Rheinisch-Westfälische Technische Hochschule Aachen

Bachelor Thesis

The Recursive Algorithm for Parity Games

by

Jan-Christoph Kassing

Lehrstuhl für Informatik 7
Logik und Theorie diskreter Systeme

First Examiner:
Priv.-Doz. Dr. Christof Löding

Second Examiner:
Prof. Dr. Martin Grohe

March 2026

Rheinisch-Westfälische Technische Hochschule Aachen

Abstract

Lehrstuhl für Informatik 7
Logik und Theorie diskreter Systeme

by Jan-Christoph Kassing

Parity games are widely used throughout the areas of computer science. It is still unknown whether there exists an algorithm, which solves parity games in polynomial runtime. One of the first and simplest algorithms to solve parity games is the divide-and-conquer algorithm by Zielonka. It turns out that it has exponential worst-case complexity, but it is one of the fastest algorithms in practice. Since 2017, there are quasi-polynomial runtime algorithms as well, which are not very easy to understand and in fact, no improvement in practice. Recently, Parys proposed a new version of Zielonka's algorithm, which is very simple again, and additionally, it is quasi-polynomial as well. Sadly, it turns out that this new algorithm is slow in practice as well. In this thesis we will present both of these algorithms (The recursive one by Zielonka and the new version by Parys) and present sample families of game graphs so that we can compare the two algorithms to each other. In particular, we will present worst-case families for the recursive algorithm (i.e., the recursive algorithm by Zielonka needs exponential runtime to solve those game graphs) and explain the computation of both algorithms on them. We will see that the new version by Parys can solve those families in polynomial time.

Contents

Chapter 1

Introduction

Parity Games are two-player games of perfect information and infinite duration, played on a directed Graph (V, E) together with a labeling function $\pi : V \to \mathbb{N}$ that gives ever node a corresponding natural number, called the priority. The two players, namely *Even* and *Odd* (or just 0 and 1), move a token from node to node along the edges of the graph until an infinite path is formed. Furthermore, the set of nodes V is partitioned into the sets V_0 and V_1 to determine, which player can move the token from a certain node. Finally, the winner is determined by the parity of the highest priority, which occurs infinitely often in the path of the token. Player 0 wins if and only if the parity is even. Otherwise, player 1 takes it all.

This game has been intensively studied in theoretical computer science. The corresponding decision problem, one has to solve, is whether player 0 or player 1 wins the game if it started at a given node v_0. The fundamental role of parity games is so enormous that we cannot enumerate all of its applications in computer science here. Many problems in formal verification, synthesis and model-checking can be reformulated in terms of solving a parity game. Emerson, Jutla and Sistla have shown that the model-checking problem for the modal μ-calculus is linear time equivalent to computing winning strategies for parity games [5]. It is also noteworthy that parity games play a crucial role in automata theory [4, 14], e.g., they can be used to solve the emptiness test for nondeterministic parity automata on infinite trees [14]. The impact of parity games even reaches far areas of computer science like linear programming [8].

Let us come to the complexity theoretic point of view for parity games. It is a long-standing open question whether parity games can be solved in polynomial time. Several results show that solving a parity game belongs to classes slightly-above **P**. In particular, it was shown that this problem lies in **NP** $\cap$ **co-NP** [5] and even in **UP** $\cap$ **co-UP** [10], but whether it is contained in **P** remains illusive.

When someone talks about the algorithmic side of solving parity games, one has to mention the recursive algorithm by Zielonka [21]. This algorithm feels very simple and way more natural than any other algorithm for parity games. We will explain this algorithm in detail in chapter 3 and look at the computation of some sample families in chapter 4. For the rest of this thesis we will refer to it as the *recursive algorithm.* Sadly, its running time is exponential in the worst case [7]. There are a lot of different approaches for solving a parity game, like the Small progress Measure algorithm [11], the strategy improvement algorithm [20], the dominion decomposition algorithm [12], or the priority promotion algorithm [2]. Still, all of them have exponential runtime in the worst case. Then, there was a breakthrough with the first quasi-polynomial time algorithm proposed by Calude, Jain, Khoussainov, Li, and Stephan [3]. Shortly following more quasi-polynomial time algorithms (e.g., [15, 6]). Most of them are quite involved and not very trivial to understand. Finally, Parys created a quasi-polynomial time algorithm [17] completely based on the recursive algorithm by adding only two new precision parameters. This algorithm, not as natural as the recursive algorithm, still feels more simple than the other quasi-polynomial time algorithms. In this thesis, we want to present this new algorithm and explain its computation on some sample families. We will refer to it as the *new algorithm.* For the comparison, we will use worst-case families for the recursive algorithm. This shall illustrate the improvement caused by the precision values, which were added in the new algorithm.

Before we start, we will also mention the practical side of parity game solvers. It turns out that although the recursive algorithm is exponential in the worst-case, and there exists quasi-polynomial time algorithms, it is still one of the best performing solvers in practice. The biggest comparison of existing solvers was performed by Tom van Dijk [19]. In his Oink tool he has implemented several algorithms, with different optimizations. Then, he has evaluated them on a benchmark of Keiren [13], containing multiple parity games obtained from model-checking and equivalence checking tasks, as well as on different classes of random games. It turns out that the recursive algorithm performs the best. Sadly, the new algorithm performs only as good as the worst algorithms implemented in the Oink tool. Therefore, the algorithm is not (yet) very interesting for practice. Still, it is an incredible discovery that such a small improvement to the recursive algorithm produces a quasi-polynomial time algorithm.

In this thesis, we want to compare the recursive algorithm to the new variant. For this, we will introduce the basic notion and definition of parity games in chapter 2. Then, we will present both of the algorithms in chapter 3 and prove their correctness and their runtime bounds. In chapter 4, we will compare the algorithms on worst-case families for the recursive algorithm (i.e., the recursive algorithm needs exponential amount of time to solve the game). We will see that those sample families can be solved in polynomial time by the new algorithm. Additionally, we will see how the new algorithm solves a parity game compared to the classical version of the recursive algorithm. We will end this thesis with a conclusion, where we talk about some improvement ideas that might help the new algorithm in practice.

Chapter 2

Preliminaries

At the start of this thesis, we want to go over some basic definitions and lemmata about infinite games. We start with the notion of infinite words over an arbitrary alphabet.

Definition 2.1 (Infinite Words)**.** *Let Σ be an alphabet. We denote Σ^* as the set of all finite words over Σ and Σ^+ as the set of all non-empty finite words over Σ. The empty word will be denoted as ε.*

Further, we denote Σ^ω as the set of infinite words that have a starting point and no endpoint over Σ. Those ω-words can also be seen as a function $\alpha : \mathbb{N} \to \Sigma$. So for $\alpha \in \Sigma^\omega$ we can write $\alpha(i)$ for the letter at position i (starting from position 0).

Next, we will present the basic definition of specific kind of games that parity games belong to. We start with the basic definition of a *game graph* and a *game* in general.

Definition 2.2 (Game Graph, Play, Game)**.** *A* game graph *is a tuple $G = (V_0, V_1, E, c)$ with $V = V_0 \mathbin{\dot\cup} V_1$, where*

- *V_0 is the set of nodes for player 0 (these nodes will be represented as circles)*
- *V_1 is the set of nodes for player 1 (these nodes will be represented as squares)*
- *$E \subseteq V \times V$ is the set of directed edges such that, the set $vE = \{v' \in V \mid (v, v') \in E\}$ of v's* successors *is non-empty, for all $v \in V$. [1].*
- *$c : V \to C$ is a coloring function with a finite set of colors C.*

If G is not clear from the context, we will write $V(G)$ for the set of nodes of G, and a set of nodes $A \subseteq V$ will also be called a region.

[1]We need the set of the successors to be non-empty because otherwise there would be potentially no infinite game possible. Once we reach a vertex with no successor, the play would stop and could not go on forever.

Let $U \subseteq V$ be any region of a game graph $G = (V_0, V_1, E, c)$. The subgraph of G induced by U will be denoted as $G[U]$

$$G[U] = (V_0 \cap U, V_1 \cap U, E \cap (U \times U), c|_U)$$

where $c|_U$ is the restriction of c to U.

$G[U]$ is a subgame graph of G iff. $G[U]$ is a game graph (i.e., each node in $G[U]$ has at least one successor).

A play *in G is an infinite sequence $\alpha = v_0 v_1 v_2 \ldots$ of nodes such that $(v_i, v_{i+1}) \in E$ holds for all $i \in \mathbb{N}$. We denote the corresponding sequence of colors by $c(\alpha) = c(v_0)c(v_1)c(v_2)\ldots$.*

For an infinite sequence $c = c_0 c_1 c_2 \ldots$ we denote the set of all elements that occur at least once by $\mathrm{Occ}(c)$ and the set of all elements that occur infinitely often by $\mathrm{Inf}(c)$.

A game *is of the form $\mathfrak{G} = (G, \mathrm{Win})$ with a game graph G and a Winning condition $\mathrm{Win} \subseteq C^\omega$.*

A play *α is winning for player 0 iff. $c(\alpha) \in \mathrm{Win}$. Otherwise, player 1 wins the play.*

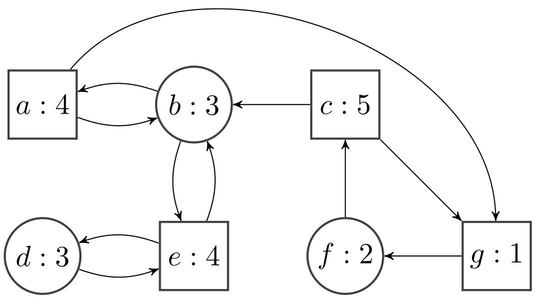

Figure 2.1: An example game graph $G = (\{b, d, f\}, \{a, c, e, g\}, E, c)$. Here, colors are given as natural numbers.

There are several kinds of winning conditions. We will briefly talk about the most important ones:

1. Safety Games:
   In a safety game, player 0 has to remain in a certain region, given by a set $S \subseteq C$ of colors, for the whole duration of the game. The corresponding winning condition is
   $$\mathrm{Win}_{\mathrm{Safety}}(S) = \{c \in C^\omega \mid \mathrm{Occ}(c) \subseteq S\}$$
2. Reachability Games:
   In a reachability game, player 0 has to reach a certain region, given by a set $S \subseteq C$ of colors. The corresponding winning condition is
   $$\mathrm{Win}_{\mathrm{Reach}}(S) = \{c \in C^\omega \mid \mathrm{Occ}(c) \cap S \neq \emptyset\}$$

3. Büchi Games:
   In a Büchi game, player 0 has to reach a node from a certain region, given by a set $S \subseteq C$ of colors, infinitely often throughout the play. The corresponding winning condition is
   $$\text{Win}_{\text{Büchi}}(S) = \{c \in C^\omega \mid \text{Inf}(c) \cap S \neq \emptyset\}$$
4. Parity Games:
   In a parity game, we have to use a more complex description of the winning condition. We will no longer only consider the color for each node but instead we will give each color a corresponding natural number $n \in \mathbb{N}$ and call it the priority of the node. This means that the coloring function is now a function $\pi : V \to \mathbb{N}$ (We will explicitly write π for a coloring function into the natural numbers $\mathbb{N}$). Player 0 wins the play iff. the highest number that occurs infinitely often in the play is even. The corresponding winning condition is
   $$\text{Win}_{\text{Parity}} = \{c \in \mathbb{N}^\omega \mid \max(\text{Inf}(c)) \text{ is even}\}$$

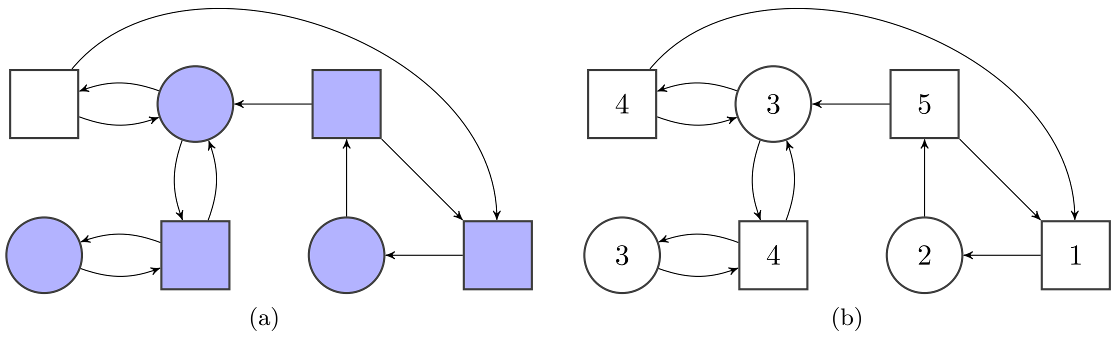

Figure 2.2: A set of two games:(a) describes a safety game "stay in blue";(b) describes a parity game. The name of the nodes are omitted here.

Most of this work only considers parity games. Therefore, we may omit the notation $\mathfrak{G} = (G, \text{Win}_{\text{Parity}})$ and just talk about the game graph G (or in short: the game G). If we want to talk about another winning condition, we will explicitly state that. Additionally, one can also describe a safety (reachability, Büchi) game by giving the region of nodes explicitly, without their corresponding colors. This is useful as we can use multiple winning conditions at once, which would otherwise require multiple colorings. Consider the game $(G, \text{Win}_{\text{Parity}} \cap \text{Win}_{\text{Safety}}(S)))$. Here, the coloring $\pi : V \to \mathbb{N}$ for the parity condition does not necessarily have something to do with the safety condition that we want to express. Therefore, we can just state the set S by the actual nodes, instead of their colors.

Next, we talk about the players in an infinite game and their strategies.

Definition 2.3 (Strategy). *A* strategy *δ for player P is a (partial) function $\delta : V^* \cdot V_P \to V$ with $\delta(xv) = v'$ implies $(v, v') \in E$ that maps every partial play that ends in a node of V_P to one of its successors.*

It is a positional strategy *iff. the choice of the successor only depends on the node we are currently in (i.e., it is a function $\delta : V_P \to V$ that maps every node to one of its successors).*

A strategy *δ for player P* agrees *with a (partial) play $v_0v_1v_2...$ iff. we have $\delta(v_0...v_i) = v_{i+1}$, whenever we have $v_i \in V_P$, for all $i \in \mathbb{N}$.*

A strategy δ is a winning strategy *for player P from a node $v \in V$ iff. every play that starts in v and agrees with δ is winning for P. Player P is winning from a node $v \in V$ if he has a winning strategy from v. We call the set $\mathrm{Win}_P(\mathfrak{G})$ of all nodes v, such that Player P is winning from v, the* winning region *for player P in the game $\mathfrak{G}$.*

A strategy δ for player P is called a uniform strategy *iff. it is a winning strategy for P from all $v \in \mathrm{Win}_P(\mathfrak{G})$.*

For a first look inside the algorithmic point of view of infinite games, we give the definition of an *attractor* of a region. These are the nodes such that player P can force to reach a region S no matter what the other player's strategy is.

Definition 2.4 (Attractor)**.** *An* attractor $\mathrm{Attr}_P(G, U)$ *for player P of a region $U \subseteq V$ in a game graph G is the smallest set such that:*

- *$U \subseteq \mathrm{Attr}_P(G, U)$*
- *if $v \in V_P$ and there exists a $v' \in vE$ such that $v' \in \mathrm{Attr}_P(G, U)$, then $v \in \mathrm{Attr}_P(G, U)$.*
- *if $v \in V_{1-P}$ and for all $v' \in vE$ we have $v' \in \mathrm{Attr}_P(G, U)$, then $v \in \mathrm{Attr}_P(G, U)$.*

Note that $\mathrm{Attr}_P(G, U)$ can be computed in polynomial time with respect to the size of the game graph G. If we delete the attractor of any given player and of any given region from a game graph, we result with a valid subgame graph.

Lemma 2.5. *Let $U \subseteq V$ be a region of a game graph G and P be a player. Then the subgraph $G[V \setminus \mathrm{Attr}_P(G, U)]$ is a subgame graph.*

Proof. We only have to show that every node $v \in V \setminus \mathrm{Attr}_P(G, U)$ has at least one successor in the game graph $G[V \setminus \mathrm{Attr}_P(G, U)]$. Assume that there is a node $v \in V \setminus \mathrm{Attr}_P(G, U)$ which has no successor in the game graph $G[V \setminus \mathrm{Attr}_P(G, U)]$. Since G is a valid game graph, we know that each node has at least one successor. Therefore, all of the successors of v must be contained in $\mathrm{Attr}_P(G, U)$. But this would imply that we have $v \in \mathrm{Attr}_P(G, U)$, which is a contradiction. □

As a counterpart to an attractor, we can define a *trap*, where player P can force to remain in a specific region U, which then *traps* player $1 - P$ inside of this region.

