithm caused a lot of variability in the run-time of individual instances, we opted for an implicit representation of the justification graph in tangle learning; it derives the justification graph from the existing information about nodes. Figure 5b, which compares `prty-tl` with `prty-tl-just`, indeed shows much less variability and we see, for instances requiring more than 10 s, a modest but almost consistent



(a) Comparison for Zielonka's algorithm

(b) Comparison for tangle learning

Fig. 5: Comparisons of run times

improvement. The overall time to solve the 1277 instances is reduced from 2541 s to 2159 s which is a 15% improvement.

The figure also shows a few examples with a significant speed-up while none with a significant slowdown. It turns out all these instances are so called Jurdzinski games [12]. A detailed profiling learned us that up to 80% of the time is spent on attracting nodes and that justifications reduce the number of attracted and reset nodes in Jurdzinski games with several orders of magnitude. Table 2 shows details about the Jurdzinski game instances as well as some other hard instances. It also includes the 500 by 500 Jurdzinski game, the most difficult Jurdzinski game in the data set, which is not part of Figure 5b as it requires more than 100 s to solve. In this game, the number of attracted nodes is reduced from 126M to 501k nodes and the run-time from 190 seconds to less than 8 seconds. The other instances in Table 2 are representative for the instances in the top right of Figure 5b. Justifications reduce the number of attracted nodes by only a fraction of the total. Still, there are a small improvements in run-time. We conclude that justifications are a significant improvement for tangle learning.

Conclusion. Our experiments demonstrate that adding a justification graph to a parity game solver algorithm is beneficial to the performance. Also, we have shown that `prty-tl-just`, a tangle learning solver with justifications, is improving upon the state of the art.

6 Conclusion

In this paper we explored the use of justification graphs in parity game solvers. We started with the nested fixpoint induction algorithm for parity games [3]. Besides storing the winning strategy of a node, the justification can also be used

instance	runtime (s)		# attracted nodes	
	prty-tl	-just	prty-tl	-just
jurdzinskigame(500,50)	2.8	0.7	663k	51k
jurdzinskigame(500,100)	7.2	1.3	2.6M	101k
jurdzinskigame(100,500)	9.4	1.9	13M	101k
jurdzinskigame(500,200)	28.6	2.0	10M	201k
jurdzinskigame(200,500)	30.2	3.3	25M	201k
jurdzinskigame(500,500)	(190)	7.7	63M	502k
ABP_Onebit_(3,1,1,weak-bisim)	70.6	66.7	177M	40M
rn-1000000-1000000-1-2-4	75.8	58.4	163M	157M
rn-1000000-1000000-1-2-9	79.0	62.6	168M	164M
SWP(4,3,infinitely_often_rw)	79.1	78.9	104M	27M
rn-700000-700000-1-2-5	94.6	81.9	217M	235M
rn-700000-700000-1-2-4	99.4	81.4	224M	220M

Table 2: The reduction of runtime is coupled to a reduction in number of attracted nodes

to save work. Indeed, upon each update of winning nodes, the basic algorithm resets all lower priority nodes to the static overestimation of their winner. The justification allows one to dynamically select the nodes for which the current winner is invalidated and only resets those nodes to the default assumption of their winner. Experimental evaluation showed that this results in a substantial speed-up of the algorithm.

Encouraged by these results we also explored the use of justifications in other algorithms. Our analysis learned us that also Zielonka's algorithm [21], priority promotion [1, 2] and tangle learning [18] can potentially benefit from justifications as it allows one to reset fewer nodes to their default settings at the beginning of a new iteration. So far we could only implement two of these three algorithms, namely Zielonka's algorithm and tangle learning. Our evaluation meets our expectations, for both algorithms we obtained a good performance improvement for the whole of our benchmarkset. Moreover, our best algorithm improves upon the state of the art.

Future work. For Zielonka's algorithm, we expect that replacing the explicit justification graph with an implicit one will improve the performance and will result in a more consistent speed-up over all instances of the benchmark.

We predict that adding justifications to priority promotion speed up the algorithm. Moreover, the region-recovery variant of priority promotion shows parallels to justification-based resetting: it uses the witness strategy to safeguard regions (a set of nodes) if none of the nodes depend on a reset region. This is, however, coarser than justifications: if a single nodes depends on a reset region then all nodes in that region are reset which cascades for all dependent regions. We predict that games improved by region recovery are also improved when using a justifications-based algorithm.

There is a need to develop a benchmark set with harder instances: The best algorithm solves 1289 out of 1297 instances and solves all instances in 400 seconds with 15 GB of memory. Furthermore, only 82 instances need more than 10 seconds to solve.

Finally, one can imagine that justifications are excellent for incremental parity game solving. When a small part of a parity game is revised, justifications allows one to identify the affected part of the solution, to reset the nodes in that part to the default winners and to start the search for a solution form there. So far we are not aware of applications that could use such an incremental parity game solver.

References

[1] Massimo Benerecetti, Daniele Dell'Erba, and Fabio Mogavero. "Improving Priority Promotion for Parity Games". In: *Hardware and Software: Verification and Testing - 12th International Haifa Verification Conference, HVC 2016, Haifa, Israel, November 14-17, 2016, Proceedings*. Ed. by Roderick Bloem and Eli Arbel. Vol. 10028. Lecture Notes in Computer Science. 2016, pp. 117–133. ISBN: 978-3-319-49051-9.

[2] Massimo Benerecetti, Daniele Dell'Erba, and Fabio Mogavero. "Solving Parity Games via Priority Promotion". In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Vol. 9780. Lecture Notes in Computer Science. Springer, 2016, pp. 270–290. ISBN: 978-3-319-41539-0.

