Algorithmic Analysis of Parity Games

Jan Obdržálek

Doctor of Philosophy

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

2006

Abstract

Parity games are discrete infinite games of two players with complete information. There are two main motivations to study parity games. Firstly the problem of deciding a winner in a parity game is polynomially equivalent to the modal μ-calculus model checking, and therefore is very important in the field of computer aided verification. Secondly it is the intriguing status of parity games from the point of view of complexity theory. Solving parity games is one of the few natural problems in the class NP∩co-NP (even in UP∩co-UP), and there is no known polynomial time algorithm, despite the substantial amount of effort to find one.

In this thesis we add to the body of work on parity games. We start by presenting parity games and explaining the concepts behind them, giving a survey of known algorithms, and show their relationship to other problems. In the second part of the thesis we want to answer the following question: Are there classes of graphs on which we can solve parity games in polynomial time? Tree-width has long been considered the most important connectivity measure of (undirected) graphs, and we give a polynomial algorithm for solving parity games on graphs of bounded tree-width. However tree-width is not the most concise measure for directed graphs, on which the parity games are played. We therefore introduce a new connectivity measure for directed graphs called DAG-width. We show several properties of this measure, including its relationship to other measures, and present a polynomial-time algorithm for solving parity games on graphs of bounded DAG-width of [BDHK06]. In the third part we analyze the strategy improvement algorithm of Vöge and Jurdziński, providing some new results and comments on their algorithm. Finally we present a new algorithm for parity games, in part inspired by the strategy improvement algorithm, based on spines. The notion of spine is a new structural way of capturing the (possible) winning sets and counterstrategies. This notion has some interesting properties, which can give a further insight into parity games.

Acknowledgements

I would like to thank my supervisor, Colin Stirling, for his help, support and encouragement during the course of my study in Edinburgh. I am even more grateful for what he once called "turning you into a mathematician" – that would probably never happened if it had not been for Colin's keen interest in showing me new and exciting research areas.

My thanks go to the many people at LFCS who kindly offered me support and advice when I needed them most. I would especially like to thank Kousha Etessami and Julian Bradfield. Thanks are also due to my office mates Robert Atkey and Uli Schöpp for putting up with my erratic working habits.

I am very grateful to Erich Grädel for letting me spend five months in his group at RWTH Aachen. I greatly enjoyed my stay there and learned a lot from my discussions with the members of his and Wolfgang Thomas' groups. I wish to thank also my advisors and colleagues in Brno, Mojmír Křetínský and Tony Kučera, for their help, support and introducing me to research in the first place.

Special thanks go to my friends from the New Scotland Country Dance Society, who made my stay in Edinburgh as fun and enjoyable as humanly possible. I am also indebted to Mark and Gillian for their friendship and many great days out.

My parents have been constantly encouraging and supportive when I needed them. Last, but not least, I would like to thank my wonderful wife Martina for always being there.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Jan Obdržálek*)

To my parents.

Table of Contents

List of Algorithms

Chapter 1

Introduction

Since its discovery in early sixties by Büchi [Büc60] and Elgot [Elg61], scientists started to explore the close connection between automata and logic. The two works we mentioned showed the (then surprising) result that finite automata and monadic second-order logic have the same expressive power on the class of finite words. This equivalence was in the following years shown to exist also between finite automata and monadic second-order logic over infinite words and trees by now the classical results of Büchi [Büc62], McNaughton [McN66] and Rabin [Rab69]. One of the techniques developed in these works has been an effective translation of monadic second-order formulas into finite automata on words and trees, reducing the satisfiability problem for logic to non-emptiness problem for the automata.

Infinite-duration two-player games proved to be a technically useful way of describing the runs of automata on infinite words and trees. A prime example of this is the fact that by using infinite games one can simplify the most difficult part of the proof of the famous Rabin's result [Rab69] that the monadic second order theory of the binary infinite tree is decidable – the complementation lemma for automata on infinite trees. Rabin implicitly showed determinacy of parity games, but did not explicitly use games in his proof. The idea to use games was first proposed by Büchi [Büc77], and the successful application to Rabin's proof is due to Gurevich and Harrington [GH82] and, in a more elegant version, Emerson and Jutla [EJ91]. A nice proof can be found in [Tho97].

The automata on infinite trees and words can use wide variety of different acceptance conditions. In Büchi's paper [Büc62] the first such condition has

been proposed, which has since been called the Büchi condition. Other conditions followed – Muller condition [Mul63], Rabin condition [Rab72] and Streett condition [Str82]. Parity winning condition was first introduced by Mostowski in [Mos84], where it was called the 'Rabin chain condition'[1]. The name 'parity condition' was given to it by Emerson and Jutla in [EJ91], where it was independently discovered and applied as a winning condition for games at the same time as [Mos91]. Out of the many different winning conditions for two-player infinite games the parity condition is the most fundamental one. Every other (commonly used) winning condition can be reduced this condition. Moreover it can be easily dualised and is the most expressive one for which memoryless strategies always work.

The determinacy of parity games follows from the much more general result of Martin [Mar75], who showed that Borel games (a class of games which contains parity games) are determined. As we already mentioned, the determinacy of parity games was already implicitly present in Rabin's paper [Rab69]. Whereas the result of Martin [Mar75] relies on infinite strategies, Gurevich and Harrington [GH82] showed that finite memory strategies suffice for a class of games containing parity games. The fact that memoryless strategies suffice is due to Mostowski [Mos91] and Emerson and Jutla [EJ91]. First constructive proof is due to McNaughton [McN93], explicitly adapted to parity games by Zielonka [Zie98].

The modal μ-calculus, a fixed-point logic of programs, has been introduced by Kozen in [Koz83]. The close relationship between the modal μ-calculus and parity games has been observed by several authors, most notably Emerson and Jutla [EJ88], Herwig [Her89] and Stirling [Sti95]. There are indeed linear reductions between the modal μ-calculus model checking problem and the problem of solving parity games (see [GTW02] for a broad survey). As the modal μ-calculus subsumes all other widely used temporal logics this connection to parity games only gained on importance and provided an extra incentive to find a polynomial-time algorithm for solving parity games.

Another reason why we should be interested in parity games is their complexity theoretical status. The problem of solving parity games is one of only a few natural problems in the interesting complexity class NP∩co-NP. It is widely believed that there is no complete problem for this class and it is quite

[1] Very rarely the name 'Mostowski condition' is used for the parity condition.

possible that this class is even equal to P. Other famous problems in this class are graph isomorphism [KST93] (under some assumptions – see [KvM99, MV99]), prime factorisation [Pra75] and PRIMALITY. The latter problem has been recently (2002) shown to be in P [AKS04], thus settling a long-standing open question. It is interesting to note that while for parity games the proofs of membership to NP and co-NP are dual to each other, for primality they are completely different.

We actually have a slightly better upper bound on the complexity of solving parity games. Jurdziński [Jur98] showed that the problem belongs to UP∩co-UP, and thus is 'not too far above P' [Pap94]. Even more encouraging is the fact that there exist sub-exponential algorithms [BSV03, JPZ06] and there is also the strategy improvement algorithm [VJ00]. For this algorithm there is currently no known example of a parity game which needs more than a linear number of iterations, each running in cubic time (in the size of the game).

There are also several other related classes of games which belong to the same complexity class NP∩co-NP. The two most important examples are mean-payoff games [EM79] and simple stochastic games [Con92]. There exists a reduction from parity games to mean-payoff games [Pur95, Sti95], which in turn can be reduced to simple stochastic games [ZP96]. Therefore parity games are the most obvious candidate when looking for a polynomial-time algorithm for all the mentioned classes of games.

From what we have mentioned above it comes as no surprise there have been a substantial effort of the community [EJS93, Zie98, Jur00, VJ00, Obd03, BSV03, JPZ06] to find a polynomial algorithm for solving parity games. Despite of all this effort the problem remains an open question. During the years there have been several announcements that the problem has been solved (the author knows about two such cases just in the year 2005), but all of them proved to be incorrect.

In this thesis we want to add to the body of knowledge on parity games. We present a new general algorithm for solving parity games, deal with the complexity of existing algorithms, and also give algorithms working in polynomial time on restricted classes of graphs. As the problem of solving parity games has been of considerable interest to researchers involved in the area, many interesting special cases have been studied and some partial results have been obtained. However these usually do not get published and therefore remain

largely unknown. To help to remedy this situation we present some of these results in this thesis.

The rest of this thesis is organised as follows: In Chapter 2, we start by giving the basic definitions and introduce parity games. Then we present the known facts about parity games – memoryless determinacy, complexity and some normal forms. Next we present the modal μ-calculus, and show that the model-checking problem for the modal μ-calculus is equivalent to the problem of solving parity games by giving linear reductions in both directions. Finally we present two other infinite-duration two-player games with a close relationship to parity games.

In Chapter 3 we give an overview of the algorithms for (solving of) parity games known so far. We start with a simple discrete exponential algorithm, and mention also other (slightly better) discrete algorithms. Then we look at known randomised sub-exponential algorithms and finally present a very recent deterministic sub-exponential algorithm. We finish by discussing the complexity of solving parity games on some restricted classes of graphs, specifically mentioning undirected graphs and trees with back edges.

In Chapter 4 we introduce graphs of bounded tree-width and give a polynomial time algorithm for solving parity games on this class of graphs. This chapter is based on the paper [Obd03].

Chapter 5 deals with the question posed by the author in [Obd03]: Whether there is some natural decomposition for directed graphs. We answer that question positively by presenting a new connectivity measure called DAG-width. Part of this work was published in [Obd06]. Independently and shortly later Berwanger et al.[BDHK06] came with almost exactly the same definition. In addition to the results presented in [Obd06] (the definition of DAG-width and related results like comparison with other measures or game characterisation) the paper [BDHK06] also contains a polynomial-time algorithm for solving parity games on graphs of bounded DAG-width. In Chapter 5 we present an adapted version of this algorithm.

In Chapter 6 we discuss the strategy improvement algorithm for parity games of [VJ00]. We start by giving an overview of the algorithm, and continue by examining some aspects in more detail. We also present some new results.

Finally in Chapter 7 we present a brand new algorithm for solving parity

games. This algorithm, partly inspired by the strategy improvement algorithm mentioned above, is based around the notion of spine, a structural way of representing the possible winning sets and counter-strategies. We conclude with Chapter 8.

Chapter 2

Parity Games and Modal μ-calculus

In this chapter we present the material which will be needed in later chapters. We start by giving and explaining the definition of parity games, strategies etc. Also some concepts used in more than one chapter, like force sets, are explained here. We give brief information regarding complexity and determinacy of parity games. Then we present the modal μ-calculus, and show that there are linear reductions between the problem of solving parity games and modal μ-calculus model checking problem. Finally we present some related infinite games and show their relationship and relevance of these games to parity games.

2.1 Definitions

A *parity game* $\mathcal{G} = (V, E, \lambda)$ consists of a directed graph $G = (V, E)$, where V is a disjoint union of V_0 and V_1 (in the rest of the thesis we assume that this partition is implicit), and a parity function $\lambda : V \to \mathbb{N}$ (we assume $0 \notin \mathbb{N}$). As it is usually clear from the context, we sometimes talk about a parity game G – i.e. we identify the game with its game graph. For technical reasons we also assume that the edge relation $E : V \times V$ is total: that is, for all $u \in V$ there is $v \in V$ such that $(u, v) \in E$. The game $\mathcal{G}$ is played by two players P_0 and P_1 (also called EVEN and ODD[1]), who move a single token along edges of the graph G. The game starts in an initial vertex and players play indefinitely as follows: if the token is on a vertex $v \in V_0$ ($v \in V_1$), then P_0 (P_1) moves it along some edge

[1] Adam and Eve, Al and Ex, and many other names are also used in literature. We are not concerned that our second player is being 'odd', and this way it is much easier to remember who are we talking about.

$(v,w) \in E$ to w. As a result, a play of $\mathcal{G}$ is an infinite path $\pi = \pi_1\pi_2\ldots$, where $\forall i > 0.(\pi_i, \pi_{i+1}) \in E$.

Let $Inf(\pi) = \{v \in V \mid v \text{ appears infinitely often in } \pi\}$. Player P_0 *wins the play* π if $\max\{\lambda(v) \mid v \in Inf(\pi)\}$ is even, and otherwise player P_1 wins. (Often a dual winning condition is used: Player P_0 *wins the play* π iff $\min\{\lambda(v) \mid v \in Inf(\pi)\}$ is even. It does not matter which of these condition we use as long as we have a finite number of priorities. The two versions are sometimes referred to as 'big endian' and 'little endian' parity games.)

Example 2.1. *Fig. 2.1 shows a parity game of six vertices. The game is drawn in standard*[2] *graphical notation for parity games. Circles denote the vertices of player* P_0 *and boxes the vertices of player* P_1*. Priorities are written inside vertices.*

In this game player P_0 *can win from the shaded vertices by forcing a play to the vertex with priority four. Player* P_1 *has no choice in that vertex and must play to the vertex with priority three. The play will stay in the cycle with the highest priority four and therefore* P_0 *wins. Similarly* P_1 *wins the remaining (non-shaded) vertices by forcing the play to the cycle* $2,3,2$*.*

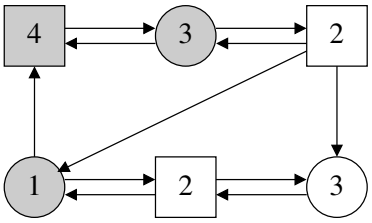


Figure 2.1: A parity game

If we fix a parity game $\mathcal{G} = (V,E,\lambda)$, we will often use the constants n, m and d to mean the following:

$n = |V|$ is the number of vertices in G

$m = |E|$ is the number of edges in G

$d = |\lambda(V)|$ is the number of priorities in $\mathcal{G}$

When defining and investigating algorithms for parity games, we quite often want to restrict ourselves to just a part of the game graph. We say that

[2]Some literature uses exactly the opposite notation, where circles are used to denote the vertices of player P_1. Some authors even use diamonds instead of circles. We stick to ours because circles are more 'even' and resemble the figure 0 in P_0.

the game $\mathcal{G}' = (V', E', \lambda')$ is a *subgame* of $\mathcal{G}$, if the game graph $G' = (V', E')$ is a subgraph of $G = (V, E)$ and for all $u \in V'$ there is $v \in V'$ such that $(u, v) \in E'$.

