Semester Project

Spring 2022

The Computational Complexity of Parity Games

Alexandre Hayderi

Under the supervision of

Pr. Mika Göös

With the help of

Deniz Imrek

Contents

Chapter 1

An introduction to Parity Games

1.1 Defining the game

Parity games are infinite-duration two-player games played on finite directed graphs. The players are usually called Even and Odd. For a more formal definition,

Definition 1.1.1 (Parity game graph)**.** A parity game graph
$G = (V, V_E, V_O, E, \pi)$ is a directed graph (V, E) where V_E and V_O form a partition of V. As the names suggest, V_E are the vertices that belong to player Even and V_O to player Odd. Every vertex is also assigned an integer *priority* by $\pi : V \to \{0, 1, \ldots, h\}$ with $h \in \mathbb{N}$.

As additional requirements, we assume that there are no self-loops and no sink states in the graph. All vertices thus have at least one outgoing edge.

Some notations:

- We extend the notation of π on the set of all vertices V by noting $\pi(V)$ to be the set of all vertex priorities.
- When going over arguments, we sometimes use the notation P which could refer to either of the players Even or Odd, the notation $\overline{P}$ then respectively refers to player Odd or Even.
- For a given graph G, the important parameters for a complexity analysis are its number of vertices as well as its maximal priority, for conciseness, these parameters are often referred to as n and h where $n = |V|$ and $h = \max_{v \in V}(\pi(v))$.

To play a parity game, we need a parity game graph G as well as a starting state v_0. The game starts by placing a token on v_0, then the player that owns

v_0 (e.g. player Even if $v_0 \in V_E$) moves the token to a neighbouring vertex v_1 through one of v_0's outgoing edges. The game continues in a similar fashion *forever*: at any point in the game, if the token is on a vertex owned by a player, that player will choose a neighbouring vertex to move it on. Since there are no sink states, every player always has at least one possible vertex to move the token to.

Playing a parity game defines a run on G, more formally,

Definition 1.1.2 (Game run)**.** A run on a parity game defines an infinite path in G that we denote $\rho \in (V \times \pi(V))^\omega$ where every vertex on which the token passed, starting from the initial state v_0, is assigned its priority. A general run thus looks like: $\rho = (v_0, \pi(v_0)), (v_1, \pi(v_1)), \ldots$

Now for the interesting question, how to determine which player won a run?

Definition 1.1.3 (Winning a run)**.** Let $\rho \in (V \times \pi(V))^\omega$ be a run on G, ρ is winning for player Even if the *highest priority occurring infinitely often on ρ is even*, otherwise, the run is winning for player Odd. Another way to say this is to say that the winner on ρ is the player that has same parity as $\limsup(\rho)$ (where we only consider the priority values in ρ, not the vertices).

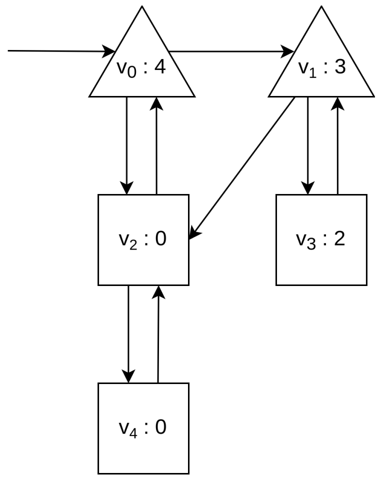

Figure 1.1: A simple parity game

This is an example of a parity game with its starting vertex (denoted by an incoming arrow) in the top left. The Even vertices are denoted by squares and the Odd vertices by triangles. Vertex priorities are noted inside the vertex (e.g. vertex v_0 has priority 4).

One possible run on this parity game is $\rho = (v_0, 4), ((v_1, 3), (v_3, 2))^\omega$ (where the notation $\texttt{subpath}^\omega$ denotes the fact that this subpath occurs an infinite number of times). In ρ, the priorities occurring an infinite number of times are 2 and 3, the highest priority occurring an infinite number of times is thus 3, an odd number, so Odd wins the run.

1.2 Strategies

Definition 1.2.1 (Even cycle)**.** An even cycle in a parity game graph G is a cycle in which the greatest priority is even. We similarly define odd cycles.

Definition 1.2.2 (Strategy)**.** A strategy for Even is a function
$\sigma : (V \times \pi(V))^{<w} \to V$ which maps every finite play ending in a vertex owned by Even to a move.

A strategy σ for Even is called a *winning strategy* if by following σ Even wins all runs beginning on the start state.

Theorem 1.2.1 (Determinacy of parity games)**.** In any parity game graph, from any starting node, either Even has a winning strategy, or Odd has a winning strategy.

Definition 1.2.3 (Memoryless strategy)**.** A memoryless strategy for Even is a function $\sigma : V_E \to V$ which maps every vertex v_E owned by Even to a move, that is, a neighbouring vertex of v_E.

Theorem 1.2.2 (Memoryless determinacy of parity games)**.** In any parity game graph, the winner has a memoryless winning strategy. [1, 2]

Consequently, we will use the term strategy to refer to a memoryless strategy.

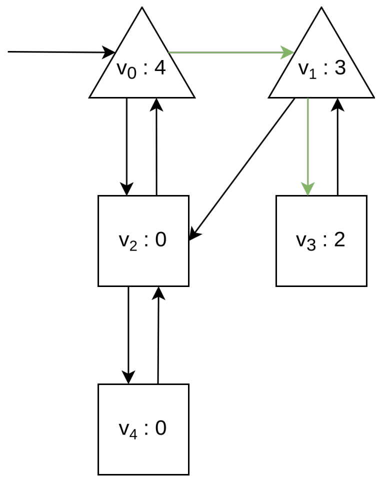

Figure 1.2: A memoryless winning strategy for Odd

1.3 Describing the winning regions

When we consider parity games, the main algorithmic question is to determine the *winning region* for Even, that is, all the starting nodes from which Even has a winning strategy. In order to facilitate the description of these winning regions, we will now introduce a few classic notions,

Definition 1.3.1 (Trap)**.** A trap T for a player P is a set of vertices on which player $\overline{P}$ has a strategy such that P can never escape T.