[3] Florian Bruse, Michael Falk, and Martin Lange. "The Fixpoint-Iteration Algorithm for Parity Games". In: *Proceedings Fifth International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2014, Verona, Italy, September 10-12, 2014*. Ed. by Adriano Peron and Carla Piazza. Vol. 161. EPTCS. 2014, pp. 116–130.

[4] Cristian S. Calude et al. "Deciding parity games in quasipolynomial time". In: *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*. Ed. by Hamed Hatami, Pierre McKenzie, and Valerie King. ACM, 2017, pp. 252–263. ISBN: 978-1-4503-4528-6.

[5] Sjoerd Cranen et al. "An Overview of the mCRL2 Toolset and Its Recent Advances". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Nir Piterman and Scott A. Smolka. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 199–213. ISBN: 978-3-642-36742-7.

[6] E. Allen Emerson and Chin-Laung Lei. "Efficient Model Checking in Fragments of the Propositional Mu-Calculus (Extended Abstract)". In: *Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986*. IEEE Computer Society, 1986, pp. 267–278. ISBN: 0-8186-0720-3.

[7] John Fearnley et al. “An ordered approach to solving parity games in quasi polynomial time and quasi linear space”. In: *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, July 10-14, 2017*. Ed. by Hakan Erdogmus and Klaus Havelund. ACM, 2017, pp. 112–121. ISBN: 978-1-4503-5077-8.

[8] Oliver Friedmann and Martin Lange. “Solving Parity Games in Practice”. In: *Automated Technology for Verification and Analysis, 7th International Symposium, ATVA 2009, Macao, China, October 14-16, 2009. Proceedings*. Ed. by Zhiming Liu and Anders P. Ravn. Vol. 5799. Lecture Notes in Computer Science. Springer, 2009, pp. 182–196. ISBN: 978-3-642-04760-2.

[9] Oliver Friedmann and Martin Lange. “The PGSolver collection of parity game solvers”. In: *University of Munich* (2009).

[10] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, eds. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*. Vol. 2500. Lecture Notes in Computer Science. Springer, 2002. ISBN: 3-540-00388-6.

[11] Ping Hou, Broes De Cat, and Marc Denecker. “FO(FD): Extending classical logic with rule-based fixpoint definitions”. In: *TPLP* 10.4-6 (2010), pp. 581–596.

[12] Marcin Jurdzinski. “Small Progress Measures for Solving Parity Games”. In: *STACS 2000, 17th Annual Symposium on Theoretical Aspects of Computer Science, Lille, France, February 2000, Proceedings*. Ed. by Horst Reichel and Sophie Tison. Vol. 1770. Lecture Notes in Computer Science. Springer, 2000, pp. 290–301. ISBN: 3-540-67141-2.

[13] Gijs Kant and Jaco van de Pol. “Efficient Instantiation of Parameterised Boolean Equation Systems to Parity Games”. In: *Proceedings First Workshop on GRAPH Inspection and Traversal Engineering, GRAPHITE 2012, Tallinn, Estonia, 1st April 2012*. Ed. by Anton Wijs, Dragan Bosnacki, and Stefan Edelkamp. Vol. 99. EPTCS. 2012, pp. 50–65.

[14] Jeroen J. A. Keiren. “Benchmarks for Parity Games”. In: *Fundamentals of Software Engineering - 6th International Conference, FSEN 2015 Tehran, Iran, April 22-24, 2015, Revised Selected Papers*. Ed. by Mehdi Dastani and Marjan Sirjani. Vol. 9392. Lecture Notes in Computer Science. Springer, 2015, pp. 127–142. ISBN: 978-3-319-24643-7.

[15] David E. Long et al. “An Improved Algorithm for the Evaluation of Fixpoint Expressions”. In: *Computer Aided Verification, 6th International Conference, CAV ’94, Stanford, California, USA, June 21-23, 1994, Proceedings*. Ed. by David L. Dill. Vol. 818. Lecture Notes in Computer Science. Springer, 1994, pp. 338–350. ISBN: 3-540-58179-0.

[16] Sven Schewe. “An Optimal Strategy Improvement Algorithm for Solving Parity and Payoff Games”. In: *Computer Science Logic, 22nd International Workshop, CSL 2008, 17th Annual Conference of the EACSL, Bertinoro, Italy, September 16-19, 2008. Proceedings*. Ed. by Michael Kaminski and Simone Martini. Vol. 5213. Lecture Notes in Computer Science. Springer, 2008, pp. 369–384. ISBN: 978-3-540-87530-7.

[17] Helmut Seidl. "Fast and Simple Nested Fixpoints". In: *Inf. Process. Lett.* 59.6 (1996), pp. 303–308.

[18] Tom van Dijk. "Attracting Tangles to Solve Parity Games". In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*. Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10982. Lecture Notes in Computer Science. Springer, 2018, pp. 198–215. ISBN: 978-3-319-96141-5.

[19] Tom van Dijk. "Oink: An Implementation and Evaluation of Modern Parity Game Solvers". In: *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*. Ed. by Dirk Beyer and Marieke Huisman. Vol. 10805. Lecture Notes in Computer Science. Springer, 2018, pp. 291–308. ISBN: 978-3-319-89959-6.

[20] Igor Walukiewicz. "Monadic second-order logic on tree-like structures". In: *Theor. Comput. Sci.* 275.1-2 (2002), pp. 311–346.

[21] Wieslaw Zielonka. "Infinite Games on Finitely Coloured Graphs with Applications to Automata on Infinite Trees". In: *Theor. Comput. Sci.* 200.1-2 (1998), pp. 135–183.