Solving Parity Games: Combining Progress Measures and Tangle Learning

Master Thesis
Alexander Stekelenburg
January 2024

Supervisors / Graduation Committee:

dr. Tom van Dijk
dr.ir Pieter-Tjerk de Boer
dr. Milan Lopuhaä-Zwakenberg

UNIVERSITY
OF TWENTE.

Faculty of Electrical Engineering, Mathematics & Computer Science (EEMCS)

Master Computer Science: Software Technology Specialisation

Abstract

An oft-used method for verifying the correctness of software is model checking. One technique for performing model checking involves converting a model and specification into a parity game and solving this parity game to obtain information about the model. There are many algorithms for solving parity games. Most of these run in exponential time, but recently some were found which run in quasi-polynomial time. Most of these quasi-polynomial algorithms are from the family of progress measure-based algorithms. However, these algorithms tend to be impractically slow on large games. On the other hand, there are the attractor-based algorithms which tend to be fast in practice but have an exponential time complexity. In this thesis we present an algorithm which accelerates the value iteration of any progress measure by using attractors. We prove the correctness of this algorithm and some of its variants. To do this we first establish a set of requirements on what makes a suitable progress measure. Finally, we perform some empirical performance tests and an analysis of the complexity of the algorithm which show that the algorithm significantly accelerates the progress measure algorithms while having a time complexity that is equivalent to the progress measure that is used.

Acknowledgements

To start I would like to thank my supervisors/members of my graduation committee Tom van Dijk, Pieter-Tjerk de Boer, and Milan Lopuhaä-Zwakenberg for their diligent reading and feedback.

Secondly, I would like to thank Suzanne van der Veen who worked on her parity game related thesis at the same time as me and with whom I had productive conversations about our theses and the graduation procedure in general.

Finally, I would like to thank my family and friends for their support during this somewhat prolonged endeavour and for providing me useful tips and feedback despite their distance from the subject.

- Alexander

Contents

Chapter 1

Introduction

Ensuring the correctness and reliability of software is crucial in a world that depends so heavily on computing. One common approach for the formal verification of programs is model checking. For model checking we construct a model of the program (usually a finite automaton) and provide a specification describing the properties of the program we want to verify. This model and specification can be combined and converted into a parity game.

Parity games are infinite games played by an even and an odd player on a finite directed graph partitioned into vertices owned by the even player and vertices owned by the odd player. Every vertex has an associated priority which is a natural number. For every play through the game the winner is determined by checking the parity (even or odd) of the highest priority that occurs infinitely often. Every play is an infinite path along the vertices of the graph where the owner of a vertex determines the next vertex in the play.

When we solve a parity game we are attempting to find for each vertex which player, the even player or the odd player, has a strategy to win every play starting at that vertex and what that strategy is. The winner of a vertex and the associated strategy give information about whether the model conformed to the specification, and if it did not, what a counter-example looks like. The specifics of how the solution to the parity game relates to the model and specification from which it was synthesized depend on how this synthesis was done and are beyond the scope of this thesis.

For a long time algorithms that solved parity games had an exponential worst-case time complexity. However, since 2017 some quasi-polynomial ($2^{\mathtt{poly}(\mathtt{log}\,n)}$) algorithms have emerged. Two of the most prominent families of parity game solving algorithms are the attractor-based and progress measure-based algorithms. These two families of algorithms work in a very different manner and as such exhibit their worst-case running time on different graph structures.

Attractor-based algorithms reason about the backward reachability of vertices in the graph. We can compute an attractor set for a target vertex and a player which comprises all the vertices for which the player can ensure that a play continues towards the target vertex. By computing attractor sets and removing them from the game in different orders we can detect structures in the game graph which eventually allow us to solve the game. The attractor-based algorithms tend to solve most parity games quickly. However, most of these algorithms have an exponential theoretical time complexity.

The TANGLE LEARNING algorithm uses an extended variant of the attractor called the tangle attractor. When structures in the game graph called tangles are detected they can be used to attract more vertices to the target. Eventually enough tangles are found to solve the game.

Progress measure-based algorithms assign two measures to every vertex, one for each player. This measure is a value that represents how much the player 'prefers' to play towards the vertex. By updating the measures of every vertex based on the measures of its direct successors using a consistent set of update rules we eventually reach a fixed point. At this fixed point, called a progress measure, we can determine the winner of every vertex by checking whether the measure of that vertex got assigned the largest possible measure $\top$. In practice, progress measure-based algorithms tend to be considerably slower than other parity game solving algorithms making them unsuitable for use on larger parity games. However, there are multiple progress measure-based algorithms which feature a quasi-polynomial theoretical time complexity.

The progress measure-based algorithms are also referred to as value iteration algorithms since they mirror the value iteration algorithmic framework that is commonly used for Markov Decision Processes.

We introduce and analyse a new algorithm for solving parity games which is a synthesis of these two families. We will henceforth refer to this algorithm as PMTL (PROGRESS MEASURES AND TANGLE LEARNING). This algorithm takes any existing progress measure and accelerates the propagation of measures using (tangle) attractors. The aim is to combine the quasi-polynomial time complexity of progress measure-based algorithms with the much greater speed of (tangle) attractor-based algorithms on most real-world and random games.

In this thesis we prove the correctness of three variants of this algorithm and give a set of requirements on a progress measure for it to be used with PMTL. Additionally, we test and analyse the performance of the algorithm's variants when combined with different progress measures against attractor-based algorithms and against those measures when put in a value iteration framework.

Chapter 2

Preliminaries

Introduction

In this chapter we introduce parity games and the existing concepts surrounding them. After introducing the abstract concepts underlying TANGLE LEARNING (an attractor-based algorithm for solving parity games) we discuss the generic concept of a progress measure which will be expanded on in chapter 3. This chapter contains only (reformulations of) existing concepts surrounding parity games.

2.1 Basic definitions

A parity game is a two-player game played on a directed graph which is partitioned into two sets: the vertices owned by the even player and the vertices owned by the odd player. We denote the even player as ○ and the odd player as ⬠. Commonly we refer to a player as $\alpha \in \{○, ⬠\}$ or $\bar{\alpha}$ which is the opposing player for player α. Every vertex has priority $p \in \mathbb{N}$, i.e. some natural number. Additionally, we also use ○ and ⬠ to refer to their respective parities.

Definition 2.1 (Parity Game) A parity game $\mathcal{G}$ is a tuple $(V, E, \texttt{pr}, (V_{○}, V_{⬠}))$. It consists of a directed graph (V, E), a function mapping vertices to a priority $\texttt{pr} : V \to \mathbb{N}$, and a partition into even ○ and odd ⬠ vertices $(V_{○}, V_{⬠})$. ▫

We define some more notation. The set $E(u)$ is the set of successors of the vertex u, i.e. $E(u) = \{v \mid (u, v) \in E\}$. Similarly, we refer to the set $E^{-1}(u)$ as the set

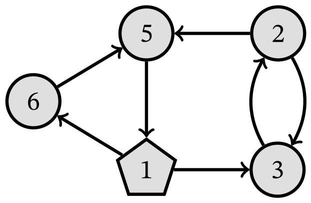


Figure 2.1: An example of a parity game

of predecessors of the vertex u. Lastly, we use the $\equiv_2$ operator to determine whether two natural numbers have the same parity. For example, we use $p \equiv_2$ ○ to denote that $p \in \mathbb{N}$ is an even number.

Figure 2.1 shows an example of a parity game graph. On these graphs we define the concept of a play. In a play the player who owns the current vertex determines the next vertex in the play.

Definition 2.2 (Play) A play π is an infinite path along the edges of the graph such that $\forall_{i>0}(\pi_{i-1}, \pi_i) \in E$. □

We use the word *path* to mean a sequence of vertices connected by edges in the graph. This concept is also commonly referred to as a *walk*.

A play is won by a player $\alpha \in \{$○, ⬠$\}$ if among the vertices that occur infinitely often the highest priority is of α's parity. That is, every play ends up in some infinite sequence and the highest priority in that sequence determines the winner of the play.

When reasoning about parity games we often fix a *positional strategy* for a player. This strategy is used to determine, for each vertex in the graph owned by that player, what the successor must be in the play.

Definition 2.3 (Positional Strategy) Given a player α, a positional strategy σ_α is a partial function $V_\alpha \to V$ such that $\forall_{u \in \mathtt{dom}(\sigma_\alpha)}(u, \sigma_\alpha(u)) \in E$. □

We sometimes use σ to refer to a specific σ_α when this is clear from the context. Given a strategy σ_α we write $\mathtt{Plays}(V(\mathcal{G}), \sigma_\alpha)$ to refer to the set of plays which are consistent with that strategy, that is every play contains only edges (u, v) where either $u \in V_{\bar{\alpha}}$, $u \notin \mathtt{dom}\,\sigma_\alpha$, or $\sigma_\alpha[u] = v$. The player α wins a vertex u if there exists a strategy σ_α such that in all infinite sequences in every play in $\mathtt{Plays}(u, \sigma_\alpha)$ (meaning the set of all plays starting in u consistent with σ_α) the highest priority is of α's parity.

Parity games are memoryless determined. This means that every vertex is won by either even ○ or odd ⬠, and for both players there exists a positional strategy to win their respective vertices[17]. A positional strategy means that the choice of successor is only determined by the current position, no other information needs to be stored. As such, such a strategy is memoryless. For example, in Figure 2.1 the odd ⬠ player can choose to always play towards the vertex with priority 3 in which case all the cycles in the subgame *induced* by the odd ⬠ player's strategy have a highest priority that is odd ⬠. We say that these cycles are won by odd ⬠.

In general, every play eventually enters a repeating sequence, i.e. a cycle. Whichever priority is the highest among the vertices in the cycle determines the winner of the cycle and consequently the winner of the play.

An *induced* subgame is the game with all edges that are not consistent with the strategy removed. In this example, the strategy for the odd ⬠ vertex (i.e. the vertex owned by the odd ⬠ player) with priority 1 is to play towards the even ○ vertex with priority 3. Because of this the induced subgame does not contain the edge that ends in the even ○ vertex with priority 6. As such, the remaining cycles are the smaller cycle with the priorities 3 and 2 which is won by odd ⬠ and the larger cycle with the priorities 1, 3, 2, and 5 which is also won by odd.

A play will always end in a cycle if a positional strategy is used. Since we have found a strategy for odd ⬠ which wins the game for all plays starting in any vertex we find that odd ⬠ wins all the vertices in the game. It may also occur that some set of vertices is won by the odd ⬠ player and another set is won by the ○ player. However, there is no situation where neither player, or both players win a vertex.

2.2 Structures of a parity game

Within parity games we can identify certain structures that allow us to analyse the properties of these games. We commonly refer to sets of vertices in the game as *regions*. We can identify closed regions. An α-closed region is a set of vertices for which α has a strategy σ_α such that for every vertex v in the region $\sigma_\alpha[v]$ does not leave the region, or if v is owned by $\bar{\alpha}$ there is no successor that is not in the region.

Definition 2.4 (Closed region) An α-closed region in a game $\mathcal{G}$ is any non-empty set $U \subseteq V(\mathcal{G})$ such that:

1. $\forall_{v \in U \cap V_\alpha(\mathcal{G})} E(v) \cap U \neq \varnothing$
2. $\forall_{v \in U \cap V_{\bar{\alpha}}(\mathcal{G})} E(v) \subseteq U$ □

Another structure we can identify is the tangle. A tangle is a set of vertices for which a player α has a strategy σ_α to win all the cycles within. The subgraph restricted to the strategy is strongly-connected.

Definition 2.5 (Tangle) A p-tangle is a set of vertices $\tau \subseteq V(\mathcal{G})$ where the highest priority vertex in τ has priority $p \equiv_2 \alpha$ (it has α's parity), and the player α has a strategy σ_α such that $\forall_{v \in \tau \cap V_\alpha(\mathcal{G})} \sigma_\alpha[v] \in \tau$ and α wins all the cycles in the subgraph (τ, E') with $E' = \{(u,v) \in E(\mathcal{G}) \mid u \in \tau \wedge v \in \tau \wedge (\sigma_\alpha[u] = v \vee u \notin \mathtt{dom}\,\sigma_\alpha)\}$. Additionally, the subgraph (τ, E') is strongly-connected. □

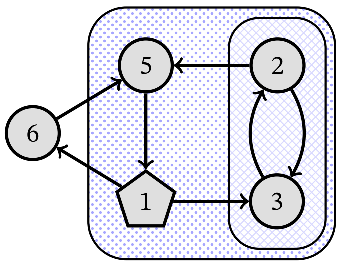


Figure 2.2: The tangles of the game in Figure 2.1 highlighted

If a p-tangle with $p \equiv_2 \alpha$ is an α-closed region, i.e. $\bar{\alpha}$ cannot escape it, then it is a *closed p-tangle*. For example, in Figure 2.2 we can see that the 3-tangle is not a closed region because the vertex with priority 2 can play towards a vertex outside the region. However, the 5-tangle is a ⬠-closed region (odd) since there are no even vertices which can leave the region and the odd vertex has the option to stay in the region.

Parity games contain hierarchies of tangles. A p-tangle may contain tangles with maximum priorities $q \leq p$, i.e. q-tangles. We can identify two tangles in the game in Figure 2.1, these are highlighted in Figure 2.2. We can see two tangles won by odd ⬠, a 3-tangle contained in a 5-tangle.

Finally, parity games contain dominions.

Definition 2.6 (Dominion) A p-dominion is an α-closed region such that the highest infinitely recurring priority in every play restricted to the region is p. As such, there exists a strategy σ_α for α such that α wins any play starting from one of the vertices in the region. □

Every p-dominion contains at least one p-tangle but may also contain more vertices that, for example, have a priority $\geq p$ but for which α has a strategy such that that priority does not infinitely recur. (the vertices with these priorities are not visited infinitely often in any play) Furthermore an ○-dominion (even) is a p-dominion with $p \equiv_2$ ○ and an ⬠-dominion (odd) is a q-dominion with $q \equiv_2$ ⬠. We identify three well-known properties of dominions:

Lemma 2.1 (Closed under union) *Given a p-dominion U and a q-dominion U' with $p \equiv_2 q$, we have that $U \cup U'$ is a r-dominion with $r \equiv_2 p$.* □

Lemma 2.2 (Positionally determined) *Every vertex v in a parity game $\mathcal{G}$ is either in an even dominion or in an odd dominion. Specifically, v is in either the largest even dominion or in the largest odd dominion. The proof of this is given by McNaughton[15].* □

Corollary 2.1 (Disjoint) *Given a p-dominion U and a q-dominion U' with $p \not\equiv_2 q$, we have that $U \cap U' = \emptyset$.* □

Note that if a tangle is closed, it is a dominion. But a dominion is not necessarily a tangle since it may contain vertices with priorities of $\bar{\alpha}$'s parity that are higher than the highest priority of α's parity in the set of vertices as long as they are not a part of a $\bar{\alpha}$-cycle. (i.e. a cycle where the highest priority is of $\bar{\alpha}$'s parity) Additionally, a dominion need not be strongly connected.

2.3 Attractors

To find the aforementioned structures in a parity game we can use attractors. Attractors reason about the backward reachability of a vertex. If a vertex u is in the α-attractor set of a vertex v then there is a strategy for α such that it can ensure a play ends up in v. Intuitively, we can reason that if α owns a vertex u and there is an edge $(u, v) \in E(\mathcal{G})$ then if α chooses to use that edge as its strategy, that will ensure that every play that reaches u will reach v. Therefore, we can say that v attracts u. However, a vertex w owned by $\bar{\alpha}$ can only be attracted if all its outgoing edges lead either to v directly or lead to a vertex which is already attracted to v.

It is important to be able to compute these attractors with some maximum priority, because when we find a region that is locally closed, i.e. closed with regard to only the current subgame, we can ensure that the highest priority vertex in that region is of α's parity. This allows us to conclude that all the vertices in that region are won by α if $\bar{\alpha}$ cannot escape the region.

Attractor sets were first specified by Zielonka[17]; he proves that the existence of the attractor implies that a positionally determined strategy exists such that the target is reached. An attractor set $\mathtt{attr}^{\mathcal{G}}_{\alpha}(\mathcal{A}, P)$ is the set of vertices for which a player α has a strategy σ_α such that a play will reach some set of vertices $\mathcal{A} \subseteq V(\mathcal{G})$ while only attracting vertices with a priority that is in the set P. The formal definition given here is based on Van Dijk[5].

Definition 2.7 (Attractor) Given a player $\alpha \in \{\bigcirc, \pentagon\}$, a parity game $\mathcal{G}$, a set of vertices $\mathcal{A}$, and a set of priorities P the attractor set $\mathtt{attr}^{\mathcal{G}}_{\alpha}(\mathcal{A}, P)$ is defined[1] as:

$$\begin{aligned}\mu Z.\, \mathcal{A} &\cup \{v \in V_\alpha(\mathcal{G}) \mid E(v) \cap Z \neq \emptyset \wedge \mathtt{pr}(v) \in P\} \\ &\cup \{v \in V_{\bar{\alpha}}(\mathcal{G}) \mid E(v) \subseteq Z \wedge \mathtt{pr}(v) \in P\}\end{aligned}$$

(To simplify the conditions we assume that $E(v) \neq \emptyset$ since any unconnected vertices may be trivially solved and excluded from the game) ◻

Attractors can be computed as such:

- Add all vertices from the set $\mathcal{A}$ to the attractor. (since they have already been reached)
- Repeat until no more vertices can be added:
  - Add all vertices $V_\alpha(\mathcal{G})$ owned by α that have a successor in the attractor set and that have a priority that is in the set P.
  - Add all vertices $V_{\bar{\alpha}}(\mathcal{G})$ owned by $\bar{\alpha}$ whose successors are all in the attractor set and that have a priority that is in the set P.

While computing the attractor we can also store a strategy σ_α (simply by setting the vertex which attracted its neighbour to be the strategy for that neighbour) for each attracted vertex which can later be used when giving the solution of the game or to determine the strategy of a tangle.

Lemma 2.3 *Given a player $\alpha \in \{\bigcirc, \pentagon\}$, a parity game $\mathcal{G}$, a set of vertices $\mathcal{A}$, a set of priorities P, and an attractor set $\mathtt{attr}^{\mathcal{G}}_{\alpha}(\mathcal{A}, P)$ there exists a strategy σ_α for α such that every play consistent with σ_α in the attractor set eventually reaches a vertex in $\mathcal{A}$.* ◻

One straightforward way in which attractors can help us discover structures in a game is that they can extend dominions.

Lemma 2.4 *Given a p-dominion U for a player α with $p \equiv_2 \alpha$ in a game $\mathcal{G}$, the attractor set $\mathtt{attr}^{\mathcal{G}}_{\alpha}(U, \mathbb{N})$ is a p-dominion.* ◻

We extend definition of an attractor to take an extra set of escapes $\mathcal{E} \subseteq \mathcal{G}$. If a vertex is not in $\mathcal{E}$ then the $\bar{\alpha}$ player cannot use it to escape, and thus the resulting attractor set may be larger than the one where $\mathcal{E} = \mathcal{G}$. We write $\mathtt{attr}^{\mathcal{G}}_{\alpha}(\mathcal{A}, \mathcal{E}, P)$ to denote the attractor set taking into account the set of escapes. As such $\mathtt{attr}^{\mathcal{G}}_{\alpha}(\mathcal{A}, P) = \mathtt{attr}^{\mathcal{G}}_{\alpha}(\mathcal{A}, \mathcal{G}, P)$. The formal definition of $\mathtt{attr}^{\mathcal{G}}_{\alpha}(\mathcal{A}, \mathcal{E}, P)$ is:

$$\begin{aligned}\mu Z.\, \mathcal{A} &\cup \{v \in V_\alpha(\mathcal{G}) \mid E(v) \cap Z \neq \emptyset \wedge \mathtt{pr}(v) \in P\} \\ &\cup \{v \in V_{\bar{\alpha}}(\mathcal{G}) \mid E(v) \cap \mathcal{E} \subseteq Z \wedge \mathtt{pr}(v) \in P\}\end{aligned}$$

[1]Using the modal μ-calculus where μ is the least fixed point operator. The set Z is initially empty and expanded until a fixed point is reached.

Finally, we also define the function `attracts` which simply checks for a single vertex whether it is attracted to the given set in a single step, i.e. no other vertex may be attracted first such that the given vertex is attracted other than those given in the original target set.

Definition 2.8 (`attracts` function) The function `attracts` is defined as follows:

$$\texttt{attracts}_\alpha(u, \mathcal{A}, \mathcal{E}) = \begin{cases} \textbf{true} & \text{if } \texttt{attr}^{\{u\}}_\alpha(\mathcal{A}, \mathcal{E}, \mathbb{N}) = \mathcal{A} \cup \{u\} \\ \textbf{false} & \text{otherwise, i.e. } \texttt{attr}^{\{u\}}_\alpha(\mathcal{A}, \mathcal{E}, \mathbb{N}) = \mathcal{A} \end{cases}$$ ◻

2.3.1 Tangle attractors

Van Dijk[5] extended the notion of an attractor by adding the tangle attractor. If we have a tangle τ won by player α then we know that the opponent $\bar{\alpha}$ must escape this tangle to avoid losing the game. We can use this fact to attract more vertices than otherwise possible. If we are computing an attractor for player α and one of the vertices in the attractor set is also contained in τ then we can add all the vertices in τ to the set.

A tangle attractor set $\texttt{attrT}^{\mathcal{G}}_\alpha(\mathcal{A}, P)$ is an extended attractor set for which a player α has a strategy σ_α such that all plays that start at a vertex in the attractor set will reach some set of vertices $\mathcal{A} \subseteq V(\mathcal{G})$ or remain in a tangle $\tau \in \mathcal{T}_\alpha$ which is won by α where $\mathcal{T}_\alpha$ is a subset of all the tangles of α's parity in the game.

Definition 2.9 (Tangle Attractor) Given a player $\alpha \in \{\bigcirc, \pentagon\}$, a parity game $\mathcal{G}$, a set of vertices $\mathcal{A}$, and a set of priorities P the tangle attractor set $\texttt{attrT}^{\mathcal{G}}_\alpha(\mathcal{A}, P)$ is defined as:

$$\begin{aligned} \mu Z.\, \mathcal{A} & \cup \{v \in V_\alpha(\mathcal{G}) \mid E(v) \cap Z \neq \emptyset \wedge \texttt{pr}(v) \in P\} \\ & \cup \{v \in V_{\bar{\alpha}}(\mathcal{G}) \mid E(v) \subseteq Z \wedge \texttt{pr}(v) \in P\} \\ & \cup \{v \in \tau \cap V(\mathcal{G}) \mid \tau \in \mathcal{T}_\alpha \wedge E(\tau) \subseteq Z \wedge \forall_{w \in \tau} \texttt{pr}(w) \in P\} \end{aligned}$$

(To simplify the conditions we assume that $E(v) \neq \emptyset$ since any unconnected vertices may be trivially solved and excluded from the game) ◻

Tangle attractors can be computed by first calculating the normal attractor and then adding all vertices from every tangle $\tau \in \mathcal{T}_\alpha$ won by α where the only escapes $E(\tau)$ of the tangle are vertices in the attractor set.

Similarly to the regular attractor we can store a strategy σ_α for each attracted vertex. For vertices that were attracted because they were part of a tangle we simply use the strategy that was stored along with the tangle when it was found.

Lemma 2.5 *Given a player $\alpha \in \{\bigcirc, \pentagon\}$, a parity game $\mathcal{G}$, a set of vertices $\mathcal{A}$, a set of priorities P, and a tangle attractor set $\texttt{attrT}^{\mathcal{G}}_\alpha(\mathcal{A}, P)$ there exists a strategy σ_α for α such that every play consistent with σ_α in the attractor set eventually reaches a vertex in $\mathcal{A}$ or ends up in a cycle where the highest priority is of α's parity.* ◻

Similarly to the regular attractors, tangle attractors can extend dominions.

Lemma 2.6 *Given a p-dominion U for a player α with $p \equiv_2 \alpha$ in a game $\mathcal{G}$, the attractor set* $\texttt{attrT}_\alpha^{\mathcal{G}}(U, \mathbb{N})$ *is a p-dominion.* ◻

We extend the definition of a tangle attractor to take an extra set of escapes $\mathcal{E} \subseteq \mathcal{G}$. If a vertex is not in $\mathcal{E}$ then the $\bar{\alpha}$ player cannot use it to escape. We write $\texttt{attrT}_\alpha^{\mathcal{G}}(\mathcal{A}, \mathcal{E}, P)$ to denote the tangle attractor set taking into account the set of escapes. As such $\texttt{attrT}_\alpha^{\mathcal{G}}(\mathcal{A}, \mathcal{G}, P) = \texttt{attrT}_\alpha^{\mathcal{G}}(\mathcal{A}, P)$. The formal definition of $\texttt{attrT}_\alpha^{\mathcal{G}}(\mathcal{A}, \mathcal{E}, P)$ is:

$$\begin{aligned}
&\mu Z.\, \mathcal{A} \cup \{v \in V_\alpha(\mathcal{G}) \mid E(v) \cap Z \neq \varnothing \wedge \texttt{pr}(v) \in P\} \\
&\qquad \cup \{v \in V_{\bar{\alpha}}(\mathcal{G}) \mid E(v) \cap \mathcal{E} \subseteq Z \wedge \texttt{pr}(v) \in P\} \\
&\qquad \cup \{v \in \tau \cap V(\mathcal{G}) \mid \tau \in \mathcal{T}_\alpha \wedge E(\tau) \cap \mathcal{E} \subseteq Z \wedge \forall_{w \in \tau} \texttt{pr}(w) \in P\}
\end{aligned}$$

2.3.2 Attractor decomposition

We can use attractors to partition a game into regions by repeatedly taking the highest priority vertex in a subgame, computing its attractor set, and removing those vertices from the subgame. This is a *top-down* attractor decomposition since we start with the highest priority in the game (the top) and we work our way down. We call the priority of the highest vertex in the region the priority of the region. As such, we can refer to regions as being lower and higher than each other based on their priority. There are multiple algorithms for solving parity games which use these regions obtained through *top-down* attractor decomposition[5, 13, 16].

The TANGLE LEARNING algorithm uses attractor decompositions and the `extractTangles` function to solve parity games[5]. By repeatedly decomposing the game into regions and finding tangles in the closed regions it eventually finds all the dominions in the game. By introducing the tangle attractor `attrT` TANGLE LEARNING is able to detect dominions in the game and remove them earlier than would be possible with older attractor-based algorithms.

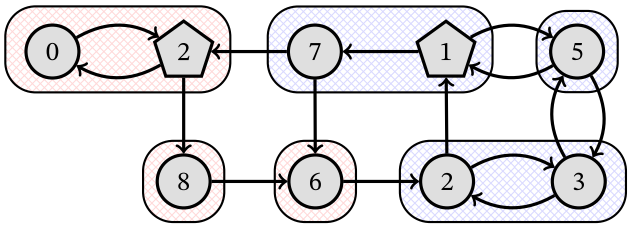


Figure 2.3: A parity game decomposed top-down into regions using attractors

Figure 2.3 shows an example of a parity game decomposed top-down into regions by the TANGLE LEARNING algorithm; even ○ regions are red and odd ⬠ regions are blue. In Example 2.1 we explain how this decomposition is done and how the TANGLE LEARNING algorithm refines the decomposition in the next iteration.

In order for the tangle attractor $\mathtt{attrT}_\alpha$ to attract more vertices than $\mathtt{attr}_\alpha$ we first need to find tangles in the game and store them in $\mathcal{T}_\alpha$. This is done using the `extractTangles` function.

Definition 2.10 (`extractTangles` Function) The $\mathtt{extractTangles}(\mathcal{A}, \sigma)$ is a function that when given a closed α-region $\mathcal{A}$ (i.e. a region for which α has a strategy σ such that any play remains within the region) computes the non-trivial strongly connected components in that region when restricted to the strategy σ and stores these as tangles in $\mathcal{T}_\alpha$. If any of these tangles have no escapes for $\bar{\alpha}$ in the $\mathcal{G}$ (note here that we are talking about outgoing edges in the entire game not only those that are left after removing vertices in the attractor decomposition) then those tangles are instead added to the set of α-dominions $\mathcal{D}_\alpha$. □

The `extractTangles` function can be efficiently implemented using Tarjan's algorithm starting in the top vertex to find the strongly connected components[5].

Example 2.1 (Executing the Tangle Learning algorithm) For this example we take the game shown in Figure 2.3. We initialise the set of even ○ tangles $\mathcal{T}_{\bigcirc}$ and odd ⬠ tangles $\mathcal{T}_{\text{⬠}}$ to the empty set.

We start by finding the highest priority in the current subgame $\mathcal{G}$ which is 8. We compute $Z = \mathtt{attrT}^{\mathcal{G}}_{\bigcirc}(\mathcal{A}, \mathbb{N})$ where $\mathcal{A}$ is the set of all vertices with priority 8. In this case $\mathcal{A} = Z$ because the *odd*-vertex with priority 2 can choose to play to the ○-vertex with priority 2 instead of the ○-vertex with priority 8. Now we remove Z from our subgame $\mathcal{G}$. Z is marked as an ○ region (red) because its highest priority is even.

The next remaining highest priority is 7. We compute $Z = \mathtt{attrT}^{\mathcal{G}}_{\text{⬠}}(\mathcal{A}, \mathbb{N})$ where $\mathcal{A}$ is the set of all vertices with priority 7. In this case Z includes the ⬠-vertex with priority 1 because the ⬠ player can choose to play towards the ○-vertex with priority 7. We computed an odd ⬠ attractor because 7 is odd. Now we remove Z form our subgame $\mathcal{G}$. Z is marked as an ⬠ region (blue) because its highest priority is odd.

The next remaining highest priority is 6. We compute $Z = \mathtt{attrT}^{\mathcal{G}}_{\bigcirc}(\mathcal{A}, \mathbb{N})$ where $\mathcal{A}$ is the set of all vertices with priority 6. In this case $Z = \mathcal{A}$ because there are no vertices in the remaining subgame which have the ○-vertex with priority 6 as a successor. Now we remove Z form our subgame $\mathcal{G}$. Z is marked as an ○ region (red) because its highest priority is even.

