Theoretical Computer Science 521 (2014) 73–91

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Alternating traps in Muller and parity games

Andrey Grinshpun [a], Pakawat Phalitnonkiat [b], Sasha Rubin [c,*], Andrei Tarfulea [d]

[a] Department of Mathematics, Massachusetts Institute of Technology, Cambridge, MA, United States
[b] Department of Mathematics, Cornell University, Ithaca, NY, United States
[c] IST Austria and TU Vienna, Austria
[d] Department of Mathematics, Princeton University, Princeton, NJ, United States

ARTICLE INFO

Article history:
Received 24 November 2010
Received in revised form 23 October 2013
Accepted 26 November 2013
Communicated by A. Fraenkel

Keywords:
Parity games
Muller games

ABSTRACT

Muller games are played by two players moving a token along a graph; the winner is determined by the set of vertices that occur infinitely often. The central algorithmic problem is to compute the winning regions for the players. Different classes and representations of Muller games lead to problems of varying computational complexity. One such class are parity games; these are of particular significance in computational complexity, as they remain one of the few combinatorial problems known to be in NP ∩ co-NP but not known to be in P. We show that winning regions for a Muller game can be determined from the alternating structure of its traps. To every Muller game we then associate a natural number that we call its *trap depth*; this parameter measures how complicated the trap structure is. We present algorithms for parity games that run in polynomial time for graphs of bounded trap depth, and in general run in time exponential in the trap depth.

1. Introduction

A Muller game [13,7] is played on a finite directed graph in which every vertex is given a label, either red or blue. There is a token on an initial vertex and two players, call them Red and Blue, move the token along edges; it is Red's move if the token is on a red vertex, and otherwise it is Blue's move. To determine the winner, a Muller game also contains a collection $\mathcal{R}$ of sets of vertices. One assumes that there are no dead ends and so the play is an infinite walk. At each turn one records the vertex under the token. The winner is determined by the set S of vertices that occur infinitely often; Red wins if S is in $\mathcal{R}$, and otherwise Blue wins.

Every two-player perfect-information game with Borel winning condition is determined [12]: one of the players has a winning strategy. In particular, every Muller game is determined: either Red or Blue has a winning strategy. To *solve a Muller game* is to determine for every vertex which player has a winning strategy when play starts from the given vertex. This set of vertices is called that player's *winning region*.

One application of these games is to solve Church's synthesis problem: construct a finite-state procedure that transforms any input sequence letter by letter into an output sequence such that the pair of sequences satisfies a given specification. The modern solution to this problem goes through Muller games [17].

Characterization of Muller games. The first part of this paper (Section 3.1) characterizes the winning region of a Muller game G in terms of a two-player reachability game. The length of this reachability game is a measure of the alternating structure of the traps in G; we call it the *trap-depth* of G. We briefly explain these concepts.

* Corresponding author.
E-mail addresses: agrinshp@mit.edu (A. Grinshpun), pp287@cornell.edu (P. Phalitnonkiat), sasha.rubin@gmail.com (S. Rubin), tarfulea@princeton.edu (A. Tarfulea).

0304-3975/$ – see front matter © 2013 Published by Elsevier B.V.
http://dx.doi.org/10.1016/j.tcs.2013.11.032

Muller games admit natural substructures, *Attractors* and *Traps*. The Red-attractor [18] of a subset X of vertices is the set of vertices from which Red can force the token into X; this may be computed in linear time. A Red-trap [18] is a subset Y of vertices in which Blue may keep the token within Y indefinitely (no matter what Red does); i.e. if the token is in Y, Blue may choose to trap Red in the set Y. It should be evident that the complement of a Red-attractor is a Red-trap. Of course, all notions here (and elsewhere) defined for Red may be symmetrically defined for Blue. Thus we talk of Blue-attractors and Blue-traps.

Now, consider the following game played on the same arena as a Muller game G. The *trap-depth game on G in which Red goes first* (Definition 3.2) proceeds as follows (the traps discussed in the following are all nonempty): Red picks a Blue-trap $X_1 \subseteq V$ (here V are the vertices of the Muller game G) which is winning for Red (i.e. $X_1 \in \mathcal{R}$). Then Blue picks a Red-trap Y_1 in the smaller game induced by X_1, where Y_1 is winning for Blue (i.e. $Y_1 \notin \mathcal{R}$). Then Red picks a Blue-trap X_2 in the game induced by Y_1 such that X_2 is winning for Red. Red and Blue continue like this, alternately choosing traps. The first player that cannot move (i.e., that cannot find an appropriate nonempty trap) loses. As shown in Theorem 3.4,

> Red has a nonempty winning region in the Muller game if and only if Red has a winning strategy in the trap-depth game in which Red goes first.

And if Red has a winning strategy in this trap-depth game, the first move of any winning strategy, X_1, contains only vertices in Red's winning region of the original Muller game.

Application to parity games. The second part of the paper (Section 4) is algorithmic and applies the characterization of winning regions to a particular class of Muller games, parity games.

A parity game [4] is played on a directed graph with vertices labeled by integers called *priorities*. This game is played between two players, Even and Odd, who move a token along edges. A vertex is called *even* if its priority is even, otherwise it is called *odd*. Even moves when the token is on an even vertex, and Odd moves when the token is on an odd vertex. Play starts from a specific vertex; we assume there are no dead ends in the graph and so a play is an infinite walk. Even wins a play if the largest priority occurring infinitely often is even, otherwise Odd wins the play.

It is evident that parity games may be expressed as Muller games: the set $\mathcal{R}$ consists of all subsets X of vertices in which the largest priority of vertices in X is even.

Parity games are intertwined with a logical problem: the model checking problem for Modal μ-calculus formulas is log-space equivalent to solving parity games [7]. In [15] we see how this work can be applied for software verification. Complexity-wise, the problem is known to be in NP ∩ co-NP [5], and even UP ∩ co-UP [9]: one of the few combinatorial problems in that category that is not known to be in P. A randomized algorithm was presented in [3] that runs subexponentially (in the size of the input game) in the worst case. Several prior results obtained polynomial-time algorithms for parity games with fixed bounds on certain structural qualities, such as DAG-width; see for instance [1], [2], [8], and [14]. Our approach is in the same vein, with a novel structural quality given by the trap-depth game.

The algorithmically-minded reader may observe a potential drawback with reinterpreting games as a game of alternating traps. The number of traps in a game can grow exponentially with the size of the game (just take a graph with only self-loops), and what's worse is that we are looking at chains of alternating traps. Nonetheless, we apply the characterization to parity games: say that a graph has *Even trap-depth at most k* if Even can guarantee that, in the trap-depth game in which Even goes first, the game ends in a win for Even within k rounds. Then, despite the previous observation, we present an algorithm $\mathrm{TDA}(G, \sigma, k)$ (here G is a parity game, σ is a player, and k an integer) that runs in time $|G|^{O(k)}$ and, as shown in Theorem 4.1,

> returns the largest (possibly empty) set starting with which σ can guarantee a win in at most k moves in the trap-depth game on G.

Note that the definition of trap depth may be applied to Muller games as well, though we do not have an algorithmic application; one might hope that there are particularly efficient algorithms for finding winning vertices in Muller games of small trap depth.

Let's put this all together. Say that a parity game has *trap-depth at most k* if either it has Even trap-depth at most k or Odd trap-depth at most k. In Fig. 1 we exhibit, for every integer k, a parity game with $O(k)$ vertices and edges that has trap-depth exactly k. By the end of the paper we will have algorithmically solved the following problems:

(1) decide if a given parity game G has trap-depth at most k.
(2) find a nonempty subset of one of the player's winning region assuming the game has trap depth at most k.

Moreover, these problems can be solved in time $O(mn^{2k-1})$ where n is the number of vertices and m the number of edges of a parity game G.

2. Muller games and parity games

A *Muller game* $G = (V, V_{\mathrm{red}}, E, \mathcal{R})$ satisfies the following conditions: (V, E) is a directed graph in which every vertex has an outgoing edge, V is partitioned into *red vertices* V_{red} and *blue vertices* $V_{\mathrm{blue}} := V \setminus V_{\mathrm{red}}$, and $\mathcal{R} \subset 2^V$ is a collection

of subsets of V. The Muller game is played between two players, Red and Blue. Red will move when the token is on a red vertex, and otherwise Blue will move. Starting from some vertex v_0, Blue's and Red's moves result in an infinite sequence of vertices, called a *play*, $P = (v_0, v_1, v_2, \ldots)$ where $(v_i, v_{i+1}) \in E$. Taking $\inf(P)$ to be the set of vertices that occur infinitely often in the play, i.e. $v \in \inf(P)$ if and only if there are infinitely many i so that $v_i = v$, we say Red *wins the play* if $\inf(P) \in \mathcal{R}$, and otherwise Blue wins the play.

Take $\sigma \in \{\text{Red}, \text{Blue}\}$ (we write $\overline{\sigma}$ for the other player, so if σ is Red then $\overline{\sigma}$ is Blue, and vice versa). A σ*-Strategy* is an instruction giving Player σ's next move given the current token position and play history. Formally, it is a function whose domain is the set of finite strings of vertices $\{v_0 v_1 \cdots v_k \colon (v_i, v_{i+1}) \in E\}$ and whose range is $N(v_k) := \{v \in V \colon (v_k, v) \in E\}$, the neighborhood of v. A σ-strategy is *winning from vertex* v_0 if, for all plays starting at v_0 and for which that strategy is followed whenever it is σ's turn, the resulting play is winning for σ. Finally, a σ-strategy is *memoryless* if it gives σ's move while taking into consideration only the current token position; i.e., it is a strategy in which the value on $v_0 \ldots v_k$ depends only on v_k. A given memoryless σ-strategy π in a Muller game G induces a subgame H in which we restrict the outgoing edges of any σ vertex to the edge defined by π. It is worth noting that if both players fix a strategy for the game, then the resulting play is completely determined by the starting vertex, since given the current history we can determine which vertex is visited next.

Muller games are determined (since they are Borel we can apply [12], although for the special case of regular games see [7]): starting from any vertex, there is a player that has a winning strategy. Determinacy partitions V into the respective *winning regions* $W_{G,\text{Red}}$ and $W_{G,\text{Blue}}$ (where $v \in W_{G,\sigma}$ if and only if σ has a winning strategy starting from v in G). In contexts where the meaning is clear, we will use W_σ for $W_{G,\sigma}$. It follows easily that for a player σ there is a single strategy that wins starting from any vertex in $W_{G,\sigma}$; such a strategy is called a winning strategy. We now introduce various important substructures of Muller games that capture some of the essential concepts of reachability and restriction (see Chapter 2.5 of [7]).

Definition 2.1. A σ*-Trap* is a collection of vertices $X \subseteq G$ where:

$$\forall x \in X \cap V_\sigma \text{ we have } N(x) \subseteq X$$

and

$$\forall x \in X \cap V_{\overline{\sigma}},\ \exists y \in X \text{ such that } (x, y) \in E.$$

No σ vertex in X has an outgoing edge leaving the trap, and every $\overline{\sigma}$ vertex in X has at least one outgoing edge that stays in the trap. Consequently, if the token ever enters X, $\overline{\sigma}$ has a strategy through which the token will never leave X, no matter what σ does. It is apparent that W_σ is a $\overline{\sigma}$-trap.

Notation. We write $\mathit{Traps}_\sigma(G)$ to denote the set of nonempty σ-traps in G.

Definition 2.2. A σ*-Attractor of a set of vertices* Y is the set of vertices starting from which σ has a strategy that guarantees Y will be reached (after finitely many, possibly 0, steps).

We denote the attractor of a set X in a graph G with respect to a player σ by $\text{Attr}(G, X, \sigma)$, and it is worth noting that the attractor of a set may be computed in time linear in the size of the graph; the algorithm for doing so is presented below [18].