For example, in figure 1.2, $T = \{v_1, v_3\}$ forms a trap for player Even since player Odd can always chose to take the edge (v_1, v_3) thus preventing Even from escaping T.

Definition 1.3.2 (Dominion)**.** A dominion D for player P is a set of vertices from which P can win without ever having to leave D.

For example, the trap T from above is actually a dominion for Odd since it is a trap for Even (and thus Even can't escape it) and Odd wins on that trap since the highest priority is 3.

Remark. The largest dominion for P in G is the *winning region* for player P.

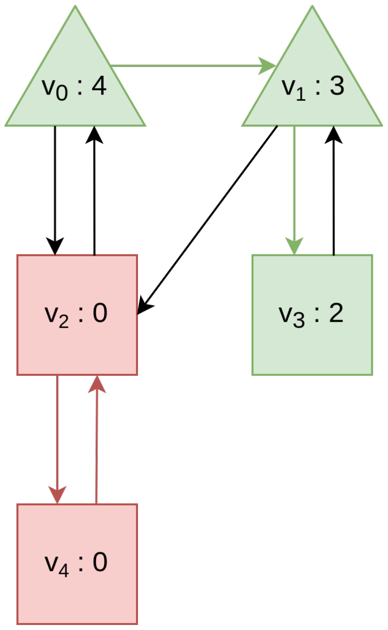

Figure 1.3: The winning regions and memoryless winning strategies on the entire graph

1.4 The complexity of parity games

The notion of memoryless determinacy and theorem 1.2.2 allow us to easily see that the problem of determining if Even has a winning strategy is in NP. Indeed, by taking as certificate a memoryless winning strategy for Even players over the entire game, we can easily construct a polynomial-time verifier that checks if that strategy is correct. Moreover, since parity games are determined, the complement of the problem (that is, deciding if Odd has a winning strategy) is also in NP by the same argument as before. Thus Parity Game $\in$ NP$\cap$co-NP [3] and is actually one of the few natural problems in NP $\cap$ coNP that haven't yet been proven to be in P.

Several other results show that the problem of deciding the winner in a parity game is also in UP $\cap$ co-UP [4] and that computing winning strategies is in CLS [5, 6], these classes are all only "slightly" above P which may hint at the fact that the problem admits a polynomial-time algorithm.

Apart from this interesting complexity theoretic status, parity games have a number of other characteristics that make them an important research subject. The problem of finding winning strategies has, for example, been shown to be linear-time equivalent to model checking for modal μ-calculus [3]. Parity games have also been used to prove lower bounds in further areas of theoretical computer science such as Markov decision processes [7] and linear programming [8].

Beyond all this, parity games are very interesting in their own right, and their algorithmic solutions can both be complex and quite surprising.

Chapter 2

Upper bounds: two families of algorithms

Current algorithms to solve parity games can be divided into 2 families. In historical order, first came the attractor decomposition algorithms, then what we'll call the separation algorithms. This distinction is interesting since it allows for a better analysis of lower bounds in chapter 3.

2.1 Attractor decomposition algorithms

2.1.1 Zielonka's algorithm

Zielonka's algorithm [9] was the first algorithm devised to solve parity games and, though it has exponential lower bounds on certain games [10, 11, 12], it is usually the best algorithm to solve parity games in practice [13]. It consists in a divide and conquer approach which requires a novel notion to be understood, that of attractors,

Definition 2.1.1 (Attractor set)**.** For a given parity game graph G, and set $S \subseteq V$, the *attractor set* of S for player P, noted: $\mathrm{Attr}_P(G, S)$ is the set of all vertices in G from which P can force a play to S. More formally, $\mathrm{Attr}_P(G, S)$ is a fixed point of the following equations:

- $A_P^0(S) = S$
- $A_P^{k+1}(S) = A_P^k(S) \cup \{u \in V_P \mid \exists v \in A_P^k(S) : (u, v) \in E\} \cup \{u \in V_{\overline{P}} \mid \forall v \; (u, v) \in E \implies v \in S\}$

Intuitively, if v is owned by P and v has an outgoing edge that goes into S, then P can force the play to go to S simply by taking that edge, thus $v \in \mathrm{Attr}_P(G, S)$. On the other hand, if v' is owned by $\overline{P}$, then P can force the play to go to S only if all outgoing edges from v' go into S, that is, if $\overline{P}$ can't escape.

We also state 2 simple but useful lemmas about attractors

Lemma 2.1.1. The graph obtained by removing the attractor for P of a set of vertices is a trap for P.

The proof of this lemma is quite direct since, if it were not a trap, e.g. if P could escape through vertex v in the remaining graph, then the attractor should contain v, a contradiction.

A corollary of this lemma is the fact that, when we remove an attractor, the remaining graph is still a well-defined parity game where all vertices have at least one outgoing edge.

Lemma 2.1.2. Given a parity game G and a set $S \subseteq V$, if W is a dominion for P in $G \setminus \text{Attr}_{\overline{P}}(G, S)$ then W is also a dominion for P in G

The reasoning behind this lemma is quite similar to that of lemma 2.1.1, indeed, since W is a dominion for P in $G \setminus \text{Attr}_{\overline{P}}(G, S)$, then $\overline{P}$ can't attract P out of W, even in G.

We can now state Zielonka's algorithm.

Algorithm 1: Zielonka's Algorithm

```
procedure $\textsc{Solve}_E(G, h)$:
    // $h$ is an even upper bound for priorities in G
    $N_h \leftarrow \{v \in G | \pi(v) = h\}$
    $H \leftarrow G \setminus \textsc{Attr}_E(G, N_h)$
    $W_O \leftarrow \textsc{Solve}_O(H, h-1)$
    if $W_O = \emptyset$ then
        return $G$
    else
        $G \leftarrow G \setminus \textsc{Attr}_O(G, W_O)$
        return $\textsc{Solve}_E(G, h)$
```

To fully define the algorithm, we also need to define $\textsc{Solve}_O(G, h)$, it is the exact same as $\textsc{Solve}_E(G, h)$ with the roles of Even and Odd switched.