The next remaining highest priority is 5. We compute $Z = \mathtt{attrT}^{\mathcal{G}}_{\text{⬠}}(\mathcal{A}, \mathbb{N})$ where $\mathcal{A}$ is the set of all vertices with priority 5. In this case $\mathcal{A} = Z$ because the *even*-vertex with priority can choose to play to the ○-vertex with priority 2 instead of the ○-vertex with priority 8. Now we remove Z form our subgame $\mathcal{G}$. Z is marked as an ⬠ region (blue) because its highest priority is odd.

The next remaining highest priority is 3. We compute $Z = \mathtt{attrT}^{\mathcal{G}}_{\text{⬠}}(\mathcal{A}, \mathbb{N})$ where $\mathcal{A}$ is the set of all vertices with priority 3. In this case Z includes the ○-vertex with priority 2 because the ○ player cannot choose to play towards the ⬠-vertex with priority 1 because it was already removed from the subgame. Now we remove Z form our subgame $\mathcal{G}$. Z is marked as an ⬠ region (blue) because its highest priority is odd. Note that this region is locally closed since the vertices in Z are unable to leave the region since all their successors are in the region or were already removed from the subgame.

Finally, the next remaining highest priority is 2. We compute $Z = \mathtt{attrT}_{\bigcirc}^{\mathcal{G}}(\mathcal{A}, \mathbb{N})$ where $\mathcal{A}$ is the set of all vertices with priority 2. In this case Z includes the $\bigcirc$-vertex with priority 0. Now we remove Z form our subgame $\mathcal{G}$. Z is marked as an $\bigcirc$ region (red) because its highest priority is even. Note that this region is locally closed since the vertices in Z are unable to leave the region since all their successors are in the region or were already removed from the subgame.

After decomposing the game into regions TANGLE LEARNING uses `extractTangles` to extract tangles from (locally) closed regions. In this case it finds an $\bigcirc$-tangle (even) with the $\pentagon$-vertex with priority 2 and the $\bigcirc$-vertex with priority 0, and an $\pentagon$-tangle (odd) with the $\bigcirc$-vertices with priorities 3 and 0. These tangles are added to $\mathcal{T}_{\bigcirc}$ and $\mathcal{T}_{\pentagon}$ respectively.

We can then start another iteration with the subgame set to the entire game again. We will not cover the entire solving of the game, but we can notice that while computing the tangle attractor for the highest region (i.e. the region with priority 8) the $\bigcirc$-tangle we just found can be attracted meaning we have found larger even region than before. This process continues until only globally closed regions remain which are dominions for whichever player has the highest priority in the region. For this particular game odd $\pentagon$ ends up winning every vertex. □

Our algorithm (PMTL) also uses regions to determine which vertices should have their measures lifted. (what a measure is will be explained in the next section) However, these regions are not computed in the *top-down* order as described above, instead the order is primarily determined by the current measure of each vertex. Additionally, the attractor decompositions used by our algorithm do not an $\bigcirc$-attractor when computing the region where the highest priority is $\bigcirc$ and an $\pentagon$-attractor when computing the region where the highest priority is $\pentagon$. Instead, we have separate $\bigcirc$-attractor decompositions and $\pentagon$-attractor decompositions which use the same parity for the attractor of every region.

We define an attractor decomposition, which formalises the repeated application of the tangle attractor computation to decompose a game into regions.

Definition 2.11 (Attractor Decomposition) Given some ordering $\leq$ of vertices in $V(\mathcal{G})$, the α-attractor decomposition $\mathcal{H}_\alpha(\mathcal{G}, \leq)$ of $\mathcal{G}$ is a sequence of regions recursively defined as:

$$\begin{aligned} \mathcal{H}_\alpha(\varnothing, \leq) &= \langle\rangle \\ \mathcal{H}_\alpha(\mathcal{G}, \leq) &= (v, \mathcal{A}) \cdot \mathcal{H}_\alpha(\mathcal{G} \setminus \mathcal{A}, \leq) \end{aligned}$$

where

$$\begin{aligned} v &= \max_{\leq} V(\mathcal{G}) \\ \mathcal{A} &= \mathtt{attrT}_{\alpha}^{\mathcal{G}}(\{v\}, \{p \in \mathbb{N} \mid p \equiv_2 \alpha \vee p \leq \mathtt{pr}(v)\}) \end{aligned}$$ □

Here, $\langle\rangle$ denotes an empty sequence, and $x \cdot s$ forms a new sequence where the first element is x and the elements from the sequence s follow. The notation $\mathtt{max}_{\leq} Z$

or $\max_{\leq} Z$ denotes the maximum element in the set Z given the ordering $\leq$. More formally a sequence is defined as such:

Definition 2.12 (Sequence) A sequence $Q : \mathbb{N} \rightharpoonup S$ is a partial function mapping indices to elements of a set S. The domain of the empty sequence $\langle\rangle$ is the empty set and the domain of any non-empty sequence of length k is the set of natural numbers in the range $[0, k)$. We write Q_i to refer to the i-th element in the sequence Q, i.e. the result of $Q(i)$. □

We refer to the v in a region $(v, \mathcal{A})$ as the *top* vertex of the region. Additionally, we say that the region has the priority $\texttt{pr}(v)$ and we consider a region whose *top* vertex was higher in the ordering $\leq$ than the *top* vertex of another region to be a higher region than that other region.

2.4 Progress measures

There are several algorithms for solving parity games which use *progress measures*. These algorithms use a framework which is similar to the value iteration framework that is commonly used for Markov Decision Processes. Namely, repeatedly applying a monotone function to a set of values until a fixpoint is reached which represents the solution to the game, i.e. the values in the fixpoint state allow us to derive the solution to the game. Since these progress measure-based algorithms all use their own progress measures we will first define the abstract concept of a *measure* and a *progress measure* in this section. Indeed, the algorithm we describe in this thesis can be used with multiple different progress measures with their exact semantics being unimportant as long as they satisfy the definition in this chapter and the properties in chapter 3. In that chapter we will also describe two concrete examples of progress measures which satisfy these definitions and properties.

Measures are values that are assigned to every vertex in the graph. We keep track of a set of measures for each player, i.e., we have an ○-measure (even) and an ⬠-measure (odd) for each vertex. Informally, each measure that is assigned to a vertex represents how much the player 'prefers' the vertex. To solve a parity game measures are lifted (i.e. replaced with a higher measure) until they cannot be lifted any further. How exactly this lifting procedure works depends on the progress measure algorithm that is used.

The solution to the game (i.e. which vertices are won by which player and their strategies) can be derived from the final values in the fixed point state of the measures. We call the fixed point state of the measures after no more measures can be lifted a *progress measure*. An α-progress measure contains information about which vertices are won by which player and from the α-measures we can extract the winning strategy for $\bar{\alpha}$. (and from the $\bar{\alpha}$-measures we would be able to extract the winning strategy for α)

We refer to the set of all measures, also known as the *measure space*, as $\mathcal{M}$. For most progress measure algorithms there is a separate measure space for each player written as $\mathcal{M}_{⬠}$ and $\mathcal{M}_{\bigcirc}$ for the odd ⬠ and even ○ player respectively. A measure space $\mathcal{M}$ is a complete lattice with a total order, it must have a least element (written as $\perp$) which is used as the starting measure of every vertex and a greatest element (written as $\top$) which denotes that the player wins the vertex. Lifting is done using a progression function $\texttt{prog}: \mathcal{M} \times \mathbb{N} \to \mathcal{M}$ also often referred to as the update function. This function determines how a measure updates when it ‘encounters’ a certain priority. We need to determine what properties this function needs to have to prove the correctness of our algorithm. We do this in chapter 3. The measures are stored using a measure function $\mu: V \to \mathcal{M}$ which maps every vertex to a measure.

Lifting a vertex v is done by taking the measure of all the successors of v and calling $\texttt{prog}$ on those measures with the priority of v: $\texttt{pr}(v)$. We say that these measures ‘encounter’ the priority $\texttt{pr}(v)$. If v is owned by the player whose measures we are calculating then v’s new measure becomes the largest of these updated measures. Otherwise, v’s new measure becomes the smallest of these updated measures. This operation always terminates since there is a finite[2] set of measures, and the progression function is monotonic. Once the final measures have been determined all the measures that have not reached $\top$ will be won by the *opponent* of the player whose progress measures we computed. Furthermore, it is now trivial to compute the strategy for this player’s vertices by simply selecting the successor with the lowest measure.

One way to visualise the process of lifting measures is that a measure is moved from vertex to vertex by following edges in the reverse direction. The measure in this manner ‘encounters’ the priorities of the prefix of a play, albeit in a reversed order. Since every play of a game eventually ends up in a cycle that is won by the player α or $\bar{\alpha}$ an α-measure will eventually converge to a value that is either $\top$ if it got stuck in a cycle won by α or lower than $\top$ if it got stuck in a cycle won by $\bar{\alpha}$.

Finally, we give a formal definition of a progress measure[3]:

Definition 2.13 (Progress measure) A measure function $\mu: V \to \mathcal{M}$ is an α-progress measure if the following conditions hold for every vertex u in the game:

- $\forall_{v \in E(u)} \texttt{prog}(\mu[v], \texttt{pr}(u)) \leq \mu[u]$ if $u \in V_\alpha(\mathcal{G})$
- $\exists_{v \in E(u)} \texttt{prog}(\mu[v], \texttt{pr}(u)) \leq \mu[u]$ if $u \in V_{\bar{\alpha}}(\mathcal{G})$
- $\mu[u] = \top$ if and only if u is won by α, that is, there exists an α-strategy to win all plays originating in u. □

Summary

In this chapter we presented definitions of a parity game and its surrounding concepts. Additionally, we gave definitions of our own modified versions of (tangle) attractors and attractor decompositions. Finally, we formalised the concept of a progress measure, but left the definition and examples of progress measure-based algorithms for solving parity game for the next chapter.

[2]For our algorithm we do not require the set of measures to be finite since we rely on the original progress measure-based algorithm’s termination proof to prove our algorithm terminates, although we do not know of any progress measure-based algorithm with an infinite measure space.

[3]This is reworded and extended from definition 5 in [1].

Chapter 3

Requirements on Progress Measures

Introduction

In this chapter we introduce some properties of the progress function `prog` for progress measures. Secondly, we introduce a generic progress measure-based algorithm which can be parametrised to resemble concrete existing progress measure-based algorithms. Furthermore, we introduce two concrete examples of existing progress measure-based algorithms: the Small Progress Measures and the Ordered Progress Measures. We show how these algorithms satisfy the properties that we introduced. Finally, we use these properties to prove lemmas which form the basis for the correctness proof of the PMTL algorithm in chapter 5.

3.1 Progress function

In section 2.4 we described the progression function `prog` but did not give a definition which can be used to determine such a function can be used with our algorithm.

Before we can describe these properties we need to introduce a new concept: the p-achievements. A measure has an achievement for every priority $p \in \mathbb{N}$. A p-achievement signifies the minimum to which a measure will be lowered if it 'encounters' a vertex with a priority $q \leq p$.

To define the p-achievement we first define a modification of `prog` which takes a sequence of priorities instead of a single priority.

Definition 3.1 (`prog` on sequences of priorities) Given a measure $m \in \mathcal{M}$ and a sequence of priorities Q we define:

$$\texttt{prog}(m, Q) \triangleq \begin{cases} m & \text{if } Q = \langle\rangle \\ \texttt{prog}(\texttt{prog}(m, Q'), \texttt{pr}(u)) & \text{if } Q = Q' \cdot u \end{cases} \qquad \square$$

Here, $\langle\rangle$ denotes an empty sequence, and $s \cdot x$ forms a new sequence where the elements from the sequence s are followed by the element x.

Definition 3.2 (p-achievement) Given a priority $p \in \mathbb{N}$, a p-achievement of an α-measure $m \in \mathcal{M}_\alpha$ is a measure $m' \in \mathcal{M}_\alpha$ such that, given a sequence of priorities Q where for every priority $q \in Q$, $q \leq p \vee q \equiv_2 \alpha$, we have $\texttt{prog}(m, Q) \geq m'$. □

We can see that the least element $\bot \in \mathcal{M}$ has p-achievements equal to $\bot$ for every $p \in \mathbb{N}$ because there exists no lower measure, and the greatest element $\top \in \mathcal{M}$ has p-achievements equal to $\top$ for every $p \in \mathbb{N}$ because `prog` is monotonic and $\top$ is the greatest possible element. (at least for all sane definitions of measures because if there were a priority $p \in \mathbb{N}$ such that $\texttt{prog}(\top, p) < \top$ then this must be true for every priority because $\top \leq \top$)

In practice, we only care about a measure's p-achievement when it equals itself. We call this p-stable.

Definition 3.3 (p-stable) Given a priority $p \in \mathbb{N}$, an α-measure $m \in \mathcal{M}_\alpha$ is p-stable, if and only if, given a sequence of priorities Q where for every priority $q \in Q$, $q \leq p \vee q \equiv_2 \alpha$, we have $\texttt{prog}(m, Q) \geq m$. □

Corollary 3.1 *Given a priority $p \in \mathbb{N}$, an α-measure $m \in \mathcal{M}_\alpha$ is p-stable, if and only if, m has a p-achievement equal to m.* □

We can now introduce the requirements on the `prog` function. These properties are sufficient for a progress measure to correctly solve a parity game within a value iteration framework and within our algorithm. However, it is probable that not all of these properties are necessary when applied within a value iteration framework. Therefore, there may be a progress measure which is not compatible with PMTL, however we have not encountered such a progress measure yet.

Definition 3.4 (Progress function) The $\texttt{prog} : \mathcal{M}_\alpha \times \mathbb{N} \to \mathcal{M}_\alpha$ function of an α-progress measure must have the following properties:

1. `prog` must be monotonic (order-preserving), i.e. given a measure $m \in \mathcal{M}_\alpha$, a measure $m' \geq m$, and a priority $p \in \mathbb{N}$ we have $\texttt{prog}(m, p) \leq \texttt{prog}(m', p)$.
2. Given a measure $m \in \mathcal{M}_\alpha$, and a priority $p \in \mathbb{N}$ we have that the resulting measure of $\texttt{prog}(m, p)$ is p-stable. (Definition 3.3)
3. Given a measure $m \in \mathcal{M}_\alpha$, a priority $p \in \mathbb{N}$ and a sequence of priorities Q where for every priority $q \in Q$ with $q \leq p \vee q \equiv_2 \alpha$, we have $\texttt{prog}(m, p) \leq \texttt{prog}(\texttt{prog}(m, Q), p)$. □

Lemma 3.1 *Every measure $m \in \mathcal{M}$ that is assigned to a vertex with priority $p \in \mathbb{N}$ must be p-stable in any algorithm where the measure assigned to a vertex is the result of some* `prog` *with its own priority.*
PROOF The second property ensures that an α-measure $m' \in \mathcal{M}_\alpha$ that is produced using a priority $p \in \mathbb{N}$ and subsequently assigned to a vertex with a priority p is p-stable such that encountering priorities that are of α's parity, or smaller or equal to p cannot decrease the measure below m'. Therefore, we can conclude that whenever a measure m' is assigned to vertex with priority p by a progress measure-based algorithm that measure must be p-stable. ■

Whereas p-stability allows us to ensure that measures with that property do not decrease beyond some level, we also need something similar for *all* measures. The third property ensures that no priority in the sequence of priorities Q where each priority is smaller than p (or of α's parity which in general should always be beneficial for α) is able to cause $\texttt{prog}(m, Q)$ to be so low that, when the priority p is encountered the resulting measure is smaller than if the measure m encountered the priority p immediately. For example, for some $\bigcirc$-measure m we have $\texttt{prog}(m, 6) \leq \texttt{prog}(\texttt{prog}(m, \langle 1, 5, 4, 3, 5\rangle), 6)$.

The necessity of the properties

Firstly, the requirement that the `prog` function is monotonic is shared by all progress measures that are used in the traditional value iteration framework since proving that those algorithms terminate requires a guarantee that an inflationary fixpoint is reached eventually. If a `prog` function has non-monotonic behaviour this is not necessarily the case.

Secondly, the second and third property of `prog` are absolutely essential for us to prove that lifting measures one edge at a time along a path is equivalent to lifting using an attractor. We need some guarantee that the measure cannot decrease too much along the path such that we can safely skip lifting one edge at a time.

3.2 Generic Progress Measures

Most existing progress measure-based parity game solving algorithms use a value iteration framework to update the measures and determine the strategy. In order to prove the correctness of our PMTL algorithm we will prove that it yields equivalent results to the Generic Progress Measures (GPM) algorithm presented in this section. That is, our PMTL algorithm and the GPM algorithm should output the same measure functions. By parametrising GPM with the appropriate measures spaces, orderings, and `prog` functions it will behave identically to these existing progress measure-based algorithms.

Algorithm 3.1: gpm

```
fn gpm():
    $pm_{○} \leftarrow \{u \leftarrow \bot \mid u \in V(\mathcal{G})\}$
    $pm_{⬠} \leftarrow \{u \leftarrow \bot \mid u \in V(\mathcal{G})\}$
    do:
        $updated \leftarrow$ false
        for every vertex $u \in V(\mathcal{G})$ :
            if liftVertex($pm_{○}$, ○, $u$) : $updated \leftarrow$ true
    while $updated$
    solve($pm_{○}$, ⬠)
    do:
        $updated \leftarrow$ false
        for every vertex $u \in V(\mathcal{G})$ :
            if liftVertex($pm_{⬠}$, ⬠, $u$) : $updated \leftarrow$ true
    while $updated$
    solve($pm_{⬠}$, ○)
```

The function `gpm` is shown in Algorithm 3.1. It starts by initialising the ○-measure (even) and ⬠-measure (odd) functions, setting the measure of every vertex to the smallest possible value, i.e. the least element $\top$ in $\mathcal{M}_{○}$ and $\mathcal{M}_{⬠}$ respectively. Then it will lift the ○-measures using `liftVertex` until they can no longer be lifted after which `solve` is used to mark vertices as won by ⬠ and set the strategies for these vertices. The process is then repeated for the ⬠-measures, marking vertices as won by ○ and setting the strategies for these vertices.

In the actual implementation it is often better to interleave these processes since in some games the ○-measures will reach their fixed point faster than the ⬠-measures or vice versa.

Algorithm 3.2: liftVertex

```
fn liftVertex($pm, \alpha, u$):
    if $u \in V_\alpha(\mathcal{G})$ : $newMeasure \leftarrow$ max{prog($pm[v]$, pr($u$)) | $v \in E(u)$}
    else: $newMeasure \leftarrow$ min{prog($pm[v]$, pr($u$)) | $v \in E(u)$}
    if $newMeasure > pm[u]$ :
        $pm[u] \leftarrow newMeasure$
        return true
    return false
```

The `liftVertex` function shown in Algorithm 3.2 applies the `prog` function to the measures of the successors of u and the priority of u. Then, for the vertices owned by α it takes the largest result, and for the vertices owned by $\bar{\alpha}$ it takes the lowest result. It stores that result in the measure function pm for u and returns true if the measure was updated.

Again, in the actual implementation this can be optimised. One such optimisation is to only lift vertices whose successors have been updated. This can be achieved by keeping track of a queue and enqueuing all the predecessors of a vertex after its measure has been updated.

Algorithm 3.3: solve

```
fn solve($pm, \alpha$):
    for every vertex $u \in V_\alpha(\mathcal{G})$ :
        if $pm[u] \neq \top$ :
            $\sigma_\alpha[u] \leftarrow \texttt{argmin}\,\{pm[v] \mid v \in E(u)\}$
    for every vertex $u \in V(\mathcal{G})$ :
        if $pm[u] \neq \top$ :
            if $u \in V_\alpha(\mathcal{G})$ : mark $u$ as won by $\alpha$ with strategy $\sigma_\alpha[u]$
            else: mark $u$ as won by $\alpha$
            $\mathcal{G} \leftarrow \mathcal{G} \setminus \{u\}$
```

The `solve` function shown in Algorithm 3.3 sets the strategy for all vertices that were won by α (which is the set of vertices whose $\bar{\alpha}$-measure did not reach $\top$) and owned by α. Then it marks all vertices won by α and removes them from the game.

Now we can introduce some lemmas to prove the correctness of the GPM algorithm for solving parity games.

Lemma 3.2 *Given a game $\mathcal{G}$, and a player $\alpha \in \{○, ⬠\}$ when* `gpm` *has finished executing it ensures that*

$$\forall_{u \in V(\mathcal{G})} pm_\alpha[u] = \begin{cases} \texttt{max}\left\{\texttt{prog}(pm_\alpha[v], \texttt{pr}(u)) \mid v \in E(u)\right\} & u \in V_\alpha(\mathcal{G}) \\ \texttt{min}\left\{\texttt{prog}(pm_\alpha[v], \texttt{pr}(u)) \mid v \in E(u)\right\} & u \in V_{\bar{\alpha}}(\mathcal{G}) \end{cases}$$

PROOF We prove this by contradiction, suppose there exists a $u \in V(\mathcal{G})$ such that

$$pm_\alpha[u] \neq \begin{cases} \texttt{max}\left\{\texttt{prog}(pm_\alpha[v], \texttt{pr}(u)) \mid v \in E(u)\right\} & u \in V_\alpha(\mathcal{G}) \\ \texttt{min}\left\{\texttt{prog}(pm_\alpha[v], \texttt{pr}(u)) \mid v \in E(u)\right\} & u \in V_{\bar{\alpha}}(\mathcal{G}) \end{cases}.$$

For `gpm` to terminate `liftVertex`(pm_α, α, u) must have returned **false** since otherwise *Updated* would have been set to **true** and either the loop on lines 4–8 or the loop on lines 10–14 of Algorithm 3.1 (depending on α) would continue. For `liftVertex` to return **false** we must have $newMeasure \leq pm_\alpha[u]$. There are two cases to consider:

Case $u \in V_\alpha(\mathcal{G})$:
In this case we have $newMeasure = \texttt{max}\left\{\texttt{prog}(pm_\alpha[v], \texttt{pr}(u)) \mid v \in E(u)\right\}$. Since we assumed that $pm_\alpha[u] \neq \texttt{max}\{\texttt{prog}(pm[v], \texttt{pr}(u)) \mid v \in E(u)\}$ we have that $pm_\alpha[u] < newMeasure$. Therefore, we know that in some previous iteration $pm_\alpha[u]$ must have been assigned a higher measure. By Definition 3.4 property 1 we know that the `prog` function is monotonic, therefore at least one vertex in $E(u)$ must have had a higher measure in the previous iteration than $\texttt{max}\left\{pm_\alpha[v] \mid v \in E(v)\right\}$ in the final iteration. However, `liftVertex` only assigns measures to vertices if they are higher than the previously stored measure in the measure function. Therefore, we have a contradiction because the measures of the vertices in $E(u)$ cannot be lower than they were in a previous iteration.

Case $u \in V_{\bar{\alpha}}(\mathcal{G})$:
In this case we have $newMeasure = \texttt{min}\left\{\texttt{prog}(pm_\alpha[v], \texttt{pr}(u)) \mid v \in E(u)\right\}$. Since we assumed that $pm_\alpha[u] \neq \texttt{min}\left\{\texttt{prog}(pm_\alpha[v], \texttt{pr}(u)) \mid v \in E(u)\right\}$ we have that

$pm_\alpha[u] < \mathit{newMeasure}$. Therefore, we know that in some previous iteration $pm_\alpha[u]$ must have been assigned a higher measure. By Definition 3.4 property 1 we know that the `prog` function is monotonic, therefore all vertices in $E(u)$ must have had a higher measure in the previous iteration than $\mathtt{min}\{pm_\alpha[v] \mid v \in E(u)\}$ in the final iteration. However, `liftVertex` only assigns measures to vertices if they are higher than the previously stored measure in the measure function. Therefore, we have a contradiction because the measures of the vertices in $E(u)$ cannot be lower than they were in a previous iteration. ■

Lemma 3.3 *Given a game $\mathcal{G}$, a player $\alpha \in \{\bigcirc, \pentagon\}$ and a $\bar{\alpha}$-progress measure $pm : V \to \mathcal{M}_\alpha$, $\mathtt{solve}(pm, \alpha)$ marks every vertex won by α as won by α and if that vertex is controlled by α it marks it with a winning strategy.*
PROOF By Definition 2.13 property 3 have that every vertex $u \in V(\mathcal{G})$ with $pm[u] \neq \top$ if and only if it is won by α. Therefore, we can conclude that the loop on lines 5–9 of Algorithm 3.3 marks every vertex won by α as won by α.

It remains to be proven that for every vertex $u \in V_\alpha(\mathcal{G})$ with $pm[u] \neq \top$ every the strategy for α determined in the loop on lines 2–4 Algorithm 3.3 and used to mark the vertices won and owned by α is a winning strategy. We can construct a play π along the edges of the game graph starting with the vertex u containing only vertices with a measure that is not $\top$ by induction on the construction of π.

Induction Hypothesis (IH): $\forall_{v \in \pi} pm[v] \neq \top$

Base case $\pi = \{u\}$:
Because $pm[u] \neq \top$ and v is the only vertex in π we find that the **IH** holds.

Step case $\pi = \pi \cdot v \cdot w$:
By the **IH** we have that $pm[v] \neq \top$. We can distinguish two cases $v \in V_\alpha(\mathcal{G})$ or $v \in V_{\bar{\alpha}}(\mathcal{G})$. If $v \in V_\alpha(\mathcal{G})$ then by Definition 2.13 property 2 and Definition 3.4 property 1 we have that there must exist a successor $w \in E(v)$ with $pm[w] \neq \top$. Otherwise, if $v \in V_{\bar{\alpha}}(\mathcal{G})$ then by Definition 2.13 property 1 and Definition 3.4 property 1 we have that all successors $w \in E(v)$ must have $pm[w] \neq \top$. As such, we have found a valid w such that the **IH** holds in all cases.

By Definition 2.13 property 3 we have that every vertex in the play π is won by α. Therefore, we know that every play starting in u restricted to the strategy for α determined by choosing the successor with the smallest measure is won by α. Therefore, the strategy that is computed by `solve` for the vertex u is a winning strategy. ■

The GPM algorithm needs to be given some parameters in order to resemble one of the existing progress measure-based algorithms.

Definition 3.5 (Concrete progress measure-based algorithm) A concrete progress measure-based algorithm is obtained by setting three parameters of our GPM algorithm: the measure spaces $\mathcal{M}_{\bigcirc}$ and $\mathcal{M}_{\pentagon}$, a total order $\leq$ for each of these measure spaces, and the progression functions $\mathtt{prog} : \mathcal{M}_{\bigcirc} \times \mathbb{N} \to \mathcal{M}_{\bigcirc}$ and $\mathtt{prog} : \mathcal{M}_{\pentagon} \times \mathbb{N} \to \mathcal{M}_{\pentagon}$. A concrete progress measure-based algorithm must satisfy the following property: the fixed point of a measure function obtained by repeatedly lifting vertices using `liftVertex` as defined in Algorithm 3.2 must be a progress measure as defined in Definition 2.13. □

Finally, we prove a theorem which ensures that the GPM correctly solves a parity game when parametrised appropriately.

Theorem 3.1 *Given a game $\mathcal{G}$ and the parameters $((\mathcal{M}_{\bigcirc}, \mathcal{M}_{\pentagon}), (\leq_{\bigcirc}, \leq_{\pentagon}), \texttt{prog})$ such that* `gpm` *yields a concrete progress measure as defined by Definition 3.5. When* `gpm` *has finished executing it ensures that every vertex is marked as won by ○ or ⬠ and if that vertex is controlled by the winning player it has marked it with a winning strategy.*

Proof By Definition 3.5 we have that the loops on lines 4–8 and 10–14 compute the ○-progress measure $pm_{\bigcirc}$ and the ⬠-progress measure $pm_{\pentagon}$ respectively. Therefore, by Lemma 3.3 we find that `gpm` marks vertices won by ○ as won by ○ and vertices won by ⬠ as won by ⬠, and if these vertices are owned by the player who wins them they are marked with a winning strategy. ∎

3.3 Existing progress measures

To adequately test PMTL we will use two existing progress measures: Small Progress Measures (SPM) and Ordered Progress Measures (OPM). These progress measures were chosen because comparing the measures from SPM is computationally inexpensive, but it has an exponential ($2^{\texttt{poly}(n)}$) time complexity because the size of its measure space $\mathcal{M}$ with respect to the size of the game is exponential. On the other hand, the measures from OPM come with more overhead but the size of OPM's measure space is quasi-polynomial ($2^{\texttt{poly}(\log n)}$) with respect to the size of the game which gives the algorithm a quasi-polynomial time complexity. We give a brief description of their rules and provide some worked examples.

3.3.1 Small Progress Measures

The simplest progress measure-based algorithm to consider is the Small Progress Measures algorithm[11]. The measure space of the ⬠-measures (odd) is defined as

$$\mathcal{M}_{\pentagon} = \left\{ \langle r_d, r_{d-2}, \dots, r_1 \rangle \mid 1 \leq i \leq d \wedge 0 \leq r_i \leq |\texttt{pr}^{-1}(i)| \right\} \cup \top$$

where d is the highest odd ⬠ priority in the game and $\texttt{pr}^{-1}(i)$ is the set of vertices with priority i in the game. The measure space of the ○-measures (even) is defined as

$$\mathcal{M}_{\bigcirc} = \left\{ \langle r_e, r_{e-2}, \dots, r_0 \rangle \mid 0 \leq i \leq e \wedge 0 \leq r_i \leq |\texttt{pr}^{-1}(i)| \right\} \cup \top$$

where e is the highest even ○ priority in the game.

An ⬠-measure keeps track of the amount of times certain odd ⬠ priorities are 'encountered' and an ○-measure does so for the even ○ priorities. Every element $r_i \in \left\{ p \in \mathbb{N} \mid p \leq |\texttt{pr}^{-1}(i)| \right\}$ of an α-measure represents how many times the priority i has been encountered.