Algorithm 1 $\text{Attr}(G = (V, E, p), X, \sigma)$.

```
C_prev := ∅
C_cur := X
while C_cur ≠ C_prev do
    C_prev := C_cur
    C_cur := C_prev ∪ {v ∈ V_σ : N(v) ∩ C_prev ≠ ∅} ∪ {v ∈ V_σ̄ : N(v) ⊆ C_prev}
end while
return C_cur
```

On each iteration, the σ vertices that have an edge into the part of the attractor that has already been computed are added, and the $\overline{\sigma}$ vertices that have only edges into that part are added. We briefly argue correctness: by induction on the number of iterations, we see that starting anywhere in the computed set, σ has a strategy to reach X, and starting outside the computed set it is easy to see that $\overline{\sigma}$ has a strategy to avoid the computed set indefinitely (every $\overline{\sigma}$ vertex outside the set has some edge that does not enter the set, and every σ vertex outside of it has no edge that enters it), so this does compute the attractor.

Definition 2.3. The *Induced Subgame of G by X* is the Muller game using the vertices $V \cap X$ and the edges $E \cap X^2$; we sometimes refer to this as "G restricted to X" and use the notation $G[X]$.

Naturally, $G[X]$ should have no dead ends if it is to be a Muller game. It is apparent that G restricted to a trap X is a Muller game. When X is a trap we use *subtraps* to mean the traps of $G[X]$.

Lemma 2.4. *(See [18].) If $X \subseteq W_{G,\sigma}$ then, taking $U = \text{Attr}(G, X, \sigma)$, we have $W_{G,\sigma} = U \cup W_{G[V\setminus U],\sigma}$.*

In other words, if we know that σ can win from a set, then we can remove that set's attractor from the graph and just find the winning region for σ in the smaller graph.

Lemma 2.5. *(See [18].) If $X \subseteq W_{G,\sigma}$ and X is a σ-trap, then $W_{G[X],\sigma} = X$.*

Intuitively, this holds because in the induced game $G[X]$ player σ can continue to use the same winning strategy that σ had in G.

We end the section with the statements of some technical lemmas that will be useful. Their proofs are routine.

Lemma 2.6. *(See [18].) If X is a σ-trap in G and Y is a σ-trap in $G[X]$, then Y is a σ-trap in G.*

The next lemma states that if we take the σ-attractor of some set Y and are interested in how it intersects with some σ-trap X, then the intersection is contained in the attractor of $X \cap Y$ in the game restricted to X.

Lemma 2.7. *(See [18].) If X is a σ-trap in G, Y is a set of vertices, and $S = \text{Attr}(G, Y, \sigma)$, then $X \cap S \subseteq \text{Attr}(G[X], X \cap Y, \sigma)$.*

Lemma 2.8. *If X is a σ-trap in G and Y is a $\overline{\sigma}$-trap in G, then $X \cap Y$ is a $\overline{\sigma}$-trap in $G[X]$.*

2.1. Parity games

A *Parity game* $G = (V, E, \rho)$ satisfies the following conditions: (V, E) is a directed graph in which every vertex has an outgoing edge, $v_0 \in V$ denotes a starting vertex, and $p : V \to \mathbb{Z}$ is a function assigning priorities to the vertices. The parity game is played between two players, Even and Odd, where each player moves the token along a directed edge of G whenever the token is on a vertex of the corresponding parity. We say a vertex is even if it has even priority and odd if it has odd priority. Even's and Odd's moves result in an infinite play: $P = (v_0, v_1, v_2, \ldots)$ where $(v_i, v_{i+1}) \in E$. Even wins the play if $\limsup_{i\in\mathbb{N}} p(v_i)$ is even and Odd wins otherwise: i.e., the largest priority that occurs infinitely often determines the winner of the play.

Note that, given a Parity game, we may define the corresponding Muller game by placing v in V_{red} if and only if $p(v)$ is even. Then $S \subseteq V$ has $S \in \mathcal{R}$ if and only if $\max(S)$ is even, and otherwise $\max(S)$ is odd and $S \notin \mathcal{R}$. The corresponding Muller game is then $(V, V_{\text{red}}, E, \mathcal{R})$. Note that a play is winning in the Muller game if and only if it is winning in the parity game.

Not only are Parity games determined, they are *Memorylessly Determined* [4]: for every vertex $v \in V$, exactly one of the two players has a memoryless strategy that guarantees a win starting from v. Moreover, for each player there is a single memoryless strategy which, if followed, will result in a winning play starting from any vertex in that player's winning region; this is a called a memoryless winning strategy. Note that Muller games are not memorylessly determined; they may require a strategy that uses some of the play history.

3. The trap-depth game

3.1. Main theorem

As mentioned in the introduction, our main result relies on a characterization stemming from chains of alternating subtraps. Each subtrap represents the decision of the corresponding player to further restrict the token's movement. This goes on until the final restriction leaves one player incapable of preventing a winning play for their opponent. We now formalize this idea. We begin by defining a set of statements related to chains of alternating traps.

Define R_σ to be $\mathcal{R}$ if σ is Red and $2^V \setminus \mathcal{R}$ otherwise. The statement $S \in R_\sigma$ says that if the set of vertices that occurs infinitely often is S, then player σ wins. Recall that $\mathit{Traps}_\sigma(G)$ is the set of *nonempty* σ-traps in G. Our boolean statements $\Delta_\sigma(G, k)$ are defined recursively and have three parameters: the player σ, the game G, and the iteration (or depth) number k.

Definition 3.1. For player σ, game G, and integer k, the value of $\Delta_\sigma(G, 0)$ is false. For $k > 0$, the value of $\Delta_\sigma(G, k)$ is true if and only if there exists $X \in \mathit{Traps}_{\overline{\sigma}}(G)$ such that

- $X \in R_\sigma$, and
- $\forall Y \in \mathit{Traps}_\sigma(G[X])$ we have $Y \in R_\sigma$ or $\Delta_\sigma(G[Y], k-1)$.

Each statement $\Delta_\sigma(G,k)$ asserts that σ can restrict the token's movement via a trap X in such a way that if every vertex in the trap occurs infinitely often, player σ wins, i.e. $X \in R_\sigma$ (intuitively then, player $\overline{\sigma}$ must choose to further restrict play) and, no matter how $\overline{\sigma}$ further restricts the token's movement via a subtrap Y, either still $Y \in R_\sigma$ or we have that $\Delta_\sigma(G[Y], k-1)$ is true. So, in particular, $\Delta_{\text{Red}}(G,1)$ states that there is a Blue-trap X in G with $X \in \mathcal{R}$ such that every Red-subtrap Y has $Y \in \mathcal{R}$.

The above definitions make it easy to see that the statements make references to natural structures in Muller games, but they can be rather cumbersome to work with, so we present an equivalent but easier to visualize way to think about them.

Definition 3.2. Let G be a Muller game. Define the *Trap-Depth game on G in which σ goes first* as follows: in the beginning of the ith round ($i \geqslant 1$) there will be some current Muller game G_i. The game starts with $G_1 = G$. In the ith round player σ moves first by choosing a trap $X_i \in \mathit{Traps}_{\overline{\sigma}}(G_i)$ with $X_i \in R_\sigma$. Player $\overline{\sigma}$ replies by choosing a σ-trap Y_i in the subgame $G_i[Y_i]$, i.e. $Y_i \in \mathit{Traps}_\sigma(G_i[X_i])$, so that $Y_i \in R_{\overline{\sigma}}$. This completes the ith round. Define $G_{i+1} = G_i[Y_i]$. The first player that has no legal move loses.

In a Muller game, this will terminate in at most $\lceil \frac{n}{2} \rceil$ rounds, as each time a player chooses a trap, a vertex must be removed. If the Muller game is a parity game, then the condition $X \in R_\sigma$ simply states that the largest priority of a vertex in X is of parity σ. For a parity game, the number of rounds is at most $\lceil \frac{|p(V)|}{2} \rceil$, since the size of the largest vertex still in play decreases twice per round. In particular, every play in this game is finite and ends in a win for one of the players. Therefore, the game is determined (i.e. one of the players has a winning strategy).

Lemma 3.3. *The value of $\Delta_\sigma(G,k)$ is* true *if and only if σ has a strategy that ensures their opponent loses the Trap-Depth game in which σ goes first in at most k rounds (so $\overline{\sigma}$ would lose on or before the* $2k$th *move).*

This is easily verified by identifying player moves with the quantifiers in the expression for $\Delta_\sigma(G,k)$. We now state the first main result of this paper; Section 3.2 is devoted to its proof.

Theorem 3.4. *Let G be a Muller game. Then $W_{G,\sigma} \neq \emptyset$ if and only if σ has a winning strategy in the trap-depth game on G in which σ goes first. Moreover, the first move X of a winning strategy for player σ satisfies $X \subseteq W_\sigma$.*

So Player σ has a nonempty winning region in the game G if and only if σ has a winning strategy in the Trap-Depth game in which σ goes first.

Note the following simple corollary:

Corollary 3.5. *The following two statements are equivalent*:

- *Parity games can be solved in polynomial time.*
- *The player with a winning strategy in the trap-depth game described by a parity game can be determined in polynomial time.*

This theorem also motivates a new parameter for parity games (the parameter applies to Muller games as well, though we do not have an algorithmic application):

Definition 3.6. The *Trap-Depth* of a parity game G is the minimum integer k such that $\Delta_{\text{Even}}(G,k)$ is true or $\Delta_{\text{Odd}}(G,k)$ is true.

One simple class of parity games that has this property is those with a bounded number of priorities, though having bounded trap-depth is much more general than having a bounded number of priorities.

One can define the *σ-trap-depth* of G as the minimum integer k (if it exists) such that $\Delta_\sigma(G,k)$ is true; so $W_\sigma \neq \emptyset$ if and only if the σ-trap-depth of G is at most $\lceil \frac{|p(V)|}{2} \rceil$. This upper bound can be achieved, as shown by Fig. 1.

3.2. Proof of Theorem 3.4

3.2.1. Proof for memoryless strategies

We will first prove the characterization of Muller games (the first two sentences of Theorem 3.4) for games in which player σ has a memoryless strategy that wins starting from any vertex in W_σ. Intuitively, traps do not distinguish between memoried and memoryless strategies; we will formalize this intuition and this will allow us to extend the main theorem to all Muller games.

Lemma 3.7. *Let G be a nonempty Muller game with $W_{G,\sigma} = V$, that is in which σ wins starting from any vertex, and π a memoryless winning strategy for σ. Then there is a nonempty $\overline{\sigma}$-trap T in G such that $W_{G[T],\sigma} = T$, $T \in R_\sigma$, and, if π is followed, then any play starting in T will not leave T (i.e. π does not prescribe leaving T).*

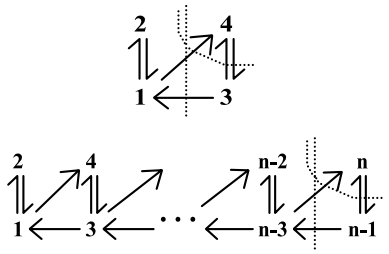

Fig. 1. Maximum trap-depth: Above: base case (G_4) with trap-depth 2; Below: G_n with n vertices (n is even); both are Even-winning from every vertex (so $\Delta_1(G_n, k)$ is never true for any k, by Theorem 3.4). The only Odd-trap is the entire graph, so this must be Even's first move in a trap-depth game. Odd could then remove the right-most top vertex, the remaining set being an Even-subtrap. Within this graph, the only remaining appropriate Odd-subtrap for Even's next move is the set formed by removing the right-most bottom vertex. We have now reduced the game to G_{n-2}. Each time we add two vertices we increase the trap-depth by 1, so the trap depth of G_n is exactly $n/2$.

Proof. Take H to be the subgame induced by π; that is, leave only one edge out of each σ vertex, the one corresponding to the strategy π. Take T to be a strongly connected component (SCC) of H such that T has no edges into any other SCC. Note that T is a $\overline{\sigma}$-trap in H, and so also in G. Since T is strongly connected and player σ only has one possible move at any vertex, $\overline{\sigma}$ has a strategy (not necessarily memoryless) such that starting from any vertex in T, if the strategy is followed, every vertex in T occurs infinitely often. Then, by the assumption that π was winning, we must have $T \in R_\sigma$. By construction, π does not prescribe leaving T. □

The following two propositions establish the theorem for Muller games in which σ has a memoryless winning strategy.