Proof Sketch. If we consider $\textsc{Solve}_E$, in the first part of the algorithm, we remove all vertices that have maximal priority h in G as well as their even attractor. We then recursively solve the game for H, there are then two cases:

1. If $W_O = \emptyset$, we note $A = \textsc{Attr}_E(G, N_h)$.
   Any play will either:

   - Go finitely many times in A then go to H where Even has a winning strategy, thus Even wins.

- Stay indefinitely in A in which case Even will always play the strategy defined by the attractor to go to a vertex with priority h. Thus, the play will visit infinitely many times a node with priority h and since h is even and maximal in G by the algorithm assumptions, Even wins.

In this case, Even wins on the entire game graph G and we return G.

2. If $W_O \neq \emptyset$, then by lemma 2.1.2, W_O is also an odd winning region in G and thus, $\textsc{Attr}_O(G, W_O)$ is also winning for Odd in G. We then simply recursively call $\textsc{Solve}_E$ on the remaining graph $G \setminus \textsc{Attr}_O(G, W_O)$ which is strictly smaller than G.

2.1.2 Quasipolynomial time attractor decomposition algorithms

After the breakthrough of the first quasipolynomial-time algorithm by Calude et al. [17], several other new quasipolynomial time algorithms were published in quick succession. One of them is a variation on the standard Zielonka's algorithm found by Paweł Parys [14]. The main intuition behind his algorithm is the following: in a given parity game with n vertices, there can only be at most one dominion of size larger than $\frac{n}{2}$. The reason why this observation is useful is that it is computationally simpler to find small dominions than large ones.

Lehtinen, Schewe and Wojtczak [15] then modified Parys' algorithm to match the complexity of other quasipolynomial-time algorithms from the separation algorithms family. We will present an improved version of that algorithm, the so-called "Liverpool algorithm" [16] since it is more succint and elegant.

Algorithm 2: The Liverpool algorithm

```
procedure $\textsc{Solve}_E(G, h, p_E, p_O)$:
    if $G = \emptyset$ or $p_O \leq 1$ then
        return $G$
    $G_1 := \textsc{Solve}_E(G, h, p_E, \lfloor p_O/2 \rfloor)$
    $N_h := \{v \in G_1 | \pi(v) = h\}$
    $H := G_1 \setminus \textsc{Attr}_E(N_h, G_1)$
    $W_O := \textsc{Solve}_O(H, h-1, p_O, p_E)$
    $G_2 := G_1 \setminus \textsc{Attr}_O(W_O, G_1)$
    $G_3 := \textsc{Solve}_E(G_2, h, p_E, \lfloor p_O/2 \rfloor)$
    return $G_3$
```

As in Zielonka's algorithm, the procedure $\textsc{Solve}_O$ is defined by switching the roles of Even and Odd in the algorithm.

We won't give a full proof of correctness of the algorithm but will try to explain the main idea behind it. The *precision parameters* p_E and p_O are the only additional parameters with respect to Zielonka's algorithm. They are used as the implementation of the observation that we made above — the fact that there can only be at most one dominion of size larger or equal to $\frac{n}{2}$ — as can be seen by the following lemma:

Lemma 2.1.3. A call to $SolveE(G, h, p_E, p_O)$ where h is even and not smaller than the maximal priority in G, returns a set that:

1. Contains all of Even's dominions in G of size up to p_E

2. Doesn't intersect with any of Odd's dominions in G of size up to p_O

This new algorithm thus separates all small even winning regions from all small odd winning regions. To solve a parity game, it then suffices to call $SolveE(G, h, n, n)$ which, by lemma 2.1.3 will return a set containing all even dominions of size up to n that doesn't intersect with any odd dominion of size up to n, that is, it will return the entire even winning region.

For the algorithm in itself, using lemma 2.1.3 we can now explain some intuition about how it works.

Lines 1 and 2 describe the base cases. Then, line 3 will return a set G_1 that contains all even dominions of any size, that don't intersect odd dominions of size up to $\frac{n}{2}$. G_1 thus contains the all of the even dominions but could also contain a large odd dominion since the induction hypothesis doesn't say anything about the intersection of the returned set with odd dominions of size larger than $\frac{n}{2}$.

This explains the second step (line 5), or the "full precision call", as in Zielonka's algorithm, we remove the attractor of the highest priority nodes then call the procedure $\text{S}{\scriptstyle\text{OLVE}}_O$ with the entire p_E and p_O precision parameters. If there was a large odd winning region in G_1, it will be contained in W_O.

We need a third recursive descent (line 8) since removing an even attractor (line 5) may actually turn odd dominions into even dominions, the third recursive call removes such odd dominions. Nevertheless, we don't need a full precision call since we know that there are no odd dominions of size larger than $\frac{n}{2}$ (if there was one, it was removed in the second recursive descent).

2.2 Separation algorithms

Calude, Jain, Khoussainov, Li, and Stephan [17] improved the upper bound for solving parity games down to $\mathcal{O}(n^{\log h})$. From there on followed a series of other quasipolynomial-time algorithms [18, 19, 14]. We won't go into detail about the different constructions since they all use specific techniques, instead, we introduce separation automata since all of the known quasipolynomial algorithms (except the attractor decomposition algorithm we just presented) can actually be reframed as reductions to separation automata.

Definition 2.2.1 (Safety game)**.** A *safety game* is a two-player game G where, as in parity games, each player owns a set of vertices. The players here are called player 0 and player 1. The winning condition for player 0 on a safety game is to never enter one of the rejecting states R. It is much simpler to solve than a parity game (there is no notion of priorities on vertices) and solving it only requires computing $\textsc{Attr}_O(G, F)$ which is computable in $\mathcal{O}(|E|)$.