For $U \subseteq V$ we define $\mathcal{G}[U]$, which is the game $\mathcal{G}$ where the game graph $G[U]$ is the subgraph of G induced by U with the following modification: For each vertex $v \in U$ which does not have a successor in U we add an extra edge (v, v). Note that $\mathcal{G}[U]$ is a subgame of $\mathcal{G}$ if there is no such extra edge. We also define the game $\mathcal{G} \setminus U = \mathcal{G}[V \setminus U]$.

Definition 2.1. For a vertex $v \in V$ we define the function o (stands for 'owns') by the following prescription:

$$o(v) = \begin{cases} 0 & \text{if } v \in V_0 \\ 1 & \text{if } v \in V_1 \end{cases}$$

So $V_{o(v)} = V_0$ iff $v \in V_0$, $P_{o(w)} = P_1$ iff $w \in V_1$ etc.

In addition to general plays, we will often talk about cycles. *A cycle* of length k is a sequence of vertices $\rho = v_1 v_2 \dots v_k v_{k+1} = v_1$ such that for each $1 \leq i \leq k$. $(v_i, v_{i+1}) \in E$, and except for v_1 and v_{k+1} all vertices are pairwise different. We say that the cycle ρ is *even* if $\max\{\lambda(v_i) \mid 1 \leq i \leq k\}$ is even, otherwise the cycle is *odd*. If v_i is a vertex of a maximum priority on the cycle ρ we say that ρ is a *cycle on* v_i (also cycle on $\lambda(v_i)$).

Another useful notation is for the sets of vertices with the same priority. For a game $\mathcal{G} = (V, E, \lambda)$ and a priority p we put $V^p = \{v \in V \mid \lambda(v) = p\}$ – i.e. V^p is the set of all vertices with priority p. Similarly $V^{\geq p} = \{v \in V \mid \lambda(v) \geq p\}$ is the set of vertices with priority at least p and $V^{\leq p} = \{v \in V \mid \lambda(v) \leq p\}$ the set of vertices with priority at most p.

2.1.1 Reward Ordering

In addition to the standard ordering of priorities (by the relation '$<$'), it is often useful to have priorities ordered from the point of their 'attractiveness' for one of the players. I.e. for player P_0 a high even priority is more attractive than a low even one, which is still more attractive than any odd priority. We define the order $\sqsubset$ in the following way:

Definition 2.2 ($\sqsubset$)**.** For two priorities $p, q \in \mathbb{N}$ we write $p \sqsubset q$ if p is odd and q is even, or both p and q are odd and $p > q$, or both p and q are even and $p < q$. We

write $p \sqsubseteq q$ if $p \sqsubset q$ or $p = q$. For a game $\mathcal{G} = (V, E, \lambda)$ and two vertices $u, v \in V$ we write $u \sqsubset v$ if $\lambda(u) \sqsubset \lambda(v)$ and $u \sqsubseteq v$ if $\lambda(u) \sqsubseteq \lambda(v)$.

The ordering $\sqsubseteq$ is also sometimes called the 'reward' ordering in the literature, where the reward is of course for the player P_0. With a little abuse of notation we can extend the order $\sqsubseteq$ to sets of priorities.

Definition 2.3. Let $W, W' \subseteq \mathbb{N}$, and let $U = W \div W' = (W \setminus W') \cup (W' \setminus W)$ be the symmetric difference of W and W'. We put $W \sqsubset W'$ iff $\max(U) \in W$ is odd or $\max(U) \in W'$ is even. We put $W \sqsubseteq W'$ if $W \sqsubset W'$ or $W = W'$.

2.1.2 Strategies

With each game there is an associated notion of a strategy. We will introduce a few different types of strategies. Here is the most general definition.

Definition 2.4. A *(total) strategy* σ (τ) for P_0 (P_1) is a function $\sigma : V^*V_0 \to V$ ($\tau : V^*V_1 \to V$) which assigns to each play $\pi.v \in V^*V_0$ ($\in V^*V_1$) a vertex w such that $(v, w) \in E$. A player *uses a strategy* σ in the play $\pi = \pi_1\pi_2\ldots\pi_k\ldots$, if $\pi_{k+1} = \sigma(\pi_1\ldots\pi_k)$ for each vertex $\pi_k \in V_i$. A strategy σ is *winning* for a player and a vertex $v \in V$ if she wins every play that starts from v using σ. (Throughout the paper we use σ to denote a strategy of P_0 and τ a strategy of P_1. If the player is not important, we also use σ. The meaning should be clear from the context.)

Using strategies we extend the notion of winning to games.

Definition 2.5. If we fix an initial vertex v, then we say player P_i *wins the game* $\mathcal{G}(v)$ if he has a strategy σ such that using σ he wins every play starting in v. By *solving* the game $\mathcal{G}$ we mean finding the winner of $\mathcal{G}(v)$ for each vertex $v \in V$. I.e. to each game $\mathcal{G}$ and a vertex $v \in V(G)$ there is an associated decision problem of finding a winner for $\mathcal{G}[v]$. When talking about solving game in this thesis we usually mean answering this decision problem. Finally we say that player *wins the game* $\mathcal{G}$ if he has a strategy σ such that using σ he wins the game $\mathcal{G}(v)$ for each $v \in V$.

Strategies do not have to be total functions. If they are not we talk about *partial strategies*. If σ is a partial strategy we say that P_0 uses σ in a play if at each prefix π' of the play π where $\sigma(\pi')$ is defined P_0 always chooses $\sigma(\pi')$ as the next vertex.

2.2 Memoryless Determinacy and Complexity

A *memoryless strategy*[3] σ (τ) for P_0 (P_1) is a function $\sigma : V_0 \to V$ ($\tau : V_1 \to V$) which assigns to each vertex $v \in V_0$ ($v \in V_1$) a vertex w such that $(v,w) \in E$. I.e. memoryless strategies do not consider the history of the play so far, but only the vertex the play is currently in. We use Σ_0 (Σ_1) to denote the set of memoryless strategies of player P_0 (P_1).

Definition 2.6 (G^τ_σ)**.** For game $\mathcal{G} = (V, E, \lambda)$ and (partial) memoryless strategies $\sigma \in \Sigma_0, \tau \in \Sigma_1$ we define $\mathcal{G}^\tau_\sigma = (V, E^\tau_\sigma, \lambda)$ to be the subgame induced by strategies σ and τ where

$$\begin{aligned} E^\tau_\sigma &= \{(v,w) \in E \mid v \in V_0, \text{ and } \sigma(v) = w \text{ or } \sigma(v) \text{ is undefined}\} \\ &\cup \{(v,w) \in E \mid v \in V_1, \text{ and } \tau(v) = w \text{ or } \tau(v) \text{ is undefined}\} \end{aligned}$$

In the case that one of the strategies σ, τ is an empty partial strategy, we omit the respective index and write just G_σ, G^τ (as well as E_σ, E^τ). In the following we often use the notation $v \to w$ and $v \to^* w$ to represent edges and paths between v and w.

Parity games are determined. By that we mean the following theorem.

Theorem 2.1. *For each parity game $\mathcal{G} = (V, E, \lambda)$ we can partition the set V into two sets W_0 and W_1 such that the player P_0 has a winning strategy for $\mathcal{G}(v)$ if, and only if, $v \in W_0$.*

The result follows from a much more general theorem of Martin [Mar75], which says that every Borel game is determined. In [Mos84] and [EJ91] it was independently proved that memoryless strategies suffice for parity games.

Using the memoryless determinacy of parity games it is easy to show that parity games are in NP∩co-NP:

Theorem 2.2. *The problem of solving parity games is in the class* NP∩*co*-NP.

Proof. To check whether a vertex v belongs to W_0 we can just guess a memoryless strategy $\sigma \in \Sigma_0$ and in polynomial time check whether there is an odd cycle in G_σ reachable from the vertex v. If not, then σ is a winning strategy

[3] Also called 'history-free' or 'positional' strategy in the literature.

for P_0 in the game $\mathcal{G}(v)$. To show that the problem is also in co-NP it suffices to note that by determinacy $v \notin W_0 \iff v \in W_1$, and we can therefore use the same algorithm as before for the player P_1. □

Thanks to Jurdziński we have a slightly tighter complexity bound.

Theorem 2.3 ([Jur98])**.** *The problem of solving parity games is in the class* UP∩*co-*UP*.*

The class UP is believed to be a rather weak subclass of NP. For completeness here is the definition of the class UP (see [Pap94] for more details).

Definition 2.7. A decision problem is in the class UP(Unambiguous Non-deterministic Polynomial Time), if there is a polynomial time non-deterministic Turing machine recognising the associated language such that for each input that is accepted it accepts by exactly one computation.

The proof of Jurdziński goes by reduction of parity games to discounted payoff games, where the UP∩co-UP upper bound follows from the result of Zwick and Paterson [ZP96].

2.2.1 Finite Parity Games

Finite parity game (FPG) $\mathcal{G} = (V, E, \lambda)$ is defined in almost the same way as the standard parity game, with two differences: The play of FPG stops as soon as we reach some vertex v for the second time (i.e. the play is of the form $\pi_1.v.\pi_2.v$, where all vertices in $\pi_1.v.\pi_2$ are pairwise distinct). The vertex w with the highest priority on the loop $v.\pi_2.v$ then determines the winner – player P_0 wins iff $\lambda(w)$ is even.

Since the parity games are memorylessly determined, finite parity games are equivalent to standard (infinite) games. More precisely σ is a winning strategy for an infinite parity game $\mathcal{G}(v)$ iff it is winning in the finite parity game $\mathcal{G}(v)$. Therefore if we have a fixed parity game $\mathcal{G}$ and strategy σ, then the player P_0 wins $\mathcal{G}(v)$ using σ if there is no odd cycle reachable from v in the graph G_σ.

In the spirit of Ehrenfeucht and Mycielski [EM79] finite parity games can be used to prove memoryless determinacy of parity games. The argument goes like this. Finite parity games are finite two-player games of perfect information

and therefore are determined. The next step is to show that FPGs are memoryless determined. Finally it is shown that a winning strategy in FPG is also a winning strategy in the associated parity game and vice versa. In [EM79] this technique was used to show memoryless determinacy of mean payoff games, the proof for parity games was explicitly written down in [BSV04].

2.3 Equivalent Definitions, Normal Forms

The definition of parity games presented in Section 2.1 is very general. For example there is no relationship between the player owning a vertex v and the priority $\lambda(v)$ of this vertex. Similarly we cannot assume that from a vertex of player P_0 we always move to a vertex of player P_1 (i.e. that the players alternate in their moves). This usually makes describing the algorithms working on parity games a bit awkward. The question is whether this is really necessary. In this section we show how we can restrict the definition of parity games while staying in the same class of games, and not necessarily changing the complexity.

In the text to follow we will often claim that two parity games $\mathcal{G} = (V, E, \lambda)$ and $\mathcal{G}' = (V', E', \lambda')$ are equivalent. By equivalence we mean here that for each vertex $v \in V$ player P_0 wins $\mathcal{G}(v)$ if, and only if, he wins $\mathcal{G}'(v)$. In all the cases $V \subseteq V'$ will hold by construction, and therefore the equivalence is well defined.

We start by showing that we can restrict ourselves to games where every vertex has out-degree at most two. We call such games *binary parity games*.

Lemma 2.1. *Any parity game $\mathcal{G} = (V, E, \lambda)$ can be converted into an equivalent game $\mathcal{G}' = (V', E', \lambda')$ where every vertex has at most two successors. Moreover $|V'| < |V|^2$.*

Proof. Let v be a vertex with k successors $v_1, v_2, \ldots, v_k$. If $k \leq 2$ we are done. For $k > 2$ we introduce a new vertex w (i.e. $V' = V \cup \{w\}$) with $\lambda'(w) = \lambda(v)$, and change the edge relation as follows: $E' = (E \setminus \{(v, v_i) \mid 1 < i \leq k\}) \cup \{(v, w)\} \cup \{(w, v_i) \mid 1 < i \leq k\}$. Finally we put $\lambda'(v) = \lambda(v)$ for all vertices $v \in V$. It is obvious that both $\mathcal{G}$ and $\mathcal{G}'$ are equivalent, v has only two successors (in G') and w has $k-1$ successors. By iterative application of the argument above we introduce $k-2$ new vertices while dealing with the vertex v. As every vertex has at most n successors and there are n vertices, the number of the new vertices introduced is bounded by $n.(n-2)$ (just for a reminder, $n = |V|$). □

Another useful restriction is to have games where the priorities of vertices are distinct.

Definition 2.8. A parity game $\mathcal{G} = (V, E, \lambda)$ is a parity *game with a maximum number of priorities*, if for each $u, v \in V$, $u \neq v$ we have $\lambda(u) \neq \lambda(v)$ (i.e. if λ is injective).

If we have a game with a maximum number of priorities we can identify vertices with their priorities, i.e. to put $V \subseteq \mathbb{N}$, and therefore also identify $\mathcal{G}$ with its game graph G. This allows us to extend our notation to omit the parity function λ, e.g. we can write directly $u \leq v$ instead of $\lambda(u) \leq \lambda(v)$. Nevertheless we still need to know the partition of V into V_0 and V_1.

Parity games with maximum number of priorities are equivalent to standard parity games.