Proposition 3.8. *If $W_{G,\sigma} \neq \emptyset$ and σ has a memoryless winning strategy, then σ has a winning strategy on the trap-depth game on G in which σ goes first.*

Proof. Fix π a memoryless winning strategy for σ. We describe a strategy for σ in the trap-depth game so that for every $i \geqslant 1$ player σ has a valid move H_i satisfying that π does not prescribe leaving X_i and any potential response Y_i satisfies the invariant $W_{G[Y_i],\sigma} = Y_i$. To get the induction going we define $Y_0 := W_{G,\sigma}$ and note that $W_{G[Y_0],\sigma} = Y_0$. Note that such a strategy ensures that player σ always has a valid move and thus wins the trap-depth game.

Suppose $i \geqslant 0$ rounds have been played, and assume by induction that σ wins the Muller game starting from any vertex in Y_i. Then, by Lemma 3.7, there is some $\overline{\sigma}$-trap X_{i+1} in $G[Y_i]$ with $X_{i+1} \in R_\sigma$ such that $W_{G[X_{i+1}],\sigma} = X_{i+1}$ and π does not prescribe leaving X_{i+1}; have σ play such an X_{i+1}. Then, if player $\overline{\sigma}$ has some response Y_{i+1}, we have that Y_{i+1} is a σ-trap in $G[X_{i+1}]$ and so by Lemma 2.5 $W_{G[Y_{i+1}],\sigma} = Y_{i+1}$, as required. □

Proposition 3.9. *If $W_{G,\sigma} = \emptyset$ and player $\overline{\sigma}$ has a memoryless winning strategy, then $\overline{\sigma}$ has a strategy that wins the trap-depth game on G in which σ goes first.*

Proof. Let X be player σ's first move. Then, since X is a $\overline{\sigma}$-trap, we have $W_{G[X],\overline{\sigma}} = X \neq \emptyset$ by Lemma 2.5. Note that now we simply play the trap-depth game on $G[X]$ in which $\overline{\sigma}$ goes first and $\pi|_X$ is a memoryless winning strategy on $G[X]$, and so by the previous proposition we have that $\overline{\sigma}$ has a winning strategy. □

The previous two propositions show the desired characterization of Muller games, assuming that players have memoryless winning strategies.

3.2.2. Proof for all Muller games

While Muller games do not, in general, have memoryless strategies, a player need only use a finite amount of memory. To formalize this notion, we define a bounded-state strategy.

Definition 3.10. For any Muller game G, any positive integer N, and any function $M : V \times [N] \to [N]$, define the M-sequence with respect to any play $v_0, v_1, \ldots$ by $M_0 = 0$ and $M_i = M(v_i, M_{i-1})$.

Intuitively, in the above, $M_i \in [N]$ is the (joint) memory used by the players and M_{i+1} depends only on M_i and on the most recent move.

Definition 3.11. For any Muller game G, any positive integer N, and any function $M : V \times [N] \to [N]$, a strategy π_σ for player σ is a bounded-state M-strategy if there is some $\pi : [N] \to V$ so that if $v_0, \ldots$ is any play consistent with π_σ and $M_0, \ldots$ is the corresponding M-sequence, then π_σ depends only on M_i. I.e., there is some function F$\pi : [N] \to V$ so that for each σ vertex v_i, we have $v_{i+1} = \pi(M_i)$.

The following theorem is proved in [13]. It states that, while Muller games may not have memoryless strategies, players need only a bounded amount of memory.

Theorem 3.12. *For any Muller game G there is some positive integer N and some $M : V \times [N] \to [N]$ so that, for each player σ, there is a bounded-state M-strategy π_σ satisfying that, starting from any vertex in σ's winning region, π_σ is a winning strategy for σ.*

Given any Muller game G, take N a positive integer and $M : V \times [N] \to [N]$ as in the previous theorem. We define the memoried Muller game associated with G, call it G_M, to have vertex set $V \times [N]$ (where N depends on G as in the previous theorem). Intuitively, G_M will simulate G, but each vertex $(v, n) \in V \times [N]$ in the memoried game records the current state of the memory, with v representing the current position in G. Thus, given $(v, n), (w, m)$ vertices in the memoried game, $((v, n), (w, m))$ is an edge of the memoried game if and only if (v, w) is an edge in G and $m = M(w, n)$. Define the vertices belonging to player σ, $V_{\sigma,M}$, by $(v, n) \in V_{\sigma,M}$ if and only if $v \in V_\sigma$. Similarly, $S \subseteq V \times [N]$ is winning for Red, i.e. has $S \in \mathcal{R}_M$, if and only if the corresponding vertices are winning for Red in the original Muller game G, i.e. if and only if $\{v\colon\ \exists n, (v, n) \in S\} \in \mathcal{R}$.

Note that by the previous theorem and the construction of the memoried games, both players have memoryless winning strategies in G_M. The remainder of this section argues that the trap-structure of G_M is very similar to that of G.

Intuitively, the following lemma says that if, when playing the trap-depth game on G_M, player $\overline{\sigma}$ simply pretends it's the trap-depth game on G, then any edge out of a $\overline{\sigma}$ vertex that would have existed were the game played on G also exists in the game on G_M.

Lemma 3.13. *Assume, in a trap-depth game on G_M, whenever the current set of vertices is X_M and it is $\overline{\sigma}$'s turn to move, that $\overline{\sigma}$'s move has the following form: taking $X := \{v\colon\ \exists n, (v, n) \in X_M\}$, there is some σ-trap Y in $G[X]$ so that $\overline{\sigma}$'s move is $X_M \cap (Y \times [N])$. Then, at every point in the game, if the current set of vertices is X_M, take $X := \{v\colon\ \exists n, (v, n) \in X_M\}$. For any $(v, n) \in X_M$ with $v \in V_{\overline{\sigma}}$ and for any $w \in X$ so that (v, w) is an edge of G, there is some m so that $((v, n), (w, m))$ is an edge of G_M and $(w, m) \in X_M$.*

Proof. We proceed by induction on the number of plays in the game. In the base case, the game is the whole graph and this is true by construction of G_M. Take X_M to be the current set of vertices and $X := \{v\colon\ \exists n, (v, n) \in X_M\}$.

If it is σ's turn to move, σ chooses some $\overline{\sigma}$ trap Y_M. Taking $Y := \{v\colon\ \exists n, (v, n) \in Y_M\}$, for any $(v, n) \in Y_M$ with $v \in V_{\overline{\sigma}}$ and for any $w \in X$ so that (v, w) is an edge of G, by induction there is some m so that $((v, n), (w, m))$ is an edge of G_M and $(w, m) \in X_M$. But Y_M is a $\overline{\sigma}$ trap, so since $(v, n) \in Y_M$ and (v, n) is a $\overline{\sigma}$ vertex we get $(w, m) \in Y_M$.

If it is $\overline{\sigma}$'s turn to move, $\overline{\sigma}$ chooses some σ trap Y_M of the form $X_M \cap (Y \times [N])$ where Y is a σ trap in X. Note that $Y = \{v\colon\ \exists n, (v, n) \in Y_M\}$, so this notation is consistent with previous notation. Given any $\overline{\sigma}$ vertex $(v, n) \in Y_M$ and any $w \in Y$ so that (v, w) is an edge of G, by induction there must be some m so that $((v, n), (w, m))$ is an edge of G_M and $(w, m) \in X_M$. But then $(w, m) \in Y_M$ since $w \in Y$ and $Y_M = X_M \cap (Y \times [N])$. □

Theorem 3.14. *Player σ has a winning strategy in a trap-depth game (in which either σ or $\overline{\sigma}$ goes first) on G if and only if player σ has a winning strategy in a trap-depth game on G_M (in which the same player goes first).*

Proof. Assume player σ has a winning strategy in a trap-depth game on G_M. Then player σ is to play a trap-depth game on G and we wish to show that player σ has a winning strategy; we define player σ's strategy by emulating the game on G_M. With each move, player σ will maintain a set of vertices X_M which represents the state of the emulated game on G_M. Assume the current set of vertices in the game on G is X, and σ has maintained the state X_M. We will inductively show that σ has a strategy that maintains $X = \{v\colon\ \exists n, (v, n) \in X_M\}$ and that, starting from X_M with the appropriate player moving, is winning for σ in the game G_M. In the base case, $X = V$ and $X_M = V_M$.

If it is $\overline{\sigma}$'s turn to move, $\overline{\sigma}$ will pick some σ trap $Y \in R_{\overline{\sigma}}$. Then we claim $Y_M := X_M \cap (Y \times [N])$ is a σ-trap in X_M: since Y is a σ-trap in X, it must be the case that given any σ vertex in Y_M, any neighbor it had in X_M is also in Y_M. Given a $\overline{\sigma}$ vertex (v, n) in Y_M, since Y is a σ-trap we have that v has a neighbor w in Y, and so v has a neighbor (w, m) in Y_M by the previous lemma, thus verifying that Y_M is a σ-trap. Then $Y = \{v\colon\ \exists n, (v, n) \in Y_M\}$ so $Y_M \in R_{\overline{\sigma},M}$ since $Y \in R_{\overline{\sigma}}$. Since X_M was winning for σ, we have Y_M is as well (since any move by $\overline{\sigma}$ must result in a winning position).

If it is σ's turn to move, by assumption we are in some winning position X_M. Then σ may choose some $\overline{\sigma}$ trap $Y_M \in R_{\sigma,M}$ in $G_M[X_M]$ so that Y_M is winning for σ. We claim $Y := \{v\colon\ \exists n, (v, n) \in Y_M\}$ is a $\overline{\sigma}$ trap in X. Given any σ vertex $v \in Y$, choose n with $(v, n) \in Y_M$; since Y_M is a $\overline{\sigma}$ trap in X_M, (v, n) must have some neighbor $(w, m) \in Y_M$, so (v, w) is an edge of Y. Given any $\overline{\sigma}$ vertex $v \in Y$ and any neighbor $w \in X$, we may choose n with $(v, n) \in Y_M$; we have that there is some m with $(w, m) \in X_M$ a neighbor of (v, n) by the previous lemma, but Y_M is a $\overline{\sigma}$-trap, so $(w, m) \in Y_M$ and so $w \in Y$, as desired. Note that, since $Y = \{v\colon\ \exists n, (v, n) \in Y_M\}$ and $Y_M \in R_{\sigma,M}$, we get that $Y \in R_\sigma$, so Y is a valid choice of move for player σ.

We've shown that, if σ has a winning strategy on G_M, then σ has a winning strategy on G. Symmetrically, if $\overline{\sigma}$ has a winning strategy on G_M, then $\overline{\sigma}$ has a winning strategy on G, thus proving the theorem. □

By combining the previous theorem with Propositions 3.8 and 3.9, we may remove the assumptions regarding having memoryless winning strategies:

Theorem 3.15. *If $W_{G,\sigma} \neq \emptyset$, then player σ has a winning strategy on the trap-depth game on G in which σ goes first.*

Theorem 3.16. *If $W_{G,\sigma} = \emptyset$, then player $\overline{\sigma}$ has a winning strategy on the trap-depth game on G in which σ goes first.*

Assume T is the first move $\overline{\sigma}$-trap in the trap-depth game on G where σ goes first and that σ wins starting from $G[T]$ if $\overline{\sigma}$ goes first. If $X := T \cap W_{\overline{\sigma}} \neq \emptyset$, then X is a σ-trap in $G[T]$ with $W_{G[X],\overline{\sigma}} = X$. So, by Lemma 3.7, if we consider X_M in G_M, $\overline{\sigma}$ has a viable move $Y_M \subseteq X_M$ such that $W_{G_M[Y_M],\overline{\sigma}} = Y_M$. By our previous arguments we then get that $\overline{\sigma}$ can win in the trap-depth game in which σ goes first on $G[Y]$ where $Y = \{v_1 \colon v \in Y_M\}$ is a σ-trap in $G[X]$. Then Y is a valid move for $\overline{\sigma}$ in $G[T]$, contradicting the assumption that σ can win from $G[T]$ if $\overline{\sigma}$ goes first.

Corollary 3.17. *If T is the first move of a winning σ-strategy in the Trap-Depth game where σ goes first, then $T \subseteq W_\sigma$.*

This completes the proof of Theorem 3.4.