SPM uses the p-truncated comparison operators $\geq_p$ and $>_p$ which compare p-truncated progress measures. When a progress measure gets p-truncated we set all elements r_i with $i < p$ to 0. The p-truncation operator is $|_p$. For example, given the ⬠-measures $a = \langle 0, 1, 3 \rangle$ and $b = \langle 0, 1, 2 \rangle$ we find that $a|_3 = b|_3 = \langle 0, 1, 0 \rangle$. (recall that the ⬠-measure has the form $\langle r_5, r_3, r_1 \rangle$ for a game where the highest odd ⬠ priority is 5) To make it clear if an element r_i of a measure equals 0 because it was truncated we often write $-$ instead of 0, as such we write $a|_3 = \langle 0, 1, - \rangle$. Because $a|_3 = b|_3$ we have that $a \geq_3 b$ but not $a >_3 b$. However, we do have $a >_1 b$ since $a|_1 = \langle 0, 1, 3 \rangle$ and $b|_1 = \langle 0, 1, 2 \rangle$.

When an ⬠-measure is lifted it 'encounters' a priority p, if p is even ○ then the resulting measure must be $\geq_p$ to the original measure, whereas if p is odd ⬠, the resulting measure must be $>_p$ than the original measure.

So when we lift a progress measure it must be greater than or equal to, or strictly greater than the original measure considering only elements r_i with $i \geq p$. Since every element r_i can only range between 0 and the amount of vertices with that priority in the game, this procedure will eventually lead to a measure which cannot be lifted any further which then becomes $\top$. This procedure is the `prog` function of the SMALL PROGRESS MEASURES.

Definition 3.6 (SMALL PROGRESS MEASURES' `prog` function) Given an α-measure $m \in \mathcal{M}_\alpha$ and a priority $p \in \mathbb{N}$ the function $\texttt{prog} : \mathcal{M}_\alpha \times \mathbb{N} \to \mathcal{M}_\alpha$ is defined as:

$$\texttt{prog}(m, p) = \begin{cases} \top & m = \top \\ \texttt{min}\{m' \in \mathcal{M}_\alpha \mid m' >_p m\} & p \equiv_2 \alpha \\ \texttt{min}\{m' \in \mathcal{M}_\alpha \mid m' \geq_p m\} & p \equiv_2 \bar{\alpha} \end{cases}$$ □

A quasi-polynomial variant of SMALL PROGRESS MEASURES is SUCCINCT PROGRESS MEASURES[12] which uses smaller measures that encode ordered trees.

We give some examples of the rules of the application of the progression function `prog` of the SMALL PROGRESS MEASURES.

Example 3.1 We use a game with 2 vertices with priority 1, 1 vertex with priority 3, 1 vertex with priority 5, and any number of ○-priority (even) vertices. For this game an ⬠-measure (odd) is of the form $\langle b_5, b_3, b_1 \rangle$.

$$\begin{aligned} b &= \langle 0, 1, 0 \rangle \\ \texttt{prog}(b, 4) &= \texttt{min}\{m \in \mathcal{M}_{⬠} \mid m \geq_4 b\} \\ &= \langle 0, -, - \rangle \end{aligned}$$

Here we see an ⬠-measure b encountering a vertex with priority 4. To get the resulting measure we must first truncate the measure b with the priority 4 which gives us $\langle 0, -, - \rangle$. Then we must find a measure m which, when also truncated with the priority 4, must be greater or equal to the truncated b measure. This gives us $\langle 0, -, - \rangle = \langle 0, 0, 0 \rangle$. □

Example 3.2 We again use a game with 2 vertices with priority 1, 1 vertex with priority 3, 1 vertex with priority 5, and any number of ○-priority (even) vertices. For this game an ⬠-measure (odd) is of the form $\langle b_5, b_3, b_1 \rangle$.

$$\begin{aligned} b &= \langle 0, 1, 2\rangle \\ \texttt{prog}(b, 1) &= \texttt{min}\{m \in \mathcal{M}_{\pentagon} \mid m >_1 b\} \\ &= \langle 1, 0, 0\rangle \end{aligned}$$

Here we see an ⬠-measure b encountering a vertex with priority 1. To get the resulting measure we must first truncate the measure b with the priority 1 which gives us $\langle 0, 1, 2\rangle$ since a p-truncation keeps all elements for priorities greater or equal to p. Then we must find a measure m which, when also truncated with the priority 1, must be greater than the truncated b measure. This gives us $\langle 1, 0, 0\rangle$ because $\langle 0, 1, 3\rangle$ would exceed the amount of vertices with priority 1 in the game and $\langle 0, 2, 0\rangle$ would exceed the amount of vertices 3 in the game. ◻

Example 3.3 We again use the same game as Example 3.1 and Example 3.2. For this game an ⬠-measure (odd) is of the form $\langle b_5, b_3, b_1\rangle$.

$$\begin{aligned} b &= \langle 1, 0, 2\rangle \\ \texttt{prog}(b, 3) &= \texttt{min}\{m \in \mathcal{M}_{\pentagon} \mid m >_3 b\} \\ &= \langle 1, 1, 0\rangle \end{aligned}$$

Here we see an ⬠-measure b encountering a vertex with priority 3. To get the resulting measure we must first truncate the measure b with the priority 3 which gives us $\langle 1, 0, -\rangle$. Then we must find a measure m which, when also truncated with the priority 3, must be greater than the truncated b measure. This gives us $\langle 1, 1, -\rangle$ because any change to the element which tracks the amount of encountered vertices with priority 1 is truncated away, therefore the smallest change we can make is to increase the element which counts the amount of encountered vertices with priority 3. ◻

Example 3.4 We reuse the game used for Example 3.1, Example 3.2, and Example 3.3. For this game an ⬠-measure (odd) is of the form $\langle b_5, b_3, b_1\rangle$.

$$\begin{aligned} b &= \langle 1, 1, 0\rangle \\ \texttt{prog}(b, 3) &= \texttt{min}\{m \in \mathcal{M}_{\pentagon} \mid m >_3 b\} \\ &= \top \end{aligned}$$

Here we see an ⬠-measure b encountering a vertex with priority 3. To get the resulting measure we must first truncate the measure b with the priority 3 which gives us $\langle 1, 1, -\rangle$. Then we must find a measure m which, when also truncated with the priority 3, must be greater than the truncated b measure. This gives us $\top$ because any change to the element which tracks the amount of encountered vertices with priority 1 is truncated away and every other counter has already reached the amount of vertices of its priority present in the game. Therefore, the smallest change we can make is to lift the measure to $\top$. ◻

Properties of the `prog` function

To use the Small Progress Measures with our algorithm its progression function `prog` must conform to the properties described in Definition 3.4.

It is straightforward to see that SPM's `prog` function is monotonic (Definition 3.4 property 1) since all three cases result in a measure that is either equal to original measure or the smallest possible step greater given some p-truncation. Therefore, if two measures encounter the same priority p their order will not change since they will be subjected to the same p-truncation.

Furthermore, we can see that SPM's `prog` function satisfies Definition 3.4 property 2 since any measure resulting from a `prog` with priority p must equal itself under a p-truncation. That is, $\texttt{prog}(m, p)|_p = \texttt{prog}(m, p)$ for every possible priority p and measure $m \in \mathcal{M}$. In fact, this measure equals itself under any q-truncation with $q \leq p$. Since the result of SPM's `prog` function with a priority q can never be lower than the q-truncation of the input measure we can conclude that the measure resulting from $\texttt{prog}(m, p)$ is p-stable.

Finally, we can see that SPM's `prog` function satisfies Definition 3.4 property 3 since for every possible priority q and measure $m \in \mathcal{M}$ we have that $\texttt{prog}(m, q) \geq m|_q$. If we have some starting measure $m \in \mathcal{M}$, a priority p, and a sequence of priorities Q where every priority $q \in Q, q \leq p$. Then we know that $\texttt{prog}(m, Q) \geq m|_p$. Therefore, because SPM's `prog` when given a measure m and a priority p always p-truncates m before taking the smallest possible step we know that if $\texttt{prog}(m, Q) \geq m|_p$ then $\texttt{prog}(\texttt{prog}(m, Q), p) \geq \texttt{prog}(m, p)$.

3.3.2 Ordered Progress Measures

Another type of quasi-polynomial progress measure was introduced by Fearnley et al.[9]. Their Ordered Progress Measures were based on the original quasi-polynomial algorithm by Calude et al.[2]. The measure space of the ⬠-measures (odd) is defined as

$$\mathcal{M}_{⬠} = \{\langle b_k, b_{k-1}, \dots, b_0 \rangle \mid 0 \leq i \leq k \wedge b_i \in \texttt{pr}(V(\mathcal{G})) \cup \{\_\}\}$$

where $\texttt{pr}(V(\mathcal{G}))$ is the set of priorities of vertices in $\mathcal{G}$, and $k = \lfloor \texttt{log}_2\, o \rfloor$ with o being the number of vertices with an odd ⬠ priority in $\mathcal{G}$. The measure space of the ○-measures (even) is defined as

$$\mathcal{M}_{○} = \{\langle b_k, b_{k-1}, \dots, b_0 \rangle \mid 0 \leq i \leq k \wedge b_i \in \texttt{pr}(V(\mathcal{G})) \cup \{\_\}\}$$

where $k = \lfloor \texttt{log}_2\, e \rfloor$ with e being the number of vertices with an even ○ priority in $\mathcal{G}$.

Every element $b_i \neq \_$ in an α-measure $b = \langle b_k, b_{k-1}, \dots, b_0 \rangle$ represents a chain of vertices derived from a partial (i.e. finite) play of the game. To define this chain of vertices we look only at the vertices with a priority of α's parity in the play and the first vertex in the play. This chain must have the property that all the vertices in the play between each pair of vertices in the chain must have a lower priority than the highest priority among the pair of vertices. We call this chain an α-chain, the value of the element b_i representing the chain is equal to the last vertex in the chain.

Example 3.5 Given a partial play through a game which encounters the following priorities: $\langle 2, 1, 3, 4, 3, 1, 2, 1, 0 \rangle$ (where the first vertex's priority is on the left and the last vertex's priority is on the right) we can identify an $\bigcirc$-chain (even) of the vertices with priorities $\langle 2, 4, 2, 0 \rangle$. Since $\mathtt{max}\{1, 3\} < \mathtt{max}\{2, 4\}$, $\mathtt{max}\{3, 1\} < \mathtt{max}\{4, 2\}$, and $1 < \mathtt{max}\{2, 0\}$ we find that the property holds for each pair of vertices in the chain. □

This property is intuitively related to properties 2 and 3 of Definition 3.4. Understanding this intuition is not necessary to understand the Ordered Progress Measures or how it satisfies the properties of Definition 3.4 but it gives insight into why it seems likely that these properties match what is needed for any progress measure-based algorithm to solve a parity game. Any pair of vertices in a chain with this property must either have a first vertex with a higher priority than the vertices in the play between the pair, in which case property 3 of Definition 3.4 might be applied, or the second vertex must have a higher priority than the vertices in the play between the pair, in which case property 2 of Definition 3.4 might be applied. While the property of a chain is only about the priorities of vertices and the properties of `prog` in Definition 3.4 are about the measures of vertices and their priorities we can see that they reason similarly about the priorities. In other words every α-chain represented by an element $b_i \neq \_$ in an α-measure corresponds to the prefix of a play which when encountered by any progress measure-based algorithm's α-measure should not cause the measure to decrease. While this intuitively connects the definition of the Ordered Progress Measures to the properties of `prog` outlined in Definition 3.4 this does not mean that OPM's `prog` function satisfies those properties. We will show that is the case after introducing the `prog` function.

The order $\sqsubseteq$ of the α-measures is determined by comparing two α-measures lexicographically from b_k to b_0 where each witness is ordered such that $\_$ is the smallest, then we have the priorities of $\bar{\alpha}$'s parity from the highest to the lowest, and finally we have the priorities of α's parity from the lowest to the highest. In other words we might say that we order the priorities by how much the α player prefers them. So for $\alpha = \bigcirc$ we prefer higher even priorities over lower even priorities, even priorities over odd priorities, lower odd priorities over higher odd priorities, and all priorities over $\_$.

Definition 3.7 (Raw Update) The raw update function $\mathtt{ru} : \mathcal{M}_\alpha \times \mathbb{N} \to \mathcal{M}_\alpha$ lifts an α-measure $b = \langle b_k, b_{k-1}, \dots, b_0 \rangle$ with a priority $p \in \mathbb{N}$ by finding all candidate indices j with $0 \leq j \leq k$ for which either:

1. All elements b_i with $i < j$ are of α's parity, or
2. $p > b_j$ and for all $i > j$ we have $b_i \geq d$

Then for each candidate index j we can construct a candidate measure $m \in \mathcal{M}_\alpha$ by setting $m_i = b_i$ for all $i > j$, setting $m_j = d$, and setting $m_i = \_$ for all $i < j$. The result of the raw update function is the largest candidate measure in the ordering $\sqsubseteq$ or the original measure b if the only candidate index is 0 and $\forall_{0 \leq i \leq k} b_i = \_ \vee b_i \geq d$.□

The intuition behind the raw update is that if the first rule is used we have found a longer α-chain by concatenating all chains represented by the b_i with $i < j$ and the new vertex. If the second rule is used only the last vertex in the chain is replaced with the newly encountered one and the chain remains the same length.

Since the second rule may only be used when the encountered priority is larger than the priority of the last vertex in the chain (b_j) we ensure that properties of the chain are preserved. Finally, it is possible to return the original measure in the exceptional where the first two rules are not applicable except for rule 1 with $j = 0$ which is always applicable but where applying rule 1 would result in a lower measure than b. An example of the latter would be $\texttt{ru}(\langle\_, 4, 2\rangle, 1)$ which we want to return $\langle\_, 4, 2\rangle$ instead of $\langle\_, 4, 1\rangle$.

Unfortunately, on its own the raw update is not a suitable progression function if we want the ORDERED PROGRESS MEASURES to be valid progress measures. Firstly, we need to define how a measure reaches $\top$. This is done using the value function `value`.

Definition 3.8 (Value function) For α-measures we define `value` as follows:

$$\texttt{value}(b) = \sum_{i=0}^{k} \begin{cases} 2^i & \text{if } b_i \text{ is of } \alpha\text{'s parity} \\ 0 & \text{otherwise} \end{cases}$$ □

The `value` function returns the sum of the lengths of the α-chains where the highest priority in the chain is of α's parity. If we find a chain that is longer than the amount of α-priority vertices in the game then we know that we have found a winning cycle for α at which point we can increase the measure to $\top$. The value of k which determines the size of the measure was chosen such that there are enough positions in the measures to record chains of at least the amount of α-priority vertices in the game.

We can now define the update function `up` which uses `value` to determine when a measure needs to be lifted to $\top$.

Definition 3.9 (Update) For α-measures we define `up` as follows:

$$\texttt{up}(b, p) = \begin{cases} \texttt{ru}(c, p) & \texttt{value}(\texttt{ru}(c, p)) \leq d \\ \top & \text{otherwise} \end{cases}$$

Where d is the amount of α-priority vertices in the game. □

One of the critical properties of the progression function `prog` is that it is monotonic (Definition 3.4 property 1), but the rules as described can result in the order between two measures switching after being lifted with the same vertex priority. (see Example 3.7) To get a monotonic progression function an antagonistic update rule is introduced which allows the opponent to increase the measure a minimal amount before the normal update rules are applied. More formally we can define the antagonistic function as such:

Definition 3.10 (Antagonistic Update) For α-measures we define `au` as follows:

$$\texttt{au}(b, p) = \min_{\sqsubseteq} \{\texttt{up}(c, p) \mid c \sqsupseteq b\}$$ □

The `prog` function of the ORDERED PROGRESS MEASURES is `au`.

It is challenging to implement the antagonistic update such that it can be computed in a reasonable amount of time. The antagonistic update and its monotonicity are discussed in chapter 4.

We give some examples of these rules being applied to clarify them.

Example 3.6 Here we assume the game has 5 ○-priority (even) and 6 ⬠-priority (odd) vertices. We look at how an ○-measure is lifted.

$$b = \langle 2, \_, 2 \rangle$$

Here we use rule 1 of the raw update `ru` (Definition 3.7) since b_i is even for $i < 2$. Therefore, rule 1 (with $j = 1$) is applicable and results in $\langle 2, 2, \_ \rangle$, applying rule 2 (with $j = 0$) yields $\langle 2, \_, 2 \rangle$. Since the raw update uses the largest result we get $\langle 2, 2, \_ \rangle$ because $2 > \_$ in our ordering.

Since $\texttt{value}(\langle 2, 2, \_ \rangle) = 2^2 + 2^1 = 6$ we have found an ○-cycle with at least 5 ○-priority vertices and the measure becomes $\top$ because we have exceeded the amount of ○-priority vertices in the game. Consequently, the result of $\texttt{up}(b, 2) = \top$.

However, to determine the result of $\texttt{au}(b, 2)$ we need to find the lowest result of $\texttt{up}(c, 2)$ where $c \sqsupseteq b$. The possible values of c depend on the sets of even and odd priorities in the game. For example, if the largest even priority in the game is 2 then there are no candidates for c where c_0 is set to $\_$ or an odd priority since b has the elements at indices 0, and 2 already set to the highest possible (most preferable for ○) priority and rule 1 can be used to set the element at index 1. Therefore, we would have $\texttt{au}(b, 2) = \texttt{up}(b, 2) = \top$.

However, if there is a larger even priority in the game, for example 4 then the measure $c = \langle 4, \_, \_ \rangle \sqsupset b$ but $\texttt{up}(c, 2) = \langle 4, \_, 2 \rangle$ because $\texttt{value}(\langle 4, \_, 2 \rangle) = 5$. To determine which c yields the lowest measure we would have to try every possible measure larger than b, but for this example it suffices that we can see that antagonistic update makes it such that $\texttt{au}(b, 2)$ is not always equal to $\texttt{up}(b, 2)$. □

Fearnley et al. also provide an example where the antagonistic update `au` is required to preserve the monotonicity of the `prog` function.

Example 3.7 We will illustrate this example here. Given a game with 5 ○-priority vertices and any number of ⬠-priority vertices we take two ○-measures: b and c.

$$b = \langle \_, 4, 2 \rangle$$
$$c = \langle 9, 8, \_ \rangle$$

We have $b \sqsubseteq c$ since $\_ < 9$ in our ordering. However, when we update these measures using a vertex with priority 6 we can see the order changes.

$$b' = \texttt{up}(b, 6) = \langle 6, \_, \_ \rangle$$
$$c' = \texttt{up}(c, 6) = \langle 9, 8, 6 \rangle$$

To obtain b' we use rule 1 of `ru` with candidate index $j = 2$ since this yields a higher measure than using rule 2 of `ru` with candidate index $j = 1$. Furthermore, $\texttt{value}(\langle 6, \_, \_ \rangle) \leq 5$ therefore $b' \neq \top$.

To obtain c' we use rule 1 of `ru` with candidate index $j = 0$ since there are no indices for which rule 2 is applicable because the priorities are higher than 6. Furthermore, $\texttt{value}(\langle 9, 8, 6\rangle) = 3 \leq 5$ therefore $c' \neq \top$.

We have $c' \sqsubseteq b'$ since $9 < 6$ in our ordering. From this we can conclude that the `up` function is not monotonic. Applying the antagonistic update function `au` instead gives:

$$b'' = \texttt{au}(b, 6) = \langle \_, 6, 6\rangle$$
$$c'' = \texttt{au}(c, 6) = \langle 9, 8, 6\rangle$$

To obtain b'' we attempt to find the smallest measure obtainable from a measure that is greater in our ordering than b. One such larger measure is $\langle \_, 6, \_\rangle$ which is in fact the smallest measure that is larger than b. We have $\texttt{up}(\langle \_, 6, \_\rangle, 6) = \langle \_, 6, 6\rangle$ using rule 1 of `ru` with candidate index $j = 0$. According to Fearnley et al. it is in practice sufficient to implement the antagonistic update by only considering the original measure b and the smallest measure that is larger than b. We have found this to not always be true; chapter 4 covers this topic in more detail. It is impractical to list every possible measure which is larger than b here but suffice it to say that there are no such measures which will result in a smaller measure than $\langle \_, 6, 6\rangle$.

To obtain c'' we attempt to find the smallest measure obtainable from a measure that is greater in our ordering than c. All measures greater than c either have a priority that is better for α than 9 at index 2, a priority that is better for α than 8 at index 1, or some priority that is not _ at index 0. In all these cases we have that rule 1 with candidate index $j = 0$ is still applicable resulting a measure where the priority at index 0 is 6 or higher and the priorities at indices 1 and 2 are also equal to or higher (in our ordering for $\bigcirc$) than 8 and 9 respectively which is a measure greater than or equal to $\langle 9, 8, 6\rangle$. Therefore, no such measure that is greater than c will yield a measure after the update that is greater than $\langle 9, 8, 6\rangle$. As such $\texttt{au}(c, 6) = \texttt{up}(c, 6)$.

Since the measure $b'' \sqsupset b$ we still make progress which means that given sufficient applications of an even priority the measure $\langle 6, \_, \_\rangle$ will still be reached eventually. However, by slowing the progress of the b measure the `au` function is monotonic. □

Finally, we give an example of solving an entire game using the ORDERED PROGRESS MEASURES. This is a simple example where we only need to compute the $\bigcirc$-measures, and we have no instances where the antagonistic update `au` is needed. Therefore, for the sake of brevity we will not be listing measures that are larger than the input measure of `au` while explaining the calculations in detail.

Example 3.8 As our example we use the game from Figure 2.1. We start by setting the $\bigcirc$-measure of every vertex to $\langle \_, \_\rangle$ as shown in Figure 3.1.

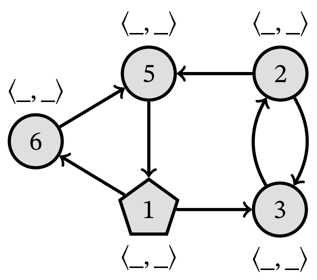


Figure 3.1: Set all measures to $\langle \_, \_ \rangle$.

For every vertex we can now attempt to lift its measure by applying the `prog` function on the measures of its successors and its priority. Since we are calculating ○-measures (even), if the vertex is owned by ○, we use the largest result of these `prog` s. If the vertex is owned by ⬠ (in this case only the vertex with priority 1), we use the smallest result.

We start by lifting the ○-vertex with priority 6, but we could have started with any other vertex, the order does not matter. Since the vertex has only one successor it computes $\texttt{prog}(\langle \_, \_ \rangle, 6) = \langle \_, 6 \rangle$, where $\langle \_, \_ \rangle$ is the measure of the successor, the ○-vertex with priority 5. The measure here results from rule 1 of the raw update `ru`. (see Definition 3.7) Recall that an element b_i ○-measure $b = \langle b_1, b_0 \rangle$ is a witness of a chain of at least 2^i ○-priority vertices where the last vertex in the chain has the priority b_i. Here we simply start recording a new chain of ○-priority vertices ending with the ○-priority 6.

We can repeat this for the ○-vertices with priority 5 and priority 3, since the rules allow for the priority of the last vertex in a chain of ○-priority vertices to be odd ⬠. The result is shown in Figure 3.2.

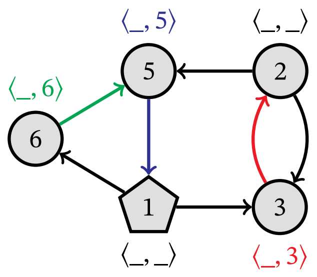


Figure 3.2: Lift the ○-vertex with priority 6 to $\langle \_, 6 \rangle$, the ○-vertex with priority 5 to $\langle \_, 5 \rangle$, and the ○-vertex with priority 3 to $\langle \_, 3 \rangle$.

Next we can lift the ○-vertex with priority 2. This vertex has two possible successors, however both of these successor vertices have a measure that only contains a witness to a chain of ○-priority (even) vertices with an ⬠-priority (odd) vertex at the end of the chain. Because of this we cannot extend the chain of vertices and must instead start anew using rule 1 of the raw update `ru`. The ○-vertex with priority 2 gets lifted to the new measure $\texttt{prog}(\langle \_, 5 \rangle, 2) = \texttt{prog}(\langle \_, 3 \rangle, 2) = \langle \_, 2 \rangle$ as shown in Figure 3.3.

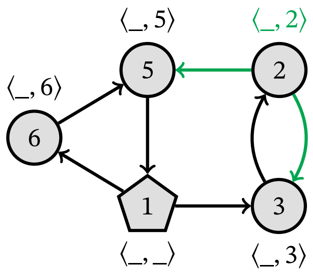


Figure 3.3: Lift the ○-vertex with priority 2 to $\langle \_, 2\rangle$.

The last remaining vertex that we have not lifted yet is the ⬠-vertex with priority 1. This vertex can play towards the ○-vertex with priority 6 and the ○-vertex with priority 3. If it were to be lifted using the ○-vertex with priority 6 we could extend the chain of ○-priority vertices witnessed by the 6 in the measure $\langle \_, 6\rangle$ to form a chain of length $2^1 = 2$. Therefore, using rule 1 of the raw update `ru` we have $\texttt{prog}(\langle \_, 6\rangle, 1) = \langle 1, \_\rangle$.

If we instead lift using the ○-vertex with priority 3 we cannot extend the chain because it already ended with an odd ⬠ priority. Therefore, we have $\texttt{prog}(\langle \_, 3\rangle, 1) = \langle \_, 1\rangle$ as shown in Figure 3.4. Note that playing towards the ○-vertex with priority 3 matches the winning strategy we highlighted when we first discussed this game in section 2.1

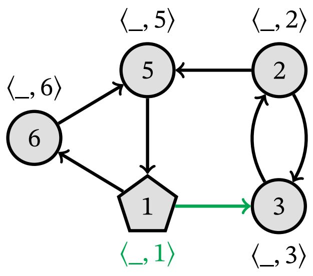


Figure 3.4: Lift the ⬠-vertex with priority 1 to $\langle \_, 1\rangle$.

Now we have lifted every vertex at least once. There are a couple of vertices that can now be lifted again because their successors got lifted.

The ○-vertex with priority 3 can be lifted because $\texttt{prog}(\langle \_, 2\rangle, 3) = \langle 3, \_\rangle \sqsupseteq \langle \_, 3\rangle$. This extends the length 1 chain of ○-priority vertices witnessed by the 2 in $\langle \_, 2\rangle$ forming a new chain with length 2.

Using this new measure the ○-vertex with priority 2 can now also be lifted again. Here $\texttt{prog}(\langle 3, \_\rangle, 2) = \langle 3, 2\rangle$, because we cannot extend the chain of ○-priority vertices witnessed by the 3 in the measure because it ended with an ⬠-priority vertex. We can however keep track of an additional chain of ○-priority vertices with length 1.

The ⬠-vertex with priority 1 can also be updated using the new measure which was assigned to the ○-vertex with priority 3. There we have $\texttt{prog}(\langle 3, \_\rangle, 1) = \langle 3, 1\rangle$ which is still smaller than the $\texttt{prog}(\langle \_, 6\rangle, 1) = \langle 1, \_\rangle$ which could be gotten by playing towards the ○-vertex with priority 6. Note that $1 > 3$ in the ordering used for ○-measures (even) for the Ordered Progress Measures. Since the vertex is owned by ⬠ and we are computing ○-measures we use the smaller of the two results.

These three updates are shown in Figure 3.5.

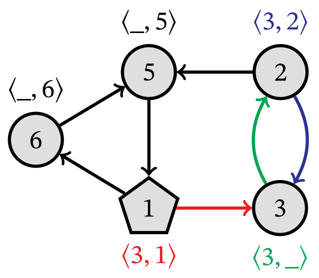


Figure 3.5: Lift the ○-vertex with priority 3 to $\langle 3, \_\rangle$, the ○-vertex with priority 2 to $\langle 3, 2\rangle$, and the ⬠-vertex with priority 1 to $\langle 3, 1\rangle$.

Using the new measure for the ⬠-vertex with priority 1 we can lift the ○-vertex with priority 5. Here we have $\texttt{prog}(\langle 3, \_\rangle, 5) = \langle 5, \_\rangle$, because we can replace the last vertex (with priority 3) in the length 2 chain of ○-priority vertices with the vertex with priority 5.

In turn, the ○ vertex with priority 6 can be lifted using this new measure. Here we have $\texttt{prog}(\langle 5, \_\rangle, 6) = \langle 6, \_\rangle$, because 6 is a higher priority than 5 we are able to replace the last vertex in the chain of ○-priority vertices even though it had an odd ⬠ priority. The result of these two updates is shown in Figure 3.6.

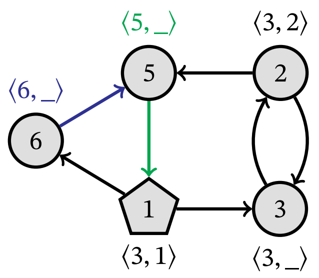


Figure 3.6: Lift the ○-vertex with priority 5 to $\langle 5, \_\rangle$, and the ○-vertex with priority 6 to $\langle 6, \_\rangle$.

In this state no further lifts can occur because the ⬠-vertex with priority 1 will never be lifted towards the ○-vertex with priority 6 because lifting towards the ○-vertex with priority 3 results in a lower measure.

Now that we have reached the fixpoint we can determine which vertices are won by which player. Since none of the vertices reached the $\top$ ◯-measure (even) we find that all the vertices are won by odd ⬠. We can determine the strategy for odd ⬠ to win the game by looking at which of its successors gives us the lowest measure, in this case that is the ◯-vertex with priority 3. □

Properties of the `prog` function

To use the Ordered Progress Measures with our algorithm its progression function `prog` must conform to the properties described in Definition 3.4.

The first property holds because the antagonistic update `au` turns the function `up` into a monotonic function. In practice, implementing this 'trick' which can make any function monotonic can be difficult. In chapter 4 we discuss the monotonicity of the Ordered Progress Measures in greater detail.

To show that the second property of Definition 3.4 holds for OPM's `prog` function we will first show that it holds for the `up` function and then show that the property is preserved when applying the antagonistic update.