Lemma 2.2. *Any parity game $\mathcal{G} = (V, E, \lambda)$ can be converted into an equivalent game $\mathcal{G}' = (V, E, \lambda')$ with a maximum number of priorities.*

Proof. As follows from the wording of the proposition, we leave V and E unchanged and modify only the parity function λ. The construction works as follows. Choose $p \in \lambda(V)$ such that $|V^p| > 1$ and let $v \in V^p$. Then we put

$$\lambda'(u) = \begin{cases} \lambda(u) & \text{if } \lambda(u) < p \vee u = v \\ \lambda(u) + 2 & \text{otherwise} \end{cases}$$

Now v is the only vertex with priority p, and $\{w \mid \lambda'(w) = p+2\} = V^p \setminus \{v\}$. It is obvious that a play in $\mathcal{G}$ is winning iff it is winning in $\mathcal{G}'$, as there is no vertex with priority $p+1$ and all other vertices keep their parity and relative ordering. By iterative application of the construction we get a game with maximum number of priorities. □

Note that this construction does not change the game graph at all, and particularly does not increase its size. However if we want to study the exact complexity of an algorithm with respect to the number of priorities, we lose this information. On the other hand if we are interested in existence of a polynomial-time algorithm this restriction (as well as all others presented in this chapter) does not matter.

Another assumption we can make is that every player owns exactly the vertices of his own priority, therefore eliminating the need for knowing the partition of V into V_0 and V_1.

Lemma 2.3. *Any parity game $\mathcal{G} = (V, E, \lambda)$ can be converted into an equivalent game $\mathcal{G}' = (V', E', \lambda')$ such that $\forall v \in V'.v \in V'_0 \iff \lambda'(v)$ is even. Moreover $|V'| < 2.|V|$.*

Proof. We can assume that no vertex of V has a priority 1 or 2. If this is not the case we can increase the priority of each vertex by two. Take a vertex $v \in V$ violating the assumption. Without loss of generality consider the case $v \in V_0$ and $\lambda(v) = p$ is odd. We introduce a new vertex v' of P_1, put $\lambda'(v') = p, \lambda'(v) = 2$, and modify E by replacing each edge $(u, v) \in E$ with a pair of edges (u, v'), (v', v). As P_1 has no choice in v' (there is only one outgoing edge) and $\max(\lambda(v'), \lambda(v)) = p$, the new game is equivalent to $\mathcal{G}$. By iterative application of the construction above we can convert $\mathcal{G}$ into a game satisfying that each player own vertices of his priority. Because each newly introduced vertex satisfies this restriction, the construction finishes in at most n iterations adding one vertex each. □

By a similar construction we can also convert any parity game into one in which players alternate their moves. The edge relation E of such a game must satisfy $E \subseteq V_0 \times V_1 \cup V_1 \times V_0$, and in that case we call such a game *0-1 bipartite parity game*.

Lemma 2.4. *Any parity game $\mathcal{G} = (V, E, \lambda)$ can be converted into an equivalent game $\mathcal{G}' = (V', E', \lambda')$ such that $E' \subseteq V_0 \times V_1 \cup V_1 \times V_0$. Moreover $|V'| \leq |V|^2 + |V|$.*

Proof. As in the previous proof assume that there is no vertex with priority 1 or 2. We replace edge $(u, w) \in V_0 \times V_0$ with two edges (u, v) and (v, w), where $v \in V'_1$ is a new vertex with $\lambda'(v) = 1$. Similarly we split each edge $(u, w) \in V_1 \times V_1$ with a new vertex $v \in V'_0$ with $\lambda'(v) = 2$. This new game is clearly equivalent to the original game and the number of new vertices is bounded by the number of edges in G, which is in turn bounded by $|V|^2$. □

To sum up, for the purposes of proving properties of parity games and establishing whether there is a polynomial-time algorithm for solving these games, we prefer to use games in the following normal form:

Definition 2.9. The parity game $\mathcal{G} = (V, E, \lambda)$ is in *normal form* if it is a game with a maximal number of priorities such that $v \in V_0$ iff $\lambda(v)$ is even.

As both the parity function λ and the partition of V are implicit, we can identify the parity game $\mathcal{G} = (V, E, \lambda)$ in normal form with its game graph $G = (V, E)$. We will therefore freely talk about 'parity game $G = (V, E)$' in this case.

That every parity game can be turned into one in normal form is a corollary of Lemma 2.2 and Lemma 2.3.

Corollary 2.1. *Each parity game $\mathcal{G} = (V, E, \lambda)$ can be turned into an equivalent parity game $\mathcal{G}' = (V', E')$ in normal form, where $|V'| \leq 2.|V|$.*

Finally combining all the requirements we get:

Definition 2.10. Parity game $G = (V, E)$ is in *strong normal form*, if

- $G = (V, E)$ is in normal form, and
- each vertex of V has out-degree at most two, and
- the game graph is bipartite.

Corollary 2.2. *Each parity game $\mathcal{G} = (V, E, \lambda)$ can be turned into an equivalent parity game $\mathcal{G}' = (V', E', \lambda')$ in strong normal form, where $|V'| = O(|V|^2)$.*

Proof. We first apply the Lemma 2.1 to get a game where vertices have out-degree at most two. Note that the number of edges of this graph is at most $2.|V|^2$. In the next step we convert the game into a bipartite one (Lemma 2.4), and follow by application of Lemma 2.3. The number of introduced edges is linear in the number of edges already present. Finally we convert the game into one with a maximal number of priorities (Lemma 2.2). □

2.4 Force Sets

A notion we use a lot in this thesis is the one of forcing and force sets [Tho95, McN93]. Starting with a set of vertices $S \subseteq V$, the force set of S for player P_i is the set of all vertices from which player P_i can force a play to S. Alternative name for force sets used in the literature is *attractor sets*. Here is a formal definition of force set:

Definition 2.11 (Force set)**.** For player P_i, and $S \subseteq V$ we define $F_i(S)$, the *force set* of S for player P_i as a fixed point of the following system of equations:

$$\begin{aligned} F_i^0(S) &= S \\ F_i^{k+1}(S) &= F_i^k(S) \cup \\ &\quad \{u \in V_i \mid \exists v \in F_i^k(S).(u,v) \in E\} \cup \\ &\quad \{u \in V_{1-i} \mid \forall v \in V.(u,v) \in E \implies v \in F_i^k(S)\} \end{aligned}$$

Definition 2.12 (Reachability set)**.** We define $R(S)$, the set of vertices from which we can reach $S \subseteq V$, as

$$R(S) = \{v \in V \mid \exists w \in S \text{ s.t. there is a path } v \rightarrow^* w \text{ in } G\}$$

In both cases $F_i(S)$ and $R(S)$ we overload the notation and write $F_i(v)$ ($R(v)$) instead of $F_i(\{v\})$ ($R(\{v\})$). We also write $R^\sigma(S)$ (and $R^\sigma(v)$) if we restrict the computation of the set R to the graph G_σ, where the strategy σ is fixed.

Definition 2.13. If $v \in F_i(X)$ then the *rank* of v, written $rank(v, F_i(X))$, is the least index k such that $v \in F_i^k(X)$. Given $F_0(X)$ and a strategy $\sigma \in \Sigma_0$ we say that σ is a *rank strategy* if for each $v \in V_0 \cap (F_0(X) \setminus X)$ we have $rank(v, F_0(X)) = rank(\sigma(v), F_0(X)) + 1$.

The following property of parity games says that solving a parity game is equivalent to having an algorithm which for each parity game identifies at least one vertex in the winning set W_0 or W_1.

Theorem 2.4. *Let $\mathcal{G} = (V, E, \lambda)$ be a parity game and $S \subseteq W_i(\mathcal{G})$ be a part of the winning region of player P_i. Then also $F_i(S) \subseteq W_i(\mathcal{G})$. Moreover $\mathcal{G}' = \mathcal{G} \setminus F_i(S)$ is a subgame of $\mathcal{G}$ and for $w \in V(G')$ we have $w \in W_i(\mathcal{G}) \iff w \in W_i(\mathcal{G}')$.*

Proof. The first claim, that $F_i(S) \subseteq W_i(\mathcal{G})$, is obvious. It follows from the fact that player P_{1-i} cannot leave (by definition of the force set) the set W_i. Next we show that $\mathcal{G}'$ is a subgame of $\mathcal{G}$. If it is not, then there must be a vertex $v \in V(G')$ s.t. it has no successor in $V(G')$. Let j be the least index such that v has all the successors in $F_i^j(S)$. By definition of force set then $v \in F_i^{j+1}(S)$, a contradiction.

For the second part first assume $w \in W_i(\mathcal{G})$. Therefore there is a winning strategy $\sigma \in \Sigma_i$ s.t. there is no opponents cycle in G. But by definition of F_i it is not possible that $v \in V(G')$ and $\sigma(v) \in F_i(S)$, so σ is winning in $\mathcal{G}'$. The opposite implication holds for the same reasons. □

Sometimes we need a slightly more general version of force sets. For two sets of vertices $U, W \subseteq V$ we want to compute the set of vertices from which player P_i can force the play to U without leaving the set W:

Definition 2.14 (Force set)**.** For player P_i, and $U, W \subseteq V$ we define $F_i(U,W)$, the *force set* of U for player P_i with respect to W, as a fixed point of the following:

$$\begin{aligned} F_i^0(U,W) &= U \cap W \\ F_i^{k+1}(U,W) &= F_i^k(U,W) \cup \\ &\quad \{u \in V_i \cap W \mid \exists v \in F_i^k(U,W).(u,v) \in E\} \cup \\ &\quad \{u \in V_{1-i} \cap W \mid \forall v \in W.(u,v) \in E \implies v \in F_i^k(U,W)\} \end{aligned}$$

Similarly we can restrict the reachability function $R(U)$. We define $R(U,W)$ to be the set of all vertices which can reach $U \subseteq V$ while staying in the set $W \subseteq V$:

$$\begin{aligned} R^0(U,W) &= U \cap W \\ R^{k+1}(U,W) &= R^k(U,W) \cup \\ &\quad \{u \in W \mid \exists v \in R^k(U,W).(u,v) \in E\} \end{aligned}$$

2.5 Modal μ-calculus

The modal μ-calculus is a fixpoint logic of Kozen [Koz83]. It is an extension of Hennessy-Milner logic with variables and fixpoint operators ν (maximal fixpoint operator) and μ (minimal fixpoint operator).

Definition 2.15 (syntax)**.** Let *Var* be a countable set of variables. The *modal μ-calculus* is a set of formulas defined by the syntax

$$\varphi ::= \mathtt{tt} \mid \mathtt{ff} \mid X \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid [\cdot]\varphi \mid \langle\cdot\rangle\varphi \mid \nu X.\varphi \mid \mu X.\varphi$$

where $X \in \mathit{Var}$.

Before we present the semantics, we need the model on which we will evaluate μ-calculus formulas. This is usually done on transition systems.

Definition 2.16. A (unlabelled) *transition system* is a pair $\mathcal{T} = (S, \to)$, where:

- S is a set of *states*,
- $\to \subseteq S \times S$ is a *transition relation*.

Instead of $(s,t) \in \to$ we write $s \to t$.

As we can see, unlabelled transition systems are just directed graphs and we will treat them as such. The semantics of the μ-calculus is defined with respect to a valuation of free variables. *Valuation* $\mathcal{V}$ is defined as a mapping $\mathcal{V} : Var \to 2^S$, assigning to every variable X a set of states. Valuation $\mathcal{V}[X := T]$, where $T \subseteq S$, is the same as the valuation $\mathcal{V}$ except for the variable X for which $\mathcal{V}[X := T](X) = T$. Now we can define the semantics as follows:

$$\begin{aligned}
[\![\mathtt{tt}]\!]^{\mathcal{T}}_{\mathcal{V}} &= S \\
[\![\mathtt{ff}]\!]^{\mathcal{T}}_{\mathcal{V}} &= \emptyset \\
[\![X]\!]^{\mathcal{T}}_{\mathcal{V}} &= \mathcal{V}(X) \\
[\![\varphi_1 \wedge \varphi_2]\!]^{\mathcal{T}}_{\mathcal{V}} &= [\![\varphi_1]\!]^{\mathcal{T}}_{\mathcal{V}} \cap [\![\varphi_2]\!]^{\mathcal{T}}_{\mathcal{V}} \\
[\![\varphi_1 \vee \varphi_2]\!]^{\mathcal{T}}_{\mathcal{V}} &= [\![\varphi_1]\!]^{\mathcal{T}}_{\mathcal{V}} \cup [\![\varphi_2]\!]^{\mathcal{T}}_{\mathcal{V}} \\
[\![[\cdot]\varphi]\!]^{\mathcal{T}}_{\mathcal{V}} &= \{s \mid \forall\, t \in S \text{ s.t. } s \to t \text{ we have } t \in [\![\varphi]\!]^{\mathcal{T}}_{\mathcal{V}}\} \\
[\![\langle\cdot\rangle\varphi]\!]^{\mathcal{T}}_{\mathcal{V}} &= \{s \mid \exists\, t \in S \text{ s.t. } s \to t \text{ and } t \in [\![\varphi]\!]^{\mathcal{T}}_{\mathcal{V}}\} \\
[\![\nu X.\varphi(X)]\!]^{\mathcal{T}}_{\mathcal{V}} &= \bigcup\{T \subseteq S \mid T \subseteq [\![\varphi]\!]^{\mathcal{T}}_{\mathcal{V}[X:=T]}\} \\
[\![\mu X.\varphi(X)]\!]^{\mathcal{T}}_{\mathcal{V}} &= \bigcap\{T \subseteq S \mid [\![\varphi]\!]^{\mathcal{T}}_{\mathcal{V}[X:=T]} \subseteq T\}
\end{aligned}$$

Let $\mathcal{T} = (S, \to)$ be a transition system, $s \in S$ a state of this transition system, $\mathcal{V}$ a valuation and φ a modal μ-calculus formula. Then we say that the formula φ holds in the state s of $\mathcal{T}$ under valuation $\mathcal{V}$, written as $(\mathcal{T}, s) \models_{\mathcal{V}} \varphi$, if $s \in [\![\varphi]\!]^{\mathcal{T}}_{\mathcal{V}}$. If the formula φ is a sentence (closed formula), then we write just $(\mathcal{T}, s) \models \varphi$ as the set of states defined by the formula does not depend on the valuation. The model checking problem for the modal μ-calculus is the question whether $(\mathcal{T}, s) \models_{\mathcal{V}} \varphi$.