Definition 2.2.2 (Deterministic safety automaton)**.** A deterministic safety automaton $D = (Q, \Sigma, q_0, \delta, R)$ has the same syntax as a DFA with 2 key differences in its semantics:

- A safety automaton accepts a word if it never reaches a rejecting state (a state in R), as if we took the complement of a DFA's accepting states and added sinks to every rejecting state.
- A safety automaton takes as input infinite words, thus it accepts a word w if none of the finite prefixes of w reaches a rejecting state.

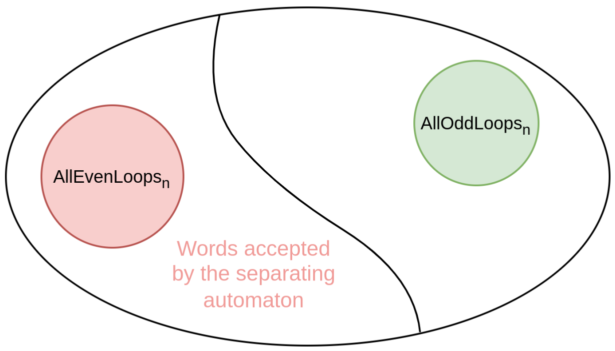

Figure 2.1: The separation regions for an (n, h)-separating automaton

We can now define separating automata for parity games, these are safety automata that are defined as follows,

Definition 2.2.3 ((n, h)-separating automata)**.** An (n, h)-separating automaton $\mathcal{A}_{n,h}$ is defined for any parity game G with n vertices and maximum priority h. It is a safety automaton defined on the language $\{0, 1, \ldots, h\}^\omega$ which, you may have noticed, is nearly the exact language a *run* on any such G is defined on, with the only difference being that we don't care about which vertices were encountered, only their priorities. Thus words taken as input to such automata can consequently be considered as runs on any such G.

For a safety automaton to be a separating automaton, it should accept all runs where all loops are even, and reject all runs where all loops are odd.

Some notations:

- We write $\textsc{LimSupEven}$ to describe all runs such that the lim sup is even, that is to say, Even wins that run, we similarly define $\textsc{LimSupOdd}$.

- We write AllEvenLoops$_n$ to denote all the runs on graphs of size at most n that only contain even loops, and similarly define AllOddLoops$_n$.

Remark. The above definition requires separating automata to *separate* runs in AllEvenLoops$_n$ from runs in AllOddLoops$_n$ (see figure 2.1), for runs with some even loops and some odd loops, the automaton could either accept or reject them.

To clarify this notion of separating automata, we will now give an example of a relatively simple instance.

Example 2.2.1. We define $\mathcal{C}_{n,h}$, a simple (n,h)-separating automaton. Assuming h is even (the case where h is odd is very similar), the set of states of $\mathcal{C}_{n,h}$ is the set of sequences $\langle c_1, c_3, \ldots, c_{h-1}\rangle$ where every c_i can take values in $\{1,\ldots,n\}$, as well as a REJECT state. The automaton is then defined as follows:

- The starting state is $\langle n, n, \ldots, n\rangle$
- $\delta(\text{REJECT}, p) = \text{REJECT}$ for any input priority p
- if p is odd, $\delta(\langle c_1, c_3, \ldots, c_{h-1}\rangle, p) =$
  - $\langle n, \ldots, n, c_p - 1, \ldots c_{h-1}\rangle$ if $c_p > 1$
  - REJECT if $c_p = 1$
- $\delta(\langle c_1, c_3, \ldots, c_{h-1}\rangle, p) = \langle n, \ldots, n, c_{p+1}, \ldots c_{h-1}\rangle$ if p is even and $c_p > 0$

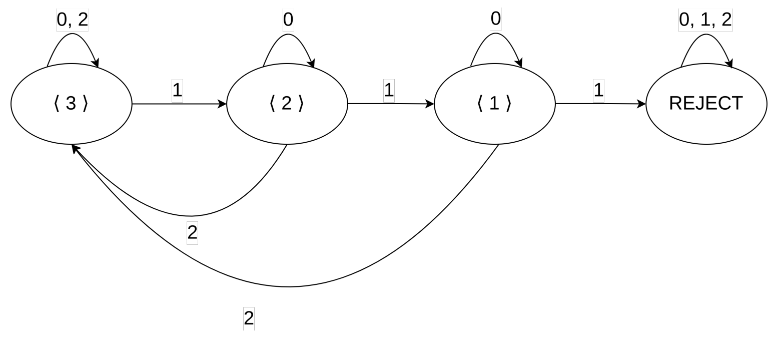

Figure 2.2: The $\mathcal{C}_{3,2}$ separating automaton

<u>Claim</u> $\mathcal{C}_{n,h}$ is an (n,h)-separating automaton.

Proof Sketch. To see this, we simply have to observe that $\mathcal{C}_{n,h}$ basically acts as a multi-counter that only rejects if it has encountered n times the same odd priority p in some substring of the input, without encountering any higher priority in between. Since there are only n vertices in G, there is thus a cycle with maximum priority p with p odd. This shows that the automaton only rejects inputs that contain at least an odd cycle, that is, all inputs won by Odd.

It is clear by construction that any word with all odd loops would be rejected, consequently, $\mathcal{C}_{n,h}$ is indeed an (n,h)-separating automaton.

Remark. $\mathcal{C}_{n,h}$ is actually a bit stronger than an (n, h)-separating automaton since not only does it reject all inputs that contain all even loops but it actually rejects all inputs that are won by Odd — it separates $\textsc{AllEvenLoops}_n$ from $\textsc{LimSupOdd}$. This is an instance of what are called *strong* (n, h)-separators [21].