Definition 2.6 (Trap)**.** *Let G be a game graph and P be a player. A* trap *for player $1-P$ is a region $U \subseteq V$ such that:*

- *if $v \in V_P$ and $v \in U$, then there exists a successors $v' \in vE$ such that $v' \in U$.*
- *if $v \in V_{1-P}$ and $v \in U$, then for all of its successors $v' \in vE$ we have $v' \in U$.*

Next, we want to point out an important fact about the winning regions in a parity game and their winning strategies. Therefore, we will introduce the notion of determinacy in infinite games.

Definition 2.7 (Determinacy)**.** *A game $\mathfrak{G}$ is* determined *iff. the winner of the game $\mathfrak{G}$ is determined by the starting node (i.e., for every node either player 0 or player 1 has a winning strategy). A game $\mathfrak{G}$ is* positionally determined *iff. the winner of the game $\mathfrak{G}$ is determined by the starting node and the corresponding winning strategy is positional.*

Theorem 2.8. *Parity games are positionally determined* [21].

Proof. We will proof this by induction on the highest priority h inside the game graph.

- *Base Case*: Assume that $h = 0$. Then, we have $\text{Win}_0(G) = V$ since there are only even priorities in the game graph and thus every positional strategy for player 0 is a positional winning strategy.
- *Induction Step*: Assume that the highest priority in G is even (otherwise swap the roles of player 0 and player 1 below) and greater than 0. Let U be the set of vertices such that player 1 has a uniform positional winning strategy δ from every node in U.

  We will show that on $V \setminus U$ player 0 has a uniform positional winning strategy. Note that $G[V \setminus U]$ is a valid subgame graph. Indeed, assume that there exists a node $v \in V \setminus U$ with all of its successors inside of U. Therefore, we have to move to U from v and thus player 1 is also winning from v (and obviously has a positional winning strategy). Thus $v \in U$, which is a contradiction to $v \in V \setminus U$.

  Consider the following two cases:

  - $G' := G[V \setminus U]$ does not contain a node of the maximal priority h:
    G' contains only priorities from $\{0, ..., h-1\}$. Therefore, we can apply the induction hypothesis and partition the subgame graph into the winning regions $\text{Win}_0(G') \dot{\cup} \text{Win}_1(G')$ with respective uniform positional winning strategies. Note that, since U already contains all nodes such that player 1 has a uniform positional winning strategy, we know that $\text{Win}_1(G')$ must be empty. In the whole game G we have $\text{Win}_1(G) = U \cup \text{Win}_1(G') = U \cup \emptyset = U$ and $\text{Win}_0(G) = \text{Win}_0(G') = V \setminus U$ with corresponding uniform positional winning strategies.

- $G' := G[V \setminus U]$ contains nodes with the maximal priority h:
  The region $V \setminus U$ is a trap for player 1 and thus player 0 can guarantee that once the play is inside $V \setminus U$ it remains there. Let C_h be the set, which contains all nodes with priority h inside of $V \setminus U$. Either the play stays forever in $(V \setminus U) \setminus \mathrm{Attr}_0(G', C_h)$ at some point or it visits $\mathrm{Attr}_0(G', C_h)$ infinitely often. In the first case, we can apply the induction hypothesis again, since we reach a point, where we can ensure that no node of priority h will be reached anymore. In the second case, player 0 wins with a uniform positional winning strategy constructed by the attractor of C_h. Again, we have $\mathrm{Win}_1(G) = U$ and $\mathrm{Win}_0(G) = V \setminus U$ with corresponding uniform positional winning strategies.

All in all, we see that on $V \setminus U$ player 0 has a uniform positional winning strategy and thus parity games are positional determined.

□

By this theorem, we know that we only have to consider positional strategies when computing the winner for a parity game. Let us look at the example from above and apply our newly introduced definitions. By the determinacy of parity games, we can partition the set of nodes into the winning regions of both players: $V = \mathrm{Win}_0(G) \,\dot\cup\, \mathrm{Win}_1(G)$.

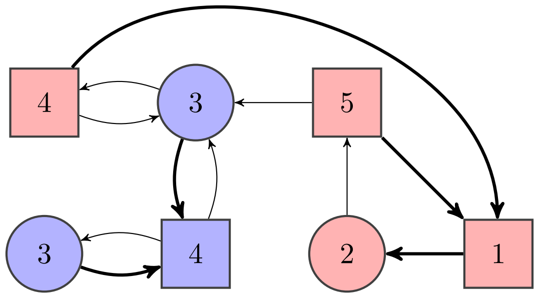

Figure 2.3: The winning region is blue for player 0 and red for player 1. The uniform positional winning strategy inside the winning region for the corresponding player is marked by thick arrows.

At the end of this chapter, we want to create a last definition about specific parts of the winning region.

Definition 2.9 (Dominion)**.** *A* dominion *for player P is a region S, such that P wins from every position $s \in S$ without leaving S (i.e., P wins the game $(G, \mathrm{Win}_{Parity} \cap \mathrm{Win}_{Safety}(S))$ from every position $s \in S$)*

Chapter 3

Recursive Algorithms

In this chapter, we will focus on the algorithmic point of view for parity games. The important question is whether player 0 or player 1 is winning from a specific node or in other words: What is the winning region for each player? For all the other kind of winning conditions we mentioned above (safety, reachability, Büchi) there already exists well known algorithms with polynomial runtime (see e.g.[18]). Yet, we only have quasi-polynomial runtime algorithms for parity games. In this thesis, we will present the recursive algorithm by Zielonka [21] and a new variant of Zielonka's algorithm that results in a better runtime complexity proposed by Parys [17]. All of the ideas in this chapter are based on those two papers.

3.1 Recursive Algorithm by Zielonka

We will start with the standard variant of Zielonka's algorithm. It has only the game graph $G = (V_0, V_1, E, \pi)$ as an input parameter and computes the winning regions for both players recursively. At the start, it evaluates player i, which corresponds to the player of the same parity as the highest priority inside the game graph. Then, it removes certain parts of the winning region of player $1 - i$ iteratively from the game until it removed the whole winning region for player $1 - i$. In the end, it returns a tuple (W_0, W_1) containing both winning regions.

Algorithm 1: Recursive Algorithm - Solve(G)

```
1  $h \leftarrow \max\{\pi(v) \mid v \in V(G)\}$;
2  $i \leftarrow h \mod 2$;
3  $(W_0, W_1) \leftarrow (\emptyset, \emptyset)$;
4  if $G = \emptyset$ then
5      return $(W_0, W_1)$;
6  repeat
7      $N_h \leftarrow \{v \in V(G) \mid \pi(v) = h\}$;
8      $H \leftarrow G[V \setminus \mathrm{Attr}_i(G, N_h)]$;
9      $(W'_0, W'_1) \leftarrow \mathrm{Solve}(H)$;
10     $A \leftarrow \mathrm{Attr}_{1-i}(G, W'_{1-i})$;
11     $G \leftarrow G[V \setminus A]$;
12     $W_{1-i} \leftarrow W_{1-i} \cup A$, $W_i \leftarrow V(G)$;
13 until $W'_{1-i} = \emptyset$;
14 return $(W_0, W_1)$;
```

Typically, one uses a version of Zielonka's algorithm that does not contain a loop (i.e., we would use a second recursive call). Here, we are using a loop instead because the new algorithm takes advantage of it. In the following, we will analyze this algorithm in detail.

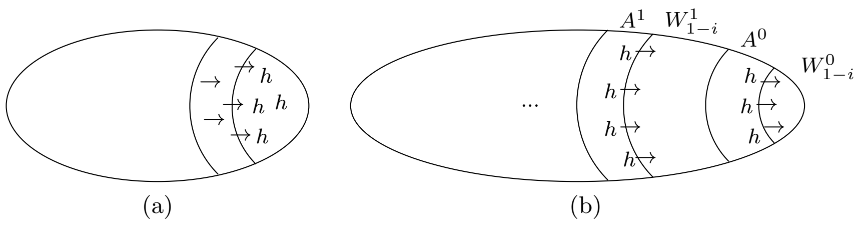

Figure 3.1: Winning regions for player i (a) and player $1 - i$ (b).

The idea of the algorithm can be understood while looking at figure 3.1. The game graph G can be partitioned into $\mathrm{Win}_i(G)$ and $\mathrm{Win}_{1-i}(G)$. Further, we can divide both winning regions into multiple parts. $\mathrm{Win}_i(G)$ can be divided into the nodes with priority h, the attractor A of nodes with priority h and $\mathrm{Win}_i(G) \setminus A$, which are the nodes, where player i can not force to see a node of priority h but still wins. $\mathrm{Win}_{1-i}(G)$ can be divided into even more parts. The first one is W^0_{1-i}, where player $1 - i$ can win without seeing a node of priority h. This region can be computed by deleting all nodes of priority h plus their attractor set for player i from the graph and then compute the winning regions for both players recursively. We can see this in line 7 to line 9 of the recursive algorithm. Then we have A^0, which are the notes that are added to W^0_{1-i}, while computing the attractor for player $1 - i$. Note that A^0 can indeed contain nodes of priority h and we may need to pass several nodes of priority h before we enter the region W^0_{1-i}

in a play that starts in A^0. If we remove $A^0 \cup W^0_{1-i}$ from the game, we will result with a valid game graph because it is an attractor set. One can see that this is exactly the set that gets deleted from the game G in the first iteration of the loop. Now, we can compute the next part W^1_{1-i} of the winning region for player $1-i$. We can do this like before (i.e., we remove all the nodes of priority h). This time, W^1_{1-i} contains all the nodes such that player $1-i$ can ensure that the play is either winning for him inside of W^1_{1-i} or enters A^0. Then we can compute A^1 and result with the region that gets deleted from the game G in the second iteration of the loop. Once player $1-i$ has an empty winning region in the subgame, we know that player i wins the whole game graph that is left.

Complexity

Let us briefly analyze the space and runtime complexity of the recursive algorithm. We will only consider the pure amount of recursive calls for the runtime, thus disregarding the runtime it needs to create an attractor set.

The space complexity is in $\mathcal{O}(n \cdot h)$, where n is the number of nodes and h is the highest priority, since the depth of the recursion is at most h and in every recursion step we have to remember some sets of nodes. The runtime is in $\mathcal{O}(n^h)$ and thus exponential in the number of the highest priority h. We know that after each iteration we delete at least one node from the game graph or stop the iteration. Once we delete a node from the game, we can be sure that this is either the first iteration or in the previous iteration we had a non-trivial attractor set A, which means that we deleted at least two nodes from the game. Therefore, we can overestimate the amount of recursive calls by n. Additionally, the highest possible occurring priority in the recursive subgames is $h-1$. Let $\text{Rec}(h)$ be the number of recursive calls, we have to do in order to solve a game graph G with a fixed $n = |V|$ and let h be the highest priority in G. Then we have $\text{Rec}(h) \leq n \cdot \text{Rec}(h-1)$, and thus it is easy to see that we have $\text{Rec}(h) \in \mathcal{O}(n^h)$.

Correctness

Now, we want to proof that the recursive algorithm is indeed correct. Because of the way the recursive algorithm is constructed, we can do this by a simple induction over the highest priority h again.

Correctness of the recursive algorithm. We will prove the correctness of the recursive algorithm by an induction over the highest priority h in the game graph G.

- *Base Case*: Assume that $h = 0$. Then clearly we have $(\mathrm{Win}_0(G), \mathrm{Win}_1(G)) = (V, \emptyset)$, since there are only even priorities in the game graph. The result of the recursive algorithm is $(V, \emptyset)$ and therefore correct.

- *Induction Step*: Assume that the highest priority h in G is even (Otherwise swap the roles of player 0 and player 1 below) and greater than 0. We will consider some execution of the procedure. By G^i, N_h^i, H^i, W_0^i and W_1^i we denote values of the variables G, N, H, W_0 and W_1 in the i-th iteration of the procedure. This means that G^{i-1} is the game graph before the i-th iteration and G^i is the game graph after the i-th iteration. The algorithm recursively deletes nodes from the game graph G and eventually results with a game graph G^m after the last iteration. We have to prove two facts here. First, we will never delete a node that belongs to the winning region of player 0. Second, once the algorithm stops, then we have $W_0^m = G^m = \mathrm{Win}_0(G)$. In the i-th iteration step, we create the new game graph as follows. Let $N_h^i = \{v \in V(G^{i-1}) \mid \pi(v) = h\}$ be the set of all nodes with the highest priority in G^{i-1}. We create $H^i = G^{i-1}[V \setminus \mathrm{Attr}_0(G^{i-1}, N_h^i)]$ without nodes of priority h. We can apply the induction hypothesis and therefore, we know that the algorithm is correct on H^i. We solve the parity game on H^i and find the winning region $\mathrm{Win}_1(H^i)$ for player 1. The set $\mathrm{Attr}_1(G^{i-1}, \mathrm{Win}_1(H^i))$ is a dominion for player 1 in G^{i-1}. We can be sure that for every node $v \in V_0 \cap \mathrm{Attr}_1(G^{i-1}, \mathrm{Win}_1(H^i))$ there are no successors outside of $\mathrm{Attr}_1(G^{i-1}, \mathrm{Win}_1(H^i))$, since we deleted the attractor of N_h^i for player 0 from the game before computing $\mathrm{Win}_1(H^i)$. We can delete the dominion from the game without loosing any nodes of $\mathrm{Win}_0(G)$. We will show this by an internal induction on i.

  - *Base Case*: Assume that $i = 1$. Then, we have $G^0 = G$ and since $\mathrm{Attr}_1(G, \mathrm{Win}_1(H^1))$ is a dominion for player 1 in G, we can be sure that $\mathrm{Attr}_1(G, \mathrm{Win}_1(H^1))$ does not contain a node from the winning region of player 0.
  - *Induction Step*: Now we prove the claim for $i+1$. $\mathrm{Attr}_1(G^i, \mathrm{Win}_1(H^{i+1}))$ is a dominion for player 1 in G^i. Player 0 can only potentially escape $\mathrm{Attr}_1(G^i, \mathrm{Win}_1(H^{i+1}))$ by moving out of $V(G^i)$ into a node of $V(G) \setminus V(G^i)$. By the induction hypothesis, we know that player 1 wins from every node in $V(G) \setminus V(G^i)$ and thus player 0 cannot win a play, that starts in $\mathrm{Attr}_1(G^i, \mathrm{Win}_1(H^{i+1}))$, as he cannot stay inside of $\mathrm{Attr}_1(G^i, \mathrm{Win}_1(H^{i+1}))$ forever, but leaving results in a winning play for player 1 as well.

To prove the second claim, note that once we stop the iteration after the m-th iteration, we have $G^{m-1} = G^m$, and we have an empty winning region for player 1 inside of H^m. Player 0 can force to reach a node of priority h on all other nodes inside of G^{m-1}. Thus, player 1 can not win by staying inside of H^m forever and can not win by moving to a node inside of $\mathrm{Attr}_0(G^{m-1}, N_h^m)$ infinitely often. All in all, player 1 can not win anywhere in the game graph $G^{m-1} = G^m$. Therefore, we have $W_0^m = G^m = \mathrm{Win}_0(G)$. $\square$

3.2 Quasi-Polynomial Time Algorithm by Parys

Let us move on to the improved version of Zielonka's algorithm and find out how we can result with a quasi-polynomial runtime. The main idea of the new algorithm is a combinatorial one. Have a look at the figure 3.1 again. While searching for the parts W^l_{1-i} of the winning region for player $1-i$, we know that they are all disjoint and all together they have a maximum size of $n := |V|$. Therefore, we have at most one region W^j_{1-i} which is bigger than $\frac{n}{2}$ and all other regions W^l_{1-i} with $j \neq l$ are smaller than $\frac{n}{2}$. We add two precision parameters p_0 and p_1 to keep track of the size of the winning regions we are looking for. The precision p_0 stands for the maximum size of winning region for player 0 that we are looking for. Equally, p_1 stands for the maximum size of winning region for player 1. Assume that $|W^j_{1-i}| \geq \frac{n}{2}$ and we have k parts in total. Then, we can search in the first $j-1$ iteration steps with decreased precision $\frac{p_{1-i}}{2}$, have one recursive call with full precision, and then search with decreased precision again. One recursive layer deeper, we can apply the same logic, either for the other player or for the same player again, if the parity of the highest priority did not change. In the algorithm, we do not know this partition of the winning region for player $1-i$ and thus, we do not know when the region W^j_{1-i}, which is bigger than $\frac{n}{2}$, occurs. Therefore, we search as long as we can find a new region W^l_{1-i}, which is non-empty, with decreased precision. Then it searches once for a winning region of full size and then again only for small ones.