The second property holds because rule 1 of the raw update `ru` (Definition 3.7) always yields a higher measure and rule 2 may only decrease a measure if the encountered priority is greater or equal to a priority stored in the tuple. Additionally, a priority can only be stored as an element in the measure if the priority is encountered. Given a starting measure $m \in \mathcal{M}_\alpha$ and a priority $p \in \mathbb{N}$ there are three cases to consider: the starting measure is $\top$, $\mathtt{up}(m,p) = m$, or $\mathtt{up}(m,p) \neq m$. It is trivial to see that if $m = \top$ that the measure cannot decrease by encountering any priority. If $\mathtt{up}(m,p) = m$ then both rules of the raw update `ru` must not have been applicable this can only occur if p is smaller than all elements in m. Consequently, any priority $q \leq p$ will also have the property $\mathtt{up}(m,q) = m$ which means that m is p-stable. Finally, if $m' = \mathtt{up}(m,p) \neq m$ then the resulting measure m' must have some index j such that for all $i > j$ we have $m'_i \geq p$, m'_j, and for all $i < j$ we have $m'_i = \_$. No priority $q \leq p$ can result in $\mathtt{up}(m',q)$ being lower than m' because the elements m'_i for all $i \geq j$ cannot be lowered by priorities less or equal to p and the remaining elements m'_i for all $i < j$ are already at their smallest possible value _.

Now what remains to be shown is that this property is preserved when the antagonistic update is applied. The result of $\mathtt{au}(m,p)$ is the smallest result of $\mathtt{up}(c,p)$ for all $c \sqsupseteq m$. Therefore, to show that the result of $\mathtt{au}(m,p)$ is p-stable we must show that the result of $\mathtt{up}(c,p)$ is p-stable for all $c \sqsupset m$. We can use the same logic we used in the previous paragraph which already showed that for any $m \in \mathcal{M}_\alpha$ and priority $p \in \mathbb{N}$ we have that $\mathtt{up}(m,p)$ is p-stable. Therefore, we can conclude that the second property of Definition 3.4 holds for OPM's `prog` (`au`) function.

Similarly to what we did to show that the second property of Definition 3.4 holds for OPM's `prog` function we will first show that the third property holds for the `up` function and then show that the property is preserved when applying the antagonistic update.

The third property holds because after a raw update `ru` of a measure $m \in \mathcal{M}_\alpha$ and a priority $p \in \mathbb{N}$ with a candidate index j the resulting measure will have elements with indices $i < j$ set to _ and all elements with indices $i > j$ will be equal to _ or greater than p. Consequently, given a starting measure $m \in \mathcal{M}_\alpha$ and a sequence of priorities Q where for every priority $q \in Q$ with $q \leq p \vee q \equiv_2 \alpha$ we find that all elements with indices $i > j$ in the result of $\texttt{up}(\texttt{up}(m, Q), p)$ are greater than or equal to the elements with those indices in the result of $\texttt{up}(m, p)$ since the priorities in Q cannot decrease the values of the elements with indices $i > j$. (either because they have a lower priority than the ones stored at those indices or because they are of α's parity which never yields a lower measure) Finally, what remains to be shown is that the same candidate index j (or a candidate index resulting in a higher measure) is chosen for $\texttt{up}(\texttt{up}(m, Q), p)$ and $\texttt{up}(m, p)$. If j is a candidate index for rule 2 of $\texttt{ru}$ then it does not depend on the elements with indices $i < j$ and is therefore still valid for the resulting measure after applying the progression function to sequence Q. On the other hand, if j is a candidate index for rule 1 of $\texttt{ru}$ then it *does* depend on all elements with indices $i < j$ being of α's parity. However, after the right-most priority in the sequence Q is encountered the resulting measure will be high enough such that any further priorities (to the left of the right-most priority) $q \in Q$ with $q \leq p$ cannot cause the measure to become low enough that encountering p would result in a measure that is lower than $\texttt{up}(m, p)$. This is because when encountering the last priority (right-most) in the sequence Q rule 1 can be used with the candidate index j. This causes the priority at index j in the measure to be increased. Since the priority at index j in the measure is now lower than p and not equal to _ when computing $\texttt{up}(\texttt{up}(m, Q), p)$ rule 2 can be used with index j resulting in the same measure as (or, if Q contained priorities of α's parity, a higher measure than) $\texttt{up}(m, p)$. Essentially, after the last priority in the sequence Q has been applied (it is the first priority to be applied according to Definition 3.1) we can use the same reasoning as was used in the rule 2 case using the fact that the resulting measure is always larger than the original measure m because rule 1 was used.

Finally, what remains to be shown is that this property is preserved when the antagonistic update is applied. The result of $\texttt{au}(m, p)$ is the smallest result of $\texttt{up}(c, p)$ for all $c \sqsupseteq m$. We can notice here that when $\texttt{up}(m, p)$ would use rule 2 of $\texttt{ru}$ with candidate index j then $\texttt{au}(m, p) = \texttt{up}(m, p)$ because if a measure $c \sqsupset m$ would have $\texttt{up}(c, p) \sqsubset \texttt{up}(m, p)$ then the priority at index j in c would have to be greater than p otherwise the same candidate index would be applicable and the same (or a higher) measure would be obtained. However, we can now distinguish two cases: either the priority at index j in c is greater than the priority at index j in m, or the priority at index j in c is smaller or equal to the priority at index j in m. If the priority at index j in c is greater, then that priority must be of α's parity because $c \sqsupset m$ which causes a contradiction because then $\texttt{up}(c, p) \sqsupset \texttt{up}(m, p)$. If the priority at index j in c is smaller or equal to the priority at index j in m then there must be an index $k > j$ such that the priority at index k of c is better for α (i.e. greater in OPM's ordering for the α-measures) than the priority at index k of m. If this is the case then the result of $\texttt{up}(c, p)$ must always be greater than $\texttt{up}(m, p)$ because a higher indexed element was improved for α resulting a contradiction.

Therefore, we only need to consider cases where the priority $q \in Q$ results in $\texttt{up}$ using rule 1 of $\texttt{ru}$ and cases where $\texttt{up}(m, p)$ uses rule 1. For the former we can consider two cases either $\exists_{r \in Q} r \equiv_2 \alpha$ or $\texttt{up}(m, p)$ uses rule 1 because either rule 1 was applicable for m or one or more priorities of α's parity increased the measure such that rule 1 became applicable. If there are priorities of α's parity in Q then this can yield a measure where rule 1 of $\texttt{ru}$ is applicable. However, this can never yield a measure that is low enough such that $\texttt{au}(\texttt{au}(m, Q), p) \sqsubset \texttt{au}(m, p)$ because the application of rule 1 always yields a higher measure, and we already found that the application of rule 2 will yield a measure that is high enough. Finally, if $\texttt{up}(m,p)$ uses rule 1 then we find using Definition 3.4 property 1 that $\texttt{au}(m,p) \sqsubseteq \texttt{au}(\texttt{au}(m,Q),p)$ because $\texttt{au}(m,p) \sqsubseteq \texttt{up}(m,p)$ and $\texttt{up}(m,q) \sqsupseteq m$ where q is the last priority in Q and because rule 1 is applicable for $\texttt{up}(m,q)$ because it was applicable for $\texttt{up}(m,p)$ where we can use that $\texttt{up}(m,q) \sqsubseteq \texttt{up}(\texttt{up}(m,Q'),q)$ where Q' is the start of the sequence Q without the last priority q.

3.4 Properties of paths

We will use the novel concept of a *forced path* which is equivalent to a (tangle) attractor but is defined for paths instead of sets of vertices.

Definition 3.11 (Forced Path) Given two vertices $u, v \in V(\mathcal{G})$ a path π from u to v is an α-*forced path* if there is some strategy σ_α for α such that all plays starting at u reach v or end up in a cycle that is won by α and the path π is one of the paths that such a play could follow. ◽

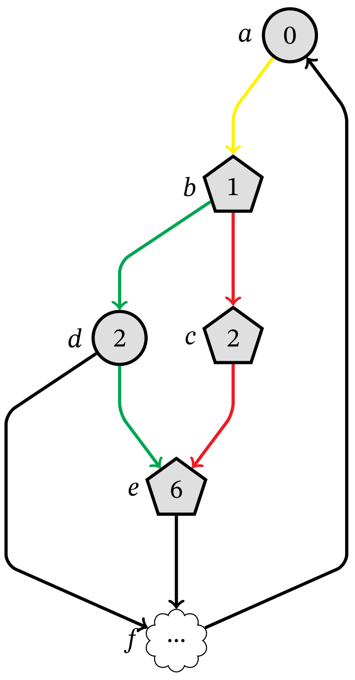


Figure 3.7: A parity game graph with two highlighted ◯-*forced paths*

Example 3.9 Figure 3.7 shows two ◯-*forced paths* from vertex a to vertex e. The even ◯ player has a strategy such that the play must continue towards vertex e. We see that regardless of the choice made by the odd ⬠ player at vertex b by making the right choice at vertex d the even ◯ player can ensure e is reached. ◽

Another example of an α-forced path is the path formed in the proof of Lemma 3.3 since if it was not an α-forced path there would be a vertex owned by $\bar{\alpha}$

which could play to a vertex with a measure equalling $\top$.

Additionally, we commonly use $\mathcal{P}$ to refer to the set of all paths between two vertices u and v in a game, and we use $\mathcal{F} \subseteq \mathcal{P}$ to refer to the set of forced paths between those two vertices.

Since the forced path is equivalent to a (tangle) attractor we can prove two lemmas to show that attractors and tangle attractors imply the existence of a forced path.

Lemma 3.4 *Given two vertices $u, v \in V(\mathcal{G})$ with $u \in \mathtt{attr}^{\mathcal{G}}_{\alpha}(\{v\}, P)$ there exists an α-forced path π from u to v containing only vertices with priorities in the set P.*
PROOF By Lemma 2.3 there exists a strategy σ_α for α such that any play must continue from u to v. The forced path is obtained by, for each vertex along the path starting at u, choosing an appropriate successor. If the vertex is owned by α the strategy σ_α should be used, if the vertex is owned by $\bar{\alpha}$ any choice of successor will suffice. Since the computation of `attr` restricts the attracted vertices to those with a priority in P, the formed path also contains only those vertices. ■

Lemma 3.5 *Given two vertices $u, v \in V(\mathcal{G})$ with $u \in \mathtt{attrT}^{\mathcal{G}}_{\alpha}(\{v\}, P)$ there exists an α-forced path π from u to v containing only vertices with priorities in the set P.*
PROOF By Lemma 2.5 there exists a strategy σ_α for α such that any play must continue from u to v, or get stuck in a winning cycle for α. The forced path is obtained by, for each vertex along the path starting at u, choosing an appropriate successor. If the vertex is owned by α the strategy σ_α should be used, if the vertex is owned by $\bar{\alpha}$ any choice of successor will suffice. Since the computation of `attrT` restricts the attracted vertices to those with a priority in P, the formed path also contains only those vertices. ■

Benerecetti et al.[1] introduce the notion of the measure $\eta(\pi)$ of a finite path. A version of their definition which has been adapted to fit the terminology used in this thesis is shown in Definition 3.12. Additionally, this definition allows for some starting measure of the first vertex in the path.

Definition 3.12 (Path Measure) The measure $\eta(\pi, m)$ of a finite path π where the last vertex $v \in \pi$ has the starting measure m is recursively defined as such:

$$\eta(\pi, m) \triangleq \begin{cases} m & \text{if } \pi = \{v\} \\ \mathtt{prog}(\eta(\pi', m), \mathtt{pr}(u)) & \text{if } \pi = u \cdot \pi',\ \text{for some unique } u \in V(\mathcal{G}) \end{cases} \qquad \square$$

Note that this is very similar to our definition of `prog` when applied to a sequence of properties. The following lemma shows the connection between the two.

Lemma 3.6 *Given a finite path π, a starting measure m, and the path's measure $\eta(\pi, m)$, we have $\eta(\pi, m) = \mathtt{prog}(m, Q)$ where Q is a sequence with all the priorities of the vertices along the path π in reverse order, excluding the last vertex in the path which had the starting measure m.* □

We can now define and prove some lemmas which will be useful in chapter 5 to prove the correctness of our algorithm.

Lemma 3.7 *Given two vertices $u, v \in V(\mathcal{G})$ where $\mathcal{F} \subseteq \mathcal{P}$ is the set of all α-forced paths from u to v, the parameters $((\mathcal{M}_{\bigcirc}, \mathcal{M}_{\pentagon}), (\leq_{\bigcirc}, \leq_{\pentagon}), \texttt{prog})$ such that $\texttt{gpm}$ yields a concrete progress measure as defined by Definition 3.5, and an α-measure function $\mu : V \to \mathcal{M}_\alpha$ where $\forall_{v \in V(\mathcal{G})} \mu[v] \leq \texttt{gpm}()[pm_\alpha][v]$*[1] *we have that:*

$$\min_{\pi \in \mathcal{F}} \eta(\pi, \mu[v]) \leq \texttt{gpm}()[pm_\alpha][u]$$

Proof We prove this by induction on the construction of an α-forced path $\pi \in \mathcal{F}$. We aim to construct such a path π such that $\eta(\pi, \mu[v]) \leq \texttt{gpm}()[pm_\alpha][u]$.

Induction Hypothesis (IH): given an α-measure function μ where $\forall_{v \in V(\mathcal{G})} \mu[v] \leq \texttt{gpm}()[pm_\alpha][v]$ we have $\eta(\pi, \mu[w]) \leq \texttt{gpm}()[pm_\alpha][u]$ where w is the last vertex in π.

Base case $\pi = \{u, w\}$:
First we fix some α-measure function μ where $\forall_{v \in V(\mathcal{G})} \mu[v] \leq \texttt{gpm}()[pm_\alpha][v]$. We distinguish two cases: either $u \in V_\alpha$ or $u \in V_{\bar{\alpha}}$. If $u \in V_\alpha$ then by Lemma 3.2 we have that $\texttt{gpm}()[pm_\alpha][u] = \texttt{max}\left\{\texttt{prog}(\texttt{gpm}()[pm_\alpha][x], \texttt{pr}(u)) \mid x \in E(u)\right\}$. Additionally, we have $\eta(\{u, w\}, \mu[w]) = \texttt{prog}(\mu[w], \texttt{pr}(u))$ by Definition 3.12 and $w \in E(u)$. Therefore, by Definition 3.4 property 1 we have that $\eta(\pi, \mu[w]) \leq \texttt{gpm}()[pm_\alpha][u]$. If $u \in V_{\bar{\alpha}}$ then by Lemma 3.2 we have that $\texttt{gpm}()[pm_\alpha] = \texttt{min}\left\{\texttt{prog}(\texttt{gpm}()[pm_\alpha][x], \texttt{pr}(u)) \mid x \in E(u)\right\}$. Now we can again distinguish two cases: either $E(u) = \{w\}$ or $E(u) \neq \{w\}$. If $E(u) = \{w\}$ then $\texttt{gpm}()[pm_\alpha][u] = \texttt{prog}(\texttt{gpm}()[pm_\alpha][w], \texttt{pr}(u)) \geq \eta(\pi, \mu[w])$ by Definition 3.4 property 1 and the requirement on μ. Otherwise, if $E(u) \neq \{w\}$ then w.l.o.g. we can choose $w \in E(u)$ to be any successor of u in an α-forced path between u and v. We can do this by Definition 3.11 because if there exists some strategy σ_α for α such that all plays starting at u eventually reach v or end up in a cycle won by α then all successors w of u must either have $\texttt{gpm}()[pm_\alpha][w] = \top$ by Theorem 3.1 and Definition 2.13 in which case they will not be the successor which grants $\texttt{gpm}()[pm_\alpha][u]$ its minimal value, or w must be in a forced path between u and v. Since we have free choice of $w \in E(u)$ we can choose w such that $\texttt{prog}(\mu[w], \texttt{pr}(u)) \leq \texttt{min}\left\{\texttt{prog}(\texttt{gpm}()[pm_\alpha][x], \texttt{pr}(u)) \mid x \in E(u)\right\}$ using Definition 3.4 property 1 and the requirement on μ.

Step case $\pi = \pi' \cdot x$:
First we fix some α-measure function μ where $\forall_{v \in V(\mathcal{G})} \mu[v] \leq \texttt{gpm}()[pm_\alpha][v]$ and we use w to be the last vertex in π'. We distinguish two cases: either $w \in V_\alpha$ or $w \in V_{\bar{\alpha}}$. If $w \in V_\alpha$ then by Lemma 3.2 we have that $\texttt{gpm}()[pm_\alpha][w] = \texttt{max}\left\{\texttt{prog}(\texttt{gpm}()[pm_\alpha][x], \texttt{pr}(w)) \mid x \in E(u)\right\}$. Additionally, we have $\eta(\{w, x\}, \mu[x]) = \texttt{prog}(\mu[x], \texttt{pr}(w))$ by Definition 3.12 and $x \in E(w)$. Therefore, by Definition 3.4 property 1 we have that $\eta(\{w, x\}, \mu[x]) \leq \texttt{gpm}()[pm_\alpha][w]$. If $u \in V_{\bar{\alpha}}$ then by Lemma 3.2 we have that $\texttt{gpm}()[pm_\alpha][w] = \texttt{min}\left\{\texttt{prog}(\texttt{gpm}()[pm_\alpha][x], \texttt{pr}(u)) \mid x \in E(u)\right\}$. Now we can again distinguish two cases: either $E(w) = \{x\}$ or $E(w) \neq \{x\}$. If $E(w) = \{x\}$ then $\texttt{gpm}()[pm_\alpha][w] = \texttt{prog}(\texttt{gpm}()[pm_\alpha][x], \texttt{pr}(w)) \geq \eta(\{w, x\}, \mu[x])$ by Definition 3.4 property 1 and the requirement on μ. Otherwise, if $E(u) \neq \{x\}$ then w.l.o.g. we can choose $x \in E(w)$ to be any successor of w in an α-forced path between u and v. We can do this by Definition 3.11 because if there exists some strategy σ_α for α such that all plays starting at u eventually reach v or end up in

[1]The notation $\texttt{gpm}()[pm_\alpha][x]$ refers to the α-measure of x after $\texttt{gpm}$ has finished executing

a cycle won by α then all successors x of w must either have $\mathtt{gpm}()[pm_\alpha][x] = \top$ by Theorem 3.1 and Definition 2.13 in which case they will not be the successor which grants $\mathtt{gpm}()[pm_\alpha][w]$ its minimal value, or x must be in a forced path between u and v. Since we have free choice of $x \in E(w)$ we can choose x such that $\mathtt{prog}(\mu[x], \mathtt{pr}(w)) \leq \mathtt{min}\{\mathtt{prog}(\mathtt{gpm}()[pm_\alpha][y], \mathtt{pr}(w)) \mid y \in E(w)\}$ using Definition 3.4 property 1 and the requirement on μ.

We have now proven that $\eta(\{w, x\}, \mu[x]) \leq \mathtt{gpm}()[pm_\alpha][w]$. Using this we can fix an α-measure function μ' where $\forall_{v \in V(\mathcal{G})} \mu'[v] \leq \mathtt{gpm}()[pm_\alpha][v]$ and $\mu'[w] = \eta(\{w, x\}, \mu[x])$. Then by the **IH** (setting μ in the **IH** to our μ') we have that $\eta(\pi', \mu'[w]) \leq \mathtt{gpm}()[pm_\alpha][u]$. Finally, because we have $\eta(\pi, \mu[x]) = \eta(\pi', \mu'[w])$ we find that $\eta(\pi, \mu[x]) \leq \mathtt{gpm}()[pm_\alpha][u]$. ∎

Lemma 3.8 *Given an attractor decomposition $\mathcal{H}_\alpha(\mathcal{G}, \preceq)$, the parameters $((\mathcal{M}_\bigcirc, \mathcal{M}_\pentagon), (\leq_\bigcirc, \leq_\pentagon), \mathtt{prog})$ such that* gpm *yields a concrete progress measure as defined by Definition 3.5, and an α-measure function $pm : V \to \mathcal{M}_\alpha$ where $\forall_{v \in V(\mathcal{G})} pm[v] \leq \mathtt{gpm}()[pm_\alpha][v]$, $\preceq$ orders vertices by their value in pm, and for each vertex $v \in V(\mathcal{G})$, $pm[v]$ is $\mathtt{pr}(v)$-stable. For every region $(v, \mathcal{A}) \in \mathcal{H}_\alpha(\mathcal{G}, \preceq)$ and every vertex $u \in \mathcal{A}$ for which there exists a forced path from u to v containing only vertices with priorities which are of α's parity or that are less or equal to $\mathtt{pr}(v)$ we have:*

$$\mathtt{prog}(pm[v], \mathtt{pr}(u)) \leq \mathtt{gpm}()[pm_\alpha][u]$$

Proof We prove this by induction on the construction of the sequence $\mathcal{H}_\alpha(\mathcal{G}, \preceq)$. (see Definition 2.11)

Induction Hypothesis (IH): For every region $(v, \mathcal{A}) \in \mathcal{H}$ and every vertex $u \in \mathcal{A}$ we have:

$$\mathtt{prog}(pm[v], \mathtt{pr}(u)) \leq \mathtt{gpm}()[pm_\alpha][u]$$

Base case $\mathcal{H} = \langle (v, \mathcal{A}) \rangle$:
Using Lemma 3.7 we have that given the set of all α-forced paths $\mathcal{F} \subseteq \mathcal{P}$ between u and v we have $\mathtt{min}_{\pi \in \mathcal{F}} \eta(\pi, pm[u]) \leq \mathtt{gpm}()[pm_\alpha][u]$. Additionally, because we know that $pm[v]$ is $\mathtt{pr}(v)$-stable by Definition 3.3 we have that for all finite paths $\pi \in \mathcal{P}$ containing only priorities of α's parity or that are lower or equal to $\mathtt{pr}(v)$, $\mathtt{prog}(pm[v], \mathtt{pr}(u)) \leq \eta(\pi, pm[v])$ because by Lemma 3.6 $\eta(\pi, pm[v]) = \mathtt{prog}(\mathtt{prog}(pm[v], Q), \mathtt{pr}(u))$ where Q is the sequence of priorities of the vertices in π except the first vertex u. Because $\mathcal{F} \subseteq \mathcal{P}$, that is the set of forced paths from u to v is a subset of all the paths from u to v, we can conclude that $\mathtt{prog}(pm[v], \mathtt{pr}(u)) \leq \mathtt{gpm}()[pm_\alpha][u]$. (Note that $\mathcal{F}$ is non-empty because by Lemma 3.5 we know there exists a forced path from u to v)

Step case $\mathcal{H} = \mathcal{H}' \cdot (v, \mathcal{A})$
For some $v' \in V(\mathcal{G}) \setminus V(\mathcal{G}')$, where $\mathcal{G}'$ is the game after the regions in $\mathcal{H}'$ have been removed, with $pm[v] \leq pm[v']$ we have:

$$\forall_{u' \in V(\mathcal{G}) \setminus \mathcal{G}'} \mathtt{prog}(pm[v'], \mathtt{pr}(u')) \leq \mathtt{gpm}()[pm_\alpha][u'] \qquad \text{by the } \textbf{IH}$$
$$\forall_{u' \in V(\mathcal{G}) \setminus \mathcal{G}'} \mathtt{prog}(pm[v], \mathtt{pr}(u')) \leq \mathtt{prog}(pm[v'], \mathtt{pr}(u')) \qquad \text{by Definition 3.4 prop. 1}$$

Note that the set $V(\mathcal{G}) \setminus V(\mathcal{G}')$ is the set of vertices which were removed from the game by the regions in $\mathcal{H}'$ which is why the **induction hypothesis** is applicable to the vertices in that set.

We can again use Lemma 3.7 but now we only consider the set of all α-forced paths $\mathcal{F}' \subseteq \mathcal{F} \subseteq \mathcal{P}$ between u and v which exclusively contain vertices from $V(\mathcal{G}')$. We can ignore the α-forced paths that pass through $\mathcal{G} \setminus \mathcal{G}'$ since those vertices were already attracted to a vertex with a higher measure than $pm[v]$. Hence, there is a strategy for α such that α can force the play to continue towards the higher region resulting in a higher measure than those obtained by paths that do not pass through $\mathcal{G} \setminus \mathcal{G}'$ because of $\forall_{u' \in V(\mathcal{G}) \setminus \mathcal{G}'} \mathtt{prog}(pm[v], \mathtt{pr}(u')) \leq \mathtt{prog}(pm[v'], \mathtt{pr}(u'))$. Now we have $\mathtt{min}_{\pi \in \mathcal{F}'} \eta(\pi, pm[v]) \leq \mathtt{gpm}()[pm_\alpha][u]$.

Additionally, because we know that $pm[v]$ is $\mathtt{pr}(v)$-stable by Lemma 3.6 and Definition 3.3 we have that for all finite paths $\pi \in \mathcal{P}$ containing only priorities of α's parity or that are lower or equal to $\mathtt{pr}(v)$, $\mathtt{prog}(pm[v], \mathtt{pr}(u)) \leq \eta(\pi, pm[v])$. (using the same logic as in the base case) Because $\mathcal{F}' \subseteq \mathcal{P}$, we can conclude that $\mathtt{prog}(pm[v], \mathtt{pr}(u)) \leq \mathtt{gpm}()[pm_\alpha][u]$. ■

Lemma 3.9 *Given an attractor decomposition $\mathcal{H}_\alpha(\mathcal{G}, \preceq)$, the parameters $((\mathcal{M}_\bigcirc, \mathcal{M}_\pentagon), (\leq_\bigcirc, \leq_\pentagon), \mathtt{prog})$ such that* $\mathtt{gpm}$ *yields a concrete progress measure as defined by Definition 3.5, and an α-measure function $pm : V \to \mathcal{M}_\alpha$ where $\forall_{v \in V(\mathcal{G})} pm[v] \leq \mathtt{gpm}()[pm_\alpha][v]$, $\preceq$ orders vertices by their value in pm, and for each vertex $v \in V(\mathcal{G})$, $pm[v]$ is $\mathtt{pr}(v)$-stable. For every region $(v, \mathcal{A}) \in \mathcal{H}_\alpha(\mathcal{G}, \preceq)$, every vertex $u \in \mathcal{A}$, and every predecessor w of u for which there exists an α-forced path from w to u we have:*

$$\mathtt{prog}(pm[v], \mathtt{pr}(w)) \leq \mathtt{gpm}()[pm_\alpha][w]$$

Proof We have w.l.o.g. $w \notin \mathcal{A}$ using Lemma 3.8, therefore $\mathtt{pr}(u) < \mathtt{pr}(w) \vee \mathtt{pr}(u) \equiv_2 \alpha$. Then, by Definition 3.3 and Definition 3.4 property 2, we have $pm[v] \leq \mathtt{prog}(pm[v], u)$. Then, by Definition 3.4 property 1 we have $\mathtt{prog}(pm[v], \mathtt{pr}(w)) \leq \mathtt{prog}(\mathtt{prog}(pm[v], \mathtt{pr}(u)), \mathtt{pr}(w))$. Finally, using Lemma 3.2 and Lemma 3.8 we have $\mathtt{prog}(pm[v], \mathtt{pr}(w)) \leq \mathtt{gpm}()[pm_\alpha][w]$.

Since we needed no additional assumptions on the region, we find that the lemma holds for all regions in $\mathcal{H}_\alpha(\mathcal{G}, \preceq)$. ■

Lemma 3.10 *Given an attractor decomposition $\mathcal{H}_\alpha(\mathcal{G}, \preceq)$, the parameters $((\mathcal{M}_\bigcirc, \mathcal{M}_\pentagon), (\leq_\bigcirc, \leq_\pentagon), \mathtt{prog})$ such that* $\mathtt{gpm}$ *yields a concrete progress measure as defined by Definition 3.5, and an α-measure function $pm : V \to \mathcal{M}_\alpha$ where $\forall_{v \in V(\mathcal{G})} pm[v] \leq \mathtt{gpm}()[pm_\alpha][v]$ and $\preceq$ orders vertices by their value in pm. For every region $(v, \mathcal{A}) \in \mathcal{H}_\alpha(\mathcal{G}, \preceq)$, every priority $\mathtt{pr}(v) < max \leq d$ where d is the highest priority in the game we have, and every vertex u with $\mathtt{pr}(u) \geq max$ for which there exists an α-forced path π containing only vertices with priorities which are of α's parity or that are less or equal to max we have:*

$$\mathtt{prog}(pm[v], \mathtt{pr}(u)) \leq \mathtt{gpm}()[pm_\alpha][u]$$

Proof By the premise of the lemma we have $\mathtt{pr}(u) \geq \mathtt{max}_{x \in \pi \wedge \mathtt{pr}(x) \not\equiv_2 \alpha} \mathtt{pr}(x)$. By Definition 3.4 property 3 and Lemma 3.6 we have that $\mathtt{prog}(pm[v], \mathtt{pr}(u)) \leq \eta(\pi, pm[v])$. By Lemma 3.7 we have that given the set of all α-forced paths $\mathcal{F} \subseteq \mathcal{P}$ between u and v we have $\mathtt{min}_{\pi \in \mathcal{F}} \eta(\pi, pm[v]) \leq \mathtt{gpm}()[pm_\alpha][u]$. Because $\mathcal{F} \subseteq \mathcal{P}$ we conclude that $\mathtt{prog}(pm[v], \mathtt{pr}(u) \leq \mathtt{gpm}()[pm_\alpha][u])$ ■

Summary

In this chapter we presented some requirements on the progression function `prog` of a progress measure which make it suitable for use with our PMTL algorithm. Then we introduced the Generic Progress Measures which can be parametrised to resemble existing progress measure algorithms allowing us to rely on their correctness proof to prove the correctness of our PMTL algorithm in the next chapter. We went on to give a description of the Small Progress Measures and Ordered Progress Measures, and discussed how these existing progress measure-based algorithms satisfy the properties we introduced. Finally, we presented the novel concept of a forced path and proved some lemmas which will allow us to prove the correctness of the PMTL algorithm in chapter 5.