2.5.1 Alternation

The number of alternations between the minimal and maximal fixed points in a μ-calculus formula φ is an important factor in the complexity of model checking problem for φ. The alternation hierarchies have been first defined and studied by Emerson and Lei [EL86] and Niwiński [Niw86]. See also [Niw97] for a comparison of the two slightly different definitions.

Even though we could simply count the syntactic alternations between the least and greatest fixed point in the formula, we present here the more precise definition of Niwiński, which also gives tighter complexity bounds.

Definition 2.17. For a formula φ of modal μ-calculus we define the *alternation depth* $\delta(\varphi)$ inductively as:

$$\delta(\mathtt{tt}) = \delta(\mathtt{ff}) = \delta(X) = 0$$

$$\delta(\varphi_1 \wedge \varphi_2) = \delta(\varphi_1 \vee \varphi_2) = \max(\delta(\varphi_1), \delta(\varphi_2))$$

$$\delta(\langle\cdot\rangle\varphi) = \delta([\cdot]\varphi) = \delta(\varphi)$$

$$\delta(\nu X.\varphi) = \max(\{1, \delta(\varphi)\} \cup \{\delta(\mu Y.\psi) + 1 \mid \mu Y.\psi \text{ is a subformula of } \varphi \text{ and } X \text{ is free in } \mu Y.\psi\})$$

$$\delta(\mu X.\varphi) = \max(\{1, \delta(\varphi)\} \cup \{\delta(\nu Y.\psi) + 1 \mid \nu Y.\psi \text{ is a subformula of } \varphi \text{ and } X \text{ is free in } \nu Y.\psi\})$$

2.5.2 Parity Games to μ-calculus

It is not very hard to show how to reduce the problem of solving a parity game to the μ-calculus model checking problem. The first to present such a reduction were Emerson and Jutla in [EJ91]. Let $\mathcal{G} = (V, E, \lambda)$ be a parity game. Without loss of generality we can assume that the set $\lambda(V) = \{1, \ldots, n\}$ and the highest priority n is even (if it is odd, the formula φ would start $\mu Z_n.\nu Z_{n-1}\ldots$ instead). Take the graph $G = (V, E)$ as the transition system $\mathcal{T}$, and the formula

$$\varphi = \nu Z_n.\mu Z_{n-1}.\ldots.\mu Z_1.\left(\bigvee_{i\leq n}(Y_0 \wedge X_i \wedge \langle\cdot\rangle Z_i) \vee \bigvee_{i\leq n}(Y_1 \wedge X_i \wedge [\cdot] Z_i)\right)$$

Finally let $\mathcal{V}$ be a valuation satisfying

$$\begin{aligned} \mathcal{V}(X_i) &= \{v \in V \mid \lambda(v) = i\} \\ \mathcal{V}(Y_0) &= V_0 \\ \mathcal{V}(Y_1) &= V_1 \end{aligned}$$

Theorem 2.5. *Let $\mathcal{G} = (V, E, \lambda)$ be a parity game, and be $\mathcal{T}$, $\mathcal{V}$, and φ be the transition system, valuation and μ-calculus formula given by the translation above. Then*

$$v \in W_0(\mathcal{G}) \iff (\mathcal{T}, v) \models_{\mathcal{V}} \varphi$$

Note that the alternation depth of this formula is equal to the number of priorities in the parity game. This is no coincidence. In the next section we will see that for the translation going the opposite direction this holds as well.

2.5.3 μ-calculus to Parity Games

In this section we show how to reduce the model checking problem for the modal μ-calculus to the problem of solving parity games. This construction can be described as first translating the μ-calculus formula to a parity tree automaton and taking the synchronised product of this automaton and the system to be checked in the spirit of [EJ91, Sti95]. The translation given here is adapted from [Sti01].

Let $\mathcal{T} = (S, \to)$ be a transition system and φ a μ-calculus formula. We will construct the parity game $\mathcal{G} = (V, E, \lambda)$ as follows: For the set of vertices we take all pairs $S \times \mathit{Sub}(\varphi)$, where $\mathit{Sub}(\varphi)$ is the set of all subformulas of φ. Moreover let $\delta(\varphi)$ be the alternation depth of φ as described in Section 2.5.1. Finally take $\psi \in \mathit{Sub}(\varphi)$, $s \in S$ and $v = (s, \psi)$. We define the edge relation E, the partition of V into V_0 and V_1, and the priority function λ by the following set of rules:

1. $\psi = X$, X is free in φ, $s \in \mathcal{V}(X)$
   $\lambda(v) = 2$, $(v, v) \in E$

2. $\psi = X$, X is free in φ, $s \notin \mathcal{V}(X)$
   $\lambda(v) = 1$, $(v, v) \in E$

3. $\psi = \mathtt{tt}$
   $\lambda(v) = 2$, $(v, v) \in E$

4. $\psi = \mathtt{ff}$
   $\lambda(v) = 1$, $(v, v) \in E$

5. $\psi = \psi_1 \wedge \psi_2$
   $v \in V_1$, $(v, (s, \psi_1)) \in E$ and $(v, (s, \psi_2)) \in E$

6. $\psi = \psi_1 \vee \psi_2$
   $v \in V_0$, $(v, (s, \psi_1)) \in E$ and $(v, (s, \psi_2)) \in E$

7. $\psi = [\cdot]\psi'$ and $\{t \mid s \to t\} = \emptyset$
   $\lambda(v) = 2$, $(v, v) \in E$

8. $\psi = [\cdot]\psi'$ and $T = \{t \mid s \to t\} \neq \emptyset$
   $v \in V_1$, $(v, (t, \psi')) \in E$ for all $t \in T$

9. $\psi = \langle\cdot\rangle\psi'$ and $\{t \mid s \to t\} = \emptyset$
   $\lambda(v) = 1$, $(v,v) \in E$

10. $\psi = \langle\cdot\rangle\psi'$ and $T = \{t \mid s \to t\} \neq \emptyset$
    $v \in V_0$, $(v,(t,\psi')) \in E$ for all $t \in T$

11. $\psi = \nu X_i.\psi'$
    $(v,(s,\psi')) \in E$, and $\lambda(v) = \begin{cases} \delta(\psi)+2 & \text{if } \delta(\psi) \text{ is even} \\ \delta(\psi)+1 & \text{otherwise} \end{cases}$

12. $\psi = \mu X_i.\psi'$
    $(v,(s,\psi')) \in E$, and $\lambda(v) = \begin{cases} \delta(\psi)+2 & \text{if } \delta(\psi) \text{ is odd} \\ \delta(\psi)+1 & \text{otherwise} \end{cases}$

13. X_i and $\rho X_i.\psi \in Sub(\varphi)$
    $(v,(s,\rho X_i.\psi)) \in E$

In the cases where $\lambda(v)$ is not defined we put $\lambda(v) = 1$, and similarly where it is not given we put $v \in V_0$. Finally we put into V only those pairs (t,ψ) reachable from the vertex (s,φ) for some $s \in S$.

Example 2.2. *In Fig. 2.3 you can see the parity game created by this construction for the formula $\mu X.P \vee (Q \wedge [\cdot]X)$), transition system $\mathcal{T}$ (Fig. 2.2), and valuation $\mathcal{V}$ such that $\mathcal{V}(P) = \{c\}$ and $\mathcal{V}(Q) = \{a,b\}$.*

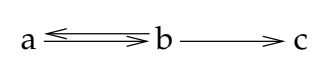
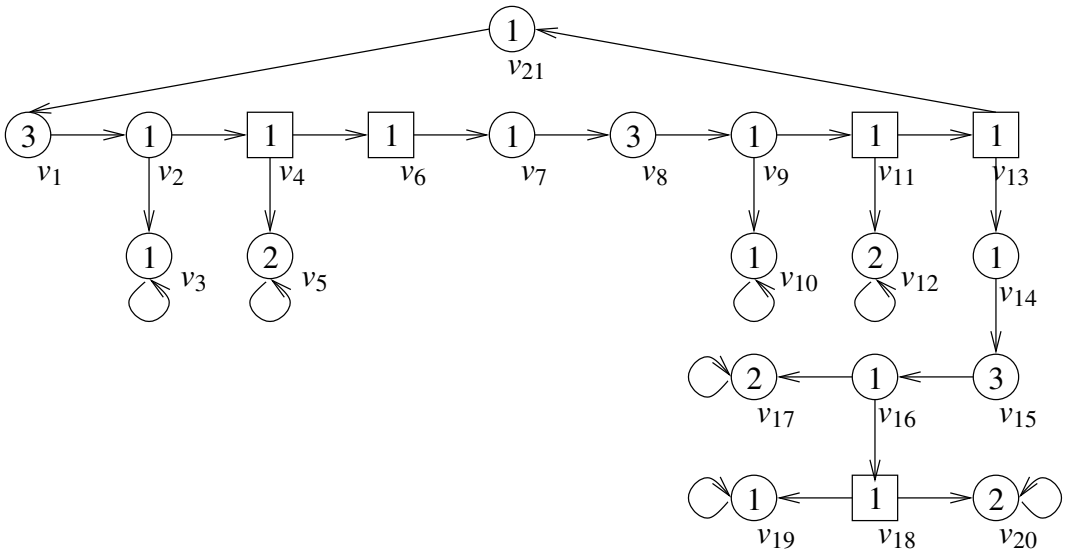


Figure 2.2: Transition system $\mathcal{T}$

Theorem 2.6. *Let $\mathcal{T} = (S,\to)$ be a transition system, φ a μ-calculus formula, and $\mathcal{G} = (V,E,\lambda)$ the parity game given by the translation above. Then*

$$(\mathcal{T},s) \models \varphi \iff (s,\varphi) \in W_0(\mathcal{G})$$

Also note that the number of priorities is equal to the alternation depth of a formula. (More precisely its depth plus two. We could modify the construction to get rid of this artefact, but for the price of losing simplicity.)



$v_1 : (a, \mu X.P \vee (Q \wedge [\cdot]X))$
$v_2 : (a, P \vee (Q \wedge [\cdot]X))$
$v_3 : (a, P)$
$v_4 : (a, Q \wedge [\cdot]X)$
$v_5 : (a, Q)$
$v_6 : (a, [\cdot]X)$
$v_7 : (b, X)$
$v_8 : (b, \mu X.P \vee (Q \wedge [\cdot]X))$
$v_9 : (b, P \vee (Q \wedge [\cdot]X)$
$v_{10} : (b, P)$
$v_{21} : (a, X)$

$v_{11} : (b, Q \wedge [\cdot]X)$
$v_{12} : (b, Q)$
$v_{13} : (b, [\cdot]X)$
$v_{14} : (c, X)$
$v_{15} : (c, \mu X.P \vee (Q \wedge [\cdot]X))$
$v_{16} : (c, P \vee (Q \wedge [\cdot]X))$
$v_{17} : (c, P)$
$v_{18} : (c, Q \wedge [\cdot]X)$
$v_{19} : (c, Q)$
$v_{20} : (c, [\cdot]X)$

Figure 2.3: Parity game for $\mathcal{T}$, $\mu X.P \vee (Q \wedge [\cdot]X)$ and $\mathcal{V}(P) = \{c\}$, $\mathcal{V}(Q) = \{a, b\}$.

2.6 More General Games

In this section we are going to present two more two-player games related to parity games. The reason why we mention them here is that 1) the complexity of solving these games is also in NP∩co-NP, and 2) the problem of finding a winner in a parity game can be reduced to the problem of finding a winner in either of these two games. Moreover the strategy improvement algorithm we will talk about in Chapter 6 originated in the strategy improvement algorithm for stochastic games [HK66], which also can be used to solve simple stochastic games [Con92].

2.6.1 Mean Payoff Games

Mean payoff games have been introduced by Ehrenfeucht and Mycielski in [EM79], and their associated decision problem was shown to belong NP∩co-NP by Zwick and Paterson [ZP96]. Here we present a decision version of the game.

The *mean payoff game* $\mathcal{G} = (V, E, \omega, \nu)$ consists of a directed graph $G = (V, E)$, where the vertex set V is a disjoint union of V_0 and V_1, a weight function $\omega : E \to \{-w, \ldots, 0, \ldots w\}$ assigning an integral weight between $-w$ and w to each edge of G, and finally an integral threshold $\nu \in \mathbb{N}$. The game is played in the same way as the parity game, the only difference is the winning condition. Player P_0 wins the infinite play $\pi = \pi_1 \pi_2 \ldots$ iff

$$\liminf_{n \to \infty} \frac{1}{n} \sum_{i=1}^{n} \omega((\pi_i, \pi_{i+1})) \geq \nu$$

The reduction from parity games to mean payoff games has been discovered independently by Puri [Pur95] and Jerrum [Sti95]. In [Jur98] the reduction has been used in the proof that the problem of solving parity games belongs to UP∩co-UP.

2.6.2 Simple Stochastic Games

Unlike both parity games and mean payoff games, simple stochastic games (SSG) are games of chance. They were introduced originally by Shapley [Sha53]. Condon [Con92] was the first to study simple stochastic games from the com-

plexity perspective, showing that the associated decision problem is also in NP∩co-NP. We present here the definition of SSG used in the latter paper.

The *simple stochastic game* $\mathcal{G} = (V, E, v_0, v_1)$ consists of a game graph $G = (V, E)$ with two special vertices $v_0, v_1 \in V$ called 0-sink and 1-sink, which have no successors. The vertex set $V \setminus \{v_0, v_1\}$ is partitioned into three sets of vertices V_0, V_1 and $V_{1/2}$. As in previous cases, player P_0 controls the vertices in V_0 and player P_1 the vertices in V_1. The vertices of $V_{1/2}$ are called *average vertices*, and have exactly 2 successors. Play is defined similarly as in the previous cases, with the following exception: if a play reaches an average vertex v, the successor of v is chosen uniformly at random (each with the probability $1/2$). Player P_0 wins the simple stochastic game $\mathcal{G}$ if he is able to reach the 0-sink with probability of at least $1/2$.