You can see the algorithm in the following page. We nearly copied the recursive algorithm three times. In the first copy, we are searching with decreased precision for the winning region of player $1-i$. The precision is decreased to $\lfloor \frac{p_{1-i}}{2} \rfloor$ as we described before. When we can not find anymore winning regions for player $1-i$ with decreased precision, we go to our second copy and search once with full precision. After that, we go to the third copy and search with decreased precision again. One starts the algorithm by calling Solve$(G, |V|, |V|)$.

Due to the fact that the precision values only impact the computation, when they reach a value of 1 or lower, it is only relevant how often one can decrease them until they reach a value of 1, instead of the actual value itself. To be precise, we have the exact same behaviour for different precision values if they are in between the same powers of 2 (i.e., for precision values p and p' the following holds: If there exists an $n \in \mathbb{N}$ such that $2^n \leq p \leq p' < 2^{n+1}$ then we have the exact same behaviour in the new algorithm). Therefore, it suffices to look only at the precision values $\{2^n \mid n \in \mathbb{N}\}$ when trying to figure out how the new algorithm solves the sample game graphs in the next chapter.

Additionally, we will assume that the game graph has no self-loops. One can preprocess a game graph in polynomial time such that we delete all self-loops in an appropriate way. We take a closer look at this preprocessing at the end of this chapter.

Algorithm 2: Quasi-Polynomial-Time Algorithm - Solve(G, p_0, p_1)

```
$h \leftarrow \max\{\pi(v) \mid v \in V(G)\}$;
$i \leftarrow h \mod 2$;
$(W_0, W_1) \leftarrow (\emptyset, \emptyset)$;
if $G = \emptyset$ then
    return $(W_0, W_1)$;
if $p_0 \leq 1$ then
    return $(\emptyset, V(G))$;                 // assuming that there are no self loops
if $p_1 \leq 1$ then
    return $(V(G), \emptyset)$;                 // assuming that there are no self loops
repeat
    $N_h \leftarrow \{v \in V(G) \mid \pi(v) = h\}$;
    $H \leftarrow G[V \setminus \mathrm{Attr}_i(G, N_h)]$;
    $p'_i \leftarrow p_i$, $p'_{1-i} \leftarrow \lfloor \frac{p_{1-i}}{2} \rfloor$;          // decreasing precision for player 1-i
    $(W'_0, W'_1) \leftarrow \mathrm{Solve}(H, p'_0, p'_1)$;
    $A \leftarrow \mathrm{Attr}_{1-i}(G, W'_{1-i})$;
    $G \leftarrow G[V \setminus A]$;
    $W_{1-i} \leftarrow W_{1-i} \cup A$, $W_i \leftarrow V(G)$;
until $W'_{1-i} = \emptyset$;
$N_h \leftarrow \{v \in V(G) \mid \pi(v) = h\}$;
$H \leftarrow G[V \setminus \mathrm{Attr}_i(G, N_h)]$;
$(W'_0, W'_1) \leftarrow \mathrm{Solve}(H, p_0, p_1)$;                      // once with full precision
$A \leftarrow \mathrm{Attr}_{1-i}(G, W'_{1-i})$;
$G \leftarrow G[V \setminus A]$;
$W_{1-i} \leftarrow W_{1-i} \cup A$, $W_i \leftarrow V(G)$;
repeat
    $N_h \leftarrow \{v \in V(G) \mid \pi(v) = h\}$;
    $H \leftarrow G[V \setminus \mathrm{Attr}_i(G, N_h)]$;
    $p'_i \leftarrow p_i$, $p'_{1-i} \leftarrow \lfloor \frac{p_{1-i}}{2} \rfloor$;   // again, decreasing precision for player 1-i
    $(W'_0, W'_1) \leftarrow \mathrm{Solve}(H, p'_0, p'_1)$;
    $A \leftarrow \mathrm{Attr}_{1-i}(G, W'_{1-i})$;
    $G \leftarrow G[V \setminus A]$;
    $W_{1-i} \leftarrow W_{1-i} \cup A$, $W_i \leftarrow V(G)$;
until $W'_{1-i} = \emptyset$;
return $(W_0, W_1)$;
```

Complexity

We can see that the space complexity does not change with the new algorithm. The space complexity is still in $\mathcal{O}(n \cdot h)$, where n is the number of nodes and h is the maximal priority, since the depth of the recursion does not increase nor decrease in general in comparison to the recursive algorithm and in every recursion step we still have to remember some sets of nodes.

Now, we come to the runtime. The new algorithm was created to show that we can adapt Zielonka's algorithm just a little bit and result with a quasi-polynomial runtime. Therefore, we do not have to prove a really tight bound but just prove that the new algorithm is really quasi-polynomial.

Let $\mathrm{Rec}(h,l)$ be the number of (non-trivial) executions of the procedure Solve performed during one call to $\mathrm{Solve}(G, p_0, p_1)$ with $\lfloor \log p_0 \rfloor + \lfloor \log p_1 \rfloor = l$, and with G having at most n nodes and the highest priority in G is h. We only count non-trivial executions, which means that we ignore executions that leave the procedure at line 5, 7 or 9. We have $\mathrm{Rec}(0,l) = \mathrm{Rec}(h,0) = 0$. Looking at the algorithm, we see that we have at least one call with full precision, and like before, we can overestimate the amount of recursive calls in the iterations with decreased precision by n. All in all, for all $h, l \geq 1$ holds that

$$\mathrm{Rec}(h,l) \leq 1 + n \cdot \mathrm{Rec}(h-1, l-1) + \mathrm{Rec}(h-1, l)$$

Using this inequation, we can prove that $\mathrm{Rec}(h,l) \leq n^l \cdot \binom{h+l}{l} - 1$ holds.

Proof. We will proof this new inequation by induction on the lexicographic order of (k,l)

- *Base Case*: Assume that $k = 0$ or $l = 0$. We have:

$$\mathrm{Rec}(k,l) = \mathrm{Rec}(k,0) = \mathrm{Rec}(0,l) = 0 = 1 - 1 \leq n^l \cdot \binom{k+l}{l} - 1$$

- *Induction Step*: For $k, l \geq 1$ we have:

$$\begin{aligned}
\mathrm{Rec}(k,l) &\leq 1 + n \cdot \mathrm{Rec}(h-1, l-1) + \mathrm{Rec}(h-1, l) \\
&\overset{\text{IH}}{\leq} 1 + n \cdot \left(n^{l-1} \cdot \binom{k-1+l-1}{l-1} - 1 \right) + n^l \cdot \binom{k-1+l}{l} - 1 \\
&= 1 + n^l \cdot \binom{k-1+l-1}{l-1} - n + n^l \cdot \binom{k-1+l}{l} - 1 \\
&= n^l \cdot \left(\binom{k-1+l-1}{l-1} + \binom{k-1+l}{l} \right) - n \\
&\leq n^l \cdot \left(\binom{k-1+l-1}{l-1} + \binom{k-1+l}{l} \right) - 1 \\
&\leq n^l \cdot \left(\binom{k-1+l}{l-1} + \binom{k-1+l}{l} \right) - 1 \\
&= n^l \cdot \binom{h+l}{l} - 1
\end{aligned}$$

□

Using this new inequation, we know that $\mathrm{Rec}(k,l) \leq n^l \cdot \binom{h+l}{l} - 1 \leq n^l \cdot (h+l)^l$ and since we started with $l = 2 \cdot \lfloor \log n \rfloor$, we see that $\mathrm{Rec}(k,l)$ is quasi-polynomial in n and h.

Correctness

Now, we justify the correctness of the new algorithm. To do so, we prove the following statement:

Theorem 3.1. *Let G be a game graph. Assume that the highest priority inside of the game graph G has parity i. The procedure* $\mathrm{Solve}(G, p_0, p_1)$ *returns a tuple (W_0, W_1) with $V(G) = W_0 \mathbin{\dot\cup} W_1$, such that for every $S \subseteq V$ we have*

- *if S is a dominion for player i, and $|S| \leq p_i$, then $S \subseteq W_i$*
- *if S is a dominion for player $1-i$, and $|S| \leq p_{1-i}$, then $S \cap W_i = \emptyset$*

Note that there may be nodes in G that do not belong to any dominion smaller than p_0 or p_1. For those nodes we do not specify where they are. Here, we use the notion of a dominion in the theorem, because it is much more suitable for induction than a statement about the whole winning region.

Before we prove this, we look at two observations that we will use to prove Theorem 3.1.

Lemma 3.2. *If S is a dominion for player P in a game graph G, and $U \subseteq V$ is a region of G, then $S \setminus \mathrm{Attr}_P(G,U)$ is a dominion for P in $G[V \setminus \mathrm{Attr}_P(G,U)]$*

Proof. Denote $S' = S \setminus \mathrm{Attr}_P(G,U)$ and $G' = G[V \setminus \mathrm{Attr}_P(G,U)]$. By definition, player P wins the parity game from every node $v \in S$ in G with a strategy such that the play will never leave S. The same strategy is valid in G' because every node of player P that remains in G' has the same successors in G' as in G (if some of the successors of v belongs to $\mathrm{Attr}_P(G,U)$, then v also belongs to $\mathrm{Attr}_P(G,U)$). Therefore, Using the same strategy, player P will win the parity game in G' from every node $v \in S'$ such that the play will never leave S'. $\square$

Lemma 3.3. *If S is a dominion for player P in a game graph G, and $U \subseteq V$ is a region of G such that $S \cap U = \emptyset$, then S is a dominion for P in $G[V \setminus \mathrm{Attr}_{1-P}(G,U)]$.*

Proof. Denote $G' = G[V \setminus \mathrm{Attr}_{1-P}(G,U)]$. Suppose that there is some $v \in S \cap \mathrm{Attr}_{1-P}(G,U)$. On the one hand, P can guarantee that, while starting from v, the play stays in S. On the other hand, $1-P$ can force to reach the set U, which is disjoint from S. Thus such a node v could not exist, and we have $S \subseteq V'$. Since S stays the same in G', we can use the same strategy to win from a node $v \in S$ in G' as in G. $\square$

Finally, we are ready to prove theorem 3.1:

Proof. We will prove the theorem by induction on the highest priority h.

- *Base Case*: Assume that $h = 0$. Then clearly we have $(\mathrm{Win}_0(G), \mathrm{Win}_1(G)) = (V, \emptyset)$ since there are only even priorities in the game graph. The result of the recursive algorithm is $(V, \emptyset)$ and therefore the theorem holds for all p_0 and p_1.

- *Induction Step*: Assume that the highest priority h in G is even (Otherwise swap the roles of player 0 and player 1 below) and greater than 0. We will consider some execution of the procedure. By G^i, N_h^i, H^i, W_0^i and W_1^i we denote values of the variables G, N, H, W_0 and W_1 in the i-th iteration of the procedure. This means that G^{i-1} is the game graph before the i-th iteration and G^i is the game graph after the i-th iteration. In particular, G^0 is equal to the original game graph G, and at the end of a loop we return $(V(G^m), V(G) \setminus V(G^m))$, where m is the amount of recursive calls to Solve. By $W_1^{H^i}$ we will denote the returned region for player 1 by the recursive call in the i-th iteration. Further, let k be the number of the recursive call to Solve with full precision in line 21 (calls number $1, ..., k-1$ are performed in line 14 and calls number $k+1, ..., m$ are performed in line 29). Additionally, we assume that $p_1 > 1$ because otherwise the claim follows directly, as we would return $(V(G), \emptyset)$.

We will tackle each part of the theorem individually. We start by proving the first part concerning the dominions of player 0. Fix a dominion S for player 0 in G such that $|S| \leq p_0$. Assume that $S \neq \emptyset$. Note that a non-empty dominion has at least two nodes because there are no self loops in G. Therefore, we have $V(G) \neq \emptyset$ and $p_0 > 1$ and thus we won't return before the first loop. Now we need to prove that $S \subseteq V(G^m)$.

Actually, we prove a more general result by proving $S \subseteq V(G^i)$ for all $0 \leq i \leq m$. We show this by an internal induction on i.

– *Base Case*: Assume that $i = 0$. Then, we have $G^i = G^0 = G$ and obviously we have $S \subseteq V(G)$.

– *Induction Step*: Now we prove the claim for $i + 1$. By the induction hypothesis, we know that S is a dominion for player 0 in G^i. Let us take a closer look at the set $S^{i+1} = S \cap H^{i+1} = S \setminus \mathrm{Attr}_0(G^i, N_h^{i+1})$. By lemma 3.2 we know that S^{i+1} is also a dominion for player 0 in $H^{i+1} = G^i[V \setminus \mathrm{Attr}_0(G^i, N_h^{i+1})]$, and obviously we have $|S^{i+1}| \leq |S| \leq p_0$. By the assumption of the external induction, which we can apply to the recursive call, it follows that $S^{i+1} \cap W_1^{H^i} = \emptyset$, and thus $S \cap W_1^{H^i} = \emptyset$. Thus by lemma 3.3 we know that S is a dominion for player 0 in $G^{i+1} = G^i[V \setminus \mathrm{Attr}_1(G^i, W_1^{H^i})]$

All in all, we have shown the first part of the theorem.

Now, we will go to the second part concerning the dominions for player 1. Fix a dominion S for player 1 in G such that $|S| \leq p_1$. Note that we have $p_1 > 1$ and we will not return before the loop. Denote $S^i = S \cap V(G^i)$. First, we will prove that S^i is a dominion for player 1 in G^i. Again, we do an internal induction over i.

– *Base Case*: Assume that $i = 0$. Then, we have $G^i = G^0 = G$ and thus $S^i = S \cap V(G^i) = S$ is a dominion by assumption.

– *Induction Step*: Now we prove the claim for $i + 1$. By the induction hypothesis, we know that S^i is a dominion for player 0 in G^i. By definition we have $G^{i+1} = G^i[V \setminus \mathrm{Attr}_1(G^i, W_1^{H^i})]$ and $S^{i+1} = S^i \setminus \mathrm{Attr}_1(G^i, W_1^{H^i})$. Thus by lemma 3.2 we know that S^{i+1} is a dominion for player 1 in G^{i+1}

For $0 \leq i < m$, let Z^i be the set of nodes in $S^i \setminus N_h^{i+1}$ from which player 1 wins the game $(G^i, \mathrm{Win}_{\mathrm{Parity}} \cap \mathrm{Win}_{\mathrm{Safety}}(S^i \setminus N_h^{i+1}))$ (that is the set of nodes, where player 1 wins without seeing a node of the highest priority h). Remember that the G^i and S^i denote the values of G and S before the i-th recursive call to Solve. Now, we will prove some statements about Z^i and after that we can conclude the second part of the theorem.

(1) If $S^i \neq \emptyset$ then $Z^i \neq \emptyset$.

Suppose that $Z^i = \emptyset$ and consider a strategy for player 1 that allows him to win the game $(G^i, \mathrm{Win}_{\mathrm{Parity}} \cap \mathrm{Win}_{\mathrm{Safety}}(S^i))$ from some node $v_0 \in S^i$. Because $v_0 \notin Z^i$, this strategy is not winning from v_0 in the game $(G^i, \mathrm{Win}_{\mathsf{Parity}} \cap \mathrm{Win}_{\mathsf{Safety}}(S^i \setminus N_h^{i+1}))$. Therefore, player 0 can reach a node $v_1 \in N_h^{i+1}$ (as he cannot violate the parity condition nor leave S^i). For the same reason, because $v_1 \notin Z^i$, player 0 can continue and reach a node $v_2 \in N_h^{i+1}$. Repeating this forever, player 0 reaches infinitely many nodes of priority h, which is even and the highest priority in G^i. This is a contradiction to the assumption that the strategy was winning for player 1. Thus, we have shown that (1) holds.

From nodes of Z^i, player 1 can actually win the game $(G^i, \mathrm{Win}_{\mathsf{Parity}} \cap \mathrm{Win}_{\mathsf{Safety}}(Z^i))$ using the strategy that allows him to win the game $(G^i, \mathrm{Win}_{\mathsf{Parity}} \cap \mathrm{Win}_{\mathsf{Safety}}(S^i \setminus N_h^{i+1}))$. Indeed, if a play following this strategy enters some node v, then from this node v player 1 can still win the game $(G^i, \mathrm{Win}_{\mathsf{Parity}} \cap \mathrm{Win}_{\mathsf{Safety}}(S^i \setminus N_h^{i+1}))$, which means that theses nodes belong to Z^i. It follows that Z^i is a dominion for player 1 in G^i and by lemma 3.3 and $Z^i \cap N_h^{i+1} = \emptyset$ we know that Z^i is also a dominion for player 1 in the game graph $H^{i+1} = G^i[V \setminus \mathrm{Attr}_0(G, N_h^{i+1})]$.