Chapter 4

The monotonicity of ORDERED PROGRESS MEASURES

Introduction

The basic update rules for the ORDERED PROGRESS MEASURES algorithm were described by Calude et al.[2], and modified for value iteration by Fearnley et al.[9]. (see Definition 3.9) Unfortunately, these basic update rules do not preserve the ordering introduced by Fearnley et al. To resolve this issue Fearnley et al. introduce an *antagonistic update*. (see Definition 3.10) This operation can be applied to any function to make it monotonic. However, to make the ORDERED PROGRESS MEASURES algorithm practical to use it must be possible to compute its progression function efficiently. While Fearnley et al. give an explanation of how this can be implemented, we found that the resulting function is not monotonic. All existing implementations of the ORDERED PROGRESS MEASURES algorithm sidesteps the issue by adding a check before measures are updated which seems to be sufficient to make the algorithm behave correctly. However, in our PMTL algorithm we cannot employ this same trick, therefore we require the progress measure used with our algorithm to be monotonic for all inputs. In this chapter we will explore some examples that show that the function as implemented is not monotonic, analyse the situations in which this occurs, and provide an alternative definition which fixes this issue.

4.1 Implementation as described by Fearnley et al.

According to Fearnley et al.[9] the antagonistic update $\mathtt{au}(b, v)$ can be implemented as follows. First, we find some measure d:

$$\min_{\sqsubseteq}\{d = \langle d_k, d_{k-1}, \dots, d_0\rangle \mid d \sqsupseteq b \wedge d_0 = \_\}$$

Then, we compare $\mathtt{up}(b, v)$ and $\mathtt{up}(d, v)$, taking the smallest of the two (in the order $\sqsubseteq$) as the result of $\mathtt{au}(b, v)$.

However, as mentioned in the introduction to this chapter, the resulting function is not order-preserving for all input measures. In this section we will discuss the two kinds of situations in which this non-monotonic behaviour presents itself.

4.1.1 Multiple rule 2 candidates

In some cases when there are two or more candidate indices that can be used to apply rule 2 of the raw update function `ru` (Definition 3.7) to a measure. This results in a measure that is too high. Take, for example, the $\bigcirc$-measures (even) $b = \langle \_, 9, 3, 2\rangle$ and $b' = \langle \_, 5, \_, \_\rangle$. We can see that $b \sqsubset b'$ (since 9 and 5 are odd and $9 > 5$), but $\mathtt{au}(b, v) \sqsupset \mathtt{au}(b', v)$ for a vertex v with $\mathtt{pr}(v) = 9$. We will look at the calculation of $\mathtt{au}(b, v)$ and $\mathtt{au}(b', v)$ in detail to see that this is true.

For $\mathtt{au}(b, v)$ we first find d. Setting b_0 to $\_$ gives us $\langle \_, 9, 3, \_\rangle \sqsubseteq b$, the smallest step we can take to make this measure larger than or equal to b is to lower the 3 to a 1. (remember the order $\sqsubseteq$ prefers lower odd priorities over higher ones) This yields $d = \langle \_, 9, 1, \_\rangle \sqsupseteq b$. Now we calculate $\mathtt{up}(b, v)$. If we use rule 1 of Definition 3.7 we get $\langle \_, 9, 9, \_\rangle$, and if we use rule 2 we get $\langle \_, 9, 9, \_\rangle$ too. As such that is the result of $\mathtt{au}(b, v)$.

For $\mathtt{au}(b', v)$ we first find d'. Setting b'_0 to $\_$ gives us $d' = \langle \_, 5, \_, \_\rangle \sqsupseteq b'$. In this case we do not need to find a $d \sqsupseteq b'$ since b'_0 was already equal to $\_$. Therefore, we only need to calculate $\mathtt{up}(b', v)$. We cannot use rule 1 of Definition 3.7 because there are no even priorities in the measure. However, we can use rule 2 which yields $\mathtt{au}(b', v) = \langle \_, 9, \_, \_\rangle$. We can see that $\langle \_, 9, 9, \_\rangle \sqsupset \langle \_, 9, \_, \_\rangle$.

In general, the problem occurs when an α-measure b has some $b_j = p$ and some $b_i = q \leq p$ with $i < j$. When updating this measure with a vertex v with a priority $r > q$, i can be used to apply rule 2 of the raw update `ru`. Additionally, we assume p, q, and r are not of α's parity. It is possible for there to be multiple candidates b_i which fit the above description. Since Fearnley et al. specify the largest measure must be chosen we assume w.l.o.g. that i is the index for rule 2 which results in the largest measure out of all possible i candidates. We can distinguish two cases:

Case $r \leq p$:
Because $r \leq p$, j cannot be used as an index for rule 2. Therefore, i will be chosen to update the measure with rule 2. This means that in the measure c after the update, all c_k with $k > i$ will be equal to their respective b_k.

Case $r > p$:
Since choosing j as the index for rule 2 results in a smaller measure, we know that i will be used. This does conflict with the later instruction in Fearnley et al.[9] which states 'the rule is to simply update the left-most position in a measure that can be updated'. This instruction would indicate that j must be chosen instead, resulting in a smaller measure after the update. However, if i is used then all c_k with $k > i$ in the measure c after the update will be equal to their respective b_k.

We can now compare this to another measure $b' \sqsupset b$ with $b'_j < p$, $b'_k = b_k$ for all $k > j$, and there is no $k < j$ for which $b'_k < r$. Because there is no other index for which rule 2 can be applied the index j must be used. Therefore, the measure c' after the update has $c'_j = r$ and $c'_k = c_k$ for all $k > j$. Hence, $c' \sqsubseteq c$ and $\mathtt{up}(b', v) \sqsubseteq \mathtt{up}(b, v)$. Since we had $b' \sqsupset b$, we can conclude that the definition of rule 2 does not yield a monotonic update function. The example given above shows that the implementation of the antagonistic update as provided by Fearnley et al. is not able to resolve this case, unlike the non-monotonic behaviour induced by rule 1 which is resolved by their implementation.

4.1.2 Reaching the top measure prematurely

There is also another way that the update rules violate monotonicity. Take, for example, the $\bigcirc$-measures (even) $b = \langle \_, 2, \_, 2\rangle$, and $b' = \langle \_, 6, \_, \_\rangle$ for a game with 5 $\bigcirc$-priority vertices. We can see that $b \sqsubset b'$ (since 2 and 6 are even $\bigcirc$ and $2 < 6$), but $\mathtt{au}(b, v) \sqsupset \mathtt{au}(b', v)$ for a vertex v with $\mathtt{pr}(v) = 2$. We will look at the calculation of $\mathtt{au}(b, v)$ and $\mathtt{au}(b', v)$ in detail to see that this is true.

For $\mathtt{au}(b, v)$ we first find d. Setting b_0 to _ gives us $\langle \_, 2, \_, \_\rangle$, the smallest step we can take to make this measure larger than or equal to b is to set d_1 to the highest odd priority in the game, in this case 1. This yields $d = \langle \_, 2, 1, \_\rangle \sqsupseteq b$. Now we calculate $\mathtt{up}(b, v)$. If we use rule 1 of the raw update function $\mathtt{ru}$ (Definition 3.7), this results in $\langle \_, 2, 1, 2\rangle$ and if we use rule 2 this results in $\langle \_, 2, 2, \_\rangle$. Since we always choose the highest possible measure the result of the raw update $\mathtt{ru}(b, v)$ is $\langle \_, 2, 2, \_\rangle$, however, the result of the normal update $\mathtt{up}(b, v)$ is $\top$. This is because $\mathtt{value}(\langle \_, 2, 2, \_\rangle) = 6$ and there are only 5 even-priority vertices in the game. Next, we calculate $\mathtt{up}(d, v)$. If we use rule 1 this results in $\langle \_, 2, 2, \_\rangle$, since rule 2 is not applicable here this is the result of $\mathtt{ru}(d, v)$. However, again the value of this measure exceeds 5 so $\mathtt{au}(d, v) = \mathtt{up}(d, v) = \top$.

For $\mathtt{au}(b', v)$ we first find d'. Setting b'_0 to _ gives us $d' = \langle \_, 6, \_, \_\rangle = b'$. Therefore, we have $\mathtt{au}(b', v) = \mathtt{up}(b', v)$. Now we calculate $\mathtt{up}(b', v)$. We use rule 1, since rule 2 is not applicable, this results in $\mathtt{ru}(b', v) = \langle \_, 6, \_, 2\rangle$. Since $\mathtt{value}(\langle \_, 6, \_, 2\rangle) = 5$, we have $\mathtt{up}(b', v) = \mathtt{ru}(b', v)$. We can now see that $\mathtt{au}(b, v) = \top \sqsupset \mathtt{au}(b', v) = \langle \_, 6, \_, 2\rangle$, whereas $b \sqsubset b'$. We can therefore conclude that the implementation of the antagonistic update as described by Fearnley et al. does not yield a monotonic function.

4.2 Unanswered questions

In section 4.1 we showed that the implementation of the antagonistic update yields a non-monotone function. However, despite this, the available implementations of the ORDERED PROGRESS MEASURES have provided correct results for all games that have been fed to them[6, 8]. These implementations have an additional check that will only allow the measure of a vertex to be updated if the new measure is higher in the ordering $\sqsubseteq$. By removing this check, which should be unnecessary if the update function is monotonic, one can verify that the implemented update function is non-monotonic. To prove correctness of these implementations we would need to prove that for every measure b that results from the implemented antagonistic update, if it is higher than the measure c that the theoretical antagonistic update would have produced, two properties hold:

- The measure b must not be higher than the measure that the vertex gets after the algorithm with the theoretical antagonistic update has finished running.
- Since $\mathtt{au}(b, v)$ may be lower than $\mathtt{au}(c, v)$ for some vertex v, using the measure b instead of the measure c must not cause the final measure (after running

the algorithm to completion) of v to be lower than it would have been were the algorithm to use the theoretical antagonistic update.

Proving these properties is outside the scope of this research since the PMTL algorithm described in chapter 5 requires the progression function to be monotonic. The extra check does not make the implemented antagonistic update function monotonic.

4.3 The solution

As described in section 4.1 we have identified two different mechanisms that cause the update function to be non-monotonic: by choosing the wrong index for rule 2 of the raw update function `ru` (Definition 3.7), and by reaching $\top$ faster than a lower measure would because of a higher `value`. The latter is easily resolved by modifying the `up` function (Definition 3.9) to no longer use the `value` for determining when a measure should become $\top$, instead an α-measure b becomes top when b_k (i.e. the left-most element of b) is of α's parity and the `value` is still sufficiently large. It is clear that this new rule allows strictly fewer measures to become $\top$, additionally, if a measure reached $\top$ with the old rule then the game has an α-cycle which causes the measure to continue to rise such that it will eventually also set b_k to an α-priority. As such this new rule will not change the results of the Ordered Progress Measures algorithm. Another way to implement this rule is to increase the size of the measure by increasing k by one. This means that when b_k is set to a priority that is of α's parity, then there must also be an α-priority chain of vertices that is at least $2^{\lfloor \log_2 d \rfloor + 1} > d$ in length.

To solve the choice of the wrong index for rule 2 we must replace two parts of the raw update function `ru`. Firstly, currently the choice of which rule to apply and which index to use for that rule is determined by seeing which one results in the largest possible measure. We will change this to instead choose the highest index for which a rule can be applied. It is clear that for rule 1 this will still select the same index since applying rule 1 with a higher index will always yield a higher measure than selecting any lower index.

For rule 2 there are cases where this results in a higher index (and therefore lower measure) being chosen. However, switching to the highest index rule does not, on its own, change anything for the example given in subsection 4.1.1. For this reason we also modify rule 2. When the encountered vertex has a priority p that is of α's priority we only want to apply rule 2 on an index j for which $b_j < p$ since, otherwise we would be making the measure smaller if $b_j \neq p$ or $b_i \neq \_$ for any $i < j$. Therefore, we only change the rule for cases in which p is of $\bar{\alpha}$'s priority. In those cases we want to allow indices j for which $b_j \leq d$ instead of being strictly less than d. This means that in our example $j = 2$ could and would be chosen resulting in $\langle \_9, \_, \_ \rangle$ instead of $\langle \_, 9, 9, \_ \rangle$.

By making these two changes we fix both mechanisms that we identified could cause the update function to be non-monotonic. While we did not prove this is the case, the modification of the Ordered Progress Measures described by Dell'Erba and Schewe[3] likely also bypasses the mechanisms we described.

Summary

In this chapter we discussed the difficulty of implementing the ORDERED PROGRESS MEASURES' antagonistic update. We showed two examples that represent different ways in which the update function's implementation as described by Fearnley et al. is non-monotonic. We then discussed why this deficiency has not caused any problems in practice. Finally, we described two changes to the update function which fix the two mechanism by which the non-monotonicity was introduced thereby making the algorithm suitable to be used with the PMTL algorithm described in the next chapter.

Chapter 5

The algorithm

Introduction

This chapter describes three variants of the PMTL algorithm starting with its most basic form. The PMTL algorithm accelerates the value iteration of progress measures to more quickly solve a parity game. It does this by lifting all measures of vertices in the attractor set using the measure of the target vertex of the attractor set. By doing this a high-valued measure that would previously need multiple iterations to influence a vertex's measure can be used immediately. We also describe a variant which uses an alternative strategy for lifting the measures of vertices with a higher priority than the top vertex of the region. Finally, we incorporate the techniques from the Tangle Learning algorithm into a third variant. This includes the use of the tangle attractor `attrT` and the `extractTangles` function to find tangles and quickly remove dominions from the game. This allows the game to be solved even faster. Additionally, we discuss the reasoning behind the different operations the algorithm performs. Finally, we prove the correctness of the three variants.

5.1 The base algorithm

We will first examine the simplest version of the algorithm which only uses regular attractors, leaving out the discovery of tangles for now. This will make it easier to prove correctness and the lemmas we prove along the way will be reused for the tangle learning version. Additionally, this base version only attracts vertices of a lower priority since it is much easier to prove correctness in those cases.

5.1.1 Description of the algorithm

We can solve a parity game by iteratively lifting a set of progress measures for each player and removing the dominions from the game when the progress measure has reached a fixpoint. This iteration is implemented in the `pmtl` function in Algorithm 5.1. Note that in the interest of brevity the game $\mathcal{G}$ parameter is omitted for all functions presented.

Algorithm 5.1: pmtl

```
fn pmtl():
    $pm_{◯} \leftarrow \{u \leftarrow \bot \mid u \in V(\mathcal{G})\}$
    $pm_{⬠} \leftarrow \{u \leftarrow \bot \mid u \in V(\mathcal{G})\}$
    $c_{◯} \leftarrow$ true, $c_{⬠} \leftarrow$ true
    while $c_{◯} \vee c_{⬠}$ :
        if $c_{◯}$ :
            $c_{◯} \leftarrow$ update($pm_{◯}$, ◯)
            if $\neg\, c_{◯}$ : solve($pm_{◯}$, ⬠)
        if $c_{⬠}$ :
            $c_{⬠} \leftarrow$ update($pm_{⬠}$, ⬠)
            if $\neg\, c_{⬠}$ : solve($pm_{⬠}$, ◯)
```

The `pmtl` function in Algorithm 5.1 uses `update` to iteratively lift two sets of progress measures and `solve` to remove the vertices won by $\bar{\alpha}$ from $\mathcal{G}$ after pm_α has reached a fixpoint, i.e. after c_α becomes **false**.

Algorithm 5.2: update

```
fn update($pm$, $\alpha$):
    $target \leftarrow V(\mathcal{G}) \mapsto \_$
    $\mathcal{E} \leftarrow \mathcal{G}$
    for $(top, \mathcal{A}) \in \mathcal{H}_\alpha(\mathcal{G}, \sqsubseteq)$ :
        for $u \in V(\mathcal{G})$ :
            if $target[u] = \_ \wedge$ attracts$_\alpha(u, \mathcal{A}, \mathcal{E})$ :
                $target[u] \leftarrow top$
        $\mathcal{E} \leftarrow \mathcal{E} \setminus \mathcal{A}$
    $updated \leftarrow$ false
    for $u \in V(\mathcal{G})$ :
        if progressVertex($pm$, $target[u]$, $u$) : $updated \leftarrow$ true
    return $updated$
```

The `update` function is shown in Algorithm 5.2. It starts by computing the attractor decomposition $\mathcal{H}_\alpha(\mathcal{G}, \sqsubseteq)$ where $\sqsubseteq$ orders vertices by their measure and then by their priority. For every region $(top, \mathcal{A})$ we find all the vertices which are attracted to the attractor set $\mathcal{A}$. Note that by attracted we mean ‘attracted in a single step’ here, essentially we only check whether the vertices $E^{-1}(\mathcal{A})$ (direct predecessors of the vertices $\mathcal{A}$) are attracted. For each attracted vertex u which has not already been assigned a target vertex in a higher region, we assign *top*. This is stored in the $target : V \to (V \cup \{\_\})$ variable which maps every vertex to the top vertex of a region.

Finally, we use `progressVertex` on every vertex to lift it towards its target vertex. If the measure of any vertex was lifted in this way we return true.

The actual implementation of `update` differs quite substantially from this description. Instead of iterating over the attractor decomposition twice and then over every vertex in the game, we do all of this during a single iteration over the attractor decomposition. By separately tracking vertices which may be attracted (which includes the top vertices of higher regions) and the escapes we can determine which region attracts which top vertex.

Finally, once at least one vertex's measure has been lifted (and after we have finished all steps for that vertex's region) we return **true** without iterating further. This should theoretically allow a higher measure to be used again earlier. Since the `update` function is called until it no longer returns **true** this gives the same results as completing the entire decomposition each time, but it has different performance characteristics. In subsection 7.3.4 we will investigate what the impact of this decision is.

There is another way to conceptualise what this algorithm does which makes it easier to prove its correctness. We split the for loop into two loops. In the first for loop the target of every vertex will be assigned to the top of the region which contains it, except for the top vertices themselves. In other words, for all $u \in \mathcal{A} \setminus \{top\}$ we set $target[u] = top$.

After this we need to set the target of the *top* vertices of each region. It is possible that these vertices are attracted by their own region, but they may also be attracted by lower regions, i.e. a region whose *top* vertex has a lower measure. Additionally, since the attractor decomposition uses a maximum priority for its calculation of the attractor there may be some vertex which may be attracted to higher region because it lies only one edge beyond the original region.

To set the target vertex of the top vertices and to update the target of the aforementioned vertices to a higher region we iterate over every region $(top, \mathcal{A}) \in \mathcal{H}_\alpha(\mathcal{G}, \sqsubseteq)$ once more. We keep track of a variable $\mathcal{E}$ tracking the remaining vertices to which the opponent may escape and use this to determine whether any vertices around the periphery of the region's attractor set $\mathcal{A}$ can be attracted. If the vertex can be attracted, its target vertex is set to the current *top* vertex if it is higher in the ordering $\sqsubseteq$.

Only after the second for loop do we use the stored targets to attempt to lift the measures of the vertices with `progressVertex`.

Algorithm 5.3: progressVertex

```
fn progressVertex($pm, v, u$):
    $newMeasure \leftarrow$ prog($pm[v]$, pr($u$))
    if $newMeasure > pm[v]$ :
        $pm[v] \leftarrow newMeasure$
        return true
    else: return false
```

The `progressVertex` function shown in Algorithm 5.3 attempts to lift a vertex v using the progress measure's `prog` function based on the measure of the vertex u. It only does so if the new measure is strictly larger than the original measure.

An important note is that care must be taken to not override the 'old' measures before they are passed to `prog` because this might cause measures to be lifted higher than they should have been. To prevent this we can keep track of two separate sets of measures and switch them at the end of the `update` function. Alternatively, we can keep track of whether a measure was updated before we call `prog`, and if it was, we simply skip progressing that vertex, but we still return **true**. That vertex will then end up being updated in a later iteration.

5.1.2 Proving correctness

To prove that the algorithm described in subsection 5.1.1 is able to correctly solve all parity games we will proof two lemmas which in turn prove that the algorithm yields the same measures as the original progress measure-based algorithm would have in its value iteration framework. We use the GENERIC PROGRESS MEASURES (GPM) algorithm described in chapter 2 as an analogue for the original progress-measure based algorithm. These lemmas are:

Lemma 5.1 *Given an α-measure function $pm : V \rightarrow \mathcal{M}_\alpha$ where for each vertex $v \in V(\mathcal{G})$, $pm[v]$ is $\mathtt{pr}(v)$-stable, the function* `update` *will lift these measures to be no higher than those acquired by running the original progress measure algorithm* `gpm` *when both are parametrised with the same set of parameters $((\mathcal{M}_{\bigcirc}, \mathcal{M}_{\pentagon}), (\leq_{\bigcirc}, \leq_{\pentagon}), \mathtt{prog})$ such that* `gpm` *yields a concrete progress measure as defined by Definition 3.5.*
PROOF The `update` function computes the attractor decomposition $\mathcal{H}_\alpha$ of $\mathcal{G}$ with an ordering $\leq$ based on the current measure stored in pm. For every region $(top, \mathcal{A}) \in \mathcal{H}_\alpha(\mathcal{G}, \leq)$ it updates the measures of all $u \in \mathcal{A}$ in pm to be $\mathtt{prog}(pm[top], \mathtt{pr}(u))$, or it keeps them at a higher measure. We have $u \in \mathtt{attr}_\alpha^{\mathcal{G}'}(\{top\}, \{p \in \mathbb{N} \mid p \equiv_2 \alpha \vee p \leq \mathtt{pr}(top)\}) \setminus \{top\}$ where $\mathcal{G}'$ is the set of vertices which were not in a higher region. By Lemma 3.4 we have an α-forced path from u to top containing only vertices with priorities which are of α's parity or that are less or equal to $\mathtt{pr}(top)$. Note that since the set of found tangles $\mathcal{T}_\alpha$ is empty $\mathtt{attrT} = \mathtt{attr}$ which means we can use lemmas that use `attr` even though the definition of $\mathcal{H}$ uses `attrT`. By Lemma 3.8 we have $\mathtt{prog}(pm[top], \mathtt{pr}(u)) \leq \mathtt{gpm}()[pm_\alpha][u]$.

Then for every vertex in the $\mathcal{A}$ set the measure of its successor w becomes $\mathtt{prog}(pm[top], \mathtt{pr}(w))$ if it is attracted to u in a single step (e.g. $w \in \mathtt{attr}_\alpha^{E^{-1}(\mathcal{A})}(\{u\}, \mathcal{E}, \mathbb{N})$ again using $\mathtt{attrT} = \mathtt{attr}$), or it remains higher. By Lemma 3.4 we have an α-forced path from w to u. Finally, by Lemma 3.9 we have $\mathtt{prog}(pm[top], \mathtt{pr}(u)) \leq \mathtt{gpm}()[pm_\alpha][w]$. Since all the updated measures are no higher than the measures obtained from `gpm` the lemma holds. ■

Lemma 5.2 *Given an α-measure function $pm : V \rightarrow \mathcal{M}_\alpha$, the function* `update` *will lift at least one measure, if running the original progress measure algorithm* `gpm` *would have lifted a measure when both are parametrised with the same set of parameters $((\mathcal{M}_{\bigcirc}, \mathcal{M}_{\pentagon}), (\leq_{\bigcirc}, \leq_{\pentagon}), \mathtt{prog})$ such that* `gpm` *yields a concrete progress measure as defined by Definition 3.5.*
PROOF We will prove this by structural induction on the construction of a graph:

Induction Hypothesis: the function `update` will lift at least one measure in pm. if running `gpm` would have lifted a measure.

Base case $V(\mathcal{G}) = E(\mathcal{G}) = \emptyset$:
Since there are no vertices to be lifted by `gpm` or `update` the **IH** holds.

Step case adding a vertex $V(\mathcal{G}) = V(\mathcal{G}') \cup \{u\}$, $E(\mathcal{G}) = E(\mathcal{G}')$:
By the **IH** we have that for all vertices in $\mathcal{G}'$ if `gpm` had lifted the measure then `update` will also lift a measure. Since the newly added vertex u is not connected to

any other vertex it does not affect the measure of any other vertex, and it can never be lifted itself. Hence, the **IH** holds for $\mathcal{G}$.

Step case adding an edge $V(\mathcal{G}) = V(\mathcal{G}'),\ E(\mathcal{G}) = E(\mathcal{G}') \cup \{(u, v)\}$:
By the **IH** we have that in $\mathcal{G}'$, if `gpm` had lifted the measure then `update` will also lift a measure. To prove that this is still the case in $\mathcal{G}$ it suffices to prove that if `gpm` lifted the measure of u using the edge (u, v) then `update` must also lift a measure. This suffices because any difference in the final measures returned by `gpm` after adding the edge must come about by lifting along that edge. We will prove this by contradiction. Suppose that u is lifted by `gpm` along the edge (u, v), but `update` does not lift any measures. We know that $\texttt{prog}(pm[v], \texttt{pr}(u)) > pm[u]$. The `update` function computes the attractor decomposition $\mathcal{H}_\alpha$ of $\mathcal{G}$ with an ordering $\leq$ based on the current measure stored in pm. We know there must exist a region $(top, \mathcal{A}) \in \mathcal{H}_\alpha(\mathcal{G}, \leq)$ with $u \in \mathcal{A}$. Additionally, we know that $\texttt{prog}(pm[top], \texttt{pr}(u)) \leq pm[u]$ and $pm[top] < pm[v]$. Furthermore, we know that there must exist another higher value region $(w, \mathcal{A}')$ containing v because it has a higher measure than top. This region must have a top vertex w with $pm[v] \leq pm[w]$. There are two cases to consider:

Case $u \in V_\alpha(\mathcal{G})$:
Because $u \in V_\alpha(\mathcal{G})$ we have that $u \in \texttt{attr}_\alpha^{E^{-1}(\mathcal{A})}(\mathcal{A}, \mathcal{E}, \mathbb{N})$. However, that means that `update` would have lifted the measure of u to $\texttt{prog}(pm[w], \texttt{pr}(u))$ which contradicts the premise that no measures would be lifted.

Case $u \in V_{\bar{\alpha}}(\mathcal{G})$:
By the logic of the other case we know that it must be the case that $u \notin \texttt{attr}_\alpha^{E^{-1}(\mathcal{A})}(\mathcal{A}, \mathcal{E}, \mathbb{N})$. Since u is not attracted there must exist a vertex $x \in \mathcal{E} \backslash \mathcal{A}$. However, because `gpm` used the edge (u, v) to lift u, we know that v must have the lowest measure among all the successors of u. Since all vertices with a higher measure than v must either be in $\mathcal{A}$ or in some higher region that means they cannot be in $\mathcal{E} \backslash \mathcal{A}$. Therefore, we know that u must be attracted to $\mathcal{A}$ which by the same logic as the other case contradicts the premise that no measures would be lifted.

We have now proven by induction that for any parity game graph the lemma holds.■

Finally, we can use Lemma 5.1 which proved that the measures computed by our algorithm are no higher than the measure computed by GPM, and Lemma 5.2 which proved that if there are any measures that could be lifted by GPM then our algorithm will lift at least one, to prove that our algorithm correctly solves a parity game.

Theorem 5.1 *Given a game $\mathcal{G}$. When* `pmtl` *has finished executing it ensures that every vertex is marked as won by ○ or ⬠ and if the vertex is controlled by the winning player it has it with a winning strategy.*
Proof For all vertices we have by Lemma 3.1, Lemma 5.1, and Lemma 5.2 that the measures computed by `pmtl` and `gpm` must be the same. Therefore, this theorem holds by Theorem 3.1. ■

5.2 Going up

The base algorithm only uses attractors which attract vertices with a priority that is less or equal to the priority of the *top* vertex. (or the attracted vertex has a priority that is of the current player's parity) It is relatively straightforward to prove that this cannot result in measures that are too high since we can rely on Definition 3.4 property 2. However, even for the base algorithm we needed an exception since there would otherwise be cases where we would not lift a 'liftable' vertex. This exception entailed attracting one more step around the boundary of the attractor. The proof for this mainly relies on the fact that the newly attracted vertex must have a larger priority than the vertex in the attractor that it was attracted to. Attracting more vertices to a region is beneficial since these vertices will be attracted to a higher region than they would otherwise be, therefore, their measure will increase faster.

For the 'going up' variant of the algorithm we attempt to attract more vertices by using Definition 3.4 property 3. Using this property we find that if the vertex whose measure we are lifting has a higher priority than all the vertices between it and the target vertex then we are allowed to skip progressing the measure vertex-by-vertex.

5.2.1 Description of the algorithm

Algorithm 5.4: update

```
fn update($pm, \alpha$):
    $target \leftarrow V(\mathcal{G}) \mapsto \_$
    $\mathcal{E} \leftarrow \mathcal{G}$
    for $(top, \mathcal{A}) \in \mathcal{H}_\alpha(\mathcal{G}, \sqsubseteq)$ :
        for $u \in V(\mathcal{G})$ :
            if $target[u] = \_ \wedge \texttt{attracts}_\alpha(u, \mathcal{A}, \mathcal{E})$ :
                $target[u] \leftarrow top$
        for every priority $max \in (\texttt{pr}(top); d]$ :
            for $u \in \texttt{attrT}^{\mathcal{G}}_\alpha(\mathcal{A}, \mathcal{E}, \{p \in \mathbb{N} \mid p \equiv_2 \alpha \vee p \leq max\})$ :
                if $target[u] = \_ \wedge \texttt{pr}(u) \geq max$ :
                    $target[u] \leftarrow top$
        $\mathcal{E} \leftarrow \mathcal{E} \setminus \mathcal{A}$
    $updated \leftarrow$ false
    for $u \in V(\mathcal{G})$ :
        if progressVertex($pm$, $target[u]$, $u$) : $updated \leftarrow$ true
    return $updated$
```

We reuse most of the base algorithm only modifying the `update` function in Algorithm 5.4. In addition to attracting a single layer of vertices around the attractor set we start iterating over all priorities higher than $\texttt{pr}(top)$. For every priority we attempt to expand our original attractor set by attracting vertices with a priority $\leq max$. It is important to do this in steps where we only call `progressVertex` on the attracted vertices with *max* priority and the attracted vertices with a priority of α's parity. This is required because if we were to lift vertices with a lower priority that only got attracted through some higher $\bar{\alpha}$-priority vertex this would no longer be analogous to lifting along that path using the successors.