Simple stochastic games were the first of the three games we have seen here for which a sub-exponential ($2^{O(\sqrt{n})}$) algorithm was shown to exist [Lud95]. This has been later improved to a strongly sub-exponential (sub-exponential on graphs with unbounded vertex out-degree) algorithm running in $2^{O(\sqrt{n \cdot \log n})}$ in [Hal04]. The reduction from mean payoff games (through a variant of mean payoff games, called discounted payoff games) to simple stochastic games has been discovered by Zwick and Paterson in [ZP96].

Chapter 3

Algorithms for Solving Parity Games

In this chapter we are going to give a brief overview of the existing algorithms for solving parity games. The algorithms considered are not covered to the same extent. In some cases we give the full algorithm, whereas sometimes we just mention the complexity bound achieved. This chapter is meant to give an overview of the various ways of solving parity games, particularly focusing on the current state of the art. For detailed information on the algorithms pointers to the relevant sources are given.

We start by presenting a simple algorithm which is a consequence of the memoryless determinacy proof of McNaughton [McN93], adapted to parity games by Zielonka [Zie98]. In this only case we also prove correctness of this algorithm. After that we mention several other algorithms with better complexity bounds, which were published as algorithms for the modal μ-calculus model checking. Then we go on to, up till very recently, the best (from the complexity point of view) deterministic algorithm for solving parity games [Jur00]. This algorithm by Jurdziński is based on small progress measures, a concept defined for parity games by Walukiewicz [Wal96].

Next we discuss the available randomised sub-exponential algorithms of which the currently best one is by Björklund et al. [BSV03]. These results are ultimately based on non-trivial randomisation schemes of Ludwig [Lud95] and Kalai [Kal92]. We explain what is the underlying machinery and describe the exact complexity bounds.

A special place belongs to the strategy improvement algorithm of Vöge and Jurdziński [VJ00]. We deal with this algorithm in Chapter 6. Here we just mention that for this algorithm no exponential counterexample is known,

which is in contrast with most of the other algorithms.

Very recently Jurdziński, Paterson and Zwick [JPZ06] came up with a deterministic algorithm whose complexity matches the complexity of the best randomised algorithms. What may be surprising is that their algorithm is 'just' a clever modification of the simple algorithm presented in Section 3.1.

All the algorithms above deal with general parity games. In the last two sections we mention two fast algorithms for solving parity games on special classes of graphs. In the first of the two sections we consider 'undirected' graphs, i.e. graphs where the edge relation is symmetric. The observation of Serre [Ser03] is that for this class of graphs we have a linear-time algorithm for solving parity games. In the second section we consider parity games on trees with back edges. This class of graphs is in a sense both simple and complicated, occurring naturally in many areas of computer science. As observed by Niwiński [Niw], we have a polynomial time algorithm for solving parity games on trees with back edges. Neither of the two results have been published before and we think they present another facet of the challenging problem of solving parity games.

3.1 Simple Algorithm for Parity Games

The following simple exponential-time algorithm for solving parity games is based on the work of McNaughton [McN93]. For parity games it was first explicitly presented by Zielonka [Zie98]. The algorithm is obtained from a constructive proof of memoryless determinacy of parity games, and is implemented by the procedure `PGSolve`.

Theorem 3.1. *Let $\mathcal{G}$ be a parity game. Then* `PGSolve`*$(G)=(W_0(G), W_1(G))$. Moreover the running time of* `PGSolve` *is $2^{O(n)}$, where $n = |V(G)|$.*

Proof. The proof goes by induction on the size of $V(G)$. If $V(G) = \emptyset$ we are finished, since the theorem obviously holds. Otherwise $V(G)$ is non-empty and the algorithm works as follows. We start with the set $V^p = Y \subseteq V(G)$ of vertices of the highest priority p. W.l.o.g. we can assume p is even. Then we compute the solution (W_0, W_1) for the smaller game $\mathcal{G}' = \mathcal{G} \setminus F_0(Y)$. By induction hypothesis W_0 and W_1 are the winning sets of players P_0 and P_1 in the smaller game $\mathcal{G}'$.

Procedure `PGSolve(`G`)`

if $V(G) = \emptyset$ **then return** $(\emptyset, \emptyset)$
p:=max$\{\lambda(v) \mid v \in V(G)\}$
Y:=$\lambda^{-1}(p)$; i:=$o(p)$
(W_0, W_1):=`PGSolve` $(G \setminus F_i(Y))$
if $W_{1-i} = \emptyset$ **then**
    W_i:=$V(G)$
else
    (W_0, W_1):=`PGSolve` $(G \setminus F_{1-i}(W_{1-i}))$
    W_i:=$V(G) \setminus W_{1-i}$
return (W_0, W_1)

There are two separate cases to be considered. If $W_1 = \emptyset$, then the player P_0 has a winning strategy σ in the game $\mathcal{G}'$. By definition of force set player P_0 cannot leave the set $V(G')$, whereas player P_1 cannot leave the set $F_0(Y)$. Therefore if P_0 uses the strategy σ for vertices in $V(G')$ and a rank strategy for vertices in $F_0(Y)$, each play in the whole game $\mathcal{G}$ either stays in $V(G')$, or passes infinitely often through a vertex of priority p. In the first case the play is winning for the player P_0 since σ is a winning strategy in $\mathcal{G}'$ and in the second case the play is winning since the highest priority seen infinitely often is even.

In the second case W_1 is non-empty. As player P_0 cannot leave the set $V(G')$, $W_1 \subseteq W_1(\mathcal{G})$. By Theorem 2.4 also $F_1(W_1) \subseteq W_1(\mathcal{G})$. The algorithm now asks for solution of the game $\mathcal{G} \setminus F_1(W_1)$. By a similar argument as for the previous case, player $W_0 \subseteq W_0(\mathcal{G})$, and loses for all other vertices.

To obtain the complexity bound notice that at every iteration the procedure `PGSolve` is called recursively at most twice, in both cases on a smaller game. Except for the recursive calls the time of one iteration is bounded by the number of edges of G (which is how long the computation of the force sets could take), therefore is in $O(n^2)$. If we denote $T(n)$ the running time of the algorithm for a game with n vertices, we have $T(n) \leq 2.T(n-1) + O(n^2)$, and therefore $T(n) \in 2^{O(n)}$. □

Note that in this case we get a bound which is not dependent on the number of priorities. By a more careful analysis it is actually possible to decrease the bound on the running time to roughly $O(m \cdot (n/d)^d)$ [Jur00].

3.2 Better Deterministic Algorithms

Many algorithms which are used to solve parity games were originally formulated as algorithms for solving the modal μ-calculus model checking problem. We know there is a linear translation from parity games to the modal μ-calculus (see Section 2.5.2). Moreover this translation preserves the graph of the parity game, and the alternation depth of the resulting formula is equal to the number of priorities. Therefore the complexity bounds for the modal μ-calculus model checking problem translate directly to the problem of solving parity games.

Before continuing further let us remember that for a game $\mathcal{G} = (V, E, \lambda)$ we have defined $n = |V|, m = |E|$ and $d = |\lambda(V)|$. The standard algorithm of Emerson and Lei [EL86] has time complexity $O(m \cdot n^d)$. This has been later improved by Long, Browne, Clarke, Jha, and Marrero [LBC+94] to roughly $O(d^2 \cdot m \cdot n^{\lceil d/2 \rceil})$. A further improvement came from Seidl [Sei96], who showed how to decrease this bound to $O(d \cdot m \cdot \frac{n+d}{d}^{\lceil d/2 \rceil})$. Up till recently the best known algorithm has been the algorithm of Jurdziński based on small progress measures [Jur00]. Its time complexity is shown to be in $O(d \cdot m \cdot \frac{n}{\lfloor d/2 \rfloor}^{\lfloor d/2 \rfloor})$ and the algorithm can be made to work in time $O(d \cdot m \cdot \frac{n+d}{d}^{\lceil d/2 \rceil})$, thus matching the complexity of the previous algorithms. Moreover this algorithm works in space $O(d \cdot n)$, whereas the other two algorithms have exponential worst case space behaviour. As the small progress measures algorithm is quite interesting, we present it in the next section.

3.2.1 Small Progress Measures

Progress measures [KK91] are decorations of graphs whose local consistency guarantees some global, and often infinitary, properties of graphs. Progress measures have been used successfully for complementation of automata on infinite words and trees [Kla91, Kla94]. A similar notion, called signature, occurs in the study of modal μ-calculus [SE89], and signatures have also been used to prove the determinacy of parity games [EJ91, Wal96]. The algorithm is based on the notion of game parity progress measures, which were called consistent signature assignments by Walukiewicz [Wal96]. The algorithm presented here was obtained by Jurdziński [Jur00]. For detailed description of the algorithm and the related proofs we refer the reader to [Jur00].

In this section we stick as much as possible to the notation of [Jur00]. Therefore the parity games we consider the lowest priority appearing infinitely often is winning, and 0 is the lowest priority. The algorithm is built around a data structure M_G. If $\mathcal{G} = (V, E, \lambda)$ is a parity game, we use $n_p = |V^p|$ to denote the size of the set of vertices of priority p. Let $|k| = \{0, 1, \ldots, k-1\}$ be the set of k elements 0 to $k-1$. We can assume that priorities come from the set $[d]$, i.e. the highest priority is $d-1$. Then $M_G \subseteq \mathbb{N}^d$ is for even d defined as

$$M_G = [1] \times [n_1 + 1] \times [1] \times [n_3 + 1] \times \cdots \times [1] \times [n_{d-1} + 1]$$

and for odd d we have the same equation except $\cdots \times [1] \times [n_{d-2} + 1] \times [1]$ being at the end. In other words M_G is the finite set of d-tuples of integers with zeros on even positions, and non-negative integers bounded by $|V^p|$ at every odd position p. We define $M_G^\top$ to be the set $M_G \cup \{\top\}$, where $\top$ is an extra element. We use the standard comparison symbols $\leq, =, \geq$ to denote the order on $M_G^\top$ which extends the standard lexicographic order on $M_G^\top$ by taking $\top$ as the maximal element, i.e. $m < \top$ for all $m \in M_G$. When subscripted by $i \in \mathbb{N}$ (e.g. $\geq_i$, $>_i$) they denote the extended lexicographic order restricted to the first i components.

For a function $\rho : V \to M_G^\top$ and an edge $(v, w) \in E$ by $Prog(\rho, v, w)$ we denote the least $m \in M_G^\top$ such that $m \geq_{\lambda(v)} \rho(w)$, and if $\lambda(v)$ is odd, then either the inequality is strict, or $m = \rho(v) = \top$.

Definition 3.1. A function $\rho : V \to M_G^\top$ is a *game parity progress measure* if for all $v \in V$ it satisfies:

- if $v \in V_0$ then $\rho(v) \geq_{\lambda(v)} Prog(\rho, v, w)$ for some $(v, w) \in E$, and
- if $v \in V_1$ then $\rho(v) \geq_{\lambda(v)} Prog(\rho, v, w)$ for all $(v, w) \in E$.

By $\|\rho\|$ we denote the set $\|\rho\| = \{v \in V \mid \rho(v) \neq \top\}$.

For every parity game progress measure ρ we define the associated strategy $\widetilde{\rho} \in \Sigma_0$ by putting $\widetilde{\rho}(v)$ to be the successor w of v which minimises $\rho(w)$.

Theorem 3.2. *If ρ is a game parity progress measure then $\widetilde{\rho}$ is a winning strategy for P_0 from vertices in $\|\rho\|$. In addition there is a game parity progress measure such that $\|\rho\| = W_0$.*

Before we can present the algorithm, we need to define an ordering on measures. For $\mu, \rho : V \to M_G^\top$ we put $\mu \sqsubseteq \rho$ if $\mu(v) \leq \rho(v)$ for all $v \in V$. We write $\mu \sqsubset \rho$ iff $\mu \sqsubseteq \rho$ and $\mu \neq \rho$. The relation $\sqsubseteq$ gives a complete lattice structure on the set of functions $V \to M_G^\top$. Finally we define operator $\mathit{Lift}(\rho, v)$ for $v \in V$ as

$$\mathit{Lift}(\rho, v)(u) = \begin{cases} \rho(u) & \text{if } u \neq v \\ \max\{\rho(v), \min_{(v,w)\in E} \mathit{Prog}(\rho, v, w)\} & \text{if } u = v \in V_0 \\ \max\{\rho(v), \max_{(v,w)\in E} \mathit{Prog}(\rho, v, w)\} & \text{if } u = v \in V_1 \end{cases}$$

The following two lemmas are easy to prove.

Lemma 3.1. *For every $v \in V$ the operator $\mathit{Lift}(\cdot, v)$ is $\sqsubseteq$-monotone.*

Lemma 3.2. *A function $\rho : V \to M_G^\top$ is a game parity progress measure iff it is a simultaneous pre-fixed point of all $\mathit{Lift}(\cdot, v)$ operators, i.e. if $\mathit{Lift}(\rho, v) \sqsubseteq \rho$ for all $v \in V$.*

From Knaster-Tarski theorem it follows that the $\sqsubseteq$-least game parity progress measure must exist and can be computed by the procedure `ProgressMeasureLifting`.