It is interesting to understand how these nested traps will interact with modifications to the graph. The following theorem says that via one such modification not much information is lost; this is particularly useful if one wishes to run the algorithms discussed in the next section.

Theorem 3.18. *Let G be a Muller game. Assume that, in the trap-depth game on G, X is a valid first move for σ that allows σ to guarantee a win in at most k rounds and that A is a σ-trap so that $A \cap X$ is nonempty. Then there is $Y \subseteq X \cap A$ where Y is a valid first move for σ that allows σ to win in at most k rounds on the trap-depth game on $G[A]$.*

Proof. Since X is a $\overline{\sigma}$ trap in G and A is a σ trap in G, we have $X \cap A$ is a $\overline{\sigma}$ trap in $G[A]$. Furthermore, $X \cap A$ is a σ trap in $G[X]$.

If $X \cap A$ is in $R_{\overline{\sigma}}$ then $X \cap A$ is a valid move for $\overline{\sigma}$ in the trap-depth game on G after σ plays X. Therefore, σ must have a response Y that leads to a win in at most $k-1$ rounds; then Y is a $\overline{\sigma}$ trap in $X \cap A$ and therefore also in A, so it is a valid first move for σ in $G[A]$.

Otherwise, $X \cap A$ is in R_σ. We will make $X \cap A$ player σ's first move in the trap-depth game on $G[A]$. Assume $\overline{\sigma}$ has a response X' so that σ cannot win from X' in at most $k-1$ rounds. Then X' is a σ trap in $G[X \cap A]$ and therefore also in $G[X]$, so X' is a valid response for $\overline{\sigma}$ in the trap-depth game on G to the play X, contradicting the assumption. □

Finally, in preparation for the next section, we translate the above into the language of parity games.

Define the "max" of a set of vertices to be those vertices in the set with maximum priority. Recall that $Traps_\sigma(G)$ is the set of nonempty σ-traps in G. Then the condition $X \in R_\sigma$ becomes $\max(X) \subseteq V_\sigma$. For example, we may rewrite the statements Δ:

$$\Delta_\sigma(G,0) := \text{FALSE};$$

$$\Delta_\sigma(G,k+1) := \left[\exists X \in Traps_{\overline{\sigma}}(G) \text{ such that } \max(X) \subseteq V_\sigma\right] \text{ and } \left[\forall Y \in Traps_\sigma\left(G[X]\right) \text{ we have } \left(\Delta_\sigma\left(G[Y],k\right) \text{ or } \max(Y) \subseteq V_\sigma\right)\right].$$

In the definition of trap-depth game, for example, when it is player σ's turn, player σ will choose a $\overline{\sigma}$ trap whose largest priority is of parity σ.

Recall for a parity game G that $\mathrm{TDA}(G,\sigma,k)$ returns the largest set X which, as a first move for σ, allows σ to win in at most k rounds in the trap-depth game on G. Theorem 3.18 tells us that the kth trap-depth algorithm is robust in the following sense: if one determines that some vertices are winning for σ and removes their attractor from the graph, either one removes all of $\mathrm{TDA}(G,\sigma,k)$ or else one can find the rest of $\mathrm{TDA}(G,\sigma,k)$ by repeatedly running the kth trap-depth algorithm on the remaining set.

4. Trap-Depth Algorithms for parity games

In this section of the paper, all discussions are regarding parity games. We present a collection of algorithms that return subsets of the vertices of a parity game, culminating in the Trap-Depth Algorithm (TDA). We will have two versions of TDA (which take different inputs). We will discuss the first of the algorithms later. The characterization of the second algorithm, $\mathrm{TDA}(G,\sigma,k)$, follows easily from the first, and it takes as its inputs a parity game G, a player σ, and an integer k. We will ultimately show the following characterization of the second TDA algorithm:

Theorem 4.1. $\mathrm{TDA}(G,\sigma,k)$ *returns the largest (possibly empty) set S so that if σ uses S as a first move, σ can guarantee a win in at most k moves in the trap-depth game on G.*

Note that, by Theorem 3.4, this implies in particular that $\mathrm{TDA}(G,\sigma,k) \subseteq W_\sigma$.

4.1. Büchi games

Due to the complexity of the TDA, we will first introduce some simpler algorithms. Understanding the simpler algorithms will help significantly in understanding the TDA. The overall structure of the TDA resembles that of a classical algorithm for solving Büchi games, which we present here. A *simple Büchi game*[1] is a parity game in which all of the priorities are either 0 or 1. In the discussions that follow, we will simply say Büchi games; note that the discussions do apply to Büchi games as well as simple Büchi games. Odd wins a Büchi game if and only if Odd has a strategy that reaches vertices of priority 1 infinitely many times. The algorithm takes as input a Büchi game G and returns the winning region for Odd.

Algorithm 2 Büchi$(G = (V, E, p))$.

```
1: σ := Odd
2: T_prev := ∅
3: T_cur := V_σ
4: while T_cur ≠ T_prev do
5:     T_prev := T_cur
6:     C := Attr(G, T_prev, σ)
7:     T_cur := T_prev \ {v ∈ V_σ : N(v) ∩ C = ∅}
8: end while
9: return C
```

The classical algorithm for Büchi games begins each iteration of its while-loop with a set of target vertices $T_{\text{prev}} \subseteq V_{\text{Odd}}$. It then computes the attractor of T_{prev}. This provides the largest set of vertices from which Odd has a strategy for reaching the set T_{prev}. The attractor by itself, however, does not provide any strategy for σ after the token reaches T_{prev}. To remedy this, the algorithm then tests each vertex in T_{prev} to check if it has the option to continue this strategy by returning the token back to the attractor of T_{prev}. If not, then that vertex is removed from being a target. This process repeats until the set T_{prev} stabilizes.

We will first observe that the algorithm outputs a subset of Odd's winning region. To see this, take C to be the output of the algorithm and T to be the final set of target vertices (so that $C = \text{Attr}(G, T, \text{Odd})$). Consider the following strategy for Odd: from any vertex of T, Odd chooses to enter C (this is possible by the termination condition of the algorithm). From any vertex of $C \setminus T$, Odd follows a strategy to reach T. This guarantees that the vertices of T are visited infinitely often; since these vertices have priority 1, this is a winning strategy for Odd.

Conversely, Even has a winning strategy from any vertex not in C. To see this, let $T_0, T_1, \ldots$ be the sequence of values of T_{cur} at the beginning of the while-loop in the execution of Büchi(G) (with the final value repeated). Take $C_i = \text{Attr}(G, T_i, \text{Odd})$. Note that T_i, and therefore C_i, is a decreasing sequence. We claim that any odd vertex in C_i must be in T_i. If some Odd vertex w were in C_i but not in T_i, then there must be some edge from w into C_i. However, w must have been removed from T_j for some $j < i$, which means there is no edge from w into C_j. However, the sequence of C_j is decreasing, a contradiction.

We now proceed by induction to show that Even has a winning strategy from any vertex not in C_i. Note that $T_0 = V_{\text{Odd}}$. Therefore, any vertex not in C_0 is not in the attractor of V_{Odd}, so from any such vertex Even has a strategy that never visits any vertex of priority 1. Take for an inductive hypothesis that, for some i, any vertex not in C_i is winning for Even. Then $T_{i+1} = T_i \setminus \{v \in V_{\text{Odd}} \colon N(v) \cap C_i = \emptyset\}$. Assume for contradiction that there is some vertex v that is not in C_{i+1} so that v is winning for Odd. Then v must be in $C_i \setminus C_{i+1}$, as otherwise v is winning for Even by the inductive hypothesis. Since v is not in C_{i+1}, Even has a strategy starting from v that avoids T_{i+1} indefinitely. Then consider Even playing the following strategy: as long as the token remains in C_i, Even plays any strategy that avoids entering T_{i+1}. If the token ever leaves C_i, then Even has a winning strategy and uses it. Since v is winning for Odd, Odd must have some winning strategy. Consider the play induced by Even playing the aforementioned strategy and Odd playing a winning strategy. This play cannot leave C_i, as otherwise Even wins. However, some vertex $w \in C_i$ of priority 1 must be visited. Since Even avoids T_{i+1} indefinitely, we must have that w is not in T_{i+1}. Since all Odd vertices in C_i are in T_i, we therefore have that w is in $T_i \setminus T_{i+1}$. However, by definition of T_{i+1}, this means that there are no edges from w into C_i, so the next vertex in the play is not in C_i, a contradiction. This completes the proof of the correctness of the classical Büchi games algorithm.

4.2. $k = 1$

The algorithm for solving trap-depth 1 parity games closely resembles that for Büchi games. The main difference is that, rather than taking the attractor of the target set, we take a "safe" version of the attractor. This takes a parameter λ; the

[1] This is actually a simpler problem than Büchi games. The way we defined parity games, the player that chooses where to move the token from a given vertex v is just based on the parity of $p(v)$. If we had instead defined parity games in such a way that the player who chooses where to move the token from v may depend on v itself (rather than just on $p(v)$), then parity games in which all priorities are either 0 or 1 would be Büchi games. Under our simplified definition of Büchi games, they may be solved by simply determining if the vertices labeled with 0 contain a cycle. However, the algorithm for solving Büchi games is instructive, even when applied to the simplified Büchi games, so we present it here.

λ-safe attractor in G of a set X for player σ is the set of vertices from which σ has a strategy that guarantees X will be reached and that, in the process, no vertices (excluding those in X) of priority at least λ are visited.

Algorithm 3 SafeAttr$(G = (V, E, p), \lambda, X, \sigma)$.

```
C_prev := ∅
C_cur := X
while C_cur ≠ C_prev do
    C_prev := C_cur
    C_cur := C_prev ∪ {v ∈ V_σ : p(v) < λ ∧ N(v) ∩ C_prev ≠ ∅} ∪ {v ∈ V_σ̄ : p(v) < λ ∧ N(v) ⊆ C_prev}
end while
return C_cur
```

At each iteration of the "while" loop, the set C_{cur} (initially X) is enlarged by adding any vertices (of priority less than λ) in V_σ or in $V_{\overline{\sigma}}$ that, respectively, have an edge going into C_{cur} or have only edges going into C_{cur}. Note the similarities to the attractor, Algorithm 1.

Indeed, one sees that $\text{SafeAttr}(G, \lambda, X, \sigma) = \text{Attr}(G, X, \sigma)$ if $\lambda \geqslant p(\max(V \setminus X))$. And, just like the regular Attractor, one sees that the Safe Attractor stabilizes its own output; i.e., $\text{SafeAttr}(G, \lambda, \text{SafeAttr}(G, \lambda, X, \sigma), \sigma) = \text{SafeAttr}(G, \lambda, X, \sigma)$.

However, it is not obvious how to directly substitute the safe attractor into Algorithm 2, as there does not appear to be a canonical choice for the parameter λ. This motivates the Sequential Safe Attractor algorithm, which, in the trap-depth $k = 1$ case, iteratively applies the λ-safe attractor to σ vertices of priority at least λ. Recall that we defined the max of a set of vertices to be the vertices of largest priority in that set. Below, if S is a set of vertices that all have the same priority, then $p(S)$ is that priority (rather than the singleton containing that priority).

Algorithm 4 $\text{SeqAttr}_1(G = (V, E, p), X, \sigma)$.

```
W := V_σ ∩ X
C := ∅
while W ≠ ∅ do
    S := max(W)
    C := SafeAttr(G, p(S), C ∪ S, σ)
    W := W \ C
end while
return C
```

At the beginning of the "while" loop above, we have a set C and a list W of target vertices to process. Each iteration of the loop calls the Safe Attractor Algorithm. The Sequential Attractor removes the issue of a priority bound inside the Safe Attractor. For any vertex $v \in V$, $\text{SeqAttr}_1(G, X, \sigma)$ tests if σ has a strategy to move the token from v towards some $w \in X$ in which any resulting path did not visit any vertices of priority at least $p(w)$.

The algorithm for solving trap-depth 1 parity games, $\text{TDA}_1(G, \sigma)$, simply substitutes the Sequential Safe Attractor for the Attractor in the Büchi games algorithm, Algorithm 2.