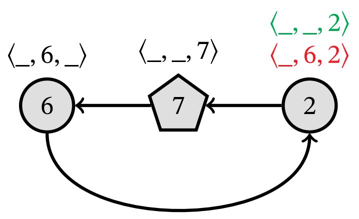

$\langle \_, \_, 2\rangle$
$\langle \_, 6, \_\rangle$
$\langle \_, \_, 7\rangle$
$\langle \_, 6, 2\rangle$
6
7
2

Figure 5.1: A simple parity game showing why we need to be careful when attracting vertices with a higher priority.

To illustrate this Figure 5.1 shows a simple parity game where the vertex with priority 2 gets attracted to the vertex with priority 6 through the vertex with priority 7. The progress measures above each vertex are those that are gotten by using the Ordered Progress Measures' update rule twice for every vertex. The figure shows that when we naively lift all vertices attracted to 6 that 2 gets a higher value (shown in red) than would otherwise be possible (shown in green), in fact, with more iterations this would allow the value of 6 and 2 to reach $\top$ which should not be possible since the vertices are clearly won by the odd ⬠ player in every possible infinite play.

In the actual implementation it is not needed to both attract a single layer of vertices around the attractor set and 'attract up' since the latter will also attract all the vertices attracted by the former.

5.2.2 Proving correctness

Since most of the base algorithm is left the same, proving the correctness of this modification is relatively straightforward.

We will again need to prove two lemmas to prove that the algorithm yields the same measures as the original progress measure algorithm:

Lemma 5.3 *Given an α-measure-function $pm : V \to \mathcal{M}_\alpha$ where for each vertex $v \in V(\mathcal{G})$, $pm[v]$ is* `pr`(v)*-stable, the function* `update` *will lift these measures to be no higher than those acquired by running the original progress measure algorithm* `gpm` *when both are parametrised with the same set of parameters $((\mathcal{M}_{\bigcirc}, \mathcal{M}_{⬠}), (\leq_{\bigcirc}, \leq_{⬠}$), `prog`$)$ such that* `gpm` *yields a concrete progress measure as defined by Definition 3.5.*
Proof Both versions of the `update` function (Algorithm 5.2 Algorithm 5.4) compute the attractor decomposition $\mathcal{H}_\alpha$ of $\mathcal{G}$ with an ordering $\leq$ based on the current measure stored in pm. For every region $(top, \mathcal{A}) \in \mathcal{H}_\alpha(\mathcal{G}, \leq)$ it sets the targets of every vertex $u \in \mathcal{A} \setminus \{top\}$ to top if they had not been set to a higher region's top. Furthermore, both version of `update` set the target of top to top if the region is closed, i.e. $\texttt{attracts}_\alpha(top, \mathcal{A}, \mathcal{E})$. If no more changes are made to the targets of the vertices the two versions are equivalent. Therefore, we can use the same reasoning as Lemma 5.1 (which uses Lemma 3.4 and Lemma 3.8) to find that $\texttt{prog}(pm[top], \texttt{pr}(u)) \leq \texttt{gpm}()[pm_\alpha][u]$ for every $u \in V(\mathcal{G})$ with $target[u] = top \neq \_$. It remains to be proven that this holds after the targets have been modified by the 'going up' part of Algorithm 5.4.

For every priority $\mathtt{pr}(top) < max \leq d$ where d is the highest priority in the game, and every vertex $u \in \mathtt{attr}^{\mathcal{G}}_{\alpha}(\mathcal{A}, \mathcal{E}, \{p \in \mathbb{N} \mid p \equiv_2 \alpha \vee p \leq max\})$ with $\mathtt{pr}(u) \geq max$ we have an α-forced path from u to top by Lemma 3.4. As with Lemma 5.1 we use $\mathtt{attrT} = \mathtt{attr}$ because the set of found tangles $\mathcal{T}_{\alpha}$ is empty. This path only contains vertices with priorities which are of α's parity or that are less or equal to max. By Lemma 3.10 we have $\mathtt{prog}(pm[top], \mathtt{pr}(u)) \leq \mathtt{gpm}()[pm_{\alpha}][u]$. Since all the updated measures are no higher than the measures obtained from gpm the lemma holds. ∎

Lemma 5.4 *Given an α-measure function $pm : V \to \mathcal{M}_{\alpha}$, the function* `update` *will lift at least one measure, if running the original progress measure algorithm* `gpm` *would have lifted a measure when both are parametrised with the same set of parameters $((\mathcal{M}_{○}, \mathcal{M}_{⬠}), (\leq_{○}, \leq_{⬠}), \mathtt{prog})$ such that* `gpm` *yields a concrete progress measure as defined by Definition 3.5.*

Proof Since both versions of the `update` function (Algorithm 5.2 and Algorithm 5.4) use the same attractor decomposition the regions that are used are the same. By Lemma 5.2 we have that the base version (Algorithm 5.2) always lifts at least one vertex if there is one that can be lifted. As mentioned in the proof for Lemma 5.3 the computed targets for the vertices in each region are the same. The only difference arises when comparing which vertices get assigned a higher target than their region's top vertex.

However, the 'attract up' version (Algorithm 5.4) of the algorithm only ever assigns a higher target to more vertices than the base version did. We know this is true since the added lines 10–11 can only cause a higher target to be set and never reduce an already set target. Therefore, this lemma holds by Lemma 5.2. ∎

Finally, we can use Lemma 5.3 which proved that the measures computed by our algorithm are no higher than the measure computed by GPM, and Lemma 5.4 which proved that if there are any measures that could be lifted by GPM then our algorithm will lift at least one, to prove that our algorithm correctly solves a parity game.

Theorem 5.2 *Given a game $\mathcal{G}$. When* `pmtl` *has finished executing it ensures that every vertex is marked as won by ○ or ⬠ and if the vertex is controlled by the winning player it has it with a winning strategy.*

Proof For all vertices we have by Lemma 3.1, Lemma 5.3, and Lemma 5.4 that the measures computed by `pmtl` and `gpm` must be the same. Therefore, this theorem holds by Theorem 3.1. ∎

5.3 Adding tangles

Using attractors instead of only successors for lifting progress measures accelerates the computation because it allows a higher measure to propagate further without lifting all vertices in between the higher and the lower area over the course of multiple iterations.

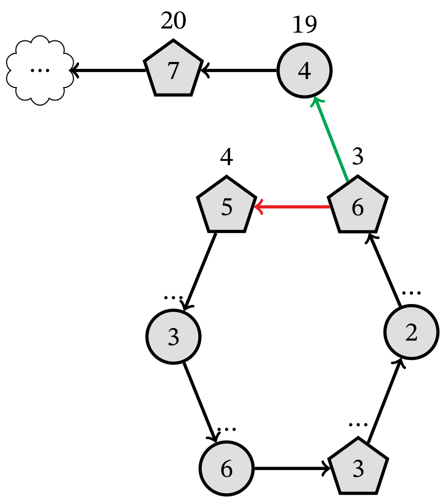


Figure 5.2: A game which is able to delay a progress measure-based algorithm

However, applying the techniques from TANGLE LEARNING (the tangle attractor `attrT` and `extractTangles`) to this algorithm can yield a more significant acceleration. Progress measure-based algorithms can be significantly slowed by the presence of tangles.

An example of this is shown in Figure 5.2. There we see the odd ⬠ vertex with priority 6 which tries to avoid high ○-measures (even), here we have used arbitrary integers to represent these measures since the specifics do not matter. We can see that if odd ⬠ chooses the red strategy it enters a cycle which will be won by even ○ since the highest priority in the cycle is 6.

However, the regular value iteration progress measure algorithms will not change their strategy from red to green until the ○-measure of the vertex with priority 5 becomes higher than the ○-measure of the vertex with priority 4. We know this must eventually happen (unless the ○-measure of the vertex with priority 4 becomes $\top$) since the ○-measures in the cycle will tend to $\top$ because the cycle is won by ○. Unfortunately, it may take many iterations of the measures being updated one-by-one across the cycle before the green edge is used.

Using attractors like we do in PMTL speeds up this process since the vertices in the cycle may attract themselves (once the top two vertices with measures have been removed from consideration in a higher region of the attractor decomposition) which allows their measures to be lifted immediately across the path instead of step-by-step. However, it may still take many iterations before the ○-measure of the vertex with priority 5 exceeds the ○-measure of the vertex with priority 4.

By applying the tangle attractor `attrT` and extracting tangles with `extractTangles`, we can detect that the aforementioned cycle is a 6-tangle. As such the ⬠-vertex with priority 7 can attract not only the vertex with priority 4 but also every vertex in the cycle, thereby in a single iteration increasing the ○ measure of the *odd*-vertex with priority 6 to at least 20 immediately. As such we can see how the techniques from TANGLE LEARNING can further accelerate the lifting of progress measures.

There is, however, another way in which Tangle Learning can accelerate our algorithm. When we find tangles, we may also find inescapable tangles which are dominions which we may immediately mark as solved. In fact, as we will discuss in more detail in chapter 7 most games are entirely solved in this manner. This may prompt the question: How is this any different from just doing Tangle Learning?

The crucial difference is that whereas Tangle Learning orders its attractor decomposition based on the priority of every vertex (although there are variants with other orders), PMTL orders its attractor decomposition based on the measures of every vertex. One way to interpret Tangle Learning is that it keeps trying different attractor decompositions (because it keeps learning more tangles) until it tries one that allows it to detect a dominion which allows it to solve (part of) the game. In PMTL we are using the measures to guide the order of the decomposition which may help us find these dominions faster.

If we take, for example, a game where we have found the inflationary fixpoint of the ◯-measures (even) then we know that every vertex that has an ◯-measure that is $\top$ that it is won by even. If we now compute an attractor decomposition ordered by α-measure, we will find only closed regions since if a vertex that is won by ◯ (and hence has the highest ◯-measure) attracts a vertex, then that vertex must also have been won. (Lemma 2.4) All the remaining regions may not contain ◯-vertices which can escape these regions, these regions are won by odd. We could say that this ordering of the attractor decomposition is *ideal* since it allows us to find every dominion.

Since every iteration of PMTL will lift measures closer to this *ideal* fixpoint state we also approach the *ideal* attractor decomposition for finding dominions. Of course, this just an intuition and not a proof, and in practice, we do find that there are some games that are solved by reaching a fixpoint without a dominion being found. We are essentially doing 'one-sided' Tangle Learning because we only use ◯-attractors while updating ◯-measures and ⬠-attractors while updating ⬠-measures. Therefore, we are not guaranteed to find every tangle and dominion. In chapter 7 we will discuss whether there is evidence that this theory holds water.

5.3.1 Description of the algorithm

Algorithm 5.5: update (with tangle learning added)

```
fn update($pm, \alpha$):
    $target \leftarrow V(\mathcal{G}) \mapsto \_$
    $\mathcal{E} \leftarrow \mathcal{G}$
    for $(top, \mathcal{A}) \in \mathcal{H}_\alpha(\mathcal{G}, \sqsubseteq)$ :
        for $u \in \mathcal{A}$ :
            if $target[u] = \_ \wedge \texttt{attracts}_\alpha(u, \mathcal{A}, \mathcal{E})$ :
                $target[u] \leftarrow top$
        for every priority $max \in (\texttt{pr}(top); d]$ :
            for $u \in \texttt{attrT}^{\mathcal{G}}_\alpha(\mathcal{A}, \mathcal{E}, \{p \in \mathbb{N} \mid p \equiv_2 \alpha \vee p \leq max\})$ :
                if $target[u] = \_ \wedge \texttt{pr}(u) \geq max$ :
                    $target[u] \leftarrow top$
        if $\texttt{attracts}_\alpha(top, \mathcal{A}, \mathcal{E}) \wedge \texttt{pr}(top) \equiv_2 \alpha$ :
            $\texttt{extractTangles}(\mathcal{A}, \sigma_\alpha)$
        $\mathcal{E} \leftarrow \mathcal{E} \setminus \mathcal{A}$
    $updated \leftarrow$ false
    for $u \in V(\mathcal{G})$ :
        if $\texttt{progressVertex}(pm, target[u], u)$ : $updated \leftarrow$ true

    if $dominions \neq \emptyset$ :
        $\mathcal{A} \leftarrow \texttt{attrT}^{\mathcal{G}}_\alpha(\mathcal{D}_\alpha, \mathbb{N})$
        for $v \in \mathcal{A}$ :
            mark $v$ as won by $\alpha$ with strategy $\sigma_\alpha[v]$
        $\mathcal{G} \leftarrow \mathcal{G} \setminus \mathcal{A}$
    return $updated$
```

There are two differences between the version of `update` presented in Algorithm 5.4 and the version with Tangle Learning presented in Algorithm 5.5.

Firstly, whenever we find a closed region (i.e. $\texttt{attracts}_\alpha(top, \mathcal{A}, \mathcal{E})$) with a priority that is of α's parity we attempt to extract tangles from it with the `extractTangles` function. This function populates the set $\mathcal{T}_\alpha$ of tangles of α's parity and when it finds dominions (i.e. inescapable tangles) it stores them in $\mathcal{D}_\alpha$.

Secondly, after we finish our attractor decomposition we attempt to extend any dominions found by calls to `extractTangles` by computing the attractor. After this all the vertices in these dominions are marked as won by α and removed from the game. The strategy $\sigma_\alpha[v]$ was computed by the attractor which found the closed region in the first place. In the interest of brevity the specifics of how these strategies are stored is left out here. In practice, we find that many games are solved entirely through the removal of dominions before the measures reach a fixpoint.

5.3.2 Proving correctness

There are essentially three changes to consider when we compare the new variant to the 'going up' variant described in section 5.2. Firstly, we use tangle attractors instead of regular attractors. Proving that this still yields a correct algorithm is straightforward since we can simply apply Lemma 3.5 instead of Lemma 3.4.

Secondly, we extract tangles by finding closed regions. We need to prove that we only add a tangle to our list of tangles if it actually exists. Finally, we remove dominions from the game. We need to prove that removing the dominions from the game marks vertices with the correct winner and strategy. We now have three lemmas that we will prove:

Lemma 5.5 *All tangles found through the invocation of* `extractTangles` *by* `update` *are valid p-tangles where $p = \texttt{pr}(top) \equiv_2 \alpha$.*
PROOF The function `extractTangles` restricts the attractor set to the strategy σ that was computed by the `attrT` function and then computes the bottom SCCs of the resulting region. It is only called when the attractor set is a closed region and the *top* vertex has a priority that is of α's parity. This means that the attractor set when restricted by the strategy σ is inescapable given the current subgame. Furthermore, because of the condition on the `attrT` function we know that all vertices in the attractor set have a priority that is either lower or equal to p, or higher than p and of α's parity. As such when `extractTangles` finds strongly connected components of the game graph where the highest priority is greater or equal to p with a strategy σ, that forces any play restricted to the region to end up in a winning cycle for α, i.e. a p-tangle. ■

Lemma 5.6 *Given an α-measure function $pm : V \rightarrow \mathcal{M}_\alpha$, the function* `update` *will lift these measures to be no higher than those acquired by running the original progress measure algorithm* `gpm` *when both are parametrised with the same set of parameters $((\mathcal{M}_{\bigcirc}, \mathcal{M}_{\pentagon}), (\leq_{\bigcirc}, \leq_{\pentagon}), \texttt{prog})$ such that* `gpm` *yields a concrete progress measure as defined by Definition 3.5.*
PROOF By Lemma 5.5 we have that the tangles found by `extractTangles` are valid tangles, therefore, the `attrT` function produces correct tangle attractor sets.

The only difference with regard to the measures returned by the previous version of `update` (Algorithm 5.4) and the version which uses tangles (Algorithm 5.5) is that the tangle attractors sets may be larger than the attractor sets computed by the previous version. Proving this lemma is equivalent to proving Lemma 5.3 except we replace Lemma 3.4 with Lemma 3.5. ■

Lemma 5.7 *Given an α-measure function $pm : V \rightarrow \mathcal{M}_\alpha$, the function* `update` *will lift at least one measure, if running the original progress measure algorithm* `gpm` *would have lifted a measure when both are parametrised with the same set of parameters $((\mathcal{M}_{\bigcirc}, \mathcal{M}_{\pentagon}), (\leq_{\bigcirc}, \leq_{\pentagon}), \texttt{prog})$ such that* `gpm` *yields a concrete progress measure as defined by Definition 3.5.*
PROOF By Lemma 5.5 we have that the tangles found by `extractTangles` are valid tangles, therefore, the `attrT` function produces correct tangle attractor sets. Since tangle attractor sets are always a superset of their respective regular attractor sets this means that more vertices are attracted to higher regions which can only cause these vertices to be lifted higher than they would be had they been attracted to a lower region. Therefore, by Lemma 5.4 we find that we would always lift at least one measure if the original `gpm` algorithm would have. ■

Finally, we prove that `pmtl` correctly solves a parity game.

Theorem 5.3 *Given a game $\mathcal{G}$. When* `pmtl` *has finished executing it ensures that every vertex is marked as won by $\bigcirc$ or $\pentagon$ and if the vertex is controlled by the winning* *player it has it with a winning strategy.*

PROOF By Lemma 5.5 we have that the tangles found by `extractTangles` are valid tangles. The function `extractTangles` also marks any tangles with no escapes (edges from a vertex owned by the losing player that leave the tangle) as dominions. In `update` these dominions are maximised marked as won, marked with a strategy and removed from the game. Since the tangles are valid, so are the dominions. Since for every α-dominion there exist a strategy for α to win (computed by the tangle attractor), all the vertices in this dominion are marked with the correct winner and winning strategy.

For all remaining vertices we have by Lemma 3.1, Lemma 5.6, and Lemma 5.7 that the measures computed by `pmtl` and `gpm` must be the same. Therefore, this theorem holds by Theorem 3.1. ■

5.4 Termination

Notably missing from the proofs for the three variants of PMTL is a proof that the algorithm terminates. For this proof we simply rely on the termination proof of the original progress measure-based algorithms. Because by Definition 3.5 the fixed point obtained by repeatedly lifting vertices must be progress measure that must mean there exists such a fixed point. The termination proofs of the SMALL PROGRESS MEASURES and ORDERED PROGRESS MEASURES algorithms simply rely on the monotonicity of their progression function `prog` and the fact that their measure spaces are finite.

Summary

In this chapter we discussed three different variants of PMTL and proved their correctness. Additionally, we provided the reasoning behind choices that were made and highlighted areas which require further testing. These tests and a comparison of the variants are presented in chapter 7.

Chapter 6

Analysis

Introduction

This chapter discusses the time and space complexity of the PMTL algorithm.

6.1 Time complexity

For a given progress measure we can take the set $\mathcal{M}$ to be the set of all possible measures. We can see that to solve the game we need to perform at most $|V| \cdot |\mathcal{M}|$ lift operations per player. In the worst case there will be at most one lift in every iteration of the loop on line 5 of `pmtl` (Algorithm 5.1), meaning we need $2 \cdot |V| \cdot |\mathcal{M}|$ iterations. In the worst case every region consists of a single vertex meaning every call to `update` (Algorithm 5.2, Algorithm 5.4, and Algorithm 5.5) will need to perform $|V|$ iterations of the loop on line 4 of `update`. Every iteration of this loop consists of calculating the (tangle) attractor, lifting the vertices in the region, and extracting tangles. Finally, `update` also attracts and updates higher-priority vertices or a single layer of vertices around the region.

Computing the attractor, in the worst case, requires checking every edge in the game. This would give computing the attractor a time complexity of $O(|E|)$. The tangle attractor requires checking for every vertex of every *found* tangle whether it is attracted or can escape. While solving the game we can find at most one tangle for every region in every iteration, meaning we find at most $|V|^2 \cdot |\mathcal{M}|$ tangles. This gives computing the tangle attractor an upper-bound of $O\left(|V|^3 \cdot |\mathcal{M}|\right)$ steps. Note that it should be possible to find a much tighter upper bound since finding more tangles causes regions to grow in size which causes there to be fewer regions and more vertices that are attracted to higher regions which means they can get lifted higher which means we need less iterations.

The time complexity of lifting a vertex is dependent on the complexity of the progress measures' `prog` function. For Ordered Progress Measures the update/`prog` function has an average cost of $O(1)$, or $O(\texttt{log}\,|\mathcal{M}|)$ if we want to save $O(|E| \cdot \texttt{log}\,\texttt{log}\,|\mathcal{M}|)$ space[9]. For Small Progress Measures the lift function can be implemented to work in $O(d \cdot |E|)$ where d is the highest priority in the game[11]. Attempting to lift all the vertices in a region takes at most $|V| \cdot \mathit{LiftCost}$.

For every region we attempt to extract tangles, this takes $O(|E|)$[5].

Attracting and updating higher-priority vertices or a single layer of vertices around the region has a similar complexity to the attracting and updating of the region itself. Since attracting and updating higher-priority vertices is done in steps of increasing maximum priority it has a complexity of $O\left(d \cdot \left(|E| + |V|^3 \cdot |\mathcal{M}|\right) + |V| \cdot \mathit{LiftCost}\right)$. Attracting a single layer of vertices is simpler since it simply requires checking all incoming edges of the regions giving us a complexity of $O(|E| \cdot \mathit{LiftCost})$.

Finally, in every iteration we also attempt to extend any dominions which were found. This has the same complexity as the other attractor computations and simply happens once per iteration.

In conclusion, after simplification PMTL has a worst-case time complexity of

$$2 \cdot \left((d+2) \cdot |V|^5 \cdot |\mathcal{M}|^2 + |V|^2 \cdot |\mathcal{M}| \cdot ((d+2) \cdot |E| + (d+1) \cdot |V| \cdot \mathit{LiftCost})\right)$$

for the ‘attract up’ variant, which is in the order $O\left(d \cdot |V|^5 \cdot |\mathcal{M}|^2\right)$, or

$$2 \cdot \left(2 \cdot |V|^5 \cdot |\mathcal{M}|^2 + |V|^2 \cdot |\mathcal{M}| \cdot (2\,|E| + (|V| + |E|) \cdot \mathit{LiftCost})\right)$$

for the variant which only attracts a single layer of vertices around the attractor set, which is in the order $O\left(|V|^5 \cdot |\mathcal{M}|^2\right)$.

From this we find that PMTL has a polynomial time complexity if used with a progress measure with a polynomial amount of possible measures. Since such a progress measure has not been found the best time complexity arises from the use of PMTL with a quasi-polynomial progress measure such as Ordered Progress Measures. In this case PMTL has a quasi-polynomial time complexity.

Note that the time complexity given here is likely a gross overestimate of the actual time complexity of our algorithm since we used the worst case for the amount of tangles found and the worst case for the amount of iterations needed, but finding more tangles means we need fewer iterations.

6.2 Space complexity

Determining the space complexity is relatively trivial. For every vertex we need to keep track of two measures (the even ○ and odd ⬠ measures) which requires $O(2\,|V| \cdot \mathit{MeasureSize})$. For Ordered Progress Measures the size of a measure is $O(\log |\mathcal{M}|)$ per vertex, to get $O(1)$ time complexity for its update function we need to store an additional $O(\log\log |\mathcal{M}|)$ for every edge[9]. For Small Progress Measures every measure consists of d integers[11].

Additionally, we need to store the discovered tangles. In section 6.1 we found that we find at most $|V|^2 \cdot |\mathcal{M}|$ tangles in a game. Every tangle can be stored in $\log_2 |V|$ bits. However, to achieve a better time upper-bound we store the vertices that the opponent can escape to requiring an additional $\log_2 |V|$ bits.

In conclusion, PMTL has a space complexity of:

$$2 \cdot |V| \cdot \mathit{MeasureSize} + 4 \cdot |V|^2 \cdot |\mathcal{M}| \cdot \mathtt{log}_2 |V|$$

which is in the order of

$$O(|V|^2 \cdot \mathtt{log}\, |V| \cdot |\mathcal{M}|)$$

This is exactly the space complexity one would expect when running TANGLE LEARNING and a value-iteration progress measure algorithm concurrently.

Summary

In this chapter we determined that the time and space complexity of PMTL is primarily dependent on the time and space complexity of the progress measure chosen. The amount of work PMTL performs in each iteration is polynomial and the amount of iterations (in the worst case) is determined by the progress measure. Therefore, a quasi-polynomial progress measure yields a quasi-polynomial time complexity for PMTL, an exponential progress measure yields an exponential time complexity, and so on. The space complexity was equivalent to running the TANGLE LEARNING and progress measure algorithms concurrently.

Chapter 7

Experiments

Introduction

In this chapter we perform empirical testing of the different variants of PMTL presented in chapter 5. Additionally, we analyse the results of these benchmarks to identify what the strengths and weaknesses of the algorithm are. Finally, we test some potential improvements to the algorithm.

7.1 Methodology

In chapter 5 we presented three different variants of the PMTL algorithm. While the variant that 'attracts up' is able to propagate a higher measure further within the same iteration than the variant which does not, this does not necessarily mean that this variant is faster when implemented. Therefore, it is important to run benchmarks to compare these variants. Furthermore, to analyse whether the PMTL algorithm is a feasible alternative to existing parity game solving algorithms we compare its variants to progress measure algorithms and tangle learning variants.

We test two variants of PMTL with two measures each. The first variant we test is a hybrid of the base version (section 5.1) and the version with tangles (section 5.3) which uses tangle attractors, but only attracts inside the region and a single step outside of it. In the benchmarks we refer to this variant as 'No-Up'. The second variant is the version with tangles (section 5.3), but *with* the step where we attract upwards, attracting vertices with a higher priority than the top of the region. In the benchmarks we refer to this variant as 'Up'. Both of these variants are tested using the SMALL PROGRESS MEASURES and the ORDERED PROGRESS MEASURES.

Since the different parity game algorithms respond differently to different graph structures it is important to run our benchmarks with a variety of different graphs. Firstly, to explore whether PMTL is able to avoid the distractions which slow down tangle learning we use the 'Two Counters' games which are worst-case examples which show tangle learning to have an exponential lower bound.[4] Secondly, since there are algorithms like RECURSIVE TANGLE LEARNING which solve the 'Two Counters' game in polynomial time we also test PMTL against REcursive Tangle Learning on the ‘Two Counters+’ games which show that Recursive Tangle Learning has an exponential lower bound.[7] Thirdly, we use 4 sets of 1000 random games with 25, 50, 100, and 200 vertices respectively which we use to compare against tangle learning and the regular progress measures algorithms. These random games are generated using Oink with a maximum vertex priority equal to the number of vertices and a maximum number of edges equal to four times the number of vertices making these sparse graphs. Fourthly, we use the worst case example for Zielonka’s algorithm by Gazda[10] as implemented in Oink since this is an example where PMTL performs significantly worse than the regular progress measure algorithms. Finally, we use some games generated using Knor based on the HOA files from SYNTCOMP 2023 which resemble games that are close to the real-world usage of solving parity games.[1]

The benchmarks were run using the *hyperfine*[2] command-line benchmarking tool. All benchmarks except for the random games and some of the ‘Two Counters+’ games were run with 5 warmup runs and with a minimum of 20 actual runs of the benchmark. The random games and the longer benchmarks for the ‘Two Counters+’ games were run with 1 warmup run and a minimum of 5 actual runs. If the total benchmarking time was less than 3 seconds hyperfine adds runs until 3 seconds have passed. The generation of the games and the execution of the solvers were performed using Oink and its accompanying tools. The implementation and benchmarks can be found at https://doi.org/10.5281/zenodo.10558316.

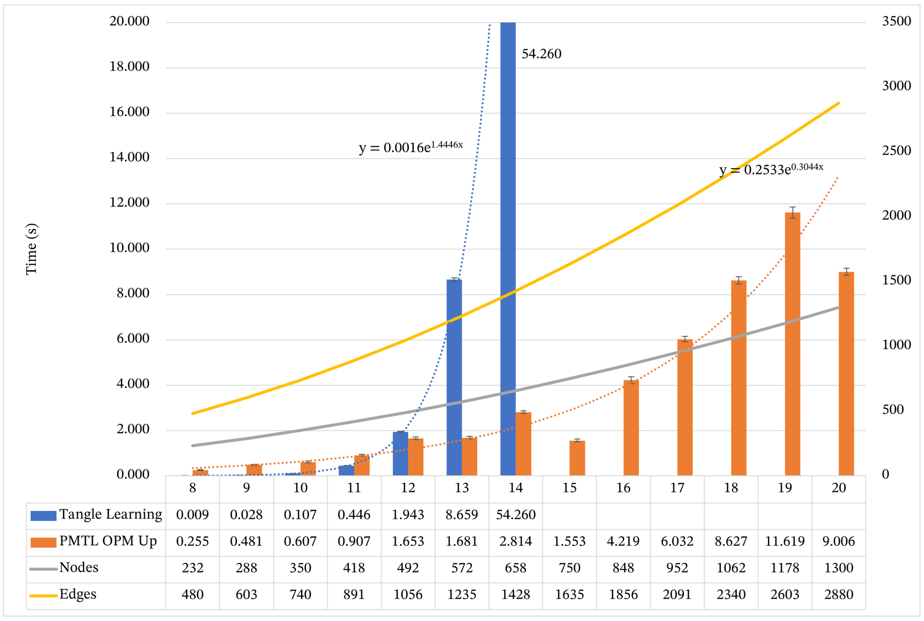
All the benchmarks were run on an Intel i7–8750H @ 2.20 Ghz with 6 cores and 12 threads, however the benchmarks only run on a single core.

[1]Knor found at https://github.com/trolando/knor. HOA files found at https://github.com/SYNTCOMP/benchmarks/releases/tag/v2023.4.

[2]Found at https://github.com/sharkdp/hyperfine.