Recall that $W_1^{H^i}$ is the set returned by the call $\mathrm{Solve}_1(H^i, p_1^i, p_0)$, where $p_1^k = p_1$ and $p_1^i = \lfloor \frac{p_1}{2} \rfloor$ for all $i \neq k$. From the assumption of the external induction , we know that, (2) if $|Z^i| \leq \lfloor \frac{p_1}{2} \rfloor$ or if $i = k-1$ we obtain $Z^i \subseteq W_1^{H^{i+1}}$.

We now prove that $|S^k| \leq \lfloor \frac{p_1}{2} \rfloor$. If $S^{k-2} = \emptyset$ then we know by $S^k \subseteq S^{k-1} \subseteq S^{k-2}$ that the statement holds. Suppose that $S^{k-2} \neq \emptyset$. Then we have $Z^{k-2} \neq \emptyset$ by (1). On the other hand, $W_1^{H^{k-1}} = \emptyset$, because we are just about to leave the first loop. By (2), if $|Z^{k-2}| \leq \lfloor \frac{p_1}{2} \rfloor$, then $Z^{k-2} \subseteq W_1^{H^{k-1}} = \emptyset$, which does not hold in our case. Thus we have $|Z^{k-2}| > \lfloor \frac{p_1}{2} \rfloor$. Because $W_1^{H^{k-1}} = \emptyset$, we simply have $G^{k-1} = G^{k-2}$, $S^{k-1} = S^{k-2}$ and $Z^{k-1} = Z^{k-2}$. Using (2) for $i = k-1$, we obtain $Z^{k-1} \subseteq W_1^{H^k}$, and because $S^k = S^{k-1} \setminus \mathrm{Attr}_1(G^{k-1}, W_1^{H^k}) \subseteq S^{k-1} \setminus W_1^{H^k} \subseteq S^{k-1} \setminus Z^{k-1}$ we obtain that $|S^k| \leq |S^{k-1}| - |Z^{k-1}| \leq p_1 - (\lfloor \frac{p_1}{2} \rfloor + 1) \leq \lfloor \frac{p_1}{2} \rfloor$, as initially claimed.

We have $S^{m-1} \subseteq S^k$ (our procedure only removes nodes from the game) and $Z^{m-1} \subseteq S^{m-1}$, so $|Z^{m-1}| \leq \lfloor \frac{p_1}{2} \rfloor$ by the above paragraph, and also $Z^{m-1} \subseteq W_1^{H^m}$. Because after the m-th call to Solve the procedure ends, we have $W_1^{H^m} = \emptyset$, so also $Z^{m-1} = \emptyset$, and thus $S^{m-1} = \emptyset$ by (1). We have $S^m \subseteq S^{m-1} = \emptyset$, so $S^m = S \cap V(G^m) = \emptyset$. This completes the proof, since $W_0 = V(G^m)$, $W_1 = V(G) \setminus W_0$ and (W_0, W_1) is returned by the procedure.

□

In the end, we prove the correctness of the new algorithm by theorem 3.1.

Correctness of the new algorithm. Let G be a game graph. Assume that the highest priority inside of the game graph G has parity i. Let (W_0, W_1) be the tuple that was returned by procedure $\mathrm{Solve}(G, |V|, |V|)$. We have to prove that $\mathrm{Win}_i(G)$ is a dominion for player i.

Assume that there exists a winning play α for player i such that α starts in a node of $\text{Win}_i(G)$ and leaves $\text{Win}_i(G)$ at least once. Thus, we will reach a node $v \in \text{Win}_{1-i}(G)$. When we reach this node, player $1-i$ can play a winning strategy from v and thus wins the whole play α. This is a contradiction to the assumption that α was winning for player i. Therefore, we will never leave $\text{Win}_i(G)$ in a winning play for player i.

We have $\text{Win}_i(G) \leq |V|$, and therefore, by theorem 3.1 we know that $\text{Win}_i(G) \subseteq W_i$. $\text{Win}_{1-i}(G)$ is a dominion for player $1-i$ and we have $\text{Win}_{1-i}(G) \leq |V|$. Therefore, by theorem 3.1 we know that $\text{Win}_{1-i}(G) \cap W_i = \emptyset$. All in all, we have $(W_0, W_1) = (\text{Win}_0(G), \text{Win}_1(G))$. $\square$

Preprocessing Self-Loops

Now, we will shortly explain how we can delete self-loops in a parity game. The idea for this preprocessing comes from [9]. Let $G = (V_0, V_1, E, \pi)$ be a game graph. Suppose that there is a node v such that $(v, v) \in E$. Then, there are two cases depending on the node's owner P and the priority $\pi(v)$ of v.

If $P \neq \pi(v) \mod 2$ then taking the edge (v, v) is always a bad choice for player P and this edge can be removed from the game. If v is it's only successor (i.e., $vE = \{v\}$) then by entering v we know that player $1-P$ wins the play. Therefore, we can delete the attractor $\text{Attr}_{1-P}(G, \{v\})$ from the game and add $\text{Attr}_{1-P}(G, \{v\})$ to the winning region of player $1-P$.

If $P = \pi(v) \mod 2$ then taking the edge (v, v) infinitely often results in a winning play for player P. Like before, we can delete the attractor $\text{Attr}_P(G, \{v\})$ from the game and add $\text{Attr}_P(G, \{v\})$ to the winning region of player P.

After these steps, we result with a game graph that has no self loops together with some preprocessed winning regions. This preprocessing can be done in polynomial time.

Chapter 4

Comparison on Worst-Case Families

In this chapter, we will compare the two algorithms on certain sample game graphs. We will present worst-case families $(G_n)_{n\in\mathbb{N}}$ and $(H_n)_{n\in\mathbb{N}}$ for the recursive algorithm (i.e., the size of the game graph G_n is in $\mathcal{O}(n)$ and the runtime of the recursive algorithm is exponential in n). For both of these families, we will introduce them first. Then, we will go into more detail, how the recursive algorithm solves these families, and why it needs an exponential amount of time to do so. In the end, we will prove that the new algorithm solves these families in polynomial time. We start with a simple family created by O. Friedmann [7] and then we have another, more advanced family created by Massimo Benerecetti, Daniele Dell'Erba, and Fabio Mogavero [1] that additionally prevents the help of a memorization technique. The second family forms the backbone for an entire class of worst-case families for the recursive algorithm. But as all those families inside of this class have basically the same behaviour, it suffices to look at this backbone family for the work of this thesis. Most of this chapter is based on those two papers. We will increase the work of these papers by applying the new algorithm on these families.

4.1 Simple Worst-Case Family

The main idea of the first family of game graphs is that the recursive algorithm has to solve the game graphs G_{i-1} and G_{i-2} independently in order to solve the game graph G_i. Therefore, we can use the Fibonacci sequence as a lower bound for the runtime and it is a well known fact that the Fibonacci sequence rises exponentially fast. The idea for this family comes from [7].

We will denote the first family by $G_n = (V_{0,n}, V_{1,n}, E_n, \pi_n)$. The set of nodes $V_n = V_{0,n} \dot\cup V_{1,n}$ is

$$V_n = \{a_1, ..., a_n, b_1, ..., b_n, c_0, ..., c_{n-1}, d_0, ..., d_{n-1}, e_0, ..., e_{n-1}\}$$

The complete game graph is described in the following table:

G_n			
Node	Player	Priority	Successors
a_i	$1-(i \mod 2)$	$1-(i \mod 2)$	$\{b_i, d_{i-1}\}$
b_i	$i \mod 2$	$1-(i \mod 2)$	$\{a_i\} \cup (\{c_i\} \cap V_n)$
c_i	$1-(i \mod 2)$	$3i+5$	$\{b_{i+1}, d_i\}$
d_i	$i \mod 2$	$3i+4$	$\{e_i\} \cup (\{d_{i-1}, d_{i+1}\} \cap V_n)$
e_i	$1-(i \mod 2)$	$3i+3$	$\{b_{i+1}, d_i\}$

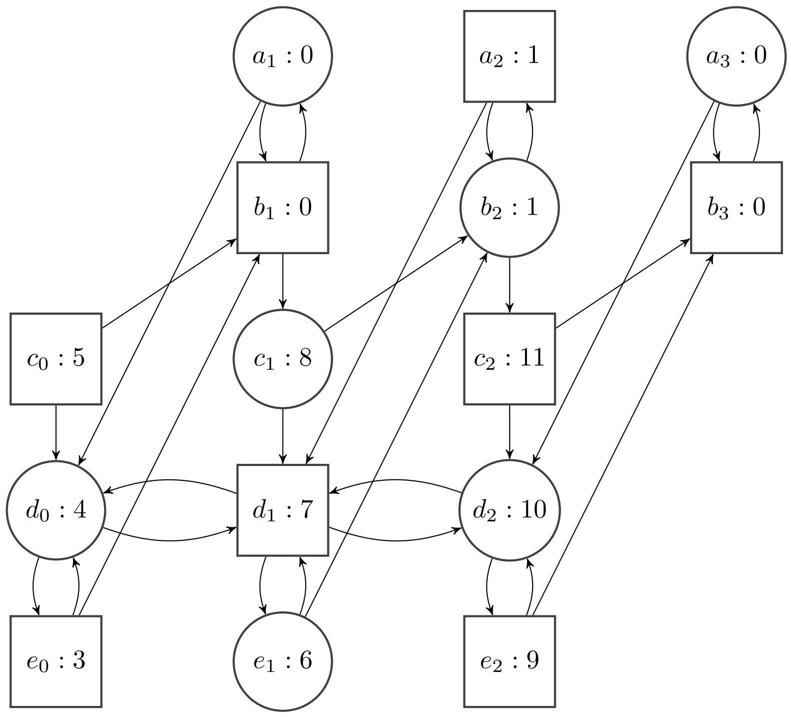

Figure 4.1: The game G_3

We start the analysis by observing that G_n is completely won by player $1-(n \mod 2)$.

Theorem 4.1. *The Game G_n is completely won by player $1-(n \mod 2)$*

Proof. We will proof this by induction on n.

- *Base Case*: It is easy to see that G_1 is won by player 0 and that G_2 is won by player 1.
- *Induction Step*: Let $n > 2$ and $i = 1-(n \mod 2)$. We know by the induction hypothesis that G_{n-2} is completely won by player i. Now attach the following strategy to the winning strategy for player i on G_{n-2}: $\delta(a_n) = b_n, \delta(b_{n-1}) = c_{n-1}, \delta(d_{n-1}) = e_{n-1}$ and $\delta(c_{n-2}) = \delta(e_{n-2}) = b_{n-1}$. Obviously, a_n and b_n are won by player i because they have the priority i and it is a trap for player $1-i$. Hence, $b_{n-1}, c_{n-1}, d_{n-1}, e_{n-1}, c_{n-2}$ and e_{n-2} are won by player i as well, as they are contained in the i-attractor of $\{a_n, b_n\}$. Therefore, G_{n-2} is

still won by player i, as moving out of G_{n-2} still results in a win for player i. Hence, a_{n-1} and d_{n-2} are also won by player i as we cannot stay inside those two nodes only.

$\square$

Now, we will move on to the observation how both of the algorithms solve each game of the family. We start with the recursive algorithm and prove the exponential runtime.

The Recursive Algorithm

As we already said, the amount of recursive calls grows as fast as the Fibonacci sequence. To be more specific, the Fibonacci sequence is a function $F : \mathbb{N} \to \mathbb{N}$ which is recursively defined by:

$$F(0) = 0$$
$$F(1) = 1$$
$$F(n+2) = F(n+1) + F(n)$$

We know that $F \in \Omega((\frac{1+\sqrt{5}}{2})^n)$, and since $\frac{1+\sqrt{5}}{2} > 1$ this implies that the Fibonacci sequence has exponential asymptotic behavior.

In order to prove that the amount of recursive calls we need to do to solve G_n using the recursive algorithm is at least as high as the Fibonacci sequence, we shall characterize a suitable induced subgame tree for this family. This tree completely describes the workflow of the recursive algorithm on the graphs G_n and uses the input parameter of the procedure as nodes. Starting from the root, which contains the original game graph G_n, we use the node G' as a child for the node G iff. the procedure Solve(G') is non-trivial and it was called recursively in Solve(G). Describing computations by a tree will be a highly used theme in this chapter. Later on, we will create an induced subgame tree for the new algorithm and use it to analyze the runtime as well. Note that we have the precision values as additional parameters in the procedure for the new algorithm and thus we will include the precision values in the nodes of the induced subgame tree for the new algorithm as well.

We will start by proving some facts about the computation of the recursive algorithm before we will create the induced subgame tree and prove the final argument.

Lemma 4.2. *For any index $n \in \mathbb{N}$ with $n > 2$ the following properties hold, where $G'_n = G_n[V \setminus \{c_{n-1}\}]$ and $G''_n = G_n[V \setminus \{a_n, b_n, b_{n-1}, c_{n-1}, c_{n-2}, d_{n-1}, e_{n-1}, e_{n-2}\}]$:*

- *In order to solve G_n, we have to solve G'_n and G''_n independently.*
- *In order to solve G'_n, we have to solve G_{n-1}.*
- *In order to solve G''_n, we have to solve G_{n-2}.*

And thus G_n requires the games G_{n-1} and G_{n-2} to be solved independently within the computation of the recursive algorithm.

Proof. Let $n > 2$ and $i = n \mod 2$. We will go through the computation of the recursive algorithm on the game graph G_n and show all of the properties at once.

In order to solve G_n, we need to solve G'_n:
The highest priority in G_n is $h = 3n + 2$, solely due to $U = \{c_{n-1}\}$, and its parity is i. The i-attractor for U is $A = U$, because the only node leading into U, namely b_{n-1}, is owned by player $1 - i$ and has more than one outgoing edge. Thus, we have to solve $G'_n = G_n[V \setminus A]$ in the first iteration of the loop.

In order to solve G'_n, we need to solve G_{n-1}:
The highest priority in G'_n is $h = 3n + 1$, solely due to $U' = \{d_{n-1}\}$, and its parity is $1 - i$. The $(1 - i)$-attractor for U' is $A' = \{a_n, b_n, d_{n-1}, e_{n-1}\}$, because the only node leading into A', namely d_{n-2}, is owned by player i and has more than one edge. Now, we have to solve $G_{n-1} = G'_n[V \setminus A']$.

In order to solve G_n, we need to solve G''_n:
Due to theorem 4.1 we know that G_{n-1} is completely won by player i, and A' is obviously won by player $1 - i$. Thus we can partition the game G' into the winning regions $W'_i = G_{n-1}$ and $W'_{1-i} = A'$.

We compute the $(1 - i)$-attractor of W'_{1-i} w.r.t. G_n, which is $B = A' \cup \{b_{n-1}, c_{n-1}, c_{n-2}, e_{n-2}\}$. Now, we have to solve $G''_n = G_n[V \setminus B]$ in the second iteration of the loop which means that we need to solve G'_n and G''_n independently in order to solve G_n.

In order to solve G''_n, we need to solve G_{n-2}:
The highest priority in G''_n is $h = 3n - 2$, solely due to $U'' = \{d_{n-2}\}$, and its parity is i. The i-attractor for U'' is $A'' = \{a_{n-1}, d_{n-2}\}$, because the only node leading into A'' is d_{n-3}, is owned by player $1 - i$ and has more than one edge. Thus, we have to solve $G_{n-2} = G''_n[V \setminus A'']$ and this completes the proof. □

Additionally, it is easy to see that we have no other non-trivial recursive calls before we reach the game graphs G_{n-1} and G_{n-2}. Now, we can define the induced subgame tree accordingly to the computation of the recursive algorithm on the game graph G_n.

Definition 4.3 ($\mathrm{IST}_{\mathrm{rec}}(G_n)$)**.** *The* induced subgame tree $\mathrm{IST}_{rec}(G_n)$ *for the recursive algorithm w.r.t. an index n is defined inductively by the following diagram:*

$$\begin{tikzcd}
 & G_n \arrow[dl] \arrow[dr] & \\
G'_n \arrow[d] & & G''_n \arrow[d] \\
G_{n-1} & & G_{n-2}
\end{tikzcd}$$

Here, we have $G'_n = G_n[V \setminus \{c_{n-1}\}]$ *and* $G''_n = G_n[V \setminus \{a_n, b_n, b_{n-1}, c_{n-1}, c_{n-2}, d_{n-1}, e_{n-1}, e_{n-2}\}]$. *The game graphs G_1 and G_2 form the leaves of the tree.*

A visualisation of the induced subgame tree can be found in the appendix in figure A.1 and A.2. There we have the induced subgame tree of G_5 and a detailed version of G_3.