Procedure `ProgressMeasureLifting`

μ:=$\lambda v \in V.(0, \ldots, 0)$
while $\mu \sqsubset \mathit{Lift}(\mu, v)$ *for some* $v \in V$ **do**
  μ:=$\mathit{Lift}(\mu, v)$

Theorem 3.3. *For a parity game $\mathcal{G}$ the procedure `ProgressMeasureLifting` computes the winning sets W_0 and W_1 and a winning strategy $\sigma \in \Sigma_0$ in space $O(d \cdot n)$ and time*

$$O\left(dm \cdot \left(\frac{n}{\lfloor d/2 \rfloor}\right)^{\lfloor d/2 \rfloor}\right)$$

3.3 Randomised Algorithms

Although there is currently no known polynomial-time algorithm for solving parity games, there are several algorithms with a known sub-exponential upper complexity bound. Historically the first such result is due to Ludwig [Lud95], who gave a randomised algorithm for simple stochastic games based on linear programming, with time complexity $2^{O(\sqrt{n})}$. (But there is a catch, as we will see later.) With only a minor modification this algorithm can be applied to parity

games, giving the same complexity bound. Petersson and Vorobyov [PV01] gave a similar algorithm based on graph optimisations.

The Ludwig-style algorithm works on *binary* parity games. For these games each strategy $\sigma \in \Sigma_0$ can be associated with a corner of n_0-dimensional hypercube (where $n_0 = |V_0|$). If there is an appropriate way of assigning values to strategies, the algorithm can be described by the following steps:

1. Start with some strategy σ_0 of player P_0.
2. Randomly choose a facet F of the hypercube containing σ_0.
3. Recursively find the best strategy σ' on F.
4. Let σ'' be the neighbour of σ' on the opposite facet $\overline{F}$. If σ' is better than or equal to σ'', then return σ'. Otherwise recursively find the optimum on $\overline{F}$, starting from σ''.

For binary parity games the upper bound on complexity is $2^{O(\sqrt{n})}$. However if the parity game to be solved is not binary, we need to translate it into one that is. In the worst case this may result in a quadratic blowup in the number of states (cf. Lemma 2.1) and the algorithm becomes exponential in n. Therefore both the algorithms are sub-exponential only for games where vertex out-degree is bounded by a constant. (This is to be expected. For example it is comparatively easy to come up with an algorithm for solving parity games in polynomial time on graphs of bounded DAG-width if the vertex out-degree is bounded.)

The first truly sub-exponential algorithm is due to Björklund, Sandberg and Vorobyov [BSV03]. Instead of applying the randomisation scheme of Ludwig, they rely on a different randomised scheme of Kalai [Kal92], used for linear programming. This scheme can be applied to games of arbitrary out-degree, without the need for the quadratic translation to binary parity games. The complexity is then bounded by $2^{O(\sqrt{n \log n})}$.

The algorithm can be described by the sequence of steps presented below. Since we allow vertices to have an arbitrary out-degree, we must redefine the notion of a facet. For a game $\mathcal{G}$, a vertex $v \in V_0$ and an edge $(v,w) \in E$, a *facet* F is the subgame of $\mathcal{G}$ created by fixing the edge (v,w) and removing all other edges leaving v. This corresponds to fixing the strategy $\sigma(v) = w$.

1. Collect a set M containing r pairs (F, σ) of σ_0-improving facets F of $\mathcal{G}$ and corresponding witness strategies σ (r is a parameter controlling the complexity).

2. Select one pair $(F, \sigma_1) \in M$ uniformly at random. Find an optimal strategy σ in F by applying the algorithm recursively, taking σ_1 as the initial strategy.

3. If σ is an optimal strategy also in $\mathcal{G}$, return σ. Otherwise let σ' be a strategy differing from σ by an attractive switch. Restart from step 1 using the new strategy σ'.

Termination is guaranteed by the fact there is an optimal strategy.

3.4 Deterministic Sub-exponential Algorithm

The sub-exponential algorithms we have seen in the previous section are all ultimately based upon the randomised sub-exponential simplex algorithms of Kalai [Kal92] and Matoušek et al. [MSW96] These are very deep results and randomness seems to play an essential role in these results. However very recently Jurdziński, Paterson and Zwick [JPZ06] came up with a deterministic algorithm which achieves roughly the same time complexity as the randomised algorithm of Björklund et al. [BSV03] – the complexity of this new algorithm is $n^{O(\sqrt{n})}$ if the vertex out-degree is not bounded.

As surprising as it may seem, this algorithm is a modification of the simple algorithm of McNaughton [McN93] and Zielonka [Zie98] we have seen in Section 3.1. The idea behind the modification is subtle: By doing some extra computation before starting the recursive descent, and also by a careful complexity analysis, we get a better complexity bound. The key notion is defined below:

Definition 3.2. A set $W \subset V(G)$ is said to be *i-dominion* if player P_i can win from any vertex of W without leaving the set W. By *dominion* we mean either 0-dominion or 1-dominion.

Clearly W_0 is a 0-dominion and W_1 is 1-dominion. The following lemma gives us an important property of dominions:

Lemma 3.3. *Let $\mathcal{G} = (V, E, \lambda)$ be a parity game, $n = |V|$, and let $l \leq n/3$. A dominion of G of size at most l, if one exists, can be found in time $O((2en/l)^l)$.*

Proof. If $l \leq n/3$ then for all $i \leq l$ we have that $\binom{n}{i} / \binom{n}{i-1} \geq 2$. Therefore the number $\sum_{i=1}^{l} \binom{n}{i}$ of subsets W of V of size at most l is $O(\binom{n}{l})$. For each such subset W if $G[W]$ is not a subgame, then obviously W is not a dominion. Otherwise we can apply the algorithm `PGSolve` to $G[W]$ and in time $O(2^l)$ find out whether $W_0(G[W]) = W$ or $W_1(G[W]) = W$, in which case W must be a dominion. The total running time is $O(2^l \binom{n}{l}) = O((2en/l)^l)$ as required. □

To get the sub-exponential algorithm we modify the procedure `PGSolve` in the following way. At the beginning the modified procedure `PGSolve2` starts by checking whether there is a dominion of size at most $l = \lceil \sqrt{n} \rceil$. The parameter l is chosen in this way to minimise the running time. If such a dominion is found, then it is easy to remove the dominion and its force set from the game (using Theorem 2.4) and recurse on the remaining subgame. If no such dominion is found, the procedure `PGSolve2` behaves exactly like `PGSolve` (except for calling `PGSolve2` instead of `PGSolve` on recursive descent).

Theorem 3.4. *Let $\mathcal{G} = (V, E, \lambda)$ be a parity game. Then `PGSolve2`($\mathcal{G}$)=($W_0(G)$, $W_1(G)$). Moreover the running time of `PGSolve2` is $n^{O(\sqrt{n})}$, where $n = |V(G)|$.*

Proof. The correctness follows from the correctness of the algorithm `PGSolve`, which was proved in Section 3.1, and the definition of dominions. By careful analysis of the algorithm we can see that the running time for a graph of n vertices is given by the following equation:

$$T(n) \leq n^{O(\sqrt{n})} + T(n-1) + T(n-l)$$

From this reccurence relation it can be derived that $T(n) = n^{O(\sqrt{n})}$ (the proof is a bit technical, but not hard). □

3.5 Games on Undirected Graphs

In this section we consider the problem of solving parity games on undirected graphs, for which we allow each edge of the graph to be traversed in both directions. This is equivalent to solving parity games for which the following is true:

$$\forall v, w \in V.(v, w) \in E \text{ iff } (w, v) \in E \tag{3.1}$$

The results in this section were first observed by Olivier Serre [Ser03].

To be able to give a clear presentation we will make the following two assumptions: 1) $\mathcal{G}$ is a parity game with a maximum number of priorities, and 2) the game graph is 0-1 bipartite.

Theorem 3.5. *Let $\mathcal{G} = (V, E, \lambda)$ be a 0-1 bipartite parity game with a maximum number of priorities satisfying (3.1) above. Then we can solve $\mathcal{G}$ in time $O(|E|)$.*

Proof. We create a graph $G' = (V, E')$ from G by taking the following prescription for E':

$$\begin{aligned} E' &= \{(v,w) \mid \{v,w\} \in E \text{ and } v \in V_0, w \in V_1, \max\{\lambda(v), \lambda(w)\} \text{ is even } \} \cup \\ &\cup \{(v,w) \mid \{v,w\} \in E \text{ and } v \in V_1, w \in V_0, \max\{\lambda(v), \lambda(w)\} \text{ is odd } \} \end{aligned}$$

The graph G' must be acyclic: the definition above and the fact that $\mathcal{G}$ is a game with a maximal number of priorities guarantee that for any edge $(v,w) \in E'. \lambda(v) > \lambda(w)$.

We can actually view $\mathcal{G}'$ as a game between players P_0 and P_1 with a reachability condition. Let

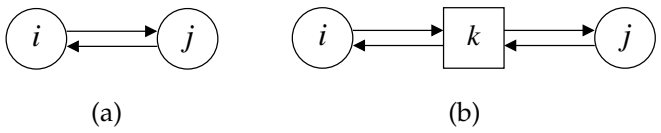
$$F = \{v \in V_1 \mid v \text{ has no successors in } G'\}$$

Then P_0 wins a play starting in v_0 iff the play reaches F. The following statement relates $\mathcal{G}$ and $\mathcal{G}'$. Let $\mathcal{G}$ and $\mathcal{G}'$ be the games as above and $v \in V$. Then $v \in W_0(\mathcal{G})$ iff $v \in F_0(F)$, where the force set $F_0(F)$ is taken in the game $\mathcal{G}'$. We will prove only one direction, the other follows from duality of parity games.

The proof goes by induction on the structure of G. The only interesting case is actually the base case. Let $v \in V_1$ be a vertex with no successors, and let $U = \{u \in V \mid (u,v) \in E'\}$. Then surely $U \subseteq V_0$ because G is bipartite. Moreover from definition of E' we have $\forall u \in U. \max\{\lambda(u), \lambda(v)\}$ is even. By setting $\sigma(u) = v$ for all $u \in U$ we get a partial winning strategy of P_0 and therefore $U \cup \{v\} \subseteq W_0$. The inductive step is trivial. □

In the previous we assumed that the graph G is bipartite. Note that in the case G is not bipartite, we cannot just simply apply the Lemma 2.4 to convert it into a one which is. The catch here is that the construction used there to make the graph bipartite splits each non-conforming edge with a new vertex, which means we would not get a graph where the edge relation is symmetric.

What we need to do is to insert just a single extra vertex for every pair of offending edges of G with the same ends. If we have two vertices of the same player with priorities $i < j$ joined by an edge, we insert a new vertex of priority k s.t. $i < k < j$ and k has the same parity as j. The conversion is shown in Fig 3.1. Having done the transformation for all edges between two vertices of the same player, the new game is obviously equivalent to the original one.



(a) (b)

Figure 3.1: Conversion to bipartite game: (a) before, and (b) after conversion

3.6 Trees with Back Edges

A tree with back edges is a structure which appears quite often at different places in computer science. For example it is the structure obtained when running Depth First Search algorithms. Second, a graph of a μ-calculus formula is a tree with back edges. Finally, trees with back edges separate different notions of directed graph decompositions. The fact that parity games can be solved in polynomial time on trees with back edges has been observed by Niwiński [Niw], but to our knowledge this is the first time the proofs were actually written down.

For a directed graph $G = (V, E)$, a subset of edges $D \subseteq E$ and two vertices $v, w \in V$ we write $v <_D w$ iff there is a directed path from v to w. We also write $v \leq_D w$ iff $v <_D w$ or $v = w$.

Definition 3.3 (Tree with back edges)**.** A directed graph $T = (V, E)$ is a *tree with back edges* if there is a partition of $E = F \uplus B$ into the sets of *tree (forward) edges* F and *back edges* B such that (V, F) is a directed tree and $(u, v) \in B$ implies $v <_F u$.

In the text to follow we will always consider trees with back edges not to contain simple loops (of size one). We also define the *complete tree with back edges* to be a tree with back edges s.t. for each two vertices $v, w \in V$ we have that $v <_F w$ implies $(w, v) \in B$. For technical convenience for vertex $v \in V$ we define the set $B(v) = \{w \in V \mid (v, w) \in B\}$ – i.e. the set of all predecessors directly

reachable form v. The following lemma shows us an important property of trees with back edges, and is very easy to prove by induction on the depth of the tree T.

Lemma 3.4. *Let $T = (V, E)$ be a tree with back edges, and $E = F \uplus B$. Then every simple cycle in T is of the form $v \to_F^+ w \to_B v$.*

It turns out that for parity games whose graphs are trees with back edges we indeed have a good decomposition of the game graph into subgames. This immediately gives us an algorithm linear in the size of the graph.

Theorem 3.6. *Let $\mathcal{G} = (V, E, \lambda)$ be a parity game whose game graph G is a tree with back edges with a root v_0. Then there is an algorithm which can solve the parity game $\mathcal{G}(v_0)$ in time $O(m)$, where $m = |E|$.*

Proof. Let us first define the graphs G_v and G_{vw} for $v, w \in V(G)$. G_v is the subgraph of G obtained by removing all backward edges for all vertices on the (unique) path $v_0 \to_F^* v$ (excluding v). $G_{v,w}$ for an edge $(v, w) \in E(G)$ is equal to G_v where all edges with the tail v are removed with the exception of the edge (v, w). Both of these subgraphs corresponds to fixing partial strategies $\sigma \in \Sigma_0$ and $\tau \in \Sigma_1$ for the vertices on the path $v_0 \to_F^* v$. Note that for $(v, w) \in F(G)$ we have $G_{vw} = G_w$.

Parity games whose graphs are trees with back edges have one important property. If we fix a strategy for v_0 (i.e. choose an edge $(v_0, w) \in F(G)$), the game $\mathcal{G}_{v_0 w} = \mathcal{G}_w$ is a subgame of $\mathcal{G}$ such that there are no edges from the part of the graph G_w reachable from v_0 to the rest of the graph. Moreover the following is true:

Claim: Let $v \in V(G)$ and let $w_1, \ldots, w_j$ be all its successors. If $v \in V_0$, then P_0 wins the game $\mathcal{G}_v(v)$ iff he wins at least one of the games $\mathcal{G}_{vw_i}(w_i)$ for $1 \leq i \leq j$. Similarly if $v \in V_1$, then P_0 wins the game $\mathcal{G}_v(v)$ iff he wins all the games $\mathcal{G}_{vw_i}(w_i)$ for $1 \leq i \leq j$.