7.2 Results

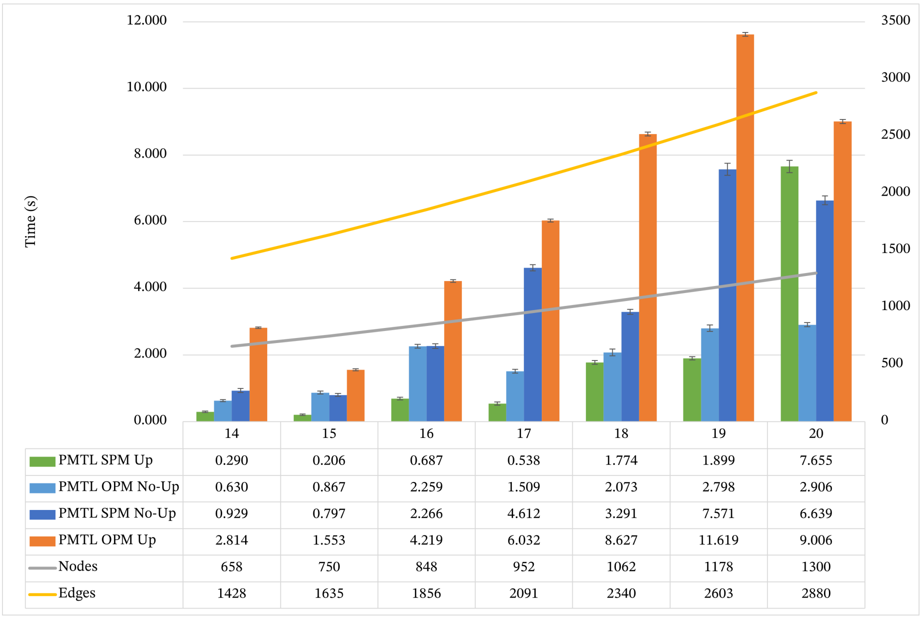
7.2.1 ‘Two Counters’



	8	9	10	11	12	13	14	15	16	17	18	19	20
🟦 Tangle Learning	0.009	0.028	0.107	0.446	1.943	8.659	54.260						
🟧 PMTL OPM Up	0.255	0.481	0.607	0.907	1.653	1.681	2.814	1.553	4.219	6.032	8.627	11.619	9.006
— Nodes	232	288	350	418	492	572	658	750	848	952	1062	1178	1300
— Edges	480	603	740	891	1056	1235	1428	1635	1856	2091	2340	2603	2880

Figure 7.1: Tangle Learning versus PMTL ‘going up’ with Ordered Progress Measures solving ‘Two Counters’ games with between 8 and 20 bits. For legibility the top of the graph has been truncated. Error bars represent 1σ.

Figure 7.1 shows that in this worst-case example of a game for tangle learning that PMTL is able to outperform tangle learning. A comparison with the regular Ordered Progress Measures is not shown here since a 6-bit two counters game already takes over 7 minutes to solve. While this result shows that PMTL is not equivalent to tangle learning there are still a lot of other variations of tangle learning which are significantly faster than PMTL. For example, recursive tangle learning solves the 20-bit two counters game in 3 milliseconds. Additionally, we can see that for the 15 and 20-bit games the solving time decreases compared to the smaller game. It is unclear what the cause of this is.

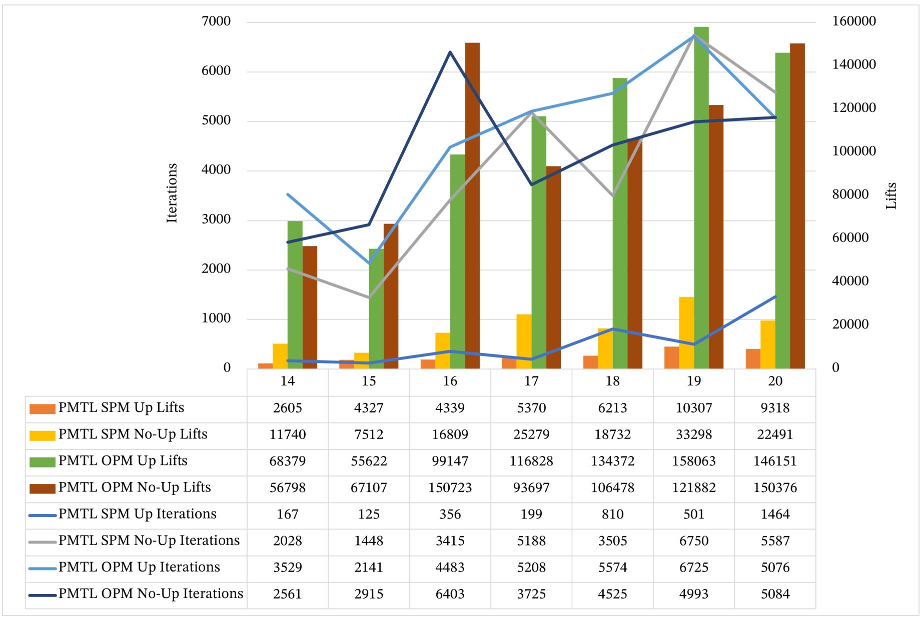


	14	15	16	17	18	19	20
🟩 PMTL SPM Up	0.290	0.206	0.687	0.538	1.774	1.899	7.655
🟦 PMTL OPM No-Up	0.630	0.867	2.259	1.509	2.073	2.798	2.906
🟦 PMTL SPM No-Up	0.929	0.797	2.266	4.612	3.291	7.571	6.639
🟧 PMTL OPM Up	2.814	1.553	4.219	6.032	8.627	11.619	9.006
— Nodes	658	750	848	952	1062	1178	1300
— Edges	1428	1635	1856	2091	2340	2603	2880

Figure 7.2: Four variants of PMTL solving 'Two Counters' games with between 14 and 20 bits. Error bars represent 1σ.

Figure 7.2 shows a comparison between the different PMTL variants. We can see that for the Ordered Progress Measures variant the PMTL version that does not attract up has an advantage, whereas for the Small Progress Measures variant the PMTL version that does attract up has an advantage. While attracting up allows a measure to 'spread' further within the same iteration, not attracting up means that there are fewer lifts per iteration which means that we reach the next iteration sooner allowing a higher measure to be 'spread'.

Additionally, we can see that on smaller games the Small Progress Measures variants of PMTL outperform the Ordered Progress Measures variants. This is likely because the comparison operation between Small Progress Measures is faster than the comparison operation between Ordered Progress Measures. Using a profiler we determined that more than 50% of the time spent solving games was spent on comparing measures for all variants. The reason that the Ordered Progress Measures variants of PMTL outperform the Small Progress Measures variants on the larger games is likely due to the quasi-polynomial lower bound for OPM and the exponential lower bound for SPM. The aforementioned dips in solving time for the 15 and 20-bit games are seen again here for the Ordered Progress Measures PMTL variants and only on the 15-bit game for the Small Progress Measures variants. This means that the effect cannot be purely explained by the quasi-polynomial nature of the Ordered Progress Measures.



	14	15	16	17	18	19	20
🟧 PMTL SPM Up Lifts	2605	4327	4339	5370	6213	10307	9318
🟨 PMTL SPM No-Up Lifts	11740	7512	16809	25279	18732	33298	22491
🟩 PMTL OPM Up Lifts	68379	55622	99147	116828	134372	158063	146151
🟫 PMTL OPM No-Up Lifts	56798	67107	150723	93697	106478	121882	150376
— PMTL SPM Up Iterations	167	125	356	199	810	501	1464
— PMTL SPM No-Up Iterations	2028	1448	3415	5188	3505	6750	5587
— PMTL OPM Up Iterations	3529	2141	4483	5208	5574	6725	5076
— PMTL OPM No-Up Iterations	2561	2915	6403	3725	4525	4993	5084

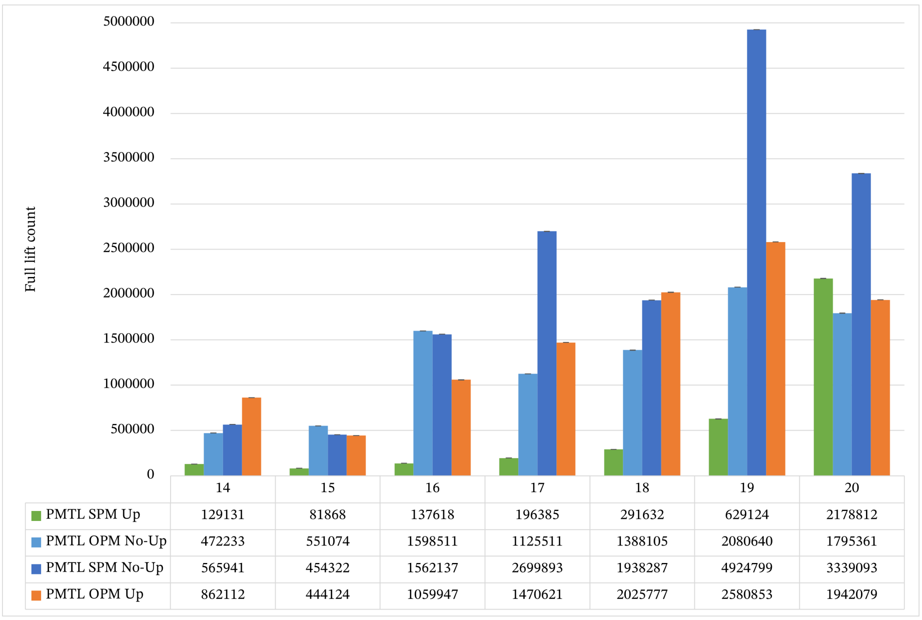
Figure 7.3: Four variants of PMTL solving ‘Two Counters’ games, showing the number of lifts and iterations that were required for games between 14 and 20 bits.

To gain a better insight into the differences between the PMTL variants we can take a look at Figure 7.3 which shows that the amount of lifts and iterations needed to solve the games differs significantly between the different variants. Note that a lift here refers only to successful lifts (if the progress function returned the same measure that was already applied to the vertex it is not counted) and iterations refers to the amount of iterations of the loop in the `pmtl` function as shown in Algorithm 5.1.

When comparing Figure 7.2 and Figure 7.3 we find that there is no clear relationship between the amount of successful lifts and iterations, and the amount of time taken to solve the game. For example the SPM Up variant performs 9318 successful lifts and 1464 iterations in around 6.6 seconds for the 20-bit game whereas the OPM No-Up variant performs 215830 successful lifts and 5061 iterations in around 2.1 seconds for the same game.

Bits	SPM Up		SPM No-Up		OPM Up		OPM No-Up	
	Lifts	Hit%	Lifts	Hit%	Lifts	Hit%	Lifts	Hit%
14	129131	2.02%	565941	2.07%	862112	7.93%	472233	12.03%
15	81868	5.29%	454322	1.65%	444124	12.52%	551074	12.18%
16	137618	3.15%	1562137	1.08%	1059947	9.35%	1598511	9.43%
17	196385	2.73%	2699893	0.94%	1470621	7.94%	1125511	8.32%
18	291632	2.13%	1938287	0.97%	2025777	6.63%	1388105	7.67%
19	629124	1.64%	4924799	0.68%	2580853	6.12%	2080640	5.86%
20	2178812	0.43%	3339093	0.67%	1942079	7.53%	1795361	8.38%

Table 7.1: Four variants of PMTL solving 'Two Counters' games, showing the *total* number of lifts and the percentage of those which were successful for games between 14 and 20 bits.



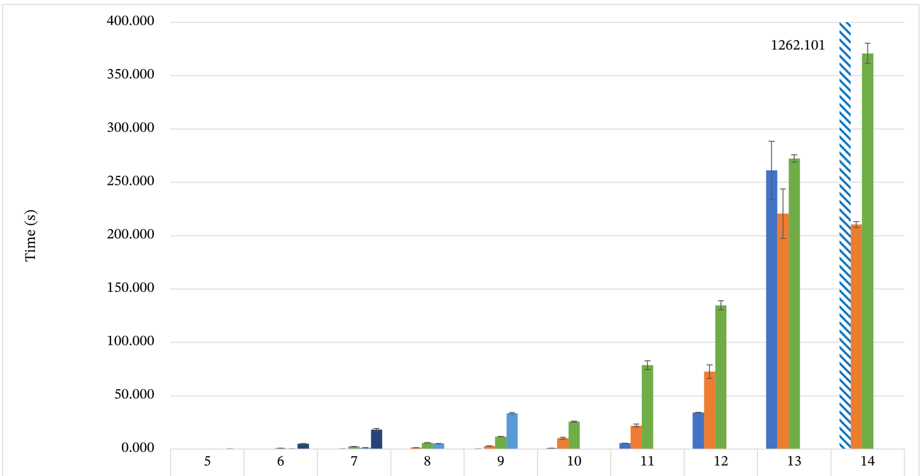
	14	15	16	17	18	19	20
🟩 PMTL SPM Up	129131	81868	137618	196385	291632	629124	2178812
🟦 PMTL OPM No-Up	472233	551074	1598511	1125511	1388105	2080640	1795361
🟦 PMTL SPM No-Up	565941	454322	1562137	2699893	1938287	4924799	3339093
🟧 PMTL OPM Up	862112	444124	1059947	1470621	2025777	2580853	1942079

Figure 7.4: Four variants of PMTL solving 'Two Counters' games, showing the *total* number of lifts required for games between 14 and 20 bits.

Measuring the *total* number of lifts, including those that did not yield an increased measure, provides more insight into the differences between the PMTL variants. This is shown in Table 7.1 which also includes the percentage of lifts which successfully increased a vertex' measure. While the ORDERED PROGRESS MEASURES variants seem to perform more successful lifts they do not necessarily perform more lifts in total. Seemingly the ORDERED PROGRESS MEASURES can measure smaller amounts of progress than the SMALL PROGRESS MEASURES this yields more overhead because the measures changes more often. However, this also means that OPM is able to make more progress sooner than SPM which is likely what explains their better performance on larger games. Comparing Figure 7.2 and

Figure 7.4 we can see a clearer correlation between the total number of lifts and the time taken than between the number of successful lifts and the time taken.

7.2.2 ‘Two Counters+’

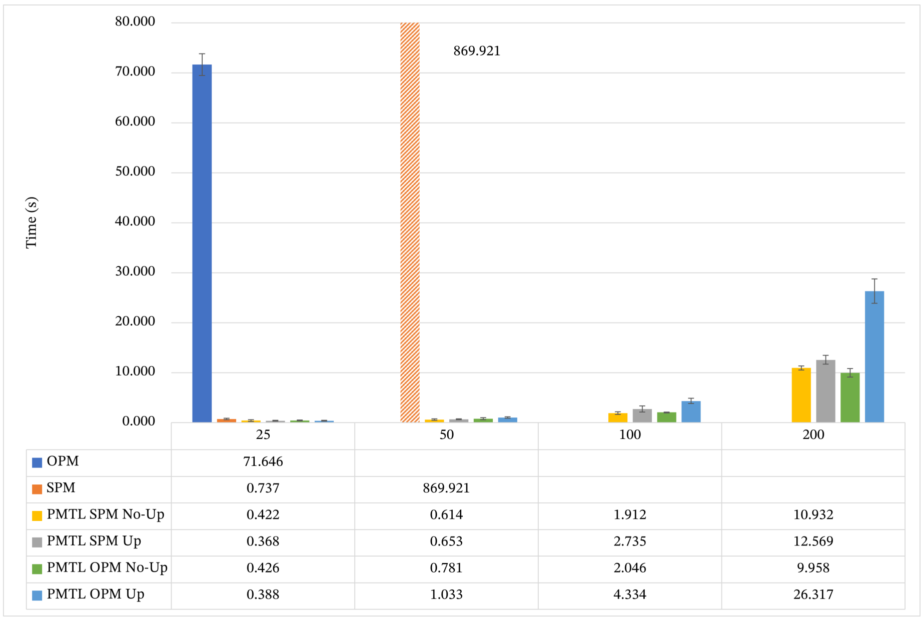


	5	6	7	8	9	10	11	12	13	14
■ RTL	0.003	0.004	0.011	0.042	0.198	0.948	5.565	34.175	261.140	1262.101
■ PMTL SPM No-Up	0.027	0.111	0.365	1.363	2.891	10.177	21.951	72.651	220.603	210.376
■ PMTL OPM No-Up	0.115	0.876	2.480	6.003	11.784	25.656	78.663	134.638	272.343	370.858
■ PMTL SPM Up	0.063	0.283	1.288	5.198	33.432					
■ PMTL OPM Up	0.250	5.045	18.295							

Figure 7.5: Recursive Tangle Learning versus four variants of PMTL solving ‘Two Counters+’ games with between 5 and 14 bits. The cut-off striped bar was a single run of Recursive Tangle Learning which was not repeated the usual 5 times because it took more than 21 minutes. Error bars represent 1σ.

Figure 7.5 shows that for ‘Two Counters+’ games between 5 and 12 bits Recursive Tangle Learning is significantly faster than the PMTL variants. However, for the larger two games PMTL shows a large advantage over Recursive Tangle Learning. Given that Ordered Progress Measures already took more than 9.5 minutes on the game with 5 bits and Small Progress Measures already took more than 3.5 minutes on the game with 6 bits (solving the 5-bit game in about 5 seconds) we can see that PMTL offers an advantage over both the ‘generally fast in practice’ tangle learning algorithms and the ‘good worst-case time complexity’ progress measure algorithms.

7.2.3 Random games



	25	50	100	200
🟦 OPM	71.646			
🟧 SPM	0.737	869.921		
🟨 PMTL SPM No-Up	0.422	0.614	1.912	10.932
⬜ PMTL SPM Up	0.368	0.653	2.735	12.569
🟩 PMTL OPM No-Up	0.426	0.781	2.046	9.958
🟦 PMTL OPM Up	0.388	1.033	4.334	26.317

Figure 7.6: Ordered Progress Measures and Small Progress Measures versus four variants of PMTL solving 4 sets of 1000 random games with 25, 50, 100, and 200 vertices respectively. The cut-off striped bar was a single run of Small Progress Measures which was not repeated the usual 5 times because it took almost 15 minutes. Error bars represent 1σ.

Figure 7.6 shows that for random games PMTL is a lot faster than the value iteration algorithms. However, not shown in the graph is that the regular Tangle Learning algorithm needed less than 0.5 seconds to solve the 1000 random games with 200 vertices. In fact, Tangle Learning needed less than 2 seconds to solve 1000 random games with 2000 vertices, whereas PMTL took more than that on a single game from that set.

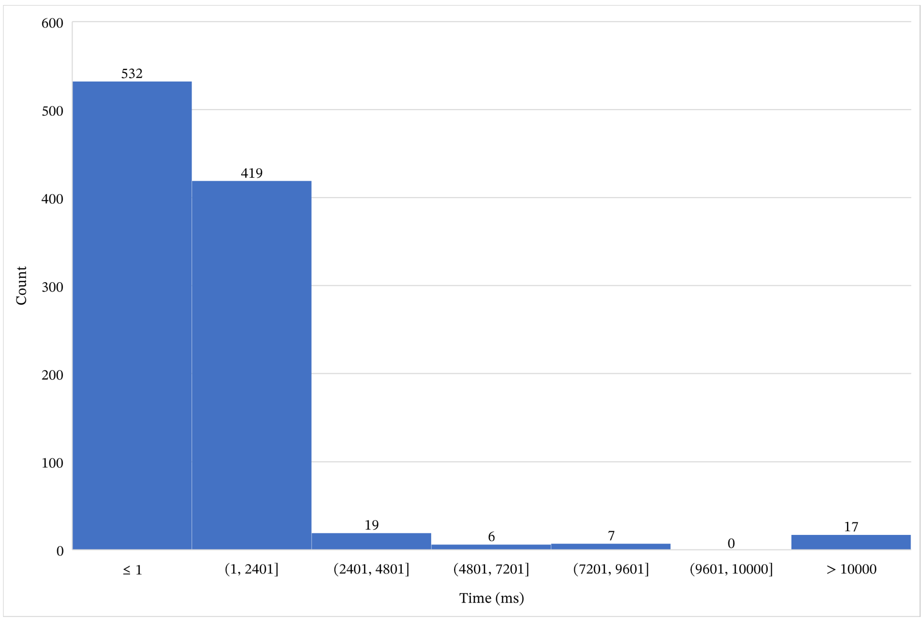


Figure 7.7: SMALL PROGRESS MEASURES solving 1000 random games with 50 vertices. Histogram of the time taken to solve individual games.

However, it should be noted that due to the random nature of the games there is a large amount of variability in the amount of time needed to solve individual games. We can see this in Figure 7.7 where we see that the time taken by SMALL PROGRESS MEASURES to solve a game with 50 vertices ranges from less than 1 ms to more than 2 minutes. Clearly TANGLE LEARNING handles sparse graphs very well whereas the progress measure based algorithms (including PMTL) are more significantly affected by other parameters like the number of priorities.

7.2.4 SYNTCOMP games

Cursory tests showed that PMTL performed in much the same way for the random games and the SYNTCOMP games. Because of this we only took a selection of games with between 7 and 35,657 vertices, since we expected solving larger games to take too long. Unfortunately, whereas TANGLE LEARNING needed only 452 ms to solve all 29 games, 'PMTL SPM Up' needed 5 minutes to a single one of the largest games in the set. 'PMTL OPM Up' did not finish solving that game even after waiting for 20 minutes. 'PMTL OPM No-Up' needed 16 minutes and 'PMTL SPM No-Up' needed 2.5 minutes. While PMTL performs well on the smaller games it struggles with large amounts of vertices.

7.3 Possible optimisations

While implementing the algorithm and testing it we came up with several ways in which it might be optimised. We did some cursory tests to determine whether these optimisations were worth pursuing further.

7.3.1 Counting escapes

There are multiple ways to implement the (tangle) attractors. The implementation of the results presented so far simply checks all the neighbours of a vertex to see whether they can act as an escape, if they can't, the vertex is attracted. Similarly, for the tangle attractor we keep track of a list of escapes and see whether all of these escapes have already been attracted to higher regions or the current regions. If all of them have, we attract the tangle. However, it may be more efficient to count the amount of escapes as we attempt to a vertex or a tangle. When such an attempt is made we can decrease the counter until it reaches 1, meaning that the only escape is the vertex we are currently trying to attract from, at which point we attract the vertex or tangle.

We implemented this and found that there was a slight improvement in the speed of the algorithm but not significant enough to warrant rerunning all the previous benchmarks.

7.3.2 Ignoring opponent-priority vertices

When first presented with Table 7.1 we speculated that the majority of the failed lifts must be attempts to lift vertices which have a priority of the opponent's parity. If a vertex with a priority of the current player is lifted it will always yield an increased measure if this vertex has not been attracted to the same measure before. However, this is not necessarily the case for vertices with the opponent's parity. Since all increases in a progress measure result from encountering a vertex of the current player's parity we can delay lifting the other vertices until the very end, at which point we have already been determined which vertex is won by which player.

We implemented this and found that the number of failed lifts almost always decreased significantly. However, in some cases the total number of lifts increased by as much as 35%. Additionally, in some cases the number of iterations increased by 50 to 70 percent. Clearly there are games where we benefit from having lifted the other vertices early and the majority of failed lifts are *not* caused by attempting to lift vertices. In terms of the amount of time taken by this variation's compared to the one which did not postpone the lifting of certain vertices we found between a 27% improvement and a 19% regression.

7.3.3 Skipping unchanged regions

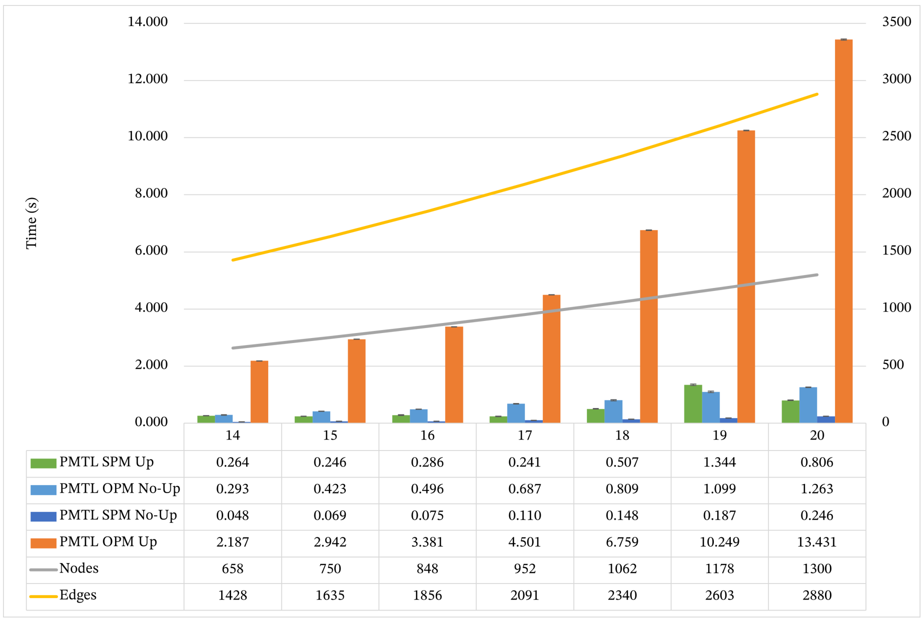
After seeing the results from the previous optimisation we realised that the majority of the failed lifts are due to repeatedly lifting unchanged regions. We added a check which skipped any regions which had not changed until reaching one which had changed. After that we can no longer skip regions since they might contain different vertices because of the region that changed.

We found that with this optimisation the number of failed lifts decreased by around 0.5% with all other metrics remaining the same. This did not yield any significant difference in the amount of time taken to solve a game since the overhead of keeping track of whether regions could be skipped likely cancelled out any improvement seen from the smaller lift count. The number of failed lifts decreased by a smaller amount than we had hoped. We speculate that the cause of this that it is common for one of the higher regions to change which means we can no longer skip the next regions even if they were the same.

To improve upon this we would need to store more information, similarly to Lapauw, Bruynooghe and Denecker's[14] approach with justifications which will be briefly covered in section 9.2.

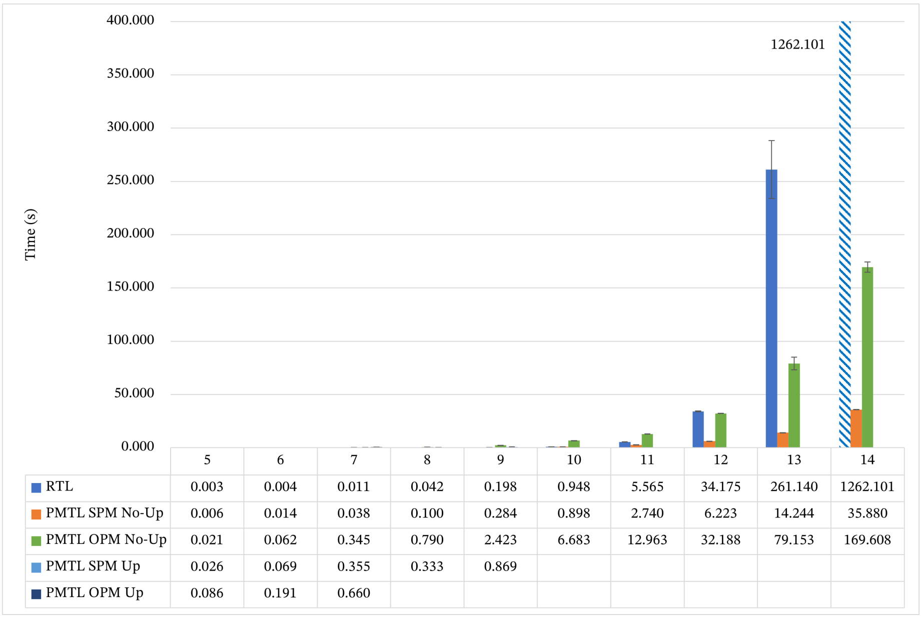
7.3.4 Completing the decomposition

While exploring the performance of PMTL on several of the counterexample games included in Oink we found that PMTL was significantly slowed by the worst case example from Gazda[10]. It was in fact the only game we found where the value-iteration progress measure algorithms outperformed PMTL. The culprit was the 'optimisation' mentioned in subsection 5.1.1 which meant that only a single region was updated in each iteration. As mentioned there this was done so that higher regions can be lifted before working on the lower regions.



	14	15	16	17	18	19	20
🟩 PMTL SPM Up	0.264	0.246	0.286	0.241	0.507	1.344	0.806
🟦 PMTL OPM No-Up	0.293	0.423	0.496	0.687	0.809	1.099	1.263
🟦 PMTL SPM No-Up	0.048	0.069	0.075	0.110	0.148	0.187	0.246
🟧 PMTL OPM Up	2.187	2.942	3.381	4.501	6.759	10.249	13.431
— Nodes	658	750	848	952	1062	1178	1300
— Edges	1428	1635	1856	2091	2340	2603	2880

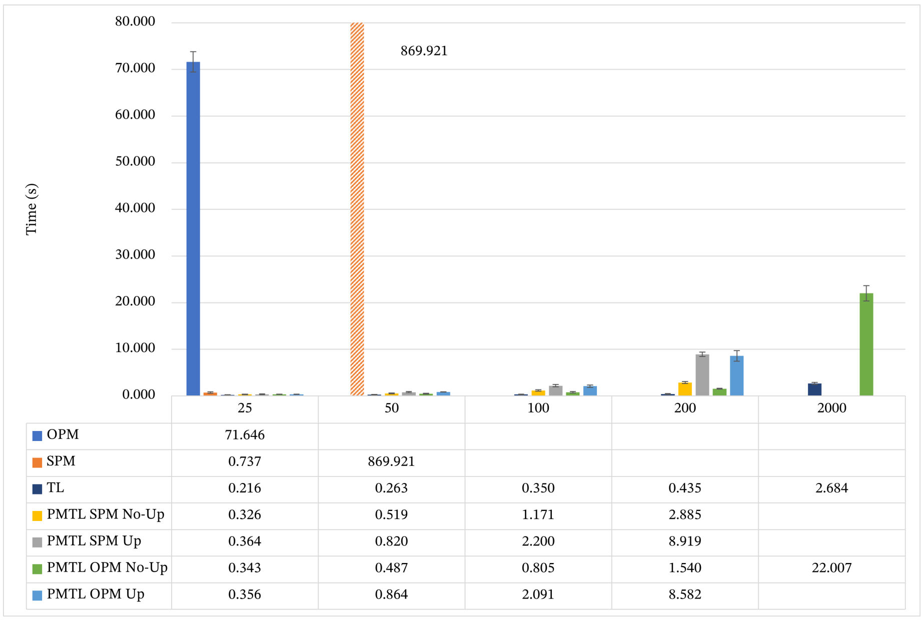
Figure 7.8: Four variants of PMTL solving 'Two Counters' games with between 14 and 20 bits. Error bars represent 1σ.