To tie all these notions together, we present the following lemma,

Lemma 2.2.1. Let $\mathcal{A}$ be an (n, h)-separating automaton and $|\mathcal{A}|$ be its number of states. Then any parity game G with n vertices and maximum priority h can be solved in

$$\mathcal{O}(n \times |\mathcal{A}| + \text{time to compute } \mathcal{A})$$

Proof Sketch. From a parity game $G = (G.V, G.V_O, G.V_E, G.E, \pi)$ and $\mathcal{A} = (Q, \Sigma, q_0, \delta, R)$, an (n, h)-separating automaton, we can construct the *product game* $PG = G \times \mathcal{A}$ where every vertex in $PG.V$ is a pair (v, c) for $v \in G.V$ and $c \in Q$. Edges in PG are defined as follows:
$((v, c), (v', c')) \in PG.E$ if $(v, v') \in G.E$ and $\delta(c, \pi(v)) = c'$

By considering states $(v, \textsc{reject})$ as rejecting states, PG is a safety game which gives us the solution to the parity game G. To show this, we will call *starting states* all states in PG of the form (v, q_o) for $v \in G.V$. We then just have to look at which of these starting states in PG are losing for player 0, we call them $(v_0^L, q_0), \dots, (v_k^L, q_0)$, the winning region for Odd in G is $W_O = \{v_0^L, \dots, v_k^L\}$. This is because, by the definition of our separation automata, starting from (v_i^L, q_0) player 0 can't avoid following an odd cycle which, in turn, means that Odd has a winning strategy in G starting from v_i.

The time bound is given directly by the fact that, as stated earlier, safety games with m states can be solved in $\mathcal{O}(m)$ time.

To make this *product game* construction a bit clearer, figure 2.4 is an example of the product game of a simple parity game 2.3 with $n = 3$ and $h = 2$, using the same $(3, 2)$-separation automaton as earlier (figure 2.2).

In orange are what we called the starting states, and it is easily noticeable that none of these starting states are losing for player 0. This directly matches with the fact that Even wins on the entire graph G.

Remark. Lemma 2.2.1 allows us to conclude that the separating automaton we built in example 2.2.1 enables us to solve parity games in $\mathcal{O}(n^{\frac{h}{2}+1})$ (since the automaton has $\mathcal{O}(n^{\frac{h}{2}})$ states). This is still exponential in n but the automaton we defined was rather simple since it just counted the number of priorities seen.

In fact, all quasipolynomial-time algorithms of what we call the "separation algorithms family" either implicitly or explicitly define separating automata of quasipolynomial size [20, 21].

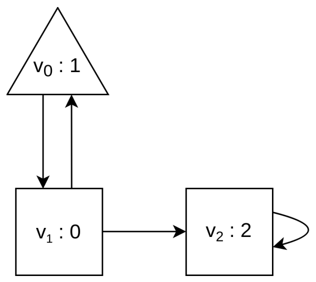

Figure 2.3: A simple parity game G

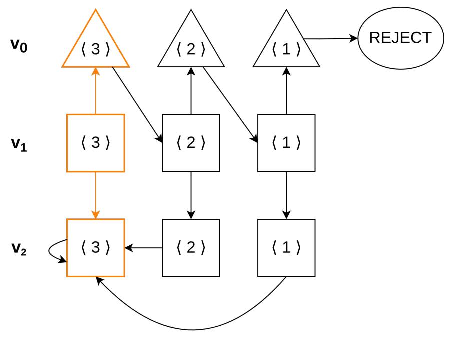

Figure 2.4: The product game of G and $\mathcal{C}_{3,2}$

Chapter 3

Lower Bounds

Before talking about lower bounds for parity game algorithms, we first have to introduce a structure called a *universal tree* that underlies all the current lower bound proofs for parity game algorithms.

3.1 Universal Trees

Definition 3.1.1 ((n, h)-ordered tree)**.** We define an (n, h) ordered tree as a tree with unbound degree, *n leaves*, and *depth h*. By *totally ordered*, we mean that every node's children are totally ordered. Additionally, we require that all leaves have the same depth h.

We also define the *size* of a tree to be its number of leaves.

Definition 3.1.2 (Tree embeddings)**.** A tree $\mathcal{T}$ *embeds* a tree $\mathcal{T}'$ if, by removing some branches from $\mathcal{T}$, we obtain $\mathcal{T}'$.

As an example, by removing the red branches from T_2 in figure 3.1.1, we obtain T_1, thus T_2 embeds T_1

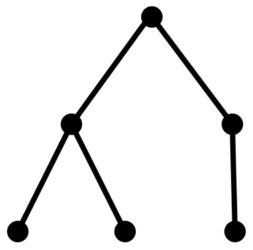
Figure 3.1: A $(3, 2)$-ordered tree T_1

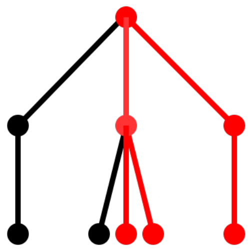
Figure 3.2: A tree embedding of T_1 in T_2

With these notions in hand, we can already define universal trees,

Definition 3.1.3 ((n, h)-universal tree)**.** A tree is (n, h)-universal if it embeds all (n, h)-trees.

Example 3.1.1. One trivial example of a universal tree is simply the complete tree (for example, the complete $(3, 2)$-universal tree 3.3). We can nevertheless observe that the complete tree seems "wasteful" when compared to T_2 (figure) which is still a universal tree but of a smaller size. In fact, complete universal trees always have an exponential n^h size.

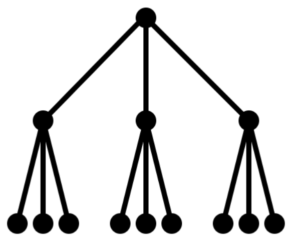

Figure 3.3: The $(3, 2)$-complete universal tree

An interesting question is then, how to construct small universal trees?