Finally, we can use the fact that G_n requires the games G_{n-1} and G_{n-2} to be solved independently to prove the exponential runtime bound. One can already see, that the induced subgame tree forms kind of a binary tree, which indicates that the runtime has to be exponential.

Theorem 4.4. *Let* $\mathrm{Rec}(G_n)$ *be the number of non-trivial executions of the procedure* Solve *performed during one call to* $\mathrm{Solve}(G_n)$. *For all $n > 0$, it holds that* $\mathrm{Rec}(G_n) \geq F(n)$.

Proof. We will proof this by induction on n.

- *Base Case*: It is easy to see that G_1 and G_2 needs at least one non-trivial recursive call.
- *Induction Step*: Let $n > 2$.

  By lemma 4.2 we know that $\mathrm{Rec}(G_n) \geq \mathrm{Rec}(G_{n-1}) + \mathrm{Rec}(G_{n-2})$ and thus by the induction hypothesis we have

$$\mathrm{Rec}(G_n) \geq \mathrm{Rec}(G_{n-1}) + \mathrm{Rec}(G_{n-2}) \overset{\mathrm{IH}}{\geq} F(n-1) + F(n-2) = F(n)$$

□

The New Algorithm

So we know that the recursive algorithm needs an exponential amount of time to solve the game G_n. How does the introduction of the precision parameters impact the computation? In this section, we will answer this question. Again, our goal is to define a suitable induced subgame tree, namely $\mathrm{IST}_{\mathrm{new}}(G_n)$, for the new algorithm such that we can estimate the size of the tree. We will develop the precise definition over the rest of this section. To do so, we explain the computation of the new algorithm with different precision values and look at different *types* of procedures. These *types* are solely defined by the amount of non-trivial recursive calls they need to do before they can return. For the recursive algorithm there were two different types of procedures: a linear type for the procedures $\mathrm{Solve}(G'_n)$ and $\mathrm{Solve}(G''_n)$ and a binary type for the procedure $\mathrm{Solve}(G_n)$. In the new algorithm there is a third type of procedure that needs to do more than two recursive calls. Later, we will create a full type characterization for all possible procedures and then, we will use this characterization to make the definition of the $\mathrm{IST}_{\mathrm{new}}(G_n)$ precise. After this definition, we can estimate the size properly.

The computation still only relies on solving the games G_n, G'_n and G''_n. Therefore, we will take a closer look at the procedures for each of those game graphs with different precision values. To ease readability, we will make an additional definition and some new notations.

Definition 4.5 (valid)**.** *Let G be a game graph. A procedure* $\mathrm{Solve}(G, p_0, p_1)$ *with precision values p_0 and p_1 is* valid *if and only if we have* $\mathrm{Solve}(G, p_0, p_1) = \mathrm{Solve}(G)$ *(i.e., the procedure returns the correct winning regions for the game graph G).*

Additionally, let G, G' and G'' be game graphs such that G' and G'' are subgame graphs of G. We may write $G - G'$ for the graph $G[V(G) \setminus V(G')]$ and $G' + G''$ for the graph $G[V(G') \cup V(G'')]$. Note that we need a super graph for the addition. This super game graph will always be G_n in the rest of this section.

In the following, we will proof that already the call $\mathrm{Solve}(G_n, 4, 4)$ is valid for n even. The same property holds for all $p_0 \geq 4$ and $p_1 \geq 4$ and can be proven analogue. Before we can prove this, we will need some additional information about the computed winning regions with lower precision values. This information also shows how we have to build the induced subgame tree and what types of procedures there are. We start with the procedures of minimal precision values greater than 1.

Lemma 4.6. *Let $n \in \mathbb{N}$ be the index with $n \geq 4$. Assume that n is even, then the following properties hold:*

1. $\mathrm{Solve}(G_n, 2, 2) = (\emptyset, V(G_n))$.

2. $\mathrm{Solve}(G'_n, 2, 2) = (V(G_{n-1} - G''_{n-1}), V(G''_{n-1} + (G'_n - G_{n-1})))$.

3. $\mathrm{Solve}(G''_n, 2, 2) = (\emptyset, V(G''_n))$.

If n is odd, the following properties hold:

4. $\mathrm{Solve}(G_n, 2, 2) = (V(G_n - G''_n), V(G''_n))$.

5. $\mathrm{Solve}(G'_n, 2, 2) = (V(G'_n - G_{n-1}), V(G_{n-1}))$.

6. $\mathrm{Solve}(G''_n, 2, 2) = (V(G''_n - G''_{n-2}), V(G''_{n-2}))$.

Proof. We will prove this by induction on the index n but we will only consider the induction step for n even as the rest of the proof is completely analogue.

- *Induction Step(s)*:
  Let $n \geq 6$ and assume that n is even. The new algorithm with precision values $p_0 = 2$ and $p_1 = 2$ works as the recursive algorithm without the loop. We have one recursive call with full precision and all the other recursive calls are trivial as we have $p_0 = 1$ or $p_1 = 1$. Therefore, we can look at $\mathrm{IST}_{\mathrm{rec}}(G_n)$ and only consider the left child of each game graph inside the tree.

  (1): We have to solve G'_n recursively. We know that the call $\mathrm{Solve}(G'_n, 2, 2)$ returns the winning regions $(V(G_{n-1} - G''_{n-1}), V(G''_{n-1} + (G'_n - G_{n-1})))$, as we will see below. The highest priority in G_n corresponds to player 0 and thus we need to create the 1-attractor of $V(G''_{n-1} + (G'_n - G_{n-1}))$ for the final result. We have $\mathrm{Attr}_1(G_n, V(G''_{n-1} + (G'_n - G_{n-1}))) = V(G_n)$ and thus we return the winning regions $(\emptyset, V(G_n))$.

  (2): We have to solve G_{n-1} recursively. By the induction hypothesis, we know that the call $\mathrm{Solve}(G_{n-1}, 2, 2)$ returns the winning regions $(V(G_{n-1} - G''_{n-1}), V(G''_{n-1}))$. The highest priority in G'_n corresponds to player 1 and thus we need to create the 0-attractor of $V(G_{n-1} - G''_{n-1})$ for the final result. We have $\mathrm{Attr}_0(G'_n, V(G_{n-1} - G''_{n-1})) = V(G_{n-1} - G''_{n-1})$ and thus we return the winning regions $(V(G_{n-1} - G''_{n-1}), V(G''_{n-1} + (G'_n - G_{n-1})))$.

  (3): We have to solve G_{n-2} recursively. By the induction hypothesis, we know that the call $\mathrm{Solve}(G_{n-2}, 2, 2)$ returns the winning regions $(\emptyset, V(G_{n-2}))$. The highest priority in G''_n corresponds to player 0 and thus we need to create the 1-attractor of $V(G_{n-2})$ for the final result. We have $\mathrm{Attr}_1(G''_n, V(G_{n-2})) = V(G''_n)$ and thus we return the winning regions $(\emptyset, V(G''_n))$.

□

One can see, using precision values $p_0 = 2$ and $p_1 = 2$, we have completely different results, whether n is even or odd, and every procedure has only one recursive call. Once we reach a procedure $\text{Solve}(G_n, 2, 2)$ or $\text{Solve}(G'_n, 2, 2)$, we will always alternate between the game graphs G_n and G'_n with decreasing n and never reach a game graph G''_n anymore.

We will call a procedure a *linear procedure* if it has only one non-trivial recursive call. The procedures for the game graphs G'_n and G''_n are always linear in the recursive algorithm, therefore the important change here is that also the procedure for the game graph G_n turns out to be linear with specific precision values.

$$\begin{tikzcd}
(G_n, 2, 2) \arrow[d] \\
(G'_n, 2, 2) \arrow[d] \\
(G_{n-1}, 2, 2) \arrow[d] \\
\cdots
\end{tikzcd}$$

Figure 4.2: Induced subgame tree for procedures with low precision values.

Next, we will talk about the procedures with precision values directly above $p_0 = 2$ and $p_1 = 2$. This time, we will not focus on all of the possible procedures, as this would not contain a lot of advancing information. Instead, we will look at the most interesting procedures, that illustrates the computation of the new algorithm the best. We will talk about the procedures $\text{Solve}(G_n, 4, 2)$, $\text{Solve}(G'_n, 4, 2)$ and $\text{Solve}(G''_n, 4, 2)$ for all index $n \geq 4$ with n even and again give their computed winning regions.

Lemma 4.7. *Let $n \in \mathbb{N}$ be the index with $n \geq 4$. Assume that n is even, then the following properties hold:*

1. $\text{Solve}(G_n, 4, 2) = (V(G''_n), V(G_n - G''_n))$.
2. $\text{Solve}(G'_n, 4, 2) = (V(G_{n-1}), V(G'_n - G_{n-1}))$
3. $\text{Solve}(G''_n, 4, 2) = (V(G''_{n-2}), V(G''_n - G''_{n-2}))$.

Proof. We will prove this by induction on the index $n \in \mathbb{N}$ as usual, and again only consider the induction step.

- *Induction Step(s)*: Assume that n is even.

  (1): The highest priority in G_n is $h = 3n + 2$, solely due to $U = \{c_{n-1}\}$, and its parity is 0 as n is even. The 0-attractor for U is $A = U$. As we have $p_1 = 2$ we cannot decrease this precision any further. We can ignore the loops and only consider the recursive call to $\text{Solve}(G'_n, 4, 2)$. We know that the call $\text{Solve}(G'_n, 4, 2)$ returns the winning regions $(V(G_{n-1}), V(G'_n - G_{n-1}))$, as we see below. Now, we need to create the 1-attractor of

$V(G'_n - G_{n-1})$. We have $\mathrm{Attr}_1(G_n, V(G'_n - G_{n-1})) = V(G_n - G''_n)$ and thus we return the winning regions $(V(G''_n), V(G_n - G''_n))$.

(2): The highest priority in G'_n is $h = 3n + 1$, solely due to $U = \{d_{n-1}\}$, and its parity is 1 as n is even. The 1-attractor for U is $A = \{a_n, b_n, d_{n-1}, e_{n-1}\}$. As we have $p_0 = 4$ we can decrease the precision. We have a recursive call to $\mathrm{Solve}(G_{n-1}, 2, 2)$ in the first loop iteration with decreased precision, which returns due to lemma 4.6 the winning regions $(V(G_{n-1} - G''_{n-1}), V(G''_{n-1}))$. Now, we need to create the 0-attractor of $V(G_{n-1} - G''_{n-1})$. We have $\mathrm{Attr}_0(G'_n, V(G_{n-1} - G''_{n-1})) = V(G_{n-1} - G''_{n-1})$ and thus we currently have the winning regions $(V(G_{n-1} - G''_{n-1}), V(G''_{n-1} + (G'_n - G_{n-1})))$. We stay in the loop with decreased precision and make a recursive call to $\mathrm{Solve}(G''_{n-1}, 2, 2)$, which returns due to lemma 4.6 the winning regions $(V(G''_{n-1} - G''_{n-3}), V(G''_{n-3}))$. We have $\mathrm{Attr}_0(G''_{n-1}, V(G''_{n-1} - G''_{n-3})) = V(G''_{n-1} - G''_{n-3})$ and thus we have the winning regions $(V(G_{n-1} - G''_{n-3}), V(G''_{n-3} + (G'_n - G_{n-1})))$. We can repeat this step and stay in the loop with decreased precision until we reach the winning regions $(V(G_{n-1}), V(G'_n - G_{n-1}))$ and return them.

(3): The highest priority in G''_n is $h = 3n - 2$, solely due to $U = \{d_{n-2}\}$, and its parity is 0 as n is even. The 0-attractor for U is $A = \{a_{n-1}, d_{n-2}\}$. As we have $p_1 = 2$ we cannot decrease this precision any further. We can ignore the loops with decreased precision and only consider the recursive call to $\mathrm{Solve}(G_{n-2}, 4, 2)$. By the induction hypothesis, we know that the call $\mathrm{Solve}(G_{n-2}, 4, 2)$ returns the winning regions $(V(G''_{n-2}), V(G_{n-2} - G''_{n-2}))$. Now, we need to create the 1-attractor of $V(G_{n-2} - G''_{n-2})$. We have $\mathrm{Attr}_1(G''_n, V(G_{n-2} - G''_{n-2})) = V(G''_n - G''_{n-2})$ and thus we return the winning regions $(V(G''_{n-2}), V(G''_n - G''_{n-2}))$.

□

Finally, we can prove that the call $\mathrm{Solve}(G_n, 4, 4)$ is valid for n even, as we wanted to show.

Theorem 4.8. *Let $n \in \mathbb{N}$ be the index with $p_0 \geq 4$ and $p_1 \geq 4$. Then the following properties hold:*

- *the call* $\mathrm{Solve}(G_n, p_0, p_1)$ *is valid.*
- *the call* $\mathrm{Solve}(G'_n, p_0, p_1)$ *is valid.*
- *the call* $\mathrm{Solve}(G''_n, p_0, p_1)$ *is valid.*

We will only prove the main fact, that the call $\mathrm{Solve}(G_n, 4, 4)$ is valid for n even.

Proof. The highest priority in G_n is $h = 3n + 2$, solely due to $U = \{c_{n-1}\}$, and its parity is 0 as n is even. The 0-attractor for U is $A = U$. As we have $p_1 = 4$ we can decrease the precision.

In the first loop iteration with decreased precision we have a recursive call to Solve$(G'_n, 4, 2)$, which returns, due to lemma 4.7, the winning regions $(V(G_{n-1}), V(G'_n - G_{n-1}))$. Now, we need to create the 1-attractor of $V(G'_n - G_{n-1})$. We have $\text{Attr}_1(G_n, V(G'_n - G_{n-1})) = V(G_n - G''_n)$ and thus we currently have the winning regions $(V(G''_n), V(G_n - G''_n))$. We stay in the loop with decreased precision and make a recursive call to Solve$(G''_n, 4, 2)$, which returns due to lemma 4.7 the winning regions $(V(G''_{n-2}), V(G''_n - G''_{n-2}))$. We have $\text{Attr}_1(G''_n, V(G''_n - G''_{n-2})) = V(G''_n - G''_{n-2})$ and thus we currently have the winning regions $(V(G''_{n-2}), V(G_n - G''_{n-2}))$. We can repeat this step and stay in the loop with decreased precision until we reach the winning regions $(\emptyset, V(G_n))$ and return them. □

$$\begin{tikzcd}
 & (G_n, p_0, p_1) \arrow[dl] \arrow[dr] & \\
(G'_n, p_0, \lfloor \frac{p_1}{2} \rfloor) \arrow[d] & & (G''_n, p_0, \lfloor \frac{p_1}{2} \rfloor) \arrow[d] \\
(G_{n-1}, \lfloor \frac{p_0}{2} \rfloor, \lfloor \frac{p_1}{2} \rfloor) \arrow[d] & & (G_{n-2}, p_0, \lfloor \frac{p_1}{4} \rfloor) \arrow[d] \\
\ldots & & \ldots
\end{tikzcd}$$

Figure 4.3: Induced subgame tree for valid procedures.

Using the theorem, lemmata and their proofs from above, we can build the rest of the induced subgame tree. We will start by looking at the call Solve(G_n, p_0, p_1), where $p_0, p_1 \geq 16$. Assume that n is even (otherwise swap the roles of the players). Then, we have in the first 2 layers of $\text{IST}_{\text{new}}(G_n)$ the same behaviour as we have in $\text{IST}_{\text{rec}}(G_n)$, but this time, we need to decrease the precision values accordingly each time we go a layer deeper. We know that all of the procedures in this part of the tree are valid. Therefore, there is no need for a call with full precision, and the computation mirrors the computation of the recursive algorithm on those game graphs.

We will call a procedure a *binary procedure* if it has exactly two non-trivial recursive calls. $\text{IST}_{\text{rec}}(G_n)$ only contains linear and binary procedures. As we can see in the proof of lemma 4.7 and theorem 4.8, we have a third type of procedures in the computation of the new algorithm. Those procedures with more than two, non-trivial recursive calls will be called a *transition procedure*. We can also see that these procedures occur in the transition from binary procedures to only linear procedures (that is where the name comes from). This transition phase might start at different layers, for different precision values and different n, but the core idea is always the same. The algorithm takes advantage of the loop with decreased precision, and performs several recursive calls instead of just one or two recursive calls. A path from the root of $\text{IST}_{\text{rec}}(G_n)$ to a leaf might go through more than one transition procedure. For example, the call Solve$(G_n, 4, 4)$ for n even is such a transition procedure, that relies on solving the procedure Solve$(G'_n, 4, 2)$, which is also a transition procedure.

$$\begin{tikzcd}
 & (G_n, p_0, p_1) \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & & \\
(G'_n, p_0, \lfloor \frac{p_1}{2} \rfloor) \arrow[d] & (G''_n, p_0, \lfloor \frac{p_1}{2} \rfloor) \arrow[d] & (G''_{n-2}, p_0, \lfloor \frac{p_1}{2} \rfloor) \arrow[d] & \cdots \arrow[d] \\
\cdots & \cdots & \cdots & \cdots
\end{tikzcd}$$

Figure 4.4: Induced subgame tree for transition procedures.