With this in mind we can give an algorithm for solving $\mathcal{G}(v)$. We start with $v = v_0$ and recursively do the following: Taking $v \in V$ with successors $w_1, \ldots, w_j$ we recursively solve the games $\mathcal{G}_{vw_i}(w_i)$. The solution to $\mathcal{G}_v(v)$ is then given by the claim above. Now if $(v, w_i) \in B(G)$, then to find the winner for $\mathcal{G}_{vw_i}(w_i)$ equals to checking the highest priority on the path from $w_i \to_F^+ v$. (Note that in G_{vw_i} each vertex on the simple cycle $w_i \to_F^+ v \to_B w_i$ has only one successor). As

leaves have only backwards edges, and we recursively follow the tree edges of G, this guarantees us that the algorithm would finish after at most m steps, where $m = |E(G)|$. Moreover the check for back-edges (v, w_i) can be done in constant time. Therefore we find the winner for $\mathcal{G}(v_0) = \mathcal{G}_{v_0}(v_0)$ in time $O(m)$.

□

Chapter 4

Bounded Tree-Width

This chapter is based on the paper [Obd03].

Tree-width is a graph theoretic concept introduced first by Robertson and Seymour [RS84] in their work on graph minors. Roughly speaking, tree-width measures how close is the given graph to being a tree. Graphs of low tree-width then allow decomposition of the problem being solved into subproblems, decreasing the overall complexity – many NP-complete problems were shown to be solvable in linear (or polynomial) time on the graphs of bounded tree-width. (Following the intuition that solving problems on trees is *much* easier than on general graphs. E.g. modal μ-calculus model checking can be done in linear time on trees.) See Bodlaender's paper [Bod97] for an excellent survey.

Even though the concept of tree-width is quite restrictive, in practice the systems considered are (may be surprisingly) often of a low tree-width. In [Tho98] it was shown that all C programs (resp. their control-flow graphs) are of tree-width at most 6, and Pascal programs of tree-width at most 3! This result does not hold for Java, as the labelled versions of `break` and `continue` can be as harmful as `goto` [GMT02]. In practice, however, programs with control-flow graphs of high tree-width do not appear (since they are written by sane humans).

The fact that parity games can be solved in polynomial time on graphs of bounded tree-width seems to be a consequence of a general theorem of Courcelle [Cou90]: For a fixed MSO formula φ and a graph of (bounded) tree-width k the model checking problem can be solved in time $O(n \cdot \alpha(k, \varphi))$, where $\alpha(k, \varphi)$ is a term which depends on k and the formula φ, but not on n. This means the

time is linear in the size of the graph if both k and φ are fixed. To get an MSO formula characterising the winning region, we could use the reduction of parity games to the modal μ-calculus model checking problem (see Section 2.5.2 for more details). The resulting modal μ-calculus formula is then easily translated into an equivalent MSO formula by using a well known algorithm – see e.g. [GTW02]. Note that using this translation the size of the formula depends on the number of priorities, i.e. on n in the worst case.

The theorem from [Cou90], however, does not provide any estimate on the size of the constant $\alpha(k,\varphi)$ (which heavily depends on the formula φ) hidden in the O notation (except for being 'large'). Fairly recently it was shown [FG02] that the function α is not even elementary. However for parity games we usually consider the number of priorities to be part of the input, and not fixed in advance. Moreover the algorithm presented in [Cou90] itself is quite complicated and does not provide any insight into what are the results/strategies in the underlying game.

In contrast, our algorithm does not require translating the winning condition to a MSO formula, and in addition one can easily follow the workings of the algorithm as well as the evolving strategies (we will actually get the winning strategy for free). We show that parity games on graphs of tree-width k can be solved in time $O(n \cdot (k+1)^2 \cdot n^{\alpha(k)})$, where $\alpha(k)$ is a polynomial in k and does not depend on the size/shape of the parity game considered (more specifically not on the number of priorities). As argued above, this result is new and does not follow from [Cou90]. We then extend this to give a new μ-calculus model checking algorithm.

4.1 Tree Decompositions

Here we present the relevant facts about tree decompositions and tree-width, which will be needed later in the text.

Definition 4.1 (Tree decomposition)**.** A *tree decomposition* of an (undirected) graph G is a pair $(T, \mathcal{X})$, where T is a tree (its vertices are called *nodes* throughout this chapter) and $\mathcal{X} = \{X_t \mid t \in T\}$ is a family of subsets of $V(G)$ satisfying the following three conditions:

(T1) $V(G) = \bigcup_{t \in V(T)} X_t$,

(T2) for every edge $\{v,w\} \in E(G)$ there exists $t \in V(T)$ s.t. $\{v,w\} \subseteq X_t$, and

(T3) for all $t,t',t'' \in V(T)$ if t' is on the (unique) path from t to t'' in T, then $X_t \cap X_{t''} \subseteq X_{t'}$.

The *width* of a tree decomposition $(T,\mathcal{X})$ is $\max_{t\in V(T)} |X_t| - 1$. The *tree-width* of a graph G (written as $tw(G)$) is the minimum width over all possible tree decompositions of G. Trees have tree-width one. One obtains an equivalent definition if the third condition is replaced by:

(T3′) For all $v \in V$, the set of nodes $\{t \in V(T) \mid v \in X_t\}$ is a connected subtree of T.

There is an easy way to generalise the concept of tree-width to directed graphs: For a directed graph G we put $tw(G) = tw(G')$ where G' is obtained from G by forgetting the orientation of the edges (i.e. an edge $\{u,v\}$ of G' can correspond to two edges (u,v) and (v,u) of G). We can therefore freely talk about tree-width and tree decompositions of directed graphs. (Later we will see that this generalisation is in some sense not 'optimal'.)

To better understand the definition look at the example in Fig. 4.1. There is a graph with six vertices a to f, together with its tree decomposition. There is a dashed line showing which triples of vertices correspond to each node of the tree decomposition. As there are at most three vertices in each node, the width of the shown decomposition is two.

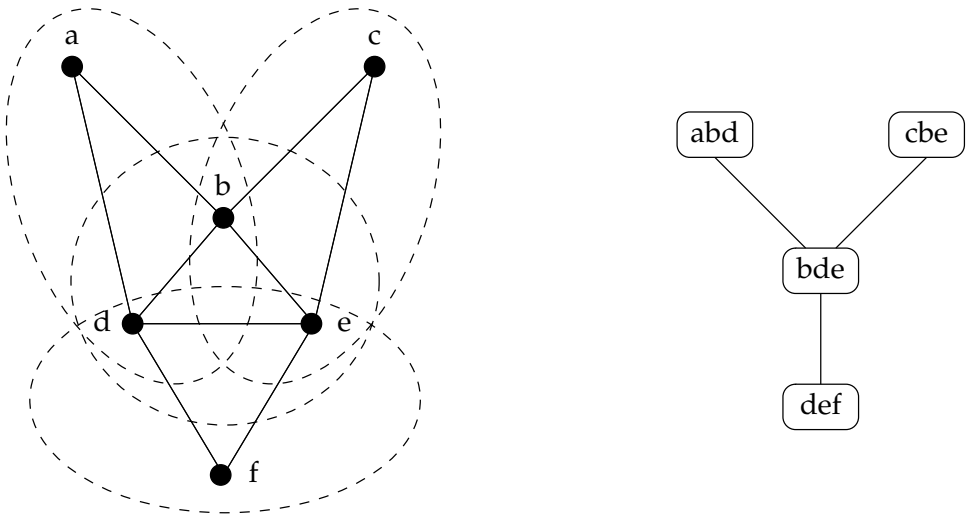


Figure 4.1: Graph and its tree decomposition

One would expect that since tree-width should measure how close a given graph is to a tree, cliques should have high tree-width and this is indeed the case:

Fact 4.1. *Let K_n be a clique of n vertices. Then K_n has tree-width $n-1$.*

Another class of structures of high tree-width are grids.

Fact 4.2. *For $n > 1$ (a graph which is) a grid of size $n \times n$ has tree-width n.*

Both facts are easy to prove using the Tree-width Duality Theorem, which we will see later (Theorem 4.1, p. 44).

Before proceeding further we will need some extra notation. For technical reasons we will assume that for each tree decomposition $(T, \mathcal{X})$ of G the tree T is a rooted tree (i.e. one vertex is designated to be the root) with all edges oriented away from the root. With this assumption in mind we define T_t for $t \in V(T)$ to be the largest subtree of T rooted in the node t. (More precisely we start in t and include all the nodes of T reachable from t by an oriented path.) Having T_t defined we use the following notation:

$V_t = \bigcup_{s \in V(T_t)} X_s$, the vertices of G appearing in T_t

$V_{>t} = V_t \setminus X_t$

The next fact about tree decompositions is one of the basic properties of graphs of bounded tree-width, which allows for all the interesting results.

Fact 4.3. *Let $(T, \mathcal{X})$ be a tree decomposition and t a node of T. Then the only vertices of V_t adjacent to vertices $V(G) \setminus V_t$ are those belonging to X_t. In other words, X_t is an* interface *between $G[V_{>t}]$ and the rest of the graph (i.e. the graph $G \setminus V_t$).*

Presenting dynamic algorithms working on general tree decompositions is often a tedious exercise. However we can at least avoid most of the purely technical problems. The following notion of *nice tree decomposition* allows us to significantly simplify the construction of our algorithm. This choice is justified by Lemma 4.1.

Definition 4.2 (Nice tree decomposition)**.** Tree decomposition $(T, \mathcal{X})$ is called *nice tree decomposition*, if the following three conditions are satisfied:

1. every node of T has at most two children,

2. if a node t has two children t_1 and t_2, then $X_{t_1} = X_{t_2} = X_t$, and

3. if a node t has one child t_1, then either $|X_t| = |X_{t_1}| + 1$ and $X_{t_1} \subseteq X_t$, or $|X_t| = |X_{t_1}| - 1$ and $X_t \subseteq X_{t_1}$.

Lemma 4.1 (See [Klo94]). *Every graph G of tree-width k has a nice tree decomposition of width k. Furthermore, if n is the number of vertices of G then there exists a nice tree decomposition with at most $4n$ nodes. Moreover given a decomposition of width k with $O(n)$ nodes, a nice tree decomposition of this size and the same width can be constructed in time $O(n)$.*

The proof of this lemma is constructive - i.e. it gives an algorithm for transforming every tree decomposition into a nice tree decomposition. The restriction to $O(n)$ nodes in the Lemma above is there just to eliminate tree decompositions where some nodes/subtrees are unnecessarily repeated along paths of the tree. As can be seen from the definition, in a nice tree decomposition $(T, \mathcal{X})$ every node is one of four possible types. These types are:

Start If a node t is a leaf, it is called a *start node*.

Join If a node t has two children t_1 and t_2, it is called a *join node* (note that by (T3) the subgraphs of its children are then disjoint except for X_t).

Forget If a node t has one child t' and $|X_t| < |X_{t'}|$, node t is called a *forget node*.

Introduce If a node t has one child t' and $|X_t| > |X_{t'}|$, node t is called an *introduce node*.

Moreover, we may assume that *start* nodes contain only a single vertex. If this is not the case, we can transform each nice tree decomposition into one having this property by adding a chain of *introduce* nodes in place of each non-conforming *start* node.

We will also need a notion of terminal graph, which is closely related to tree decompositions.

Definition 4.3 (Terminal graph). A *terminal graph* is a triple $H = (V, E, X)$, where (V, E) is a graph and X an ordered subset of vertices of V called *terminals*. The operation $\oplus$ is defined on pairs of terminal graphs with the same number of terminals: $H \oplus H'$ is obtained by taking the disjoint union of H and H' and then

identifying the i-th terminal of H with i-th terminal of H' for $i \in \{1,\ldots,l\}$. A terminal graph H is a *terminal subgraph* of a graph G iff there exists a terminal graph H' s.t. $G = H \oplus H'$. Finally we define H_t to be $G[V_t]$ taken as a terminal subgraph with X_t as a set of its terminals (the ordering of X_t is not important here).

4.2 Cops and Robber Games

Tree-width is closely related to a certain cops-and-robber game on graphs. This game not only provides us with a valuable insight into the inner working of tree decompositions, but also provides us with an alternative characterisation of the class of graphs of bounded tree-width.

The original game first appeared in [ST93]. The robber stands on a vertex of the graph, and can at any time run at great speed to any other vertex along a path of the graph. He is not permitted to run through a cop, however. There are k cops, each of whom at any time either stands on a vertex or is in a helicopter. The goal of the player controlling the cops is to land a cop via a helicopter onto a vertex currently occupied by the robber, and the robber's objective is to elude capture. (The point of the helicopter is that cops are not constrained to move along the paths of the graph.) The robber can see the helicopter landing and may run to a new vertex before it actually lands.

More formally, the game is played on a graph G by two players: the cop player, and the robber player. It is played according to the following rules: At the beginning the robber player chooses a vertex $u \in V(G)$, giving us an initial game position $(\emptyset, u)$. Given a position (X, v), the cop player chooses a set $X' \subseteq [V]^{\leq k}$, and the robber player a vertex $v' \in V(G) \setminus X'$ such that both v and v' lie in the same connected component of the graph $G \setminus (X \cap X')$, giving us the next position (X', v'). A play is a maximal sequence of positions formed from an initial game position according to the rule above. The play is winning for the cop player if it is finite – i.e. for the final position (X, v) of the play it is true that there is $X' \in [V]^{\leq k}$ such that no vertex of the graph $V(G) \setminus X'$ is in the same connected component of the graph $G \setminus \{X \cap X'\}$ as v (this immediately implies $v \in X'$). On the other hand the robber player wins if the play is infinite. If k cops can capture the robber in G (i.e. the cop player wins) we say that k cops can *search* G. Moreover if they can do so without revisiting a vertex then they

can *monotonely search G*.