Theorem 3.1.1. For all positive integers n, h, there exists an (n, h)-universal tree $\mathcal{T}$ of size $|\mathcal{T}| = n^{\mathcal{O}(\log(h))}$ [21]

Construction We define this universal tree $S_{n,h}$ recursively with the function $t(n, h)$ where $t(n, h)$ returns an (n, h)-universal tree of quasipolynomial size:

1. Base case:
   - $t(n, 1)$ is a root with n leaves
   - $t(1, h)$ is a succession of h branches (a "line" of depth h)
2. Induction case: $t(n, h) = t(\lfloor \frac{n}{2} \rfloor, h) \cup t(n, h - 1) \cup t(n - 1 - \lfloor \frac{n}{2} \rfloor, h)$ where the union means that the roots of all the subtrees are merged

Remark. The universal tree T_2 is actually an $S_{3,2}$ tree.

Remark. You may have noticed that the inductive calls in the construction of the $S_{n,h}$ universal tree are very reminiscent of the structure of the recursive calls in the Liverpool algorithm. This hints at some link between attractor decomposition algorithms and universal trees that we will begin to explore in section 3.3.

Universal trees wouldn't be very useful to prove lower bounds if we didn't have any lower bounds on their size, thankfully, that is not the case,

Theorem 3.1.2. For all positive integers n, h such that $2h \leq n$, (n, h)-universal trees have size at least $n^{\log(h/\log n)-1}$ [21].

Although there is a gap between the lower bound and the upper bound, it is quite small and in fact, the lower bound still implies that all universal trees have quasipolynomial size, a fact that is central to the lower bound arguments made on separation algorithms.

3.2 Lower bounds for separation algorithms

We won't go into detail about the proof since it is rather technical and requires additional knowledge, we will nonetheless state the interesting results that are shown in [21].

Theorem 3.2.1. For every strong (n, h)-separating automaton $\mathcal{A}_{n,h}$, there is an injective mapping of the leaves of an $(n, h/2)$-universal tree into the states of $\mathcal{A}_{n,h}$.

A direct corollary of this theorem is that it forms a barrier to all parity games algorithms that either implicitly or explicitly construct a strong separating automaton since their running time is lower bounded by $n^{\log(h/\log(n))-2}$, a quasipolynomial term.

Remark. Theorem 3.2.1 is only concerned with *strong* separating automata, that is, automata that separate $\textsc{AllEvenLoops}_n$ from $\textsc{LimSupOdd}$, it doesn't impose a barrier on automata that only separate $\textsc{AllEvenLoops}_n$ from $\textsc{AllOddLoops}_n$, though, in practice, it seems quite hard to create an automaton that only separates both with a small number of states.

We will try to give some intuition as to why universal trees would have anything to do with separating automata.

At first glance, these structures seem unrelated, universal trees are purely combinatorial constructions, with little to no relationship to runs on parity games (they don't even have a notion of priorities), while separating automata are directly defined on those runs. The unifying notion here is that of *progress measures.* Vardi [22] described them as follow:

> A *progress measure* is a mapping on program states that quantifies how close each state is to satisfying a property about infinite computations. On every program transition the progress measure must change in a way ensuring that the computation converges toward the property.

As such, the automaton $\mathcal{C}_{n,h}$ defined in example 2.2.1 is an example of a progress measure, at each step, we come closer to knowing if there are odd cycles in the run.

The link between universal trees and separating automata relies on the observation by Jurdziński and Lazić [20] that a progress measure on a game graph with n vertices and at most h distinct edge priorities is a mapping from the vertices in the game graph to nodes in an ordered tree of height at most $d/2$ and with at most n leaves.

3.3 Lower bounds for attractor decomposition algorithms

There are currently no lower bound for attractor decomposition algorithms. A first step in the direction of proving a lower bound for such algorithms is to have a general notion of an attractor decomposition algorithm which enables us to tackle the entire family of algorithms.

3.3.1 The universal attractor decomposition algorithm

Such a generalisation of attractor decomposition algorithms was done in the paper "A Universal Attractor Decomposition Algorithm for Parity Games" [23]. The authors define the following algorithm that generalises all known attractor decomposition algorithms.

Algorithm 3: The universal algorithm

```
procedure Univ_Even($\mathcal{G}, h, \mathcal{T}^{\text{Even}}, \mathcal{T}^{\text{Odd}}$):
    let $\mathcal{T}^{\text{Odd}} = \langle \mathcal{T}_1^{\text{Odd}}, \mathcal{T}_2^{\text{Odd}}, \ldots, \mathcal{T}_k^{\text{Odd}} \rangle$
    $\mathcal{G}_1 \leftarrow \mathcal{G}$
    for $i \leftarrow 1$ to $k$ do
        $D_i \leftarrow \{v \in \mathcal{G}_i | \pi(v) = h\}$
        $\mathcal{G}'_i \leftarrow \mathcal{G}_i \setminus \text{Attr}^{\mathcal{G}_i}_{\text{Even}}(D_i)$
        $U_i \leftarrow$ Univ_Odd$\left(\mathcal{G}'_i, h-1, \mathcal{T}^{\text{Even}}, \mathcal{T}_i^{\text{Odd}}\right)$
        $\mathcal{G}_{i+1} \leftarrow \mathcal{G}_i \setminus \text{Attr}^{\mathcal{G}_i}_{\text{Odd}}(U_i)$
    return $V^{\mathcal{G}_{k+1}}$
```

The parameters $\mathcal{T}^{\text{Even}}$ and $\mathcal{T}^{\text{Odd}}$ can be seen as generalisations of the parameters p_E and p_O from the Liverpool algorithm. Indeed, both $\mathcal{T}^{\text{Even}}$ and $\mathcal{T}^{\text{Odd}}$ are ordered trees that define where in the recursion tree to cut branches (since, for example if at one point in the recursion, $\mathcal{T}^{\text{Odd}}$ is just a root, then $\text{Univ}_{\text{Even}}$ will stop directly and return the entire graph, in a similar fashion to the case where $p_O \leq 1$ in the Liverpool algorithm). They *generalise* this notion of cutting branches since there are no constraints on these tree parameters.