Using the three different kind of procedures, we can create a full type characterization and then define the induced subgame tree for the new algorithm precisely. We will give a graphical representation for the type of each possible procedure.

Theorem 4.9 (Type Characterization of Procedures in the New Algorithm)**.** *The type characterization of the procedures that occur in a computation of the new algorithm w.r.t. the game graphs G_n, G'_n and G''_n with an index $n \geq 4$ and precision values p_0 and p_1 is given by the following tables:*

G_n	1	2	4	8	16	32	$\leftarrow p_1$
1	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
2	$\times$	L	L	L	L	L	L
4	$\times$	L	T	B	B	B	B
8	$\times$	L	T	B	B	B	B
16	$\times$	L	T	B	B	B	B
32	$\times$	L	T	B	B	B	B
$\uparrow p_0$	$\times$	L	T	B	B	B	B

(a) G_n with n even

G_n	1	2	4	8	16	32	$\leftarrow p_1$
1	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
2	$\times$	L	L	L	L	L	L
4	$\times$	T	T	T	T	T	T
8	$\times$	L	B	B	B	B	B
16	$\times$	L	B	B	B	B	B
32	$\times$	L	B	B	B	B	B
$\uparrow p_0$	$\times$	L	B	B	B	B	B

(b) G_n with n odd

G'_n	1	2	4	8	16	32	$\leftarrow p_1$
1	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
2	$\times$	L	L	L	L	L	L
4	$\times$	T	T	T	T	T	T
8	$\times$	L	L	L	L	L	L
16	$\times$	L	L	L	L	L	L
32	$\times$	L	L	L	L	L	L
$\uparrow p_0$	$\times$	L	L	L	L	L	L

(c) G'_n with n even

G'_n	1	2	4	8	16	32	$\leftarrow p_1$
1	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
2	$\times$	L	L	L	L	L	L
4	$\times$	L	T	L	L	L	L
8	$\times$	L	T	L	L	L	L
16	$\times$	L	T	L	L	L	L
32	$\times$	L	T	L	L	L	L
$\uparrow p_0$	$\times$	L	T	L	L	L	L

(d) G'_n with n odd

G''_n	1	2	4	8	16	32	$\leftarrow p_1$
1	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
2	$\times$	L	L	L	L	L	L
4	$\times$	L	T	L	L	L	L
8	$\times$	L	T	L	L	L	L
16	$\times$	L	T	L	L	L	L
32	$\times$	L	T	L	L	L	L
$\uparrow p_0$	$\times$	L	T	L	L	L	L

(e) G''_n with n even

G''_n	1	2	4	8	16	32	$\leftarrow p_1$
1	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
2	$\times$	L	L	L	L	L	L
4	$\times$	T	T	T	T	T	T
8	$\times$	L	L	L	L	L	L
16	$\times$	L	L	L	L	L	L
32	$\times$	L	L	L	L	L	L
$\uparrow p_0$	$\times$	L	L	L	L	L	L

(f) G''_n with n odd

Here, $\times$ stands for a trivial procedure, L stands for a linear procedure, B for a binary procedure and T for a transition procedure.

Definition 4.10. *The induced subgame tree $\mathrm{IST}_{new}(G_n)$ for the new algorithm w.r.t. an index n and precision values $p_0, p_1 \geq 2$ is defined as follows*

- *If n is even, then*
  - *a linear procedure (G_n, p_0, p_1) has the child $(G'_n, p_0, \lfloor \frac{p_1}{2} \rfloor)$ if $p_1 \geq 4$ and (G'_n, p_0, p_1) otherwise.*
  - *a binary procedure (G_n, p_0, p_1) has the children $(G'_n, p_0, \lfloor \frac{p_1}{2} \rfloor)$ and $(G''_n, p_0, \lfloor \frac{p_1}{2} \rfloor)$.*
  - *a transition procedure (G_n, p_0, p_1) has the children*

$$(G'_n, p_0, \lfloor \frac{p_1}{2} \rfloor), (G''_n, p_0, \lfloor \frac{p_1}{2} \rfloor), (G''_{n-2}, p_0, \lfloor \frac{p_1}{2} \rfloor), \ldots, (G''_4, p_0, \lfloor \frac{p_1}{2} \rfloor)$$

  - *a linear procedure (G'_n, p_0, p_1) has the child $(G_{n-1}, \lfloor \frac{p_0}{2} \rfloor, p_1)$ if $p_0 \geq 4$ and (G_{n-1}, p_0, p_1) otherwise.*
  - *a transition procedure (G'_n, p_0, p_1) has the children*

$$(G_{n-1}, \lfloor \frac{p_0}{2} \rfloor, p_1), (G''_{n-1}, \lfloor \frac{p_0}{2} \rfloor, p_1), (G''_{n-3}, \lfloor \frac{p_0}{2} \rfloor, p_1), \ldots, (G''_3, \lfloor \frac{p_0}{2} \rfloor, p_1)$$

  - *a linear procedure (G''_n, p_0, p_1) has the child $(G_{n-2}, p_0, \lfloor \frac{p_1}{2} \rfloor)$ if $p_1 \geq 4$ and (G_{n-2}, p_0, p_1) otherwise.*
  - *a transition procedure (G''_n, p_0, p_1) has the children*

$$(G_{n-2}, p_0, \lfloor \frac{p_1}{2} \rfloor), (G''_{n-2}, p_0, \lfloor \frac{p_1}{2} \rfloor), (G''_{n-4}, p_0, \lfloor \frac{p_1}{2} \rfloor), \ldots, (G''_4, p_0, \lfloor \frac{p_1}{2} \rfloor)$$

- *If n is odd, then we have the same behaviour where the decreased precision is swapped and the last child of a transition procedure has a different index.*

The procedures (G_3, p_0, p_1) and (G_4, p_0, p_1), with arbitrary precision values, form the leaves of the tree.

Finally, we will overestimate the size of $\mathrm{IST}_{\mathrm{new}}(G_n)$. Both trees look the same near the root with linear and binary procedures. Then we have a transition phase in $\mathrm{IST}_{\mathrm{new}}(G_n)$, starting with a transition procedures and after this transition phase, the computation of the new algorithm behaves in a linear fashion. Note that we have at most 2 transition procedures in a path from the root to a leaf for all game graphs G_n. After the first transition procedure, there are only linear procedures and at most one more transition procedure in the rest of the path. This can be proven, but would need a lot of case analysis and repetitive evidence management, thus we will omit this proof here.

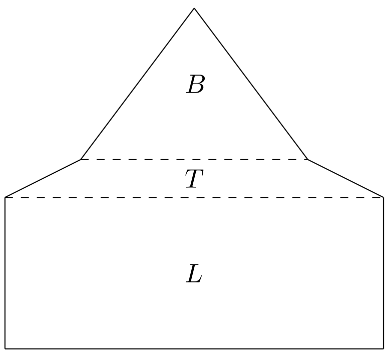

Figure 4.5: Overestimation for $\mathrm{IST}_{\mathrm{new}}(G_n)$

To overestimate the size of $\mathrm{IST}_{\mathrm{new}}(G_n)$ we will make the change between the three phases more simplistic. Assume that the transition procedures are directly after each other and all of the transition procedures are at depth k of the tree, where k is the maximal number of steps we can do until we reach the first transition procedure in $\mathrm{IST}_{\mathrm{new}}(G_n)$. This is in fact an overestimation, since we have a binary tree structure above the first transition procedure and a linear tree structure below. Therefore, we want to maximize the size of the binary tree structure and encounter the transition procedure as late as possible. Moreover, we will assume that the tree above the transition phase is a complete binary tree and does not contain a single linear procedure.

We start the computation with the call $\mathrm{Solve}(G_n, |V|, |V|)$ with an index $n \in \mathbb{N}$. Assume that $k \in \mathbb{N}$ is the maximal number of steps we can do until we reach the transition phase in $\mathrm{IST}_{\mathrm{new}}(G_n)$. Note that the degree of a transition procedure is bound by the index n of the game graph. Then we have the following three bounds for the three different phases in the overestimation of $\mathrm{IST}_{\mathrm{new}}(G_n)$:

- The binary phase has at most $\mathcal{O}(2^k)$ nodes.
- The transition phase has at most $\mathcal{O}(2^k \cdot n^2)$ nodes.
- The linear phase has at most $\mathcal{O}((2^k \cdot n^2) \cdot n)$ nodes.

All in all, the total amount of game graphs we need to solve independently is bounded by

$$\mathcal{O}(((2^k \cdot n^2) \cdot n) + (2^k \cdot n^2) + (2^k)) = \mathcal{O}(2^k \cdot ((n \cdot n^2) + n^2 + 1)) = \mathcal{O}(2^k \cdot (n^3 + n^2 + 1))$$

The last thing we need to do here, is to find a value for k. Therefore, we need to find the longest path from the root to a transition procedure.

The only position, where we can choose a path is in a node (G_n, p_0, p_1). Once we are in such a node, there are two possibilities. Assume that k is even. Then we can either choose the left child and move to $(G_{n-1}, \lfloor \frac{p_0}{2} \rfloor, \lfloor \frac{p_1}{2} \rfloor)$ or we choose the right child and move to $(G_{n-2}, p_0, \lfloor \frac{p_1}{4} \rfloor)$. Therefore, we can either decrease both precision values and move to a game graph with an index of the opposite parity or we can decrease one precision value twice and move to a game graph with an index of the same parity. As we want to stay away from the precision values $p_0 = 2$ or $p_1 = 2$ as long as possible, the best option is to decrease both precision values and to only lower the index by 1. We want to reach the precision values $p_0 = 4$ and $p_1 = 4$ because Solve$(G_n, 4, 4)$ will be the first transition node, we will see. All in all, we get the equation $4 = \frac{|V|}{2^{\frac{k}{2}}}$, which can be rewritten as $k = 2\log_2(|V|) - 4$. We have $k = 2\log_2(|V|) - 4 = \log_2(|V|^2) - 4 \leq \log_2(|V|^2)$ and in total:

$$\begin{aligned}
2^k \cdot (n^3 + n^2 + 1) &= 2^{2\log_2(|V|)-4} \cdot (n^3 + n^2 + 1) \\
&\leq 2^{\log_2(|V|^2)} \cdot (|V|^3 + |V|^2 + 1) \\
&= |V|^2 \cdot (|V|^3 + |V|^2 + 1) \\
&= |V|^5 + |V|^4 + 1 \\
&\in \mathcal{O}(|V|^5)
\end{aligned}$$

All in all, we can see that the size is polynomial in $|V|$ (i.e., the amount of non-trivial recursive steps we need to do in order to solve the game graph G_n with the new algorithm is polynomial in $|V|$).

4.2 Memorization Resilient Worst-Case Family

The main problem of the previous sample family is that it can be solved in polynomial time by a simple variant of the recursive algorithm, which uses a memorization technique just like dynamic programming works with the Fibonacci sequence. Therefore, we want to look at an even stronger counterexample, which withstands a memorization technique. We will talk about a second family that requires the recursive algorithm to solve an exponential amount of *pairwise different* subgames. This family forms the backbone of an entire class of families, with this property. This class can then be used to find other worst-case families for the recursive algorithm with other kinds of properties. For example, we can create a family that is resilient to SCC decomposition, which can be used to improve the recursive algorithm. Here, we will only focus on the core family as both algorithms act similar on all families inside this class. The following section is based on the work of [1], where one can read more about the complete class.

We will denote the second family by $H_n = (V_{0,n}, V_{1,n}, E_n, \pi_n)$. The set of nodes $V_n = V_{0,n} \dot\cup V_{1,n}$ are

$$V_n = \{a_0, ..., a_{2n}, b_0, ..., b_{2n}, c_0, ..., c_{2n}, d_0, ..., d_{2n}\}$$

The complete game graph is described in the following table:

H_n			
Node	Player	Priority	Successors
a_i	$(i \mod 2)$	$2n+i+1$	$\{b_i\}$
b_i	$(i \mod 2)$	i	$\{c_i\} \cup (\{a_{i-1}\} \cap V_n)$
c_i	$1-(i \mod 2)$	i	$\{b_i, d_i\} \cup (\{a_{i+1}\} \cap V_n)$
d_i	$1-(i \mod 2)$	i	$\{c_i\}$

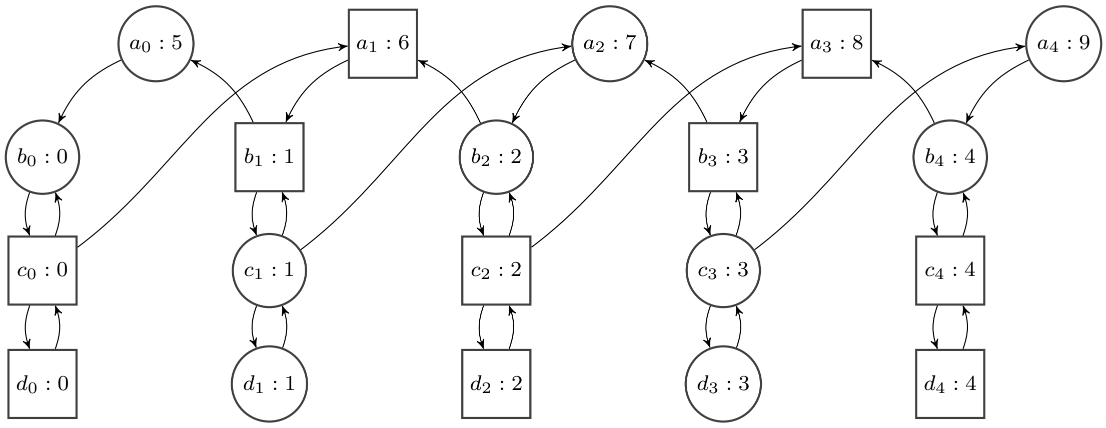

Figure 4.6: The game H_2

The key nodes of this sample family are the c nodes together with their associated d nodes. Note that by construction a c_i node is deleted from the game in the computation of both algorithms if and only if the node d_i is deleted from the game. Later, we will state a lemma about the existence of important c nodes, namely those with an odd index or an index of 0, inside specific subgame graphs. Using these nodes, we can prove that we need to solve an exponential amount of different subgames.

Again, we start the analysis by looking at the winning regions for both players and then we will go into more detail for the computation of both algorithms.

Theorem 4.11. *The Game H_n is completely won by player* 0

Proof. We will proof this by induction on n.

- *Base Case*: It is easy to see that H_1 is won by player 0.

- *Induction Step*: Let $n > 1$. We know by the induction hypothesis that H_{n-1} is completely won by player 0. Now attach the following strategy to the winning strategy δ for player 0 on H_{n-1}: $\delta(a_{2n}) = b_{2n}, \delta(b_{2n}) = c_{2n}, \delta(c_{2n-1}) = a_{2n}$ and $\delta(d_{2n-1}) = c_{2n-1}$. Obviously, b_{2n}, c_{2n} and d_{2n} are won by player 0 because they have an even priority and it is a trap for player 1. Hence, a_{2n}, c_{2n-1} and d_{2n-1} are won by player 0 as well. Therefore, player 1 has to move from b_{2n-1} to a_{2n-2}. Then, the only possible play that agrees with δ and starts in a node of $\{a_{2n-1}, b_{2n-1}, a_{2n-2}, b_{2n-2}, c_{2n-2}, d_{2n-2}\}$ either stays in c_{2n-2} and d_{2n-2} forever, where we have an even priority, or sees the node a_{2n-1} infinitely often, which has an even priority as well. Player 0 wins from all of those new nodes, and therefore, H_{n-1} is still won by player 0, as moving out of H_{n-1} still results in a win for player 0. All in all, H_n is completely won by player 0.

□

The Recursive Algorithm

In this section, we will prove that solving H_n requires an exponential amount of different subgames to be solved. In order to prove that, we shall characterize a suitable compromised subgame tree. Instead of a complete representation, like the induced subgame tree, this tree only describes specific observation points in the computation of the algorithm. The compromised subgame tree will already contain an exponential amount of pairwise different subgames inside its nodes and therefore, we can ignore the rest of the computation.