Algorithm 5 $\text{TDA}_1(G = (V, E, p), \sigma)$.

```
T_prev := ∅
T_cur := V_σ
while T_cur ≠ T_prev do
    T_prev := T_cur
    C := SeqAttr_1(G, T_prev, σ)
    T_cur := T_prev \ {v ∈ V_σ : N(v) ∩ C = ∅}
end while
return C
```

$\text{TDA}_1(G, \sigma)$ returns the largest $\overline{\sigma}$ trap X in G so that every σ-subtrap Y has that the vertices of largest priority in Y belong to player σ. We will sketch a proof of this fact; a complete proof will follow from the arguments in the next section. The proof will argue three different points, from which the characterization of $\text{TDA}_1(G, \sigma)$ follows immediately:

(1) (Monotonicity): If A is a $\overline{\sigma}$-trap in G, then we have $\text{TDA}_1(G[A], \sigma) \subseteq \text{TDA}_1(G, \sigma)$.
(2) (Completeness): If V satisfies that every σ-trap in V has a vertex whose maximum priority is of parity σ, then $\text{TDA}_1(G = (V, E, p), \sigma) = V$.
(3) (Soundness): $\text{TDA}_1(G, \sigma)$ is a $\overline{\sigma}$-trap in G whose maximum priority is of parity σ and which satisfies that every σ-subtrap has maximum priority σ.

To argue monotonicity, one first argues that the Safe Attractor Algorithm is monotonic with respect to its parameter λ, its input set X, and sometimes with respect to the parity game.

Explicitly, if $\lambda_1 \leqslant \lambda_2$, if $X_1 \subseteq X_2$, and if A is a $\overline{\sigma}$-trap in G, then

$$\text{SafeAttr}\big(G[A], \lambda_1, X_1, \sigma\big) \subseteq \text{SafeAttr}(G, \lambda_2, X_2, \sigma).$$

This is intuitive, but will be proven carefully in the next section.

From this, monotonicity of SeqAttr_1 easily follows. If $X_1 \subseteq X_2$ and if A is a $\overline{\sigma}$-trap in G, then $\text{SeqAttr}_1(G[A], X_1, \sigma) \subseteq \text{SeqAttr}_1(G, X_2, \sigma)$. A generalization of this will be proven in the next section.

Finally, from this, point (1), monotonicity of TDA_1, easily follows. Again, a generalization is carefully proved in the next section.

Now we will argue completeness. Assume the whole vertex set V satisfies that every σ-trap in V has maximum priority of parity σ. Then we claim that $\text{SeqAttr}_1(G, V, \sigma) = V$. This will imply that TDA_1 terminates after the first call to SeqAttr and returns V, as desired. To see that $\text{SeqAttr}_1(G, V, \sigma) = V$, consider the first iteration of the "WHILE" loop: since V is a σ-trap in V, the first iteration computes the σ-safe-attractor of $\max(V)$ with $\lambda = p(\max(V))$. However, since these are the vertices of maximum priority, this is the same as computing the attractor. The remaining set is a σ-trap in V, and so has maximum priority of parity σ. By induction, this continues until W is empty and SeqAttr_1 returns all of V.

Finally, we argue the soundness of TDA_1. At the end of the execution of TDA_1, there is some final set of target vertices T satisfying every vertex of T has an edge into $\text{SeqAttr}_1(G, T, \sigma)$. By our observations about SeqAttr_1, starting from any vertex in $\text{TDA}_1(G, \sigma)$, σ has some strategy to reach some vertex w in T so that along the way only priorities less than $p(w)$ are visited. Furthermore, by the terminating condition of TDA_1, every vertex in T has an edge back into $\text{TDA}_1(G, \sigma)$. This gives that $\text{TDA}_1(G, \sigma)$ is indeed a $\overline{\sigma}$-trap. Furthermore, starting from the largest vertex in any σ-trap, σ may follow the strategy that allows σ to reach some vertex w in T without seeing larger vertices along the way; σ cannot force the play to leave the trap, and therefore the largest priority in the trap must be w, a σ vertex, as desired.

4.3. General Trap-Depth Algorithm

The main difference between the general Trap-Depth Algorithm (TDA) and TDA_1 is that we change the call to Safe Attractor in TDA_1 to a stronger algorithm, the Generalized Safe Attractor. Indeed, we prove a more general result than Theorem 4.1, allowing us to strengthen any algorithm that satisfies certain conditions.

Definition 4.2. Let $\text{ParAlg}(G, \sigma)$ be an algorithm that takes as input a parity game G and a player σ and returns a subset of the vertices of G. We say ParAlg is *nice with traps* if:

- For any parity game G and any $\overline{\sigma}$ trap A in G,

$$\text{ParAlg}\big(G[A], \sigma\big) \subseteq \text{ParAlg}(G, \sigma).$$

- For any parity game G, taking $S = \text{ParAlg}(G, \sigma)$, for any $\overline{\sigma}$-trap A containing S, $\text{ParAlg}(G[A], \sigma) = S$, and for any σ-trap A intersecting S, $\text{ParAlg}(G[A], \sigma)$ is nonempty.
- $\text{ParAlg}(G, \sigma)$ always returns a $\overline{\sigma}$-trap whose largest priority belongs to player σ.

If ParAlg is nice with traps, then we can strengthen it via the TDA. The following definition defines what it means for a set of vertices X to be good; the TDA will return the largest good set of vertices.

Definition 4.3. Let $\text{ParAlg}(G, \sigma)$ be an algorithm that takes as input a parity game G and a player σ and returns a subset of the vertices of G. Given a parity game G and a set of vertices X, we say X is *good* for ParAlg with respect to player σ if X is a $\overline{\sigma}$-trap whose maximum priority is of parity σ so that for every σ-subtrap Y, either the maximum priority of Y belongs to σ or we have $\text{ParAlg}(G[Y], \sigma)$ is nonempty.

When the context is clear, we will simply say that X is good.

Theorem 4.4. *Let* $\text{ParAlg}(G, \sigma)$ *be an algorithm that takes as input a parity game* G *and a player* σ *and returns a subset of the vertices of* G. *If* ParAlg *is nice with traps, then* $\text{TDA}(G, \sigma, \text{ParAlg})$ *returns the largest set of vertices* X *which is good.*

Recursively applying Theorem 4.4 will give Theorem 4.1.

We said that TDA is obtained from TDA_1 by replacing the call to safe attractor, so let us first present the Generalized Safe Attractor Algorithm. The idea behind the safe attractor is to guarantee reaching a target set of vertices in a λ-safe way, so that along the way we don't see vertices of priority at least λ. The idea behind the generalized safe attractor is to guarantee that, if σ fails to reach a target set of vertices, then σ wins the play; in both cases, the only vertices of priority at least λ that are visited are in the target set. We informally refer to a strategy that avoids vertices of priority at least λ as "λ-safe".

In order to ensure that everything that is done is λ-safe, we will at some point remove all vertices of priority at least λ from the game by taking the restriction to vertices of lower priority: define the λ-restriction of a parity game $\text{Restrict}(G = (V, E, p), \lambda, \sigma) := V \setminus \text{Attr}(G, \{v \in V : \ p(v) \geqslant \lambda\}, \overline{\sigma})$. In words, the only vertices that remain in $\text{Restrict}(G, \lambda, \sigma)$ are those from which σ can ensure that all the priorities in any resulting play are less than λ.

We are now ready to introduce the Generalized Safe Attractor. It takes as input a parity game G, a number λ, a set of target vertices X, a player σ, and an algorithm $\mathrm{ParAlg}(G, \sigma)$. It is most useful to think of the context where $\mathrm{ParAlg}(G, \sigma)$ is nice with traps and always returns a subset of σ's winning region.

Algorithm 6 $\mathrm{GenAttr}(G = (V, E, p), \lambda, X, \sigma, \mathrm{ParAlg})$.

$C_{\mathrm{prev}} := \emptyset$
$C_{\mathrm{cur}} := X$
while $C_{\mathrm{cur}} \neq C_{\mathrm{prev}}$ **do**
  $C_{\mathrm{prev}} := C_{\mathrm{cur}}$
  $S := \mathrm{SafeAttr}(G, \lambda, C_{\mathrm{prev}}, \sigma)$
  $V' := \mathrm{Restrict}(G[V \setminus S], \lambda, \sigma)$
  $C_{\mathrm{cur}} := S \cup \mathrm{ParAlg}(G[V'], \sigma)$
end while
return C_{cur}

At the general step of the "while" loop, we begin with a set C_{prev} of vertices which we want to reach in a λ-safe way. The loop then calls the Safe Attractor Algorithm. $\mathrm{SafeAttr}(G, \lambda, C_{\mathrm{prev}}, \sigma)$ returns the largest collection of vertices S from which σ has a strategy to force the token into C_{prev} such that the token only hits vertices of priority smaller than λ along the way. Once the set S has been found, we check if, given that $\overline{\sigma}$ avoids X, player σ has any *winning* set of vertices given by ParAlg (which is λ-safe) on the remaining set $V \setminus S$; we add this to S to get C_{cur}. Each iteration either adds vertices to C_{cur} or terminates the loop. Since $|C_{\mathrm{cur}}|$ cannot increase indefinitely, GenAttr eventually halts.

For a vertex v and a subset X, v will be in GenAttr if and only if there is a σ-strategy to move the token from v towards X such that, depending on $\overline{\sigma}$'s moves, *either* the token eventually reaches X *or* the token reaches a vertex that ParAlg guarantees is winning for σ; in both cases, all the vertices visited by the token have priority less than λ, except perhaps the ones in X.

As the name suggests, the Generalized Safe Attractor is a generalization of the Safe Attractor. Consider the case where $\mathrm{ParAlg}(G, \sigma)$ simply returns the empty set for every input. Under this condition, we claim that $\mathrm{GenAttr}(G, \lambda, X, \sigma, \mathrm{ParAlg}) = \mathrm{SafeAttr}(G, \lambda, X, \sigma)$. Later we will show that SafeAttr stabilizes its own output; this immediately gives that, if ParAlg is always empty, a call to GenAttr will have at the end of the first "WHILE" loop C_{cur} equal to SafeAttr, and will subsequently terminate with this output.

Both the general case of the sequential safe attractor algorithm and the TDA are analogous to the ones before, except the sequential safe attractor calls the generalized safe attractor. As before, if S is a set of vertices that all have the same priority, then we write $p(S)$ to denote that priority.

Algorithm 7 $\mathrm{SeqAttr}(G = (V, E, p), X, \sigma, \mathrm{ParAlg})$.

$W := V_\sigma \cap X$
$C := \emptyset$
while $W \neq \emptyset$ **do**
  $S := \max(W)$
  $C := \mathrm{GenAttr}(G, p(S), C \cup S, \sigma, \mathrm{ParAlg})$
  $W := W \setminus C$
end while
return C

For any $v \in \mathrm{SeqAttr}(G, X, \sigma, \mathrm{ParAlg})$, σ has a strategy to ensure that, if the token ever reaches some $w \in X$, it hits only vertices of priority smaller than $p(w)$ along the way and, if it never reaches X, then σ wins.

TDA simply calls the sequential safe attractor.

Algorithm 8 $\mathrm{TDA}(G = (V, E, p), \sigma, \mathrm{ParAlg})$.

$T_{\mathrm{prev}} := \emptyset$
$T_{\mathrm{cur}} := V_\sigma$
while $T_{\mathrm{cur}} \neq T_{\mathrm{prev}}$ **do**
  $T_{\mathrm{prev}} := T_{\mathrm{cur}}$
  $C := \mathrm{SeqAttr}(G, T_{\mathrm{prev}}, \sigma, \mathrm{ParAlg})$
  $T_{\mathrm{cur}} := T_{\mathrm{prev}} \setminus \{v \in V_\sigma : N(v) \cap C = \emptyset\}$
end while
return C

As before, TDA calls SeqAttr on progressively smaller sets of target vertices T_{cur}.