This relationship between trees and generalisations of attractor decomposition algorithms goes even further, indeed, in their paper, Jurdziński and Morvan show the following results,

Proposition 1. The universal algorithm performs the same actions and produces the same output as Zielonka's algorithm if it is run on an (n, h)-parity game with both trees $\mathcal{T}^{\text{Even}}$ and $\mathcal{T}^{\text{Odd}}$ equal to a $(n, h/2)$-complete universal tree (as defined in example 3.1.1) and if it uses the adaptive empty-set early termination rule.

Proposition 2. The universal algorithm performs the same actions and produces the same output as the Liverpool algorithm if it is run on an (n, h)-parity game with both trees $\mathcal{T}^{\text{Even}}$ and $\mathcal{T}^{\text{Odd}}$ equal to the $S_{n,h/2}$-complete universal tree (as defined in the construction of theorem 3.1.1) and if it uses the adaptive empty-set early termination rule.

The part about the adaptive empty-set early termination rule just is just due to the fact that in the recursive algorithms as we defined them, if we once encounter an empty winning region then the algorithm can return immediately without continuing on with further recursive calls which isn't the case in the universal algorithm. This nonetheless doesn't change the analysis in any important way.

These propositions show us that this universal algorithm is clearly a good way to encompass the known attractor decomposition algorithms.

The main theorem proved in this paper shows an even more striking link, relating the tree parameters to the correctness of the algorithm,

Theorem 3.3.1. The universal algorithm is correct on a parity game G if it is run on ordered trees $\mathcal{T}^{\text{Even}}$ and $\mathcal{T}^{\text{Odd}}$ that are $(n, h/2)$-universal trees.

We knew the theorem to be correct in the instances where those universal trees were the $(n, h/2)$-complete tree or the $S_{n,h/2}$ universal tree since both Zielonka's algorithm and the Liverpool algorithm are correct, nevertheless, this theorem extends this observation to any $(n, h/2)$-universal trees.

Unfortunately, this doesn't suffice to prove a lower bound on attractor decomposition trees, we would indeed need theorem 3.3.1 to be an if and only if to form a barrier similar to the one on separation algorithms.

3.3.2 An open question

Trying to tackle a lower bound on the size of the tree parameters for the universal algorithm's isn't as easy as arguing that, if for example $\mathcal{T}^{\text{Odd}}$ isn't universal, then the returned winning region for Even is incorrect. This statement is actually wrong. Indeed, if the graph we give as parameter to $\text{Univ}_{\text{Even}}$ is entirely won by Even and if we give as parameter $\mathcal{T}^{\text{Odd}} = \langle\rangle$ (only a root), then, by following the algorithm, it directly returns V^G which, for this instance of G, is actually correct since Even wins on the entire graph.

We obviously want to avoid trivial cases like this one as well as all other cases where the algorithm just cuts of a branch at some point and was lucky that the entire subgraph was winning. To do this, we want to force the universal algorithms to "certify themselves". The winning regions are no valid certificate since there is no known way of verifying that winning regions are correct in polynomial time. An interesting alternative is to make the universal algorithm construct the winning strategy as it goes through recursive calls, indeed, unlike winning regions, a strategy is verifiable in polynomial time. More importantly, there is no way to "fake" a strategy, if the algorithm cuts off at some point where it shouldn't have, even if the winning region is correct, it won't be able to actually define the strategy on that winning region since it returned directly.

To do this, we need to describe a way to enrich the universal algorithm to make it output a strategy along with the winning region and that is what is done by the following algorithm

Algorithm 4: The universal algorithm, with strategies

```
procedure Univ_Even($\mathcal{G}, h, \mathcal{T}^{\text{Even}}, \mathcal{T}^{\text{Odd}}$):
    let $\sigma \leftarrow \sigma_\emptyset$
    let $\mathcal{T}^{\text{Odd}} = \langle \mathcal{T}_1^{\text{Odd}}, \mathcal{T}_2^{\text{Odd}}, \ldots, \mathcal{T}_k^{\text{Odd}} \rangle$
    $\mathcal{G}_1 \leftarrow \mathcal{G}$
    for $i \leftarrow 1$ to $k$ do
        $D_i \leftarrow \{v \in \mathcal{G}_i | \pi(v) = h\}$
        $\text{Attr} \leftarrow \textsc{Attr}^{\mathcal{G}_i}_{\text{Even}}(\sigma, D_i)$
        $\mathcal{G}'_i \leftarrow \mathcal{G}_i \setminus \text{Attr}$
        $U_i, \sigma' \leftarrow$ Univ_Odd $\left(\mathcal{G}'_i, h-1, \mathcal{T}^{\text{Even}}, \mathcal{T}_i^{\text{Odd}}\right)$
        $\sigma \leftarrow \sigma \cup \sigma'$
        if $U_i = \emptyset$ then
            for $v$ in $\{v' \in \mathcal{G}_i | \pi(v') = h\}$ do
                /* Adds a random strategy on the nodes with
                   maximal priority                                  */
                $\sigma.v \leftarrow \text{random.choice}(v.\text{out}())$
        else
            /* The winning region for Odd is non-empty, we
               reset the strategy on the attractor Attr of
               maximum priorities                                    */
            for $v$ in Attr.vertices() do
                $\sigma.v \leftarrow -1$
        $\mathcal{G}_{i+1} \leftarrow \mathcal{G}_i \setminus \textsc{Attr}^{\mathcal{G}_i}_{\text{Odd}}(\sigma, U_i)$
    return $V^{\mathcal{G}_{k+1}}, \sigma$
```

We will briefly explain how the strategy is constructed, considering $\text{Univ}_{\text{Even}}$ since the case for Univ_{Odd} is symmetric. As a reminder, we work with memoryless strategies so a strategy is simply a mapping $\sigma : V \to V \cup \{-1\}$ where $\{-1\}$ designates the empty strategy.