	5	6	7	8	9	10	11	12	13	14
■ RTL	0.003	0.004	0.011	0.042	0.198	0.948	5.565	34.175	261.140	1262.101
■ PMTL SPM No-Up	0.006	0.014	0.038	0.100	0.284	0.898	2.740	6.223	14.244	35.880
■ PMTL OPM No-Up	0.021	0.062	0.345	0.790	2.423	6.683	12.963	32.188	79.153	169.608
■ PMTL SPM Up	0.026	0.069	0.355	0.333	0.869					
■ PMTL OPM Up	0.086	0.191	0.660							

Figure 7.9: Recursive Tangle Learning versus four variants of PMTL solving ‘Two Counters+’ games with between 5 and 14 bits. The cut-off striped bar was a single run of RECURSIVE TANGLE LEARNING which was not repeated the usual 5 times because it took more than 21 minutes. Error bars represent 1σ.

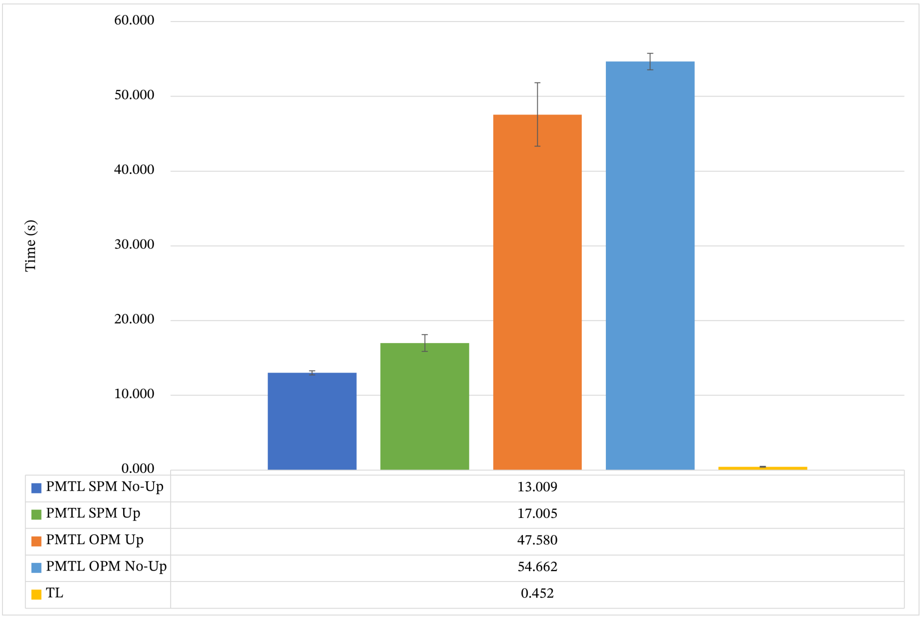
Comparing Figure 7.8 and Figure 7.9 with Figure 7.2 and Figure 7.5 shows that in almost all cases the version that does a full attractor decomposition before resetting is faster. In general, it seems that the ‘no-up’ variants get more benefit from this change, likely because the overhead of calculating the entire attractor decomposition is greater for the variants which attract up because every region needs a lot more attractor computations.



	25	50	100	200	2000
🟦 OPM	71.646				
🟧 SPM	0.737	869.921			
🟦 TL	0.216	0.263	0.350	0.435	2.684
🟨 PMTL SPM No-Up	0.326	0.519	1.171	2.885	
⬜ PMTL SPM Up	0.364	0.820	2.200	8.919	
🟩 PMTL OPM No-Up	0.343	0.487	0.805	1.540	22.007
🟦 PMTL OPM Up	0.356	0.864	2.091	8.582	

Figure 7.10: Ordered Progress Measures, Small Progress Measures and Tangle Learning versus four variants of PMTL solving 5 sets of 1000 random games with 25, 50, 100, 200, and 2000 vertices respectively. The cut-off striped bar was a single run of Small Progress Measures which was not repeated the usual 5 times because it took almost 15 minutes. Error bars represent 1σ.

The result of re-running our benchmarks for the random games is shown in Figure 7.10. Since there was a significant speed increase it was now useful to also test random games with 2000 vertices and to compare against Tangle Learning. By completing entire attractor decompositions we are able to find more tangles and dominions which has brought down the performance gap between Tangle Learning and PMTL.



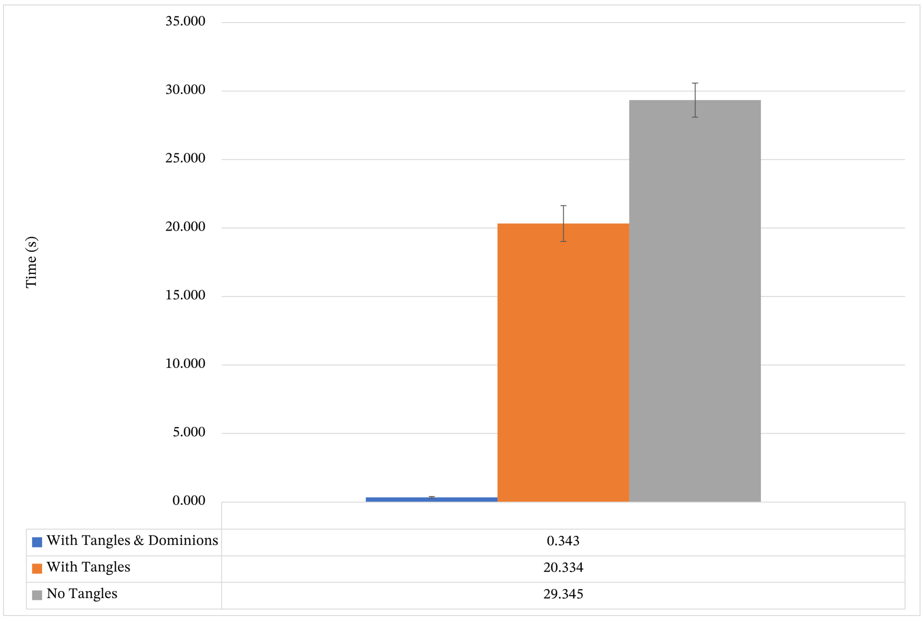
🟦 PMTL SPM No-Up	13.009
🟩 PMTL SPM Up	17.005
🟧 PMTL OPM Up	47.580
🟦 PMTL OPM No-Up	54.662
🟨 TL	0.452

Figure 7.11: Tangle Learning versus four variants of PMTL solving 29 SYNTCOMP games. Error bars represent 1σ.

In Figure 7.11 we see that, similar to the random games, this change significantly improves the solving time for the SYNTCOMP games. It is unsurprising that there is still significant gap between Tangle Learning and PMTL considering the size of the games and the performance we saw on the random games. On the large game for which we reported the execution times in subsection 7.2.4 we now see solving times between 0.7 and 2.1 seconds which is significantly faster but still more than the time Tangle Learning takes to solve all 29 games.

7.4 Do we need tangles

The motivation behind using tangle attractors instead of the regular attractors for the PMTL algorithm was twofold: tangle attractors find larger regions because they can attract more vertices, and extracting tangles also finds dominions which can be removed from the game. However, since we experimentally found that in most cases our ‘Going Up’ variant did not improve upon our most basic variant of the algorithm we can conclude that lifting vertices to higher regions is not always worth it in practice if it incurs too much overhead. Therefore, without testing we cannot conclude that tangle attractors finding larger regions will improve performance.



🟦 With Tangles & Dominions	0.343
🟧 With Tangles	20.334
⬜ No Tangles	29.345

Figure 7.12: The regular PMTL with tangles and dominions versus PMTL with tangles and no dominions versus PMTL without tangles and dominions solving 1000 random games with 25 vertices. All tests were run on the variant that uses Ordered Progress Measures and the 'No-Up' strategy. Error bars represent 1σ.

To distinguish between the effect of finding larger regions and removing dominions from the game we compare the regular version of PMTL which uses tangle attractors and detects dominions, with a version that uses tangle attractors but does not remove dominions from the game when they are found and a version that only uses regular attractors and does not detect tangles or dominions. In Figure 7.12 we compare the three versions of PMTL when solving 1000 games with 25 vertices each. We chose to use this example since for the 'Two Counters' and 'Two Counters+' games either the games were too small to show any differences or the running-time of the version without tangles became too long to obtain results in a reasonable amount of time.

From the data in Figure 7.12 we can conclude that both the use of tangle attractors and the removal of dominions improve the algorithm. Recall that in Figure 7.10 we compared variants of PMTL which used tangle attractors and removed dominions from the game against the progress measure algorithms that we are accelerating. We can see that all PMTL variants are faster than the Ordered Progress Measures.

In section 5.3 we speculated that we are not only accelerating progress measures using attractors, but that we are also accelerating the discovery of dominions through the use of progress measures. Earlier in this chapter we already showed examples of PMTL outperforming Tangle Learning and Recursive Tangle Learning on their respective worst-case example games. However, when we dig into how these games are solved we find that no vertices are marked as won by reaching a fixpoint of the measures and calling `solve`. Instead, all vertices for these 'Two Counters' and 'Two Counters+' are solved after the algorithm finds a dominion. This tells us that we must somehow be finding a more optimal attractor decomposition earlier than the regular TANGLE LEARNING algorithm.

Summary

In this chapter we compared the different variants of PMTL against each other and against other parity game solving algorithms. We found that while TANGLE LEARNING is faster than PMTL for most games, there are some games for which PMTL has better performance. In all cases, at least the improved version described in subsection 7.3.4, was significantly faster than the regular progress measure algorithms. We found that in most cases attracting higher priority regions does not accelerate the algorithm and in fact causes the algorithm to perform worse due to the overhead caused by the extra attractor calculations. Additionally, we determined that much, but not all the performance improvement over plain progress measures are due to the discovery of tangles and dominions. Finally, we tested some possible optimisations and found that most did not yield a significant improvement.

Chapter 8

Related work

Introduction

In this chapter we introduce two other attempts at synthesising progress measures and an attractor-based algorithm. We describe how they work, compare them to PMTL, and discuss whether the approach PMTL takes is sufficiently distinct.

8.1 Attractor Decomposition Lifting

Jurdziński et al.[13] already discuss a new quasi-polynomial algorithm that combines value iteration and attractor decompositions. This algorithm directly uses universal trees (the underlying concept which seems to be at the basis of all quasi-polynomial parity game algorithms) to construct a labelling for vertices. The tree is expanded (lazified) with `before` and `after` nodes for every existing node. The labelling is created by assigning vertices to nodes in the lazified tree with the same level as the priority of the vertex for all the regular nodes from the universal tree or to nodes with level greater or equal to the priority of the vertex for the lazy nodes which were added by the lazification.

The paper introduces a notion of validity for edges and vertices for labellings μ which is similar to the concept of μ-progressive edges for Small Progress Measures. When we have an edge (u, v) it is called μ-progressive for some α-measure function μ such that if $\texttt{pr}(u) \equiv_2 \bar{\alpha}$ then $\mu(u) \geq_{\texttt{pr}(u)} \mu(v)$, or if $\texttt{pr}(u) \equiv_2 \alpha$ then either $\mu(u) >_{\texttt{pr}(u)} \mu(v)$ or $\mu(v) = \mu(u) = \top$. We can use this concept to give an alternative definition of a progress measure (Definition 2.13):

Definition 8.1 (Progress Measure) A measure function $\mu : V \rightarrow \mathcal{M}$ is an α-progress measure if the following conditions hold for every vertex u in the game:

- if $u \in V_\alpha(\mathcal{G})$ then there exists an edge $(u, v) \in E(\mathcal{G})$ such that (u, v) is μ-progressive
- if $vinV_{\bar{\alpha}}(\mathcal{G})$ then all edges $(u, v) \in \mathcal{E}$ must be μ-progressive

Additionally, μ must be the smallest measure function where these conditions hold.□

Lifting the measures of vertices can then be seen as repeatedly attempting to fix the measures of vertices such that they get μ-progressive edges.

Now we can look at what the similar concept of validity looks like for labellings μ. For an edge (u, v) if v is assigned to a regular node $\mu(v)$, then v must be labelled with nodes smaller than the `after` node of that regular node. This is very similar to the μ-progressiveness requirement that $\mu(v) \leq_{\mathtt{pr}(u)} \mu(u)$. If u is assigned to a lazy node $\mu(u)$ then either $\mu(v) < \mu(u)$ or $\mu(v) = \mu(u)$ and there is tangle ensuring a vertex x with $\mu(x) < \mu(u)$ can be reached without passing through vertices assigned to a node that is greater or equal to $\mu(u)$. When either of these cases is true or if $\mu(u) = \top$ then the edge is valid.

This notion of validity can be extended to vertices in the same manner as μ-progressiveness was. A mapping μ from vertices to the lazified tree is an α-labelling (progress measure) if all vertices are valid. An α-owned vertex is valid if it has an outgoing valid edge and a $\bar{\alpha}$-owned vertex is valid if all its outgoing edges are valid. Jurdziński et al.[13] define such an α-labelling to be an α-embedded attractor decomposition in the α-universal tree used to construct the labelling.

Similarly to Small Progress Measures a winning strategy for vertices v with $\mu(v) < \top$ can be found by choosing any valid edge. Simply lifting such a labelling until all vertices are valid, results in an algorithm that runs in quasi-polynomial time when universal trees are used to construct the even ○ and odd ⬠ labellings. The resulting algorithm is referred to as Asymmetric Attractor Decomposition Lifting.

The paper also introduces an accelerated symmetric variant of this algorithm which runs on a tree that is the interleaving of the even ○ and odd ⬠ trees. Every iteration of this algorithm checks if its $\mathtt{Scope}(n)$ (all vertices that are labelled with nodes in the tree that are in the subtree of the current node n in the interleaving) contains only valid vertices for α's labelling, if that is the case, all of those vertices can immediately be labelled in $\bar{\alpha}$'s labelling with the node $\mathtt{after}(n^{\bar{\alpha}})$ where $n^{\bar{\alpha}}$ is the node in $\bar{\alpha}$'s tree from which the interleaved node n originated.

Essentially, this allows information learned through one of the labellings to be used for the other labelling in a more granular manner than what is done for the well-known progress measure algorithms. In these algorithms, only when one of the progress measures is done being lifted, can the discovered dominion be removed from the subgame that the other progress measure is iterating upon. This approach of allowing the two separate measures to interact may be similar to Benerecetti et al.'s[1] approach which uses quasi-dominions to accelerate progress measures.

Instead of directly computing attractors (and therefore attractor decompositions) Attractor Decomposition Lifting works by encoding the decompositions in the labellings (measures) and lifting those measures along their edges essentially changing the regions of the attractor decomposition edge-by-edge. Due to the size of the universal tree this yields a quasi-polynomial time algorithm whose behaviour matches attractor-based algorithms. In contrast, PMTL computes attractors and the decompositions directly and relies on lifting the progress measures of the vertices across a wider area instead of from edge-to-edge, all the while keeping track of tangles to hopefully find a dominion. In essence, Attractor Decomposition Lifting relies on finding the attractor decomposition which when given to an attractor-based algorithm would solve the game in a single iteration, whereas

PMTL merely uses the attractor decompositions to accelerate a progress measure which has its own wholly separate way of determining the solution. However, we do see that most games solved by PMTL are solved because dominions are found. Still, PMTL does not encode the attractor decompositions in the measures and consequently can be used with any existing progress measure.

Since there is no code or benchmark data available, the performance of ATTRACTOR DECOMPOSITION LIFTING and PMTL could not be compared.

8.2 QUASI-DOMINION PROGRESS MEASURES

The paper by Benerecetti et al.[1] redefines much of the existing theory around progress measures in formal terms. Specifically, the paper introduces the abstract concepts of a measure space and a progress measure. A measure space is a structure consisting of a set of measures, a total order on those measures, a truncation operator, and a stretch operator. The truncation operator maps a measure η to a measure $\leq \eta$ such that contributions of vertices with priority lower than the one provided to the truncation operator are disregarded. The p-truncation used in SMALL PROGRESS MEASURES, and SUCCINCT PROGRESS MEASURES is an example of such a truncation operator. The stretch operator adds a vertex' contribution (its priority) to a measure, the `prog` functions of most progress measure algorithms are it's equivalent, although Benerecetti et al. disallow the truncation and stretch operators from changing a non-$\top$ measure into a $\top$ measure and vice versa.

A given measure space $\mathcal{M}$ induces a measure-function space with is another structure containing the set of all measure functions μ which map vertices to a measure from the set of measures in the measure space $\mathcal{M}$. Additionally, the measure-function space includes an order on measure functions, which is simply a piecewise comparison of the measures mapped to each vertex.

Interestingly, Benerecetti et al. also introduce a notion of a measure of a finite path, which is essentially the measure that arises from applying the stretch operator to a measure for every vertex along the path. We used this same concept in the correctness proof for PMTL. The paper then defines a progress measure space as a measure space with properties that ensure that every contribution of an even vertex v to a measure cannot be negated by applying the truncation operator with v's priority, or any lower priority.

Finally, a progress measure is defined as a measure function which adheres to the properties that we expect from all progress measures. Namely, even vertices need to have a measure that is greater or equal to all the measures of their successors' measures lifted/stretched with the vertex' priority, and odd vertices need to have a measure that is greater or equal than at least one of their successors' measures lifted/stretched with the vertex' priority. In essence, the even player cannot increase the measures further, and the odd player is not forced to do so. So far, all of this is just an alternative formulation of the existing, arguably less rigidly defined, notion of a progress measure.

Benerecetti et al. use a structure similar to the tangle for their QUASI-DOMINION PROGRESS MEASURES. The paper makes a distinction between a quasi-dominion and a *weak* quasi-dominion but in practice the weak quasi-dominion is what is used.

Definition 8.2 (Quasi-dominion) An α-quasi-dominion is subgraph U for which a player α has a strategy such that for all $\bar{\alpha}$ strategies the highest priority in every path in U is of α's parity. ▫

Definition 8.3 (Weak quasi-dominion) A weak α-quasi-dominion is a subgraph U for which a player α has a strategy such that for all $\bar{\alpha}$ strategies the highest priority in every *infinite* path (also known as a play) is of α's parity. ▫

Benerecetti et al. aim to allow a measure to embed information about the game's quasi-dominions. They do this introducing a regress measure which enforces the dual conditions of the normal progress measure. This means that every contribution of an odd vertex v, must cause the measure to become smaller after applying the truncation operator with v's priority, or any lower priority. Even vertices need to have a measure that is smaller or equal to at least one of the measures of their successors' measures lifted/stretched with the vertex' priority, and odd vertices need to have a measure that is smaller or equal than all of their successors' measures lifted/stretched with the vertex' priority. In essence, the even player must have a successor that it cannot use to increase its measure, and the odd player must have no successor that allows it to lower its measure.

A regress measure guarantees that the set of non-$\bot$ non-$\top$ vertices form a weak quasi-dominion for even. If these measures are further restricted to say that the set of vertices mapped to $\top$ is a dominion for even and every measure mapped to a vertex v witnesses some finite simple path originating in v, then such a measure guarantees that the set of non-$\bot$ vertices is a quasi-dominion therefore such a measure function is a quasi-dominion measure.

Finally, the paper describes a concrete measure space which obeys the properties of a progress measure and quasi-dominion measure space. Benerecetti et al. also implement their algorithm within Oink and compare its performance with the other algorithms implemented there, showing that it is faster than most algorithms in most of the parity game types tested. A cursory comparison between QDPM and PMTL shows that QDPM is much faster in all cases, however it does not compute the correct strategies. Whether this is a bug in the code that was provided or an oversight in the algorithm itself is unclear.

Comparing QDPM and PMTL we can see they share some similar concepts. Both take a larger set of vertices which are updated at the same time. However, PMTL keeps track of its tangles (which are strongly-connected quasi-dominions) separately, allowing it to be used with many different progress measures. In contrast, QDPM encodes the quasi-dominion in the measures themselves. While we can clearly see from the empirical test that QDPM enjoys a significant speed advantage over PMTL it has a worst-case time complexity that is exponential, whereas PMTL can be quasi-polynomial. Benerecetti et al. mention that their approach may lead to a quasi-polynomial algorithm based on the SUCCINCT PROGRESS MEASURES. However, given the more complex nature of QDPM it is not immediately clear how it can be integrated with succinct progress measures.

Summary

PMTL differs greatly from Jurdziński et al.'s ATTRACTOR DECOMPOSITION LIFTING. While both offer quasi-polynomial worst-case time complexity, ATTRACTOR DECOMPOSITION LIFTING is value-iteration implementation of an attractor-based algorithm. Whereas PMTL uses an existing progress measure and accelerates the value-iteration framework with attractors.

PMTL and QDPM are a lot more similar since they both lift larger sets of vertices instead of only lifting neighbours of vertices. However, QDPM needs the measures that it uses to conform to a more strict set of requirements since it encodes the quasi-dominion in the measures, whereas PMTL keeps track of its tangles separately.

Chapter 9

Conclusions and Future Work

9.1 Conclusion

In chapter 3 we determined the requirements on the progression function `prog` of a progress measure such that it is compatible with our algorithm. Additionally, we showed that the SMALL PROGRESS MEASURES and ORDERED PROGRESS MEASURES fulfil these requirements.

In chapter 4 we discussed the difficulty of implementing the ORDERED PROGRESS MEASURES' antagonistic update function. We concluded that procedure described by Fearnley et al.[9] for implementing the antagonistic update did not yield a monotonic function. Therefore, we proposed a modification to the raw update `ru` and update `up` functions such that Fearnley et al.'s implementation of the antagonistic update does result in a monotonic function allowing us to use the ORDERED PROGRESS MEASURES with our algorithm. Finally, we speculated that a simple check making sure that the measure of a vertex is never lowered suffices to make the ORDERED PROGRESS MEASURES behave correctly within its original value iteration framework. This simpler solution is not suitable for use with our algorithm.

In chapter 5 we presented and proved the correctness of an algorithm that accelerates progress measure based algorithms using (tangle) attractors. Additionally, we showed that there are multiple variants of the algorithm which still yield correct results.

In chapter 6 we determined that the algorithm's time and space complexity is determined almost entirely by the choice of progress measure. Specifically, if a quasi-polynomial progress measure is used then PMTL is also quasi-polynomial, and if a polynomial progress measure were to exist then PMTL would become polynomial too. The space complexity was equivalent to running TANGLE LEARNING and the value-iteration progress measure algorithm concurrently.

In chapter 7 we found that for all games we tested that the PMTL is faster than the original progress measure-based algorithms, confirming that the use of attractors can accelerate progress measure-based algorithms. Additionally, we found that also detecting tangles and using tangle attractors yielded an even more significant acceleration. While competitive on small games and on certain worst case examples PMTL was not able to outperform existing attractor-based algorithms such as TANGLE LEARNING and RECURSIVE TANGLE LEARNING. Aside from some specific types of games the 'no-up' variant of PMTL was the fastest variant. Additionally, we saw that when PMTL used the SMALL PROGRESS MEASURES it was able to solve small parity games faster than when it used the ORDERED PROGRESS MEASURES. On larger games we saw the opposite PMTL with OPM significantly outperformed PMTL with SPM. This matches our initial hypothesis that SPM would perform well on small games due to its simpler comparison function and that OPM would perform well on large games due to its quasi-polynomial time complexity.

Finally, in chapter 8 we compared our algorithm to other attempts to bridge the gap between the two families of algorithms. We found that while there are some similarities, especially with Benerecetti et al.[1]'s QUASI-DOMINION PROGRESS MEASURES, there are significant enough differences that this algorithm is worth investigating further.

In conclusion, we showed that TANGLE LEARNING (and attractor-based techniques in general) can be used to accelerate a progress measure-based algorithm. This resulted in an algorithm for solving parity games which has the same time complexity class as whichever progress measure-based algorithm it is based upon while running in orders of magnitude less time than those algorithms.

9.2 Future Work

During this research we found several potential improvements and changes to the PROGRESS MEASURES AND TANGLE LEARNING algorithm that warrant further investigation.

- Testing PMTL with other progress measures. For this thesis we focused on two progress measures (SMALL PROGRESS MEASURES and ORDERED PROGRESS MEASURES) to narrow the scope of the research. However, it would be interesting to see how PMTL performs with other measures such as: SUCCINCT PROGRESS MEASURES[12], Dell'Erba and Schewe[3]'s improvement of ORDERED PROGRESS MEASURES, or even the measure from the ATTRACTOR-DECOMPOSITION LIFTING algorithm discussed in chapter 8.
- Applying TANGLE LEARNING WITH JUSTIFICATIONS[14]. Lapauw, Bruynooghe and Denecker were able to improve TANGLE LEARNING by storing extra information. This extra information allows unchanged regions to be skipped and prevents already attracted vertices from being re-attracted. Since PMTL tends to repeat similar attractor decompositions as the measures slowly increase this may yield a significant improvement.
- Experiment with different decomposition orders. The version of PMTL which we investigated in this research orders the vertices from highest to lowest measures and then from highest to lowest priority. We briefly performed some tests with different priority orders, but found no significant differences. However, it might still be worth investigating further what impact this ordering has.
- Testing out PMTL variants with different preparation stages. Since plain

TANGLE LEARNING is often faster than PMTL for random games there may be merit in running several iterations of the attractor decomposition without updating measures. This is equivalent to 'one-sided' TANGLE LEARNING. By doing this we may be able to find tangles and dominions quickly, after which we need fewer iterations where the measures are updated.

- Find a worst-case example. For many other parity game solving algorithms a construction is known for games which show the algorithms worst-case time complexity. It should be possible to find such a construction for PMTL, however this is non-trivial because of the hybrid nature of PMTL. The worst-case for tangle learning and the worst-case for the progress measure that is used might not be the worst-case for the combination of the two

Bibliography

[1] Massimo Benerecetti et al. 'From Quasi-Dominions to Progress Measures'. In: (2020). DOI: 10.48550/ARXIV.2008.04232. arXiv: 2008.04232.

[2] Cristian S Calude et al. 'Deciding Parity Games in Quasipolynomial Time'. In: *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing* 12 (2017), pp. 19–23. DOI: 10.1145/3055399.

[3] Daniele Dell'Erba and Sven Schewe. 'Smaller Progress Measures and Separating Automata for Parity Games'. In: *Frontiers in Computer Science, 4, 2022* 4 (2nd May 2022). DOI: 10.3389/fcomp.2022.936903. arXiv: 2205.00744 [cs.DS].

[4] Tom van Dijk. 'A Parity Game Tale of Two Counters'. In: *EPTCS 305, 2019, pp. 107-122* 305 (26th July 2018), pp. 107–122. DOI: 10.4204/eptcs.305.8. arXiv: 1807.10210 [cs.LO].

[5] Tom van Dijk. 'Attracting Tangles to Solve Parity Games'. In: *CoRR* abs/1804.01023 (2018). arXiv: 1804.01023.

[6] Tom van Dijk. 'Oink: An implementation and evaluation of modern parity game solvers'. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10805 LNCS (2018), pp. 291–308. ISSN: 1611-3349. DOI: 10.1007/978-3-319-89960-2_16/FIGURES/1.

[7] Tom van Dijk. 'Recursive attractor decomposition for parity games'. Unpublished.

[8] Ding Xian Fei. *qpt-parity (implementation of Ordered Progress Measures)*. GitHub, 2017. URL: https://github.com/dingxiangfei2009/qpt-parity.

[9] John Fearnley et al. 'An Ordered Approach to Solving Parity Games in Quasi Polynomial Time and Quasi Linear Space'. In: (2017). DOI: 10.48550/ARXIV.1703.01296. arXiv: 1703.01296.

[10] M.W. Gazda. 'Fixpoint logic, games, and relations of consequence'. English. Proefschrift. Phd Thesis 1 (Research TU/e / Graduation TU/e). Mathematics and Computer Science, Mar. 2016. ISBN: 978-94-028-0041-8.

[11] Marcin Jurdziński. 'Small Progress Measures for Solving Parity Games'. In: *STACS 2000*. Ed. by Horst Reichel and Sophie Tison. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 290–301. ISBN: 978-3-540-46541-6.

[12] Marcin Jurdziński and Ranko Lazić. 'Succinct progress measures for solving parity games'. In: *Proceedings - Symposium on Logic in Computer Science* (Aug. 2017). ISSN: 1043-6871. DOI: 10.1109/LICS.2017.8005092.

[13] Marcin Jurdziński et al. ‘A symmetric attractor-decomposition lifting algorithm for parity games’. In: *CoRR* abs/2010.08288 (2020). arXiv: 2010.08288.

[14] Ruben Lapauw, Maurice Bruynooghe and Marc Denecker. ‘Improving Parity Game Solvers with Justifications’. In: *Lecture Notes in Computer Science*. Springer International Publishing, 2020, pp. 449–470. DOI: 10.1007/978-3-030-39322-9_21.

[15] Robert McNaughton. ‘Infinite games played on finite graphs’. In: *Annals of Pure and Applied Logic* 65.2 (Dec. 1993), pp. 149–184. DOI: 10.1016/0168-0072(93)90036-d.

[16] K. S. Thejaswini, Pierre Ohlmann and Marcin Jurdziński. ‘A Technique to Speed up Symmetric Attractor-Based Algorithms for Parity Games’. In: *42nd IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2022)*. Ed. by Anuj Dawar and Venkatesan Guruswami. Vol. 250. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 44:1–44:20. ISBN: 978-3-95977-261-7. DOI: 10 . 4230 / LIPIcs . FSTTCS . 2022 . 44. URL: https://drops.dagstuhl.de/opus/volltexte/2022/17436.

[17] Wieslaw Zielonka. ‘Infinite games on finitely coloured graphs with applications to automata on infinite trees’. In: *Theoretical Computer Science* 200.1-2 (June 1998), pp. 135–183. DOI: 10.1016/s0304-3975(98)00009-7.