One easily sees that the set C output by $\mathrm{TDA}(G, \sigma, \mathrm{ParAlg})$ is the Sequential Attractor of some final collection $T \subseteq V_\sigma$ of target vertices. If σ follows the strategy given by the sequential attractor on C, the resulting play will either be winning for σ, as guaranteed by the conditions on ParAlg, or reach T infinitely often, and the largest priority will belong to σ and so the play will be winning for σ.

4.4. Correctness

We now outline the proof of Theorem 4.4. We will prove three propositions from which Theorem 4.4 will follow immediately. Note that, in the second statement below (completeness), V is the whole vertex set of the parity game.

Theorem 4.5. *Let* $\mathrm{ParAlg}(G, \sigma)$ *be an algorithm that takes as input a parity game* G *and a player* σ *and returns a subset of the vertices of* G*. Assume* ParAlg *is nice with traps.*

(1) (*Monotonicity*): *If* A *is a* $\overline{\sigma}$*-trap in* G*, then we have* $\mathrm{TDA}(G[A], \sigma, \mathrm{ParAlg}) \subseteq \mathrm{TDA}(G, \sigma, \mathrm{ParAlg})$.
(2) (*Completeness*): *If* V *is good, then* $\mathrm{TDA}(G, \sigma, \mathrm{ParAlg}) = V$.
(3) (*Soundness*): $\mathrm{TDA}(G, \sigma, \mathrm{ParAlg})$ *is good.*

We will now build up the machinery to prove the above.

4.4.1. General lemmas

We begin with some general lemmas that will be useful; they are all reasonably simple to prove and we omit their proofs.

The first of these notes that SafeAttr is equal to Attr for λ large enough.

Lemma 4.6. $\mathrm{SafeAttr}(G, \lambda, X, \sigma) = \mathrm{Attr}(G, X, \sigma)$ *if* $\lambda > p(\max(V \setminus X))$.

The next lemma notes that the algorithms are stable when run on their own outputs. The second and third statements follow from the ones prior.

Lemma 4.7. *Let* A *be a* $\overline{\sigma}$ *trap.*

(1) *If* $S := \mathrm{SafeAttr}(G, \lambda, X, \sigma) \subseteq A$*, then*

$$S = \mathrm{SafeAttr}(G, \lambda, S, \sigma) = \mathrm{SafeAttr}\big(G[A], \lambda, S, \sigma\big).$$

(2) *If* $S := \mathrm{GenAttr}(G, \lambda, X, \sigma, \mathrm{ParAlg}) \subseteq A$*, then*

$$S = \mathrm{GenAttr}(G, \lambda, S, \sigma, \mathrm{ParAlg}) = \mathrm{GenAttr}\big(G[A], \lambda, S, \sigma, \mathrm{ParAlg}\big).$$

(3) *If* $S := \mathrm{SeqAttr}(G, \lambda, X, \sigma, \mathrm{ParAlg}) \subseteq A$*, then*

$$S = \mathrm{SeqAttr}(G, \lambda, S, \sigma, \mathrm{ParAlg}) = \mathrm{SeqAttr}\big(G[A], \lambda, S, \sigma, \mathrm{ParAlg}\big).$$

A similar statement holds for the TDA. Observe first that $\mathrm{TDA}(G, \sigma, k)$ is a $\overline{\sigma}$ trap in G with maximum vertex of parity σ.

Lemma 4.8. *Take* $S := \mathrm{TDA}(G, \sigma, \mathrm{ParAlg})$*. Then*

$$S = \mathrm{TDA}\big(G[S], \sigma, \mathrm{ParAlg}\big).$$

As before, it is also true that $\mathrm{TDA}(G, \sigma, \mathrm{ParAlg}) = \mathrm{TDA}(G[A], \sigma, \mathrm{ParAlg})$ where A is any $\overline{\sigma}$ trap containing $\mathrm{TDA}(G, \sigma, \mathrm{ParAlg})$; this follows easily from the characterization of the TDA, but is more difficult to prove given only what we have established so far.

4.4.2. Monotonicity

We now prove monotonicity of SafeAttr for the inputs X, λ and sometimes for the graph G:

Lemma 4.9. *If* $X_1 \subseteq X_2$*,* $\lambda_1 \leqslant \lambda_2$*, and* A *is a* $\overline{\sigma}$ *trap with* $X_1 \subseteq A$*, then* $\mathrm{SafeAttr}(G[A], \lambda_1, X_1, \sigma) \subseteq \mathrm{SafeAttr}(G, \lambda_2, X_2, \sigma)$.

Proof. Take $C_0, C_1, \ldots$ to be the values of C_{cur} at the beginning of each "while" loop in the execution of $\mathrm{SafeAttr}(G[A], \lambda_1, X_1, \sigma)$ (with the final value repeating) and take $C'_0, C'_1, \ldots$ similarly from the execution of $\mathrm{SafeAttr}(G, \lambda_2, X_2, \sigma)$. Then $C_0 \subseteq C'_0$.

Note that, by definition of a trap, if $v \in A \cap V_\sigma$ then $N_{G[A]}(v) \subseteq N_G(v)$, and if $v \in A \cap V_{\overline{\sigma}}$ then $N_{G[A]}(v) = N_G(v)$.

If $C_i \subseteq C'_i$ then we have

$$\big\{v \in A \cap V_\sigma \colon p(v) < \lambda_1 \wedge N_{G[A]}(v) \cap C_i \neq \emptyset\big\} \subseteq \big\{v \in V_\sigma \colon p(v) < \lambda_2 \wedge N_G(v) \cap C'_i \neq \emptyset\big\},$$
$$\big\{v \in A \cap V_{\overline{\sigma}} \colon p(v) < \lambda_1 \wedge N_{G[A]}(v) \subseteq C_i\big\} \subseteq \big\{v \in V_{\overline{\sigma}} \colon p(v) < \lambda_2 \wedge N_G(v) \subseteq C'_i\big\}$$

and so $C_{i+1} \subseteq C'_{i+1}$, and by induction this holds for all i. □

We now simultaneously address three monotonicity properties for GenAttr: monotonicity of the output with respect to the inputs X, λ and also sometimes with respect to the graph G.

The next lemma is a weak monotonicity property for GenAttr, saying that one iteration of the 'WHILE' loop in the algorithm will be contained in the GenAttr of the stronger inputs; combining this with the previous lemma will give monotonicity properties for GenAttr.

Lemma 4.10. *Assume $X_1 \subseteq X_2$ and $\lambda_1 \leqslant \lambda_2$. Assume A is a $\overline{\sigma}$ trap in G and $X_1 \subseteq A$. Take*

$$S^* := \text{SafeAttr}\big(G[A], \lambda_1, X_1, \sigma\big),$$
$$V^* := \text{Restrict}\big(G\big[A \setminus S^*\big], \lambda_1, \sigma\big),$$
$$T^* := \text{ParAlg}\big(G\big[V^*\big], \sigma\big).$$

Then $S^* \cup T^* \subseteq \text{GenAttr}(G, \lambda_2, X_2, \sigma, \text{ParAlg})$.

Proof. Take C' to be the value of C_{cur} at the end of the execution of $\text{GenAttr}(G, \lambda_2, X_2, \sigma, \text{ParAlg})$. Taking:

$$S' := \text{SafeAttr}\big(G, \lambda_2, C', \sigma\big),$$
$$V' := \text{Restrict}\big(G\big[V \setminus S'\big], \lambda_2, \sigma\big),$$
$$T' := \text{ParAlg}\big(G\big[V'\big], \sigma\big)$$

we have that $C' = S'$ and T' is empty.

We get that $S^* \subseteq S' = C'$ by monotonicity of the safe attractor, and so we need only show that $T^* \subseteq C'$. Assume, for sake of contradiction, that T^* is not a subset of C', and so in particular $T^* \setminus S' \neq \emptyset$.

In the following, we refer to traps in induced subgraphs that are not necessarily subgames, i.e. they may have vertices without outgoing edges. The definition of trap remains unchanged.

We claim that $T^* \setminus S'$ is a $\overline{\sigma}$ trap in $G[V']$. Take $B := \{v \in A\colon\ p(v) \geqslant \lambda_1\}$. We know that T^* is a $\overline{\sigma}$ trap in $G[V^*]$. Then note that $V^* = (A \setminus S^*) \setminus \text{Attr}(G[A \setminus S^*], B, \overline{\sigma})$, and so we get that V^* is a $\overline{\sigma}$ trap in $G[A \setminus S^*]$ and so T^* is a $\overline{\sigma}$ trap in $G[A \setminus S^*]$. Since the edges of $G[A \setminus S']$ are a subset of the edges of $G[A \setminus S^*]$, we get that any $\overline{\sigma}$ vertex in $T^* \setminus S'$ has no edges leaving $T^* \setminus S'$ in the graph $G[A \setminus S']$. Given any σ vertex in $T^* \setminus S'$, since $S' = \text{SafeAttr}(G, \lambda_2, S', \sigma)$ and since $\forall v \in V^* p(v) < \lambda_1 \leqslant \lambda_2$, the σ vertex had no edges into S' in $G[A]$ (for otherwise it would be contained in S') and so the σ vertex must have some edge into $T^* \setminus S'$ and so we get that indeed $T^* \setminus S'$ is a $\overline{\sigma}$ trap in $G[A \setminus S']$, and so also in $G[V \setminus S']$ (since A is a $\overline{\sigma}$-trap in V). We have $T^* \setminus S'$ is a $\overline{\sigma}$-trap in $G[V \setminus S']$ with no vertices of priority at least λ; any such structure must be a $\overline{\sigma}$-trap in $G[V']$.

Because T^* has no vertices of priority at least λ_2, we have $T^* \setminus S' = T^* \setminus \text{Attr}(G[T^*], S', \sigma)$, and so $T^* \setminus S'$ is a σ-trap in $G[T^*]$. Since we know ParAlg is nice with traps, we have that $\text{ParAlg}(G[T^*], \sigma) = T^*$. Therefore, again since ParAlg is nice with traps, $\text{ParAlg}(G[T^* \setminus S'], \sigma)$ is nonempty. Finally, $\text{ParAlg}(G[T^* \setminus S'], \sigma) \subseteq \text{ParAlg}(G[V'], \sigma)$, but this contradicts the assumption that $T' = \emptyset$. Therefore, we must have that $T^* \subseteq S'$. □

Lemma 4.11. *If $X_1 \subseteq X_2$ and $\lambda_1 \leqslant \lambda_2$ and A is a $\overline{\sigma}$ trap in G with $X_1 \subseteq A$, then* $\text{GenAttr}(G[A], \lambda_1, X_1, \sigma, \text{ParAlg}) \subseteq \text{GenAttr}(G, \lambda_2, X_2, \sigma, \text{ParAlg})$.

Proof. Take $C_0, C_1, \ldots$ to be the values of C_{cur} at the beginning of each "while" loop in the execution of $\text{GenAttr}(G[A], \lambda_1, X_1, \sigma, \text{ParAlg})$ (with the final value repeating). Take

$$S_i := \text{SafeAttr}\big(G[A], \lambda_2, C_i, \sigma\big),$$
$$V_i := \text{Restrict}\big(G[A \setminus S_i], \lambda_2, \sigma\big),$$
$$T_i := \text{ParAlg}\big(G[V_i], \sigma\big).$$

We will proceed by induction on i to show that $C_i \subseteq \text{GenAttr}(G, n_2, X_2, \sigma, \text{ParAlg})$. Note this holds for C_0 since $C_0 = X_1 \subseteq X_2$. Then, for $i > 0$, we have $C_i = T_{i-1} \cup S_{i-1}$ and by the previous lemma and inductive hypothesis we get:

$$\begin{aligned} T_{i-1} \cup S_{i-1} &\subseteq \text{GenAttr}(G, \lambda_2, C_{i-1}, \sigma, \text{ParAlg}) \subseteq \text{GenAttr}\big(G, \lambda_2, \text{GenAttr}(G, \lambda_2, X_2, \sigma, \text{ParAlg}), \sigma, \text{ParAlg}\big) \\ &= \text{GenAttr}(G, \lambda_2, X_2, \sigma, \text{ParAlg}), \end{aligned}$$

completing the proof. □

We will now proceed to show monotonicity of SeqAttr. While the original definition was slightly more natural, the following reformulation of SeqAttr will be more useful. We leave it to the reader to verify that the following reformulation of SeqAttr is equivalent to the original. It follows immediately from monotonicity and stability of GenAttr.