At the beginning of the procedure, the strategy is set to the empty strategy (all vertices map to $\{-1\}$).

Then, when we compute the attractor for D_i, we also save the strategy defined by that attractor since it is obtainable by construction of the attractor. We then recursively call the procedure and combine both strategies as they are not defined on the same set of vertices, the original σ is defined on Attr and σ' is defined on $\mathcal{G} \setminus \text{Attr}$.

There are then two options:

1. $U_i = \emptyset$ thus Even wins on the entire graph with the strategy: on $\mathcal{G}'_i$, play strategy σ' since it was winning in the recursive call and on the rest of

the graph, play the strategy defined by Attr since it will always bring you to a node of highest priority. On the nodes of highest priority, take any strategy since you win everywhere else. Note that this is the same strategy as described in the proof sketch for Zielonka's algorithm.

2. $U_i \neq \emptyset$, Odd has a non-empty winning region and thus the strategy defined on the attractor may actually be losing for Even, we just reset it and only keep the strategy given by the recursive call.

Now that we have this self-certifying definition of the algorithm, we will hopefully be able to start using the structure of the universal algorithm to prove a lower bound on the time complexity of attractor decomposition algorithms. The substantial advantage of this method is that we now only require exhibiting a single graph that would, for example, require a quasipolynomial number of recursive steps to construct its winning strategy. Some candidates for such a graph are graphs that were used as exponential lower bounds for Zielonka's algorithm [10, 11, 12] since they seem to be inherently hard for attractor decomposition algorithms (they also experimentally seem to require a quasipolynomial number of recursive calls for the Liverpool algorithm).

Bibliography

[1] E. A. Emerson and C. S. Jutla. Tree automata, mu- calculus and determinacy. In FOCS, pages 368–377, 1991.

[2] A. W. Mostowski. Games with forbidden positions. Technical Report 78, Uniwersytet Gdańnski, 1991.

[3] E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model- checking for fragments of μ-calculus. In CAV, pages 385–396, 1993.

[4] M. Jurdziński. Deciding the winner in parity games is in UP $\cap$ co-UP. Inf. Process. Lett., 68(3):119–124, 1998.

[5] C. Daskalakis and Ch. Papadimitriou. Continuous local search. In SODA, pages 790–804, 2011.

[6] C. Daskalakis, Ch. Tzamos, and M. Zampetakis. A converse to Banach's fixed point theorem and its CLS completeness. In STOC, pages 44–50, 2018.

[7] J. Fearnley. Exponential lower bounds for policy iteration. In ICALP, pages 551–562, 2010.

[8] O. Friedmann, T. D. Hansen, and U. Zwick. Subexpo- nential lower bounds for randomized pivoting rules for the simplex algorithm. In STOC, pages 283–292, 2011.

[9] Wiesław Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theor. Comput. Sci., 200(1-2):135–183, 1998.

[10] Oliver Friedmann. Recursive algorithm for parity games requires exponential time. RAIRO - Theor. Inf. and Applic., 45(4):449–457, 2011.

[11] Massimo Benerecetti, Daniele Dell'Erba, and Fabio Mogavero. Robust exponential worst cases for divide-et-impera algorithms for parity games. In Patricia Bouyer, Andrea Orlandini, and Pierluigi San Pietro, editors, Proceedings Eighth International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2017, Roma, Italy, 20-22 September 2017., volume 256 of EPTCS, pages 121–135, 2017.

[12] Maciej Gazda. Fixpoint Logic, Games, and Relations of Consequence. PhD thesis, Eindhoven University of Technology, 2016. URL: https://pure.tue.nl/ws/files/16681817/20160315_ Gazda.pdf.

[13] Tom van Dijk. Oink: An implementation and evaluation of modern parity game solvers. In Dirk Beyer and Marieke Huisman, editors, Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I, volume 10805 of Lecture Notes in Computer Science, pages 291–308. Springer, 2018.

[14] Paweł Parys. Parity games: Zielonka's algorithm in quasi-polynomial time. In Peter Rossmanith, Pinar Heggernes, and Joost-Pieter Katoen, editors, 44th International Symposium on Mathe- matical Foundations of Computer Science, MFCS 2019, August 26-30, 2019, Aachen, Germany, volume 138 of LIPIcs, pages 10:1–10:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[15] Karoliina Lehtinen, Sven Schewe, and Dominik Wojtczak. Improving the complexity of Parys' recursive algorithm. CoRR, abs/1904.11810, 2019.

[16] Lehtinen, K., Parys, P., Schewe, S., Wojtczak, D. (2021). A Recursive Approach to Solving Parity Games in Quasipolynomial Time. CoRR, abs/2104.09717.

[17] C. S. Calude, S. Jain, B. Khoussainov, W. Li, and F. Stephan. Deciding parity games in quasipolynomial time. In STOC, pages 252–263, 2017.

[18] M. Jurdziński and R. Lazić. Succinct progress mea- sures for solving parity games. In LICS, pages 1–9, 2017.

[19] K. Lehtinen. A modal μ perspective on solving parity games in quasi-polynomial time. In LICS, pages 639– 648, 2018.

[20] M. Bojańczyk and W. Czerwiński. An automata toolbox, February 2018. https://www.mimuw.edu.pl/ bojan/papers/toolbox-reduced-feb6.pdf

[21] Wojciech Czerwiński, Laure Daviaud, Nathanaël Fijalkow, Marcin Jurdziński, Ranko Lazićc, and Paweł Parys. Universal trees grow inside separating automata: Quasi-polynomial lower bounds for parity games. In Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, pages 2333–2349. SIAM, 2019.

[22] M. Y. Vardi. Rank predicates vs. progress measures in concurrent-program verifica- tion. Chicago Journal of Theoretical Computer Science, 1996. Article 1.

[23] Jurdzinski, M., Morvan, R. (2020). A Universal Attractor Decomposition Algorithm for Parity Games. CoRR, abs/2001.04333.