Starting from the root, which contains the original game graph H_n, we fix specific observation points in the computation that are identified by sequences $w \in \{L, R\}^*$, where L denotes the recursive call and R denotes a loop iteration. Each sequence identifies two subgames $\hat{H}^w$ and H^w that correspond to the input subgames of two successive recursive calls. In the analysis of the compromised subgame tree, we shall only take the left subgame H^w of each $\hat{H}^w$ into account, thus disregarding the rest of the iteration as it is inessential to the argument. The computation of subgame graphs without a nodes is not of interest, because the recursive algorithm clears the remaining graph with linear procedures. Thus, as soon as we reach a node with no a nodes, we make it a leaf inside the tree. Let us make this definition precise with the following definition.

Definition 4.12 (Compromised Subgame Tree $\mathrm{CST}(H_n)$)**.** *The compromised subgame tree* $\mathrm{CST}(H_n)$ *for the recursive algorithm w.r.t. an index* $n \in \mathbb{N}^+$ *is defined inductively on the structure of the sequence* $w \in \{L, R\}^*$ *as follows, where* $z = 2(n - |w|)$*:*

- *The root is* $H_n = H_n^{\varepsilon}$*.*
- *The left successor of a node* H_n^w *is the node* $\hat{H}_n^{wL} = H_n^w[V \setminus \mathrm{Attr}_1(H_n^w, \{a_z\})]$*.*

- *The right successor of a node H_n^w is the node $\hat{H}_n^{wR} = H_n^w[V \setminus \mathrm{Attr}_0(H_n^w, \mathrm{Win}_0(\hat{H}_n^{wL}))]$.*
- *The only successor of a node $\hat{H}_n^w$ is the node $H_n^w = \hat{H}_n^w[V \setminus \mathrm{Attr}_0(\hat{H}_n^w, \{a_{z+1}\})]$.*
- *We stop the iteration once we reach a game graph with no a node inside.*

Before we can prove the main result of this section, we need some additional information about the compromised subgame tree. The following lemmata states some invariants of the game graphs contained in the tree. In particular, they ensure that all of them are subgames of H_n and that, depending on the sequence w, they contain the required leading position a. In addition, it states two very important properties of every left child in the tree (i.e., those elements identified by a sequence w that ends with L). The first one ensures that each such game necessarily contain a specific position c_i, with the index i depending of w. The second one characterizes the winning region for player 0 on the left child subgames $\hat{H}_n^{wL}$. It states that, in each such game, the winning positions for player 0 contained in the corresponding H_n are all its b_j, c_j and d_j, with even index greater than the maximal index of a leading position a_l in that game. Indeed, as soon as the higher positions a_i, with $i \in [l+1, 2n]$ are removed from the game, each c_j and d_j, possibly together with its associated b_j with an even index, is necessarily contained in an independent 0-dominion.

Lemma 4.13. *Let $n \in \mathbb{N}_+$ be the index, $w \in \{L, R\}^*$, and let $z = 2(n - |w|)$. By $\mathrm{lst}(w)$ we denote the last position of w. Then the following properties hold, for all defined game graphs H_n^w and $\hat{H}_n^w$:*

1. *H_n^w is a subgame graph of H_n.*
2. *$a_j \in V(H_n^w)$ iff. $j \in [0, z]$.*
3. *$c_j \in V(H_n^w)$ for all $j \in [0, z]$, and $c_{z+1} \in V(H_n^w)$ if $w \neq \varepsilon$ and $\mathrm{lst}(w) = L$.*
4. *$c_j \in V(H_n^w)$ iff. $d_j \in V(H_n^w)$ for all $j \in [0, 2n]$.*
5. *$\hat{H}_n^w$ is a subgame graph of H_n.*
6. *$a_j \in V(\hat{H}_n^w)$ iff. $j \in [0, z+1]$.*
7. *$c_j \in V(\hat{H}_n^w)$ for all $j \in [0, z]$, and $c_{z+1} \in V(\hat{H}_n^w)$, if $|w| \leq n$, and $c_0 \in V(\hat{H}_n^w)$ otherwise, when $\mathrm{lst}(w) = L$.*
8. *$c_j \in V(\hat{H}_n^w)$ iff. $d_j \in V(\hat{H}_n^w)$ for all $j \in [0, 2n]$.*
9. *$\mathrm{Win}_0(\hat{H}_n^w) = \{b_{z+2j}, c_{z+2j} \in \hat{H}_n^w \mid j \in \{1, ..., |w|\}\}$, if $\mathrm{lst}(w) = L$*

Finally, the next lemma establishes that all subgames contained in the compromised subgame tree are indeed generated by the recursive algorithm when called with input game graph H_n.

Lemma 4.14. *For any index $n \in \mathbb{N}_+$ and sequence $w \in \{L, R\}^*$ the defined game graphs H_n^w and $\hat{H}_n^w$ have to be solved independently in order to solve the game graph H_n using the recursive algorithm.*

We are now ready for the main result of this section, namely the compromised subgame tree contains elements which are all different from each other and whose number of nodes is exponential in the index n. We split the result into two parts. The first one states that any subgame in the left subtree of some H_n^w is different from any other subgame in the right subtree. The idea is that for any node in the tree, all subgames of its left subtree contain at least one position, a specific position c_i with i depending on w, that is not contained in any subgame of its right subtree.

Lemma 4.15. *For any index $n \in \mathbb{N}_+$ and sequences $w, v \in \{L, R\}^*$, with $|w| + |v| \leq n$ and $z = 2(n - |w|)$, the following properties hold:*

1. *$c_{z-1} \in H_n^{wLv}$ and $c_{z-1} \in \hat{H}_n^{wLv}$ if $|w| < n$ and $c_0 \in \hat{H}_n^{wL}$ otherwise.*
2. *$c_{z-1} \notin H_n^{wRv}$ and $c_{z-1} \notin \hat{H}_n^{wRv}$ if $|w| < n$ and $c_0 \notin \hat{H}_n^{wR}$ otherwise.*

Proof. First observe that, if $|w| < n$, by items 3 and 7 of lemma 4.13, position c_{z-1} belongs to both H_n^w and $\hat{H}_n^w$, since $0 < z - 1 < z$. Let us consider item 1 of the current lemma first and show that every position c_{z-1} belongs to all the descendants of H_n^w in its left subtree. The proof proceeds by induction on the length of the sequence v and recall, that due to item 2 (resp. 6) of lemma 4.13 the position with maximal priority in H_n^w (resp. $\hat{H}_n^w$) is a_z (resp. a_{z+1}).

- *Base Case*: Let $|v| = 0$. We have to show that $c_{z-1} \in H_n^{wL}$ and $c_{z-1} \in \hat{H}_n^{wL}$. By definition we have $\hat{H}_n^{wL} = H_n^w[V \setminus \mathrm{Attr}_1(H_n^w, \{a_z\})]$. A move entering a_z may only come from b_{z+1} or c_{z-1}. We know that the owner of c_{z-1} is player 0 and cannot be attracted since he can move to d_{z-1} instead. On the other hand, b_{z+1} will be attracted if it is present in the current game graph. This is always the case for $w \neq \epsilon$. Hence, $\mathrm{Attr}_1(H_n^w, \{a_z\}) = \{a_z, b_{z+1}\}$, if $w \neq \varepsilon$, and $\mathrm{Attr}_1(H_n^w, \{a_z\}) = \{a_z\}$ otherwise. Similarly, by definition we have $H_n^{wL} = \hat{H}_n^{wL}[V \setminus \mathrm{Attr}_0(\hat{H}_n^{wL}, \{a_{z-1}\})]$. For the same reason as before, we have $\mathrm{Attr}_0(\hat{H}_n^{wL}, \{a_{z-1}\}) = \{a_{z-1}, b_z\}$. Hence, no c node is removed from either game and the statement immediately follows.
- *Induction Step*: Assume that $|v| > 0$ and let $v = v'x$ with $x \in \{L, R\}$, and set $\bar{w} = wLv'$. By the inductive hypothesis, $c_{z-1} \in H_n^{\bar{w}}$ and $c_{z-1} \in \hat{H}_n^{\bar{w}}$. We have two cases, depending on whether $x = L$ or $x = R$. Let $r = 2(n - |\bar{w}|)$.
  - If $x = L$, we obtain $\hat{H}_n^{\bar{w}L}$ by removing $\mathrm{Attr}_1(H_n^{\bar{w}}, \{a_r\}) = \{a_r, b_{r+1}\}$ from the game $H_n^{\bar{w}}$. Similarly, we obtain $H_n^{\bar{w}L}$ by removing $\mathrm{Attr}_0(\hat{H}_n^{\bar{w}L}, \{a_{r-1}\}) = \{a_{r-1}, b_r\}$ from

the game $\hat{H}_n^{\bar{w}L}$, by observing that $|\bar{w}L| = |\bar{w}|+1$ and, therefore, $2(n-|\bar{w}L|)+1 = r-1$. In both cases the statement follows immediately.

- If $x = R$, we obtain $\hat{H}_n^{\bar{w}R}$ by removing $\mathrm{Attr}_0(H_n^{\bar{w}}, \mathrm{Win}_0(\hat{H}_n^{\bar{w}L}))$ from the game $H_n^{\bar{w}}$. Position c_{z-1} has odd priority and cannot belong to $\mathrm{Win}_0(\hat{H}_n^{\bar{w}L})$, which by item 9 of lemma 4.13, only contains nodes with even index. It holds that c_{z-1} is owned by player 0 and can only move to d_{z-1} as both a_z and b_{z-1} are not contained in the current game graph. As a consequence, it can not end up in $\mathrm{Attr}_0(H_n^{\bar{w}}, \mathrm{Win}_0(\hat{H}_n^{\bar{w}L}))$ and the thesis holds.

  Finally, we obtain $H_n^{\bar{w}R}$ by removing $\mathrm{Attr}_0(\hat{H}_n^{\bar{w}R}, \{a_{\hat{r}}\})$ from the game $\hat{H}_n^{\bar{w}R}$, with $\hat{r} = 2(n - |\bar{w}R|) + 1$. In the considered subgame $a_{\hat{r}}$, has incoming moves only from $b_{\hat{r}+1}$ and $c_{\hat{r}-1}$. However, $\hat{r} - 1 = 2(n - |\bar{w}R|) < 2(n - |w|) - 1 = z - 1$. Moreover, $\hat{r}+1$ is an even index, and thus $b_{\hat{r}+1}$ is not contained in $\hat{H}_n^{\bar{w}R}$, being in $\mathrm{Win}_0(\hat{H}_n^{\bar{w}L})$ as shown above. As a consequence $\mathrm{Attr}_0(\hat{H}_n^{\bar{w}R}, \{a_{\hat{r}}\}) = \{a_{\hat{r}}\}$ and the statement follows.

In addition, when $|w| = n$, item 7 of lemma 4.13 tells us that c_0 belongs to $\hat{H}_n^{wL}$. This ends the proof of item 1 of the lemma.

As to item 2 of the current lemma, first observe that, if $|w| < n$, then $c_{z-1} \notin \hat{H}_n^{wR}$. Indeed, we have $c_{z-1} \in \mathrm{Attr}_0(H_n^{w}, \mathrm{Win}_0(\hat{H}_n^{wL}))$. Since this set is removed from H_n^w to obtain $\hat{H}_n^{wR}$, the thesis holds for $\hat{H}_n^{wR}$. Moreover, every descendant of $\hat{H}_n^{wR}$ in the compromised subgame tree is obtained only by removing positions. As a consequence, none of them can contain position c_{z-1}. In case $|w| = n$, instead, it suffices to observe that, according to item 9 of lemma 4.13, $c_0 \in \mathrm{Win}_0(\hat{H}_n^{wL})$, hence it cannot be contained in $\hat{H}_n^{wR}$

□

The main result asserting the exponential size of the compromised subgame tree is given by the following theorem. This follows by observing that the number of nodes in the compromised subgame tree is exponential in n and by showing that the subgames associated with any two nodes in the tree are indeed different.

Theorem 4.16. $|\mathrm{CST}(H_n)| = 3(2^{k+1}-1)$*, for any index* $n \in \mathbb{N}^+$ *and all game graphs contained in* $\mathrm{CST}(H_n)$ *are pairwise different.*

Proof. We first show the second argument of the theorem, namely that all nodes contained in $\mathrm{CST}(H_n)$ are pairwise different. Therefore, we show that for all $w \neq w'$, the subgames H_n^w, $\hat{H}_n^w$, $H_n^{w'}$, $\hat{H}_n^{w'}$ are pairwise different. Let us start by showing that $H_n^w \neq \hat{H}_n^w$, for each $w \neq \varepsilon$. By item 5 of theorem 5.3 position $a_{2(n-|w|)+1} \in \hat{H}_n^w$ and by definition we delete the attractor of $a_{2(n-|w|)+1}$ to obtain H_n^w. Thus, we have $a_{2(n-|w|)+1} \notin H_n^w$ and those two subgames cannot be equal.

Let us now consider two subgames, each associated with one of the sequences w and w'. There are two possible cases: Either w is a strict prefix of w' or w and w' share a common longest prefix $\bar{w}$ that is different from both.

- Let w be a strict prefix of w' (i.e., we have $w' = wv$ for some sequence $v \neq \varepsilon$). Thus the subgame of w' is a descendant of the node containing the subgame of w. An easy induction on the length of v can prove that if $H \in \{H_n^w, \hat{H}_n^w\}$ and $H' \in \{H_n^{w'}, \hat{H}_n^{w'}\}$ then H' is a strict subgame of H. Indeed, each step further through one layer of the compromised subgame tree deletes at least one node from the game.
- Let w and w' share a common longest prefix $\bar{w}$ that is different from both. Then we know by lemma 4.15 that there is at least one position c_{z-1} contained in all subgames of the left subtree, while this c_{z-1} is not contained in all subgames of the right subtree. Therefore, each of the two subgames associated with w and w' must be different.

At the end, we show the first statement of the theorem, namely that we have $|\,\mathrm{CST}(H_n)| = 3(2^{n+1} - 1)$. It suffices to observe that the number of sequences of length at most n over the alphabet $\{L, R\}$ are precisely $2^{n+1} - 1$ different elements. Moreover, each game H_n^w for each sequence w has exactly two children $\hat{H}_n^{wL}$ and $\hat{H}_n^{wR}$. Therefore, we have exactly $3(2^{n+1} - 1)$ nodes in $\mathrm{CST}(H_n)$. □

The New Algorithm

Again, we look at the new algorithm and show how it solves the game graph H_n. The question arises whether the new algorithm only needs polynomial runtime for this sample family as well. We will see that this is in fact true, as the computation is similar to the one for the game graph G_n. Until the computation reaches a specific depth, it mirrors the computation of the recursive algorithm again. Then we have a transition phase and in the end, there are only linear procedures left. We will not prove every detail here, as one needs again a lot of case distinction and the returned winning regions of procedures with specific precision values, but give a sufficient idea what happens here.

At first, we need to talk about all the subgames that occur in the recursive algorithm. Before, we only had specific observation points, as we were interested in a lower bound for the amount of recursive calls. Now, we want to create an upper bound and thus we need to take all occuring subgame graphs into account. The subgames, which where not mentioned in the compromised subgame tree, look similar to the other ones. There is still a leading a_j node, all nodes of index $i < j$ are still present in the game graph and above the leading a_j node there might be some c and d nodes, possibly together with there corresponding b node. Additionally, the index of the leading a_j node corresponds to the depth of the recursion.

Lemma 4.17. *Let H be a game graph that occurs in the computation of the recursive algorithm on the game graph H_n for any index $n \in \mathbb{N}_+$. Further, let $d(H)$ be the recursive depth, in which H occurs and let $z = 2n - d(H)$. Then the following property hold:*

1. *$a_j \in V(H)$ iff. $j \in [0, z]$.*

In the first sample family, all occurring subgame graphs were of the form G_n, G'_n or G''_n. Here, we cannot characterize the occurring subgames in such a way, but every occurring subgame H is similar to a game graph H_n in a computational sense. As the nodes above the leading a_j node do not impact the computation of both algorithms until all a nodes are deleted from the game, and as the nodes below the leading a_j node are all present, we can look at the computation of an actual game graph H_n and take this as a representation for the computation of any occurring game graph H. We will establish all results using the game graph H_n and then we know the computational behaviour of all occurring subgame graphs.

At first, we want to identify the valid procedures again. Due to the fact that the key nodes in this game graph form pretty small dominions, just as they did in the game graph G_n, it suffices to use pretty low precision parameters in order to get a valid result.

Theorem 4.18. *Let $n \in \mathbb{N}^+$ be the index with $p_0 \geq 4$ and $p_1 \geq 4$. Then the the call* Solve(H_n, p_0, p_1) *is valid.*

We can see that we have the same computation until we reach a recursion depth k like in the first sample family. The rest of the computation below depth k is also similar to the one for the other sample family. To illustrate this, we take a closer look at the procedure Solve$(H_n, 4, 4)$, which is again a transition node.