Lemma 4.12. *If P' is a finite collection of integers and $p(X \cap V_\sigma) \subseteq P'$ then, taking P to be the priorities in P' of parity σ, the following algorithm has the same output as* SeqAttr.

Algorithm 9 $\text{SeqAttr}_P(G = (V, E, p), X, \sigma, \text{ParAlg})$.

```
C := ∅
Q := P
while Q ≠ ∅ do
    λ := max(Q)
    Q := Q \ {λ}
    S := {v ∈ X: p(v) = λ}
    C := GenAttr(G, λ, C ∪ S, σ, ParAlg)
end while
return C
```

Intuitively, we simply run the GenAttr for every priority in P, which just adds redundancy by the assumption $p(X \cap V_\sigma) \subseteq P$: if ever in the original formulation of SeqAttr some call $\text{GenAttr}(G, \lambda, C, \sigma, \text{ParAlg})$ were made, then in the above version some call will be made with the same parameter λ.

We now show monotonicity properties for SeqAttr with respect to the input X and also sometimes with respect to the graph G:

Lemma 4.13. *If $X_1 \subseteq X_2$ and A is a $\overline{\sigma}$-trap in G such that $X_1 \subseteq A$, then* $\text{SeqAttr}(G[A], X_1, \sigma, \text{ParAlg}) \subseteq \text{SeqAttr}(G, X_2, \sigma, \text{ParAlg})$.

Proof. Take $P = p(X_2 \cap V_\sigma)$. Take C_i, Q_i to be the values of C, Q respectively at the beginning of the ith iteration of the "WHILE" loop in the execution of $\text{SeqAttr}_P(G[A], X_1, \sigma, \text{ParAlg})$. Take

$$\lambda_i = \max(Q_i),$$
$$S_i = \{v \in X_1 \colon p(v) = \lambda_i\}.$$

Similarly take $C'_i, Q'_i, \lambda'_i, S'_i$ for the execution of $\text{SeqAttr}_P(G, X_2, \sigma, \text{ParAlg})$. Since $Q_0 = Q'_0$ and $Q_{i+1} = Q_i \setminus \max(Q_i)$ and $Q'_{i+1} = Q'_i \setminus \max(Q'_i)$ we get $Q_i = Q'_i$ and $\lambda_i = \lambda'_i$ for all i. Then $S_i \subseteq S'_i$ since $X_1 \subseteq X_2$. We now proceed by induction to show $C_i \subseteq C'_i$. We have $C_0 = C'_0 = \emptyset$ and

$$C_{i+1} = \text{GenAttr}\big(G[A], \lambda_i, S_i \cup C_i, \sigma, \text{ParAlg}\big) \subseteq \text{GenAttr}\big(G, \lambda'_i, S'_i \cup C'_i, \sigma, \text{ParAlg}\big) = C'_{i+1}. \qquad \square$$

We now present the monotonicity theorem for TDA:

Theorem 4.14. *If A is a $\overline{\sigma}$-trap in G, then we have* $\text{TDA}(G[A], \sigma, \text{ParAlg}) \subseteq \text{TDA}(G, \sigma, \text{ParAlg})$.

Proof. Let $T_0, T_1, \ldots$ be the values of T_{cur} at the beginning of the "WHILE" loop in the execution of $\text{TDA}(G[A], \sigma, \text{ParAlg})$. Take $C_i := \text{SeqAttr}(G[A], T_i, \sigma, \text{ParAlg})$. Similarly define T'_i, C'_i for $\text{TDA}(G, \sigma, \text{ParAlg})$. We proceed by induction to show $T_i \subseteq T'_i$. This holds for T_0, T'_0 since $A \cap V_\sigma \subseteq V_\sigma$. Then, by monotonicity of SeqAttr, we get $C_i \subseteq C'_i$. To obtain T_{i+1} and T'_{i+1} from T_i, T'_i, respectively, any vertex in T_i, T'_i without an edge into C_i, C'_i is removed. The edges of G are a superset of those of $G[A]$ and C'_i is a superset of C_i, so we get:

$$\begin{aligned} T_{i+1} &= T_i \setminus \{v \in V_\sigma \cap A \colon N_{G[A]}(v) \cap C_i = \emptyset\} = T_i \setminus \{v \in T_i \colon N_{G[A]}(v) \cap C_i = \emptyset\} \\ &\subseteq T'_i \setminus \{v \in T'_i \colon N_G(v) \cap C'_i = \emptyset\} = T'_i \setminus \{v \in V_\sigma \colon N_G(v) \cap C'_i = \emptyset\} = T'_{i+1}. \qquad \square \end{aligned}$$

4.4.3. Completeness

Lemma 4.15. *If V is good for* ParAlg *with respect to σ and if $T \subseteq V$ and λ are such that $p(\max(V \setminus T)) < \lambda$, then taking $S :=$* $\text{GenAttr}(G, T, \lambda, \sigma, \text{ParAlg})$ *we have $S = \text{Attr}(G, S, \sigma)$ and if $V \setminus S \neq \emptyset$ then* $\max(V \setminus S) \subseteq V_\sigma$.

Proof. We consider that by the terminating condition for the GenAttr algorithm, we must have

$$S = \text{SafeAttr}(G, S, \lambda, \sigma).$$

Note that $\text{SafeAttr}(G, S, \lambda, \sigma) = \text{Attr}(G, S, \sigma)$ (since $\lambda > p(\max(V \setminus T))$) and so we get

$$S = \mathrm{Attr}(G, S, \sigma).$$

Since $p(\max(V \setminus S)) < \lambda$, we also get by the terminating condition for GenAttr that $\mathrm{ParAlg}(V \setminus S, \sigma) = \emptyset$, but, by assumption, since $V \setminus S$ is a σ trap in G, either $V \setminus S = \emptyset$ (in which case we are done) or $\max(V \setminus S) \subseteq V_\sigma$. □

Lemma 4.16. *If V is good for* ParAlg *with respect to σ, then* $\mathrm{SeqAttr}(G, V_\sigma, \sigma, \mathrm{ParAlg}) = V$.

Proof. Taking $W_0, W_1, \ldots$ to be the values of W at the beginning of each while-loop in the execution of $\mathrm{SeqAttr}(G, V, \sigma, \mathrm{ParAlg})$, take

$$S_i := \max(W_i),$$
$$C_i := \mathrm{GenAttr}\big(G, p(S_i), C_{i-1} \cup S_i, \sigma, \mathrm{ParAlg}\big)$$

then we have by the previous lemma $\max(W_0) = \max(V)$. Then, by induction, if $\max(W_i) \in V_\sigma$, we have either $W_{i+1} = \emptyset$ or $\max(W_{i+1}) = \max(V \setminus C_i)$ and so the SeqAttr will not terminate until $V \subseteq C_i$. □

Theorem 4.17. *If V is good for* ParAlg *with respect to σ, then* $\mathrm{TDA}(G, \sigma, \mathrm{ParAlg}) = V$.

Proof. The previous lemma immediately gives that the TDA will terminate after the first iteration of the "WHILE" loop and return V, since SeqAttr will return the whole set of vertices. □

4.4.4. Soundness

Lemma 4.18. *If X is a σ-trap in G and if $X \cap Y = \emptyset$ then $X \cap \mathrm{SafeAttr}(G, Y, \lambda, \sigma) = \emptyset$.*

Proof. Note that $\mathrm{SafeAttr}(G, Y, \lambda, \sigma) \subseteq \mathrm{Attr}(G, Y, \sigma)$ and $\mathrm{Attr}(G, Y, \sigma) \cap X = \emptyset$. □

Lemma 4.19. *If X is a σ-trap in G with largest vertex of priority $m < \lambda$ and with* $\mathrm{ParAlg}(G[X], \sigma) = \emptyset$, *then if $X \cap Y = \emptyset$ we have $X \cap \mathrm{GenAttr}(G, \lambda, Y, \sigma, \mathrm{ParAlg}) = \emptyset$.*

Proof. Take $C_0, C_1, \ldots$ to be the value of C_{cur} at the beginning of each "WHILE" loop in the execution of $\mathrm{GenAttr}(G, \lambda, Y, \sigma, \mathrm{ParAlg})$. Take

$$S_i := \mathrm{SafeAttr}(G, \lambda, C_i, \sigma),$$
$$V_i := \mathrm{Restrict}\big(G[V \setminus S_i], \lambda, \sigma\big),$$
$$T_i := \mathrm{ParAlg}\big(G[V_i], \sigma\big).$$

We proceed by induction on i to show $C_i \cap X = \emptyset$. This holds by assumption for $C_0 = Y$. If this holds for C_i, then $S_i \cap X = \emptyset$.

Note that, because all vertices in X have priority smaller than λ, $X \cap V_i = X \setminus \mathrm{Attr}(G[X], X \setminus V_i, \overline{\sigma})$. In particular, we have that $X \cap V_i$ is a $\overline{\sigma}$-trap in $G[X]$. Since ParAlg is nice with traps, this implies that $\mathrm{ParAlg}(G[X \cap V_i], \sigma) = \emptyset$.

Since X is a σ-trap in G and $X \cap S_i = \emptyset$, we get that X is a σ-trap in $G[V \setminus S_i]$. By definition, V_i is the complement of a $\overline{\sigma}$-attractor, so V_i is a $\overline{\sigma}$-trap in $G[V \setminus S_i]$. Since X is a σ-trap and V_i is a $\overline{\sigma}$-trap, we get that $X \cap V_i$ is a σ-trap in $G[V_i]$. Since T_i is a $\overline{\sigma}$-trap in $G[V_i]$ and $X \cap V_i$ is a σ-trap in $G[V_i]$, we have that $X \cap V_i \cap T_i$ is a $\overline{\sigma}$ trap in $G[X \cap V_i]$. Therefore, since ParAlg is nice with traps, $\mathrm{ParAlg}(G[X \cap V_i \cap T_i], \sigma) = \emptyset$. If $X \cap V_i \cap T_i$ were nonempty, then, since ParAlg is nice with traps and $X \cap V_i$ is a σ-trap in $G[V_i]$, we would have $\mathrm{ParAlg}(G[X \cap V_i \cap T_i], \sigma) \neq \emptyset$, a contradiction. Therefore, we must have $X \cap V_i \cap T_i = X \cap T_i = \emptyset$.

Finally, since $C_{i+1} = S_i \cup T_i$, we have that $X \cap C_{i+1} = \emptyset$, as desired. □

Lemma 4.20. *If X is a σ-trap with largest vertex of priority λ, λ has parity $\overline{\sigma}$, and* $\mathrm{ParAlg}(G[X], \sigma) = \emptyset$, *then the largest vertices of X are not contained in* $\mathrm{SeqAttr}(G, V, \sigma, \mathrm{ParAlg})$.

Proof. Take $C_0 = \emptyset$ and $W_0, W_1, \ldots$ to be the value of W at the beginning of each "WHILE" loop in the execution of $\mathrm{SeqAttr}(G, V, \sigma, \mathrm{ParAlg})$. Take

$$S_i = \max(W_i),$$
$$C_i = \mathrm{GenAttr}\big(G, p(S_i), C_{i-1} \cup S_i, \sigma, \mathrm{ParAlg}\big).$$

If $p(S_i) > \lambda$ we have, by maximality of λ, that $X \cap S_i = \emptyset$, and so by induction that $X \cap (C_{i-1} \cup S_i) = \emptyset$. By the previous lemma we get $X \cap C_i = \emptyset$. If $p(S_i) < \lambda$, then we have $\max(X) \cap S_i = \emptyset$ and so by induction $\max(X) \cap (S_i \cup C_{i-1}) = \emptyset$. Therefore, no vertices of $\max(X)$ can be added by the call to GenAttr and so $\max(X) \cap C_i = \emptyset$. □

Theorem 4.21. TDA$(G, \sigma, \text{ParAlg})$ *returns a set that is good for* σ *with respect to* ParAlg.