It turns out that in this game there are particularly nice strategies for the robber, which correspond to the notions of bramble and haven. Let w be an integer. A *haven*[1] of order w in a graph G is a function which assigns to every set $Z \subseteq V(G)$ with $|Z| < w$ the vertex set of a connected component of G in such a way that if $Z' \subseteq Z \subseteq V(G)$ then $\beta(Z) \subseteq \beta(Z')$.

The notion of bramble is very closely related to havens. We say that two subsets of $V(G)$ *touch* if they have a common vertex, or there is an edge with one end in each of the two sets. Moreover we say that set of vertices $S \subseteq V$ *covers* $\mathcal{Y} \subseteq 2^V$ iff for each $Y \in \mathcal{Y}$ we have $S \cap Y \neq \emptyset$. A *bramble* in G is set of mutually touching subsets of $V(G)$. The least number of vertices covering a bramble is its *order*.

The following theorem comes from [ST93]:

Theorem 4.1 ([ST93])**.** *Let G be a (undirected) graph. Then the following are equivalent:*

(i) *G has a haven of order* $\geq k$

(ii) *G has a bramble of order* $\geq k$

(iii) $< k$ *cops cannot search G*

(iv) $< k$ *cops cannot monotonely search G*

(v) *G has tree-width* $\geq k-1$

The equivalence (i) $\Longleftrightarrow$ (v) is often called *Tree-width Duality Theorem*. The hardest part of the proof is the implication (v) $\Longrightarrow$ (i). It is proved by contraposition, using amalgamation of tree decompositions of subgraphs of G. A streamlined proof of the duality theorem has later appeared in [BD02].

4.3 Obtaining Tree Decompositions

To be able to use an algorithm which exploits small tree-width of input graphs we need to find a tree decomposition with tree-width bounded by a constant

[1]The notion of haven as originally defined in [ST93] is slightly less restrictive than the definition presented here, which was taken from [JRST01]. The original definition is almost the same as that of a bramble.

(not necessarily optimal). If the complexity of this step is too high, the fact that we have an algorithm which runs very fast on tree decompositions of bounded tree-width does not account for much.

We start with a bad news. In general, the problem 'Given a graph G and an integer k, is the tree-width of G at most k?' was shown to be NP-complete [ACP87]. Therefore much effort has been directed into solving this problem for the case that k is a constant. The final result is summed up by the following theorem:

Theorem 4.2 ([Bod93])**.** *For all $k \in \mathbb{N}$ there exists a linear time algorithm that tests whether a given graph G of n vertices has tree-width at most k, and if so, outputs a tree-decomposition of G of tree-width at most k.*

In practice we are quite often in a much better position. The graphs considered are usually not just random graphs we know nothing about, but have underlying structure which we may successfully exploit. For example in software verification the graphs are control-flow graphs of programs written in high-level programming languages, which are structured. The result of Thorup [Tho98] shows that tree-width (of control-flow graphs) of programs written in C is at most 6 and in PASCAL at most 3. Moreover we can easily get the desired tree decompositions by a simple syntactic analysis of the programs in question. However in the case of programs written in JAVA the tree-width can be unbounded. This is due to the presence of labelled `break` and `continue` statements in the language, as well as the exception handling mechanism [GMT02]. Nevertheless in practice the usual tree-width of JAVA programs is two or three, and programs of tree-width greater than five are virtually unheard of.

4.4 The Algorithm for Parity Games

We are now going to present a polynomial-time algorithm for solving parity games on graphs of bounded tree-width. For the rest of this section let us fix a parity game $\mathcal{G} = (V, E, \lambda)$ s.t. there is no loop. (Loop is cycle of length one. If there is such a cycle we can convert it into a cycle of length two, adding an extra vertex in order to do so.) Let G be the game graph of $\mathcal{G}$ and $(T, \mathcal{X})$ be a nice tree decomposition of G of width k (we can restrict ourselves to nice tree decompositions by Lemma 4.1).

Our algorithm follows the general approach for solving problems on graphs of bounded tree-width (see [Bod97]). The crux of the algorithm lies in computing a bounded representation of the exponential set of strategies. Given a node t of T, we only need to know the effect of any given strategy on vertices in the interface X_t, the size of which is bounded by a (small) constant. We compute the effects of strategies (called *borders*) at nodes of T in bottom-up manner. From the set of all possible borders for the root we can then quickly decide the winner for vertices in the root node. Using force-sets or some similar technique, winners for the other vertices can be found as well (the complexity then increases by at most a factor of n, where n is the number of vertices of G).

4.4.1 Borders

For a node t we need to somehow describe the effect of all plays confined to $G[V_t]$ on the rest of the graph ($G \setminus V_t$). Let us first consider the case where the strategies σ of P_0 and τ of P_1 are fixed. Let $v = \pi_0 \in X_t$, and $\pi = \pi_0\pi_1 \ldots \pi_i \ldots$ be a play of the game $\mathcal{G}$ respecting the strategies σ and τ. Let $\pi[t]$ be the maximal prefix of π when restricted to vertices of $V_{>t}$ (with the exception of π_0). We define the outcome of such path $\pi[t]$ to be

$$R^{\tau}_{\sigma}(v,t) = \begin{cases} \bot & \text{if } \pi[t] \text{ is infinite and winning for } P_1 \\ \top & \text{if } \pi[t] \text{ is infinite and winning for } P_0 \\ (w,p) & \text{if } \pi[t] : \pi_0, \ldots, \pi_j;\ w = \pi_{j+1} \text{ and } p = \max\{\lambda(\pi_i) \mid 0 \le i \le j+1\} \end{cases}$$

Note that in the last case it must be that $w \in X_t$.

The next step is to fix only a strategy σ of P_0 and try to find the best results player P_1 can achieve against this strategy. Two cases are simple. If there is a winning cycle of P_1 reachable from v in the subgame $\mathcal{G}_\sigma[V_{>t}]$, then we know P_1, starting in v, can win against the strategy σ of P_0 both in the game restricted to $V_{>t}$ and in the whole game $\mathcal{G}_\sigma$. On the other hand if all paths starting in v lead to a winning cycle for P_0, then P_1 loses every play starting in v also in the game $\mathcal{G}_\sigma$.

The third possibility is that there is no winning cycle for P_1 in $\mathcal{G}_\sigma[V_{>t}]$, but P_1 can force the play to a vertex of X_t. Then the 'value' of such play π is the highest priority of a vertex on this path. However there can be more paths starting in v which lead to some vertex $w \in X_t$. In that case it is in player P_1's interest to

choose the one with the lowest score w.r.t. the reward ordering '$\sqsubseteq$'. Moreover, there may be paths starting in v but leading to different vertices of the set X_t. Then we remember the best achievable result for each of these 'leave' vertices.

To formalise the description above, we need to extend the '$\sqsubseteq$' ordering to pairs (v,p). For two such pairs (v,p),(w,q) we put $(v,p) \sqsubseteq (w,q)$ iff $v = w$ and $p \sqsubseteq q$. Specifically if $v \neq w$ then the two pairs are incomparable. We furthermore extend the ordering $\sqsubseteq$ by adding the maximal element $\top$, and the minimal element $\bot$. For a set $X \subseteq (V \times \mathbb{N}) \cup \{\top, \bot\}$ we denote $\min_{\sqsubseteq} X$ to be the set of $\sqsubseteq$-minimal elements of X. Note that $\min_{\sqsubseteq} X = \{\bot\}$ iff $\bot \in X$ and $\min_{\sqsubseteq} X = \{\top\}$ iff $X = \{\top\}$. Moreover for $Z = \min_{\sqsubseteq} X$ it is true that if $Z \neq \{\bot\}$ and $Z \neq \{\top\}$ then Z contains at most one pair (v,p) for each vertex v.

With all the machinery in place we now define

$$R_\sigma(v,t) = \min_{\sqsubseteq}\{R^\tau_\sigma(v,t) \mid \tau \in \Sigma_1\}$$

Now we get to the definition of a border. A *border* of a node t tells us what happens *inside* the subgraph $G[V_t]$ – i.e. we take vertices of X_t as entry points for $G[V_t]$, but not as its inner vertices. We start with some useful definitions.

Definition 4.4 (border)**.** *A border* α of t is a function α mapping X_t to either $\bot$, $\top$, or a subset of $2^{X_t \times \mathbb{N}}$ in which case $\alpha(t)$ contains at most one pair (v,p) for each $v \in X_t$. Border α of t corresponds to a strategy $\sigma \in \Sigma_0$ if

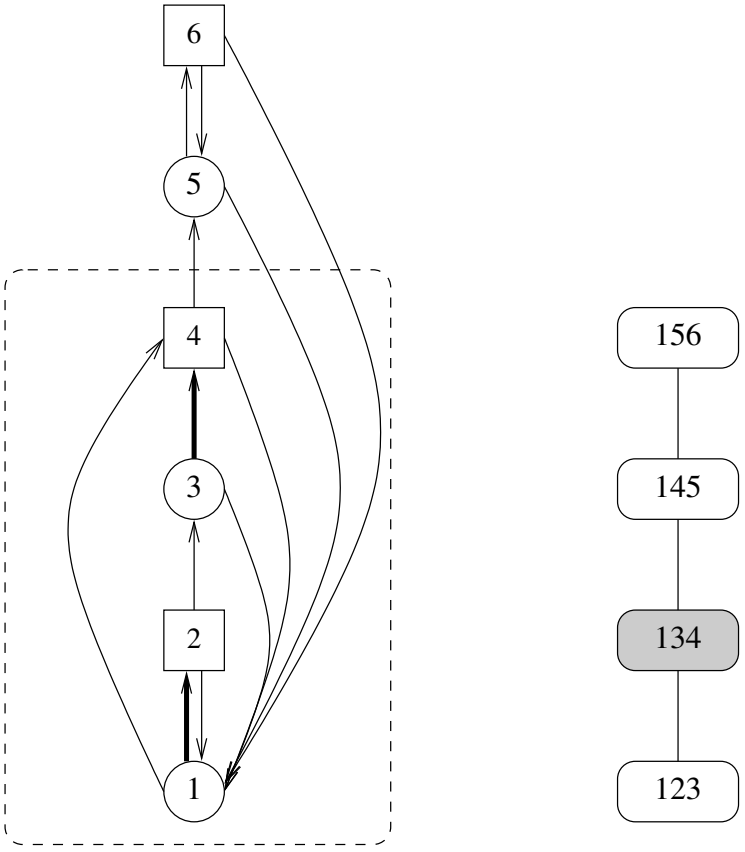
$$\forall v \in X_t . \alpha(v) = R_\sigma(v,t)$$

We will use letters from the start of the Greek alphabet to denote borders. For a node t and a strategy σ we also use α^t_σ to denote the border of t which corresponds to σ.

Note that there can be many strategies which correspond to the same border. This 'compression' is what makes the algorithm work. Elements of $\alpha(v)$ are called *entries*. For each pair $v,w \in X_t$ there is at most one entry $(w,p) \in \alpha(v)$ when $\alpha(v) \neq \bot$ or $\top$. This allows us to overload the notation a little bit and write $\alpha(v,w) = p$ as a shorthand for $(w,p) \in \alpha(v)$.

In addition to border α being a function, we can look at α as being a table of priorities with dimensions $(k+1) \times (k+1)$. In this table the rows and columns are labelled by vertices of X_t, and the value position at v,w is $\alpha(v,w)$. Likewise if symbols $\bot$ or $\top$ appear in a row, then the whole row must be marked by this symbol.

Example 4.1. *An example of a border is in Fig. 4.2. There you can see a parity game together with a tree decomposition of the game graph and border* α *for the node* t *with* $X_t = \{1,3,4\}$ *and a corresponding strategy* σ*. Just for completeness* $V_t = \{1,2,3,4\}$*.*



$\sigma = \{1 \mapsto 2, 3 \mapsto 4\}$

α	1	4	3
1	2	-	-
4	4	-	-
3	3	-	-

Figure 4.2: Example of a border

It is not hard to prove that α^t_σ contains all we need to know about the subgraph $G_\sigma[V_t]$ in order to check whether P_0 wins for some vertex in $V \setminus V_{>t}$ using the strategy σ. To do so we will need the notion of a *link*:

Definition 4.5 (link)**.** A *link* of a border α of t is a terminal graph $H = (X_t \cup W \cup \{v_\bot, v_\top\}, E \cup \{(v_\bot, v_\bot), (v_\top, v_\top)\}, X_t)$ and priority function λ s.t.:

- we start with $W = \emptyset$ and $E = \emptyset$

- for every pair $u, w \in X_t$ with $\alpha(u,w) = p$ for some $p \in \mathbb{N}$ we put a new vertex v into W and two edges (u,v) and (v,w) into E. We also set $\lambda(v) = \alpha(u,w)$.
- for every $v \in X_t$ with $\alpha(v) = \bot$ we insert an edge $(v, v_\bot)$ into E.
- for every $v \in X_t$ with $\alpha(v) = \top$ we insert an edge $(v, v_\top)$ into E.
- $\lambda(v)$ for $v \in X_t$ is the same as in original game, $\lambda(v_\bot) = 1$, and $\lambda(v_\top) = 2$.

We also write $Link(\alpha)$ for the link of a border α, using the same notation for both the game graph and the induced game (when we take λ into account).

In other words $Link(\alpha^t_\sigma)$ together with λ is just a graph having the same properties w.r.t. winning the game as $G_\sigma[V_t]$ does. The formal proof of this statement is subject of the following lemma:

Lemma 4.2. *Let α^t_σ be the border of t corresponding to σ. Let H be s.t. $G_\sigma = G_\sigma[V_t] \oplus H$ and v a vertex in $V \setminus V_{>t}$. Then P_1 has a winning strategy in $\mathcal{G}_\sigma(v)$ iff she has a winning strategy for v in $L = Link(\alpha^t_\sigma) \oplus H$.*

Proof. $\Rightarrow$ Suppose P_0 does not win in $\mathcal{G}_\sigma$. Then there must be an odd cycle $\rho : \rho_1 \dots \rho