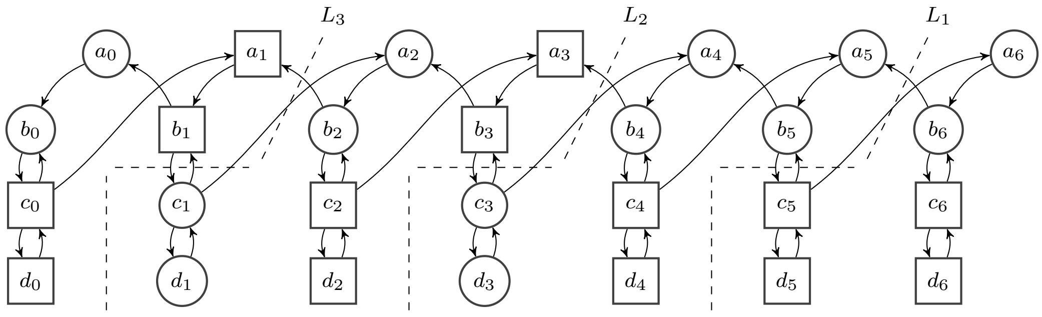

Figure 4.7: The game H_3. The nodes on the left of line L_i form the game graph after the i-th iteration of the loop in the procedure Solve$(H_3, 4, 4)$. We need a total of 4 iterations to solve this game graph.

In order to solve the procedure Solve$(H_n, 4, 4)$ we have to make exactly $n+1$ recursive calls with decreased precision. After the first recursive call we delete the nodes $\{a_n, b_n, c_n, c_{n-1}, d_n, d_{n-1}\}$

from the game graph H_n. Then we iteratively delete the next 8 nodes (i.e. the two nodes of each type with the maximal index) from the game until there are only 6 nodes left and in the last iteration of the loop we delete those nodes as well. One can see the iteration of the procedure Solve$(H_3, 4, 4)$ in the figure above.

In this sample family, different from the previous one, we only have one transition procedure directly following by a binary procedure, and then only linear procedures in a path from the root to a leaf. All in all, we have linear and binary procedures above the transition phase, we have a transition phase containing a transition procedure and a binary procedure, and in the end there are only linear procedures. One can see that we can use the exact same overestimation, which we used for the first sample family.

For the final result, we need a precise value for k again. Let $H \subseteq H_n$ be a game graph that is generated by the recursive algorithm when called with the game graph H_n, let $d(H)$ be the recursive depth, in which H occurs and let $z = 2n - d(H)$. Due to lemma 4.17 we know that the highest priority in H has parity $i = 1 - (z \mod 2)$, solely due to a_z. In depth $d(H)$ of the recursion we have to decrease the precision p_{1-i}. Therefore, we decrease the precision values alternately and reach the precision values $p_0 = 4$ and $p_1 = 4$ after exactly $2 \cdot \log_2(|V|) - 4$ recursive steps again. Therefore we have the value $k = 2 \cdot \log_2(|V|) - 4 \leq \log_2(|V|^2)$.

All in all, we can use the exact same upper bound for the amount of non-trivial recursive calls, as we used for the first sample family, and thus we result with the same conclusion as we did before: The new algorithm only needs polynomial runtime to solve the game graph H_n.

Chapter 5

Conclusion

In this thesis, we presented the recursive algorithm by Zielonka, which is used to solve a parity game and the improved version of the recursive algorithm by Parys. We have seen two worst-case sample families for the recursive algorithm. The second family even prevents the help of a memorization technique, as we need to solve an exponential amount of pairwise different subgames. Additionally, we saw that the new algorithm can solve those sample families in polynomial time. This is due to the fact that both sample families rely on pretty small dominions that we can iteratively delete from the game. As we did not prove a polynomial runtime bound for the new algorithm in general, there might be families, such that the new algorithm still has exponential runtime.

Sadly, the new algorithm is, as we said earlier, not very efficient in practice. For example, we have a lot of additional trivial procedures, caused by the precision values. Besides of the practical point, this new algorithm might be a good start for more efficient algorithms for parity games. In the end of the original paper corresponding to the quasi-polynomial algorithm [17], Parys already gave some ideas, how one could improve the efficiency of the algorithm. We used some of them in this thesis. Another idea that we did not implement here would be to decrease the precision values p_i to $min(p_i, |G|)$ after each recursive call. One could also try to get some additional information in the calls with decreased precision before we make a call with full precision. For example, one can return an additional boolean value that states whether we already found the correct winning regions. This would mean that we can omit the call with full precision.

It is also noteworthy, that shortly after the publish of Parys paper, there was already an improved version of his algorithm [16]. There, we still have the precision parameters p_0 and p_1 with the same meaning but we use a different recursive structure. This new version has a slightly better theoretical runtime, but is still quasi-polynomial. Whether, we can improve this new algorithm in a way such that we get a fast algorithm for practice remains to be shown.

Appendix

$$\begin{tikzcd}
 & & & & G_5 \arrow[dll] \arrow[drr] & & & \\
 & & G_5' \arrow[d] & & & & G_5'' \arrow[d] & \\
 & & G_4 \arrow[dl] \arrow[dr] & & & & G_3 \arrow[dl] \arrow[dr] & \\
 & G_4' \arrow[d] & & G_4'' \arrow[d] & & G_3' \arrow[d] & & G_3'' \arrow[d] \\
 & G_3 \arrow[dl] \arrow[dr] & & G_2 & & G_2 & & G_1 \\
G_3' \arrow[d] & & G_3'' \arrow[d] & & & & & \\
G_2 & & G_1 & & & & &
\end{tikzcd}$$

Figure A.1: $\mathrm{IST}_{\mathrm{rec}}(G_5)$.

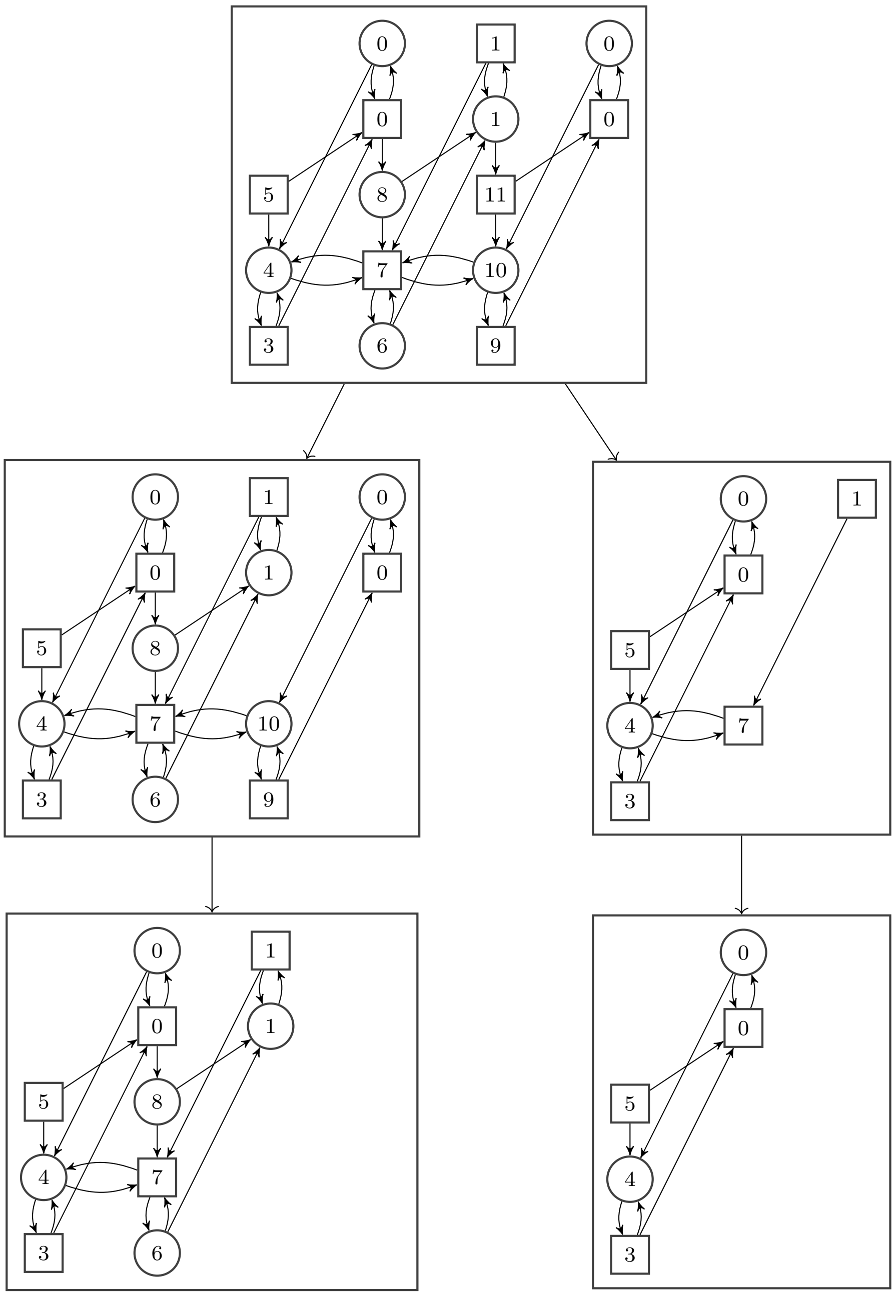

Figure A.2: $\mathrm{IST}_{\mathrm{rec}}(G_3)$ in detail.

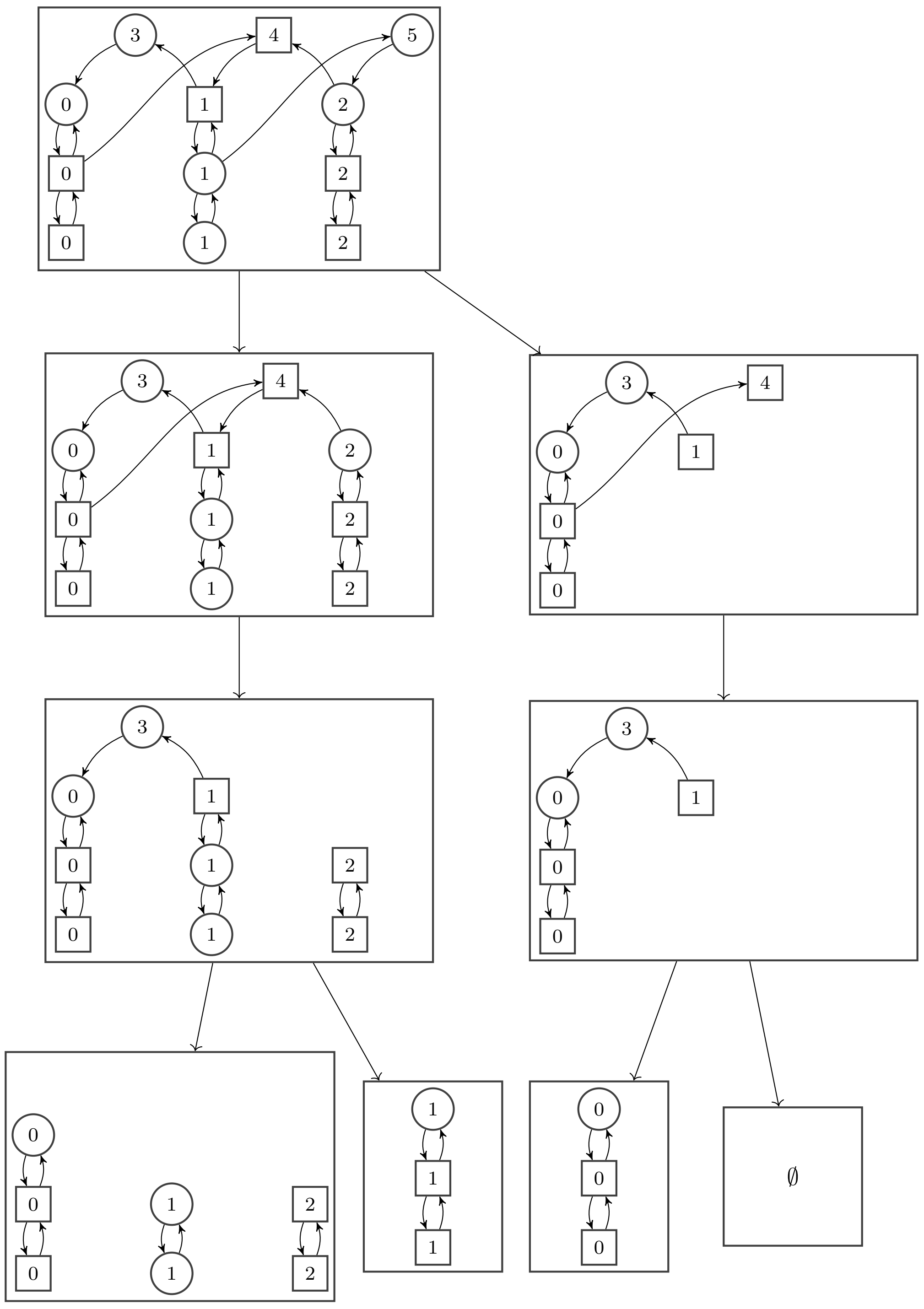

Figure A.3: CST(H_1) in detail.

Bibliography

[1] Massimo Benerecetti, Daniele Dell'Erba, and Fabio Mogavero. Robust exponential worst cases for divide-et-impera algorithms for parity games. *arXiv preprint arXiv:1709.02099*, 2017.

[2] Massimo Benerecetti, Daniele Dell'Erba, and Fabio Mogavero. Solving parity games via priority promotion. *Formal Methods in System Design*, 52(2):193–226, 2018.

[3] Cristian S Calude, Sanjay Jain, Bakhadyr Khoussainov, Wei Li, and Frank Stephan. Deciding parity games in quasi-polynomial time. *SIAM Journal on Computing*, (0):STOC17–152, 2020.

[4] E Allen Emerson and Charanjit S Jutla. Tree automata, mu-calculus and determinacy. In *FoCS*, volume 91, pages 368–377. Citeseer, 1991.

[5] E Allen Emerson, Charanjit S Jutla, and A Prasad Sistla. On model checking for the μ-calculus and its fragments. *Theoretical Computer Science*, 258(1-2):491–522, 2001.

[6] John Fearnley, Sanjay Jain, Bart De Keijzer, Sven Schewe, Frank Stephan, and Dominik Wojtczak. An ordered approach to solving parity games in quasi-polynomial time and quasi-linear space. *International Journal on Software Tools for Technology Transfer*, 21(3):325–349, 2019.

[7] Oliver Friedmann. Recursive algorithm for parity games requires exponential time. *RAIRO-Theoretical Informatics and Applications*, 45(4):449–457, 2011.

[8] Oliver Friedmann, Thomas Dueholm Hansen, and Uri Zwick. Subexponential lower bounds for randomized pivoting rules for the simplex algorithm. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 283–292, 2011.

[9] Oliver Friedmann and Martin Lange. Solving parity games in practice. In *International Symposium on Automated Technology for Verification and Analysis*, pages 182–196. Springer, 2009.

[10] Marcin Jurdziński. Deciding the winner in parity games is in up $\cap$ co-up. *Information Processing Letters*, 68(3):119–124, 1998.

[11] Marcin Jurdziński. Small progress measures for solving parity games. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 290–301. Springer, 2000.

[12] Marcin Jurdziński, Mike Paterson, and Uri Zwick. A deterministic subexponential algorithm for solving parity games. *SIAM Journal on Computing*, 38(4):1519–1532, 2008.

[13] Jeroen JA Keiren. Benchmarks for parity games. In *International Conference on Fundamentals of Software Engineering*, pages 127–142. Springer, 2015.

[14] Orna Kupferman and Moshe Y Vardi. Weak alternating automata and tree automata emptiness. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 224–233, 1998.

[15] Karoliina Lehtinen. A modal μ perspective on solving parity games in quasi-polynomial time. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 639–648, 2018.

[16] Karoliina Lehtinen, Sven Schewe, and Dominik Wojtczak. Improving the complexity of parys' recursive algorithm. *arXiv preprint arXiv:1904.11810*, 2019.

[17] Paweł Parys. Parity games: Zielonka's algorithm in quasi-polynomial time. *arXiv preprint arXiv:1904.12446*, 2019.

[18] W. Thomas. *Automata and Reactive Systems*. RWTH Aachen, 2002.

[19] Tom van Dijk. Oink: An implementation and evaluation of modern parity game solvers. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 291–308. Springer, 2018.

[20] Jens Vöge and Marcin Jurdziński. A discrete strategy improvement algorithm for solving parity games. In *International Conference on Computer Aided Verification*, pages 202–215. Springer, 2000.

[21] Wieslaw Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200(1-2):135–183, 1998.