Proof. Take $S := \text{TDA}(G, \sigma, \text{ParAlg})$. Then $S = \text{TDA}(G[S], \sigma, \text{ParAlg})$. We've observed before that S is a $\overline{\sigma}$ trap whose largest vertex has priority of parity σ, so we may assume without loss of generality that $S = V$. By the previous lemma, there is no nonempty set X that is a σ-trap with largest vertex of priority $\overline{\sigma}$ so that $\text{ParAlg}(G[X], \sigma) = \emptyset$. □

This completes the proof of Theorem 4.4.

4.4.5. Second Trap-Depth Algorithm

We now define the second Trap-Depth Algorithm, $\text{TDA}(G, \sigma, k)$, and prove Theorem 4.1. We will define $\text{TDA}(G, \sigma, k)$ by recursively applying $\text{TDA}(G, \sigma, \text{ParAlg})$. In order to do so, we will need to know that $\text{TDA}(G, \sigma, \text{ParAlg})$ is nice with traps whenever ParAlg is.

Lemma 4.22. *If* ParAlg *is nice with traps, then* $\text{TDA}(G, \sigma, \text{ParAlg})$ *is nice with traps.*

Proof. The same argument used in proving Theorem 3.18 applies to show that, for any σ-trap Y, if $\text{TDA}(G, \sigma, \text{ParAlg}) \cap Y$ is nonempty, then $\text{TDA}(G[Y], \sigma, \text{ParAlg})$ is nonempty. The rest of the properties of being nice with traps follow from Theorem 4.4. □

We now define $\text{TDA}_k(G, \sigma) = \text{TDA}(G, \sigma, k)$ recursively. Define $\text{TDA}_0(G, \sigma)$ to be the algorithm that always returns the empty set. Note this is nice with traps. Given $\text{TDA}_k(G, \sigma)$, define $\text{TDA}_{k+1}(G, \sigma)$ by $\text{TDA}_{k+1}(G, \sigma) = \text{TDA}(G, \sigma, \text{TDA}_k)$. Inductively applying Theorem 4.4 now proves Theorem 4.1.

Note that we had previously defined $\text{TDA}_1(G, \sigma)$. Recalling that

$$\text{GenAttr}(G, \lambda, X, \sigma, \text{TDA}_0) = \text{SafeAttr}(G, \lambda, X, \sigma)$$

and that SafeAttr stabilizes its own output, it is easy to see that these two definitions match.

4.4.6. Runtime

Lemma 4.23. *Let* $T(n, m)$ *be an upper bound on the runtime of* $\text{ParAlg}(G, \sigma)$ *for a graph* G *on* n *vertices and* m *edges. Then the runtime of* $\text{TDA}(G, \sigma, \text{ParAlg})$ *on a graph on* n *vertices and* m *edges is at most* $O(mn^2) + n^2 T(n, m)$.

Proof. Consider first the Safe Attractor Algorithm. Since each iteration of the "while" loop increases the size of C_{cur} or halts the algorithm, there will be at most $O(n)$ loops. If implemented carefully (in the same way that the regular Attractor is implemented) we may guarantee that each edge is only used a constant number of times and actually run the algorithm in $O(m + n) = O(m)$ time.

Next, consider the Generalized Safe Attractor Algorithm. Each iteration of the "while" loop increases the size of C_{cur} or halts the algorithm. On top of calling ParAlg, the algorithm does $O(m)$ work for each loop ($\text{Restrict}(G, \lambda, \sigma)$ can be computed in linear time). If the algorithm runs j "while" loops, it does work at most $(O(m) + T(n, m)) \times j$.

The Sequential Attractor Algorithm has C increasing every iteration or the algorithm halts. Note that each time a call to generalized attractor causes the generalized attractor to go through a "while" loop, a new vertex is added to C, so the total number of such loops done throughout the calls to generalized attractor is n, and so the total amount of work is at most $(O(m) + T(n, m)) \times n = O(mn) + nT(n, m)$.

In TDA we have T_{cur} decreasing on each iteration or the algorithm halts, and so there are at most n calls to SeqAttr, and on top of these only $O(m)$ work is done, and so we get $T(n, m) = O(mn^2) + n^2 T(n, m)$. □

Lemma 4.24. *Let* $T(n, m, k)$ *denote the runtime of* $\text{TDA}(G, \sigma, k)$ *for a graph* G *on* n *vertices and* m *edges. Then* $T(n, m, 0) = O(1)$ *and for* $k > 0$ *we have* $T(n, m, k) = O(mn^{2k-1})$.

Proof. We have $T(n, m, 0) = O(1)$ since this algorithm always returns the empty set.

The same optimizations used in the computation of the Attractor and the Safe Attractor may be used to get a runtime of $O(m)$ in the case $k = 1$ for the sequential attractor, that is for SeqAttr_1. In TDA we have T_{cur} decreasing on each iteration or the algorithm halts, and so there are at most n calls to SeqAttr_1, and on top of these only $O(m)$ work is done, and so we get $T(n, m, 1) = O(mn)$.

For $k \geqslant 1$ by the previous lemma we have $T(n, m, k + 1) \leqslant O(mn^2) + n^2 T(n, m)$. This recurrence solves to $T(n, m, k) = O(mn^{2k-1})$, as desired. □

5. Summary and critical remarks

The theorems of the previous section show the promised characterization of TDA (Theorem 4.1):

TDA(G, σ, k) returns the largest (possibly empty) set starting with which σ can guarantee a win in at most k moves in the trap-depth game on G.

We have introduced Trap-Depth games (where the moves consist of choosing subsets of the graph rather than vertices/edges) and shown their close relationship with Muller games. We have defined the trap-depth parameter and given algorithms for parity games for finding subsets of the winning regions whose runtime is bounded by an exponential in this trap-depth. Writing $d := |p(V)|$, since the trap-depth of a parity game is at most $\lceil \frac{d}{2} \rceil$, the algorithm runs in time $O(mn^d)$. If one is only interested in the class of graphs with a bounded number of priorities, there are other options. The classical algorithm of Zielonka also runs in time $O(mn^d)$ (see [7]), but there are better algorithms: Jurdziński's [10] algorithm achieves $O(dm(\frac{n}{\lfloor \frac{d}{2} \rfloor})^{\lfloor \frac{d}{2} \rfloor})$, and the subexponential algorithm of [11] as well as a polynomial improvement thereof by [16] achieve $n^{O(\sqrt{n})}$ and $mn^{\frac{d}{3}+O(1)}$. Of course, the class of graphs of bounded trap depth is much more general than the class of graphs with a bounded number of priorities.

By Lemma 2.4, finding any nonempty subset of the winning region allows us to remove part of the graph to get a smaller parity game that needs to be solved; thus, for example, Parity games in which every subgame has bounded trap depth (such as those with a bounded number of priorities) may be completely solved in polynomial time, a generalization of the result that parity games with a bounded number of priorities may be solved in polynomial time.

Parity games are just one encoding of a class of Muller games. One may ask if there are others for which the characterization of Muller games we present is algorithmically useful. One possible encoding is called *Explicit Muller games*, where an enumeration of the sets winning for Red, i.e. of the set $\mathcal{R}$, is explicitly given as input. There is a known polynomial time algorithm for solving explicit Muller games [6], but we may hope to obtain another algorithm using the characterization. If one could efficiently answer the following question, such an algorithm exists (note in the following question (V, E, V_{red}) are given explicitly):

Problem 5.1. Given a Muller game G and an explicit list $S_1, \ldots, S_k \subseteq V$, is there some polynomial time algorithm that determines if every Red-trap H contains one of the S_i as a Blue-subtrap?

To see that the above would allow us to solve the problem, let an explicit Muller game $(V, E, V_{\text{red}}, \mathcal{R})$ be given. We will first prune $\mathcal{R}$ by removing any sets $R \in \mathcal{R}$ in which some vertex has no outgoing edges in $G[R]$ (these have no impact on the game). To determine if Red has a nonempty winning region, we will find the collection W of sets in $\mathcal{R}$ from which Red will win the trap-depth game in which Blue goes first.

We will iteratively update $\mathcal{R}$ and W. Choose any minimal (under inclusion) set $R \in \mathcal{R}$. For each such set R we determine if $G[R]$ contains any Red-traps that do not contain as a Blue-trap any set in $W \cup \{R\}$. If $G[R]$ has no such Red-traps, then we add R to W. In either case, we remove R from $\mathcal{R}$ and iterate.

It is easy to argue that if in the trap-depth game the set of vertices is X and it is Red's turn to move, then a Blue-trap Y in $G[X]$ is winning for Red if and only if H is in W. To determine if Red has a non-empty winning region, we need only check if one of the sets in W is a Blue-trap in G.

Acknowledgements

This work was partially supported by NSF grant DMS-0648208 at the Cornell REU, which are both gratefully acknowledged. Andrey Grinshpun is partially supported by the NPSC. Andrei Tarfulea is partially supported by the NSF GRFP. We warmly thank Alex Kruckman, James Worthington and Ben Zax for many stimulating discussions on an early part of this work, as well as Damian Niwinski for his comments. We also thank the anonymous referee, without whose comments reading this paper would be much less pleasant.

References

[1] D. Berwanger, A. Dawar, P. Hunter, S. Kreutzer, DAG-width and parity games, in: STACS 2006, in: Lect. Notes Comput. Sci., vol. 3848, 2006, pp. 524–536.
[2] D. Berwanger, E. Grädel, Fixed-point logics and solitaire games, Theory Comput. Syst. 37 (2004) 675–694.
[3] H. Björklund, S. Sandberg, S. Vorobyov, A discrete subexponential time algorithm for parity games, in: STACS 2003, in: Lect. Notes Comput. Sci., vol. 2607, 2003, pp. 663–674.
[4] E. Emerson, C. Jutla, Tree automata, μ-calculus, and determinacy, in: Proceedings of the 32nd Annual Symposium on Foundations of Computer Science, IEEE, 1991, pp. 368–377.
[5] E. Emerson, C. Jutla, A. Sistla, On model checking for fragments of μ-calculus, in: Computer Aided Verification, STACS 2006, in: Lect. Notes Comput. Sci., vol. 697, 1993, pp. 385–396.
[6] F. Horn, Explicit Muller games are PTIME, in: Annual Conference on Foundations of Software Technology and Theoretical Computer Science, 2008.
[7] E. Grädel, W. Thomas, T. Wilke, Automata, Logics, and Infinite Games, Springer, 2002.
[8] P. Hunter, Complexity and infinite games on finite graphs, Ph.D. thesis, University of Cambridge, 2007.
[9] M. Jurdziński, Deciding the winner in parity games is in UP ∩ co-UP, Inf. Process. Lett. 68 (1998) 119–124.
[10] M. Jurdziński, Small progress measures for solving parity games, in: STACS 2000, in: Lect. Notes Comput. Sci., vol. 1770, 2000, pp. 290–301.
[11] M. Jurdziński, M. Paterson, U. Zwick, A deterministic subexponential time algorithm for solving parity games, in: Proceedings of the Seventeenth Annual ACM–SIAM Symposium on Discrete Algorithms, Symposium on Discrete Mathematics, 2006, pp. 117–123.
[12] D. Martin, Borel determinacy, Ann. Math. (2) 102 (2) (1975) 363–371.

[13] R. McNaughton, Infinite games played on finite graphs, Ann. Pure Appl. Log. 65 (2) (1993) 149–184.
[14] J. Obdržálek, Clique-width and parity games, in: Computer Science Logic, in: Lect. Notes Comput. Sci., vol. 4646, 2007, pp. 54–68.
[15] J. Obdržálek, Fast mu-calculus model checking when tree-width is bounded, in: Computer Aided Verification, in: Lect. Notes Comput. Sci., vol. 2825, 2003, pp. 80–92.
[16] S. Schewe, Solving parity games in big steps, in: Foundations of Software Technology and Theoretical Computer Science, in: Lect. Notes Comput. Sci., vol. 4855, 2007, pp. 449–460.
[17] W. Thomas, Facets of synthesis: revisiting Church's problem, in: FOSSACS, 2009, pp. 1–14.
[18] W. Zielonka, Infinite games on finitely coloured graphs with applications to automata on infinite trees, Theor. Comput. Sci. 200 (1998) 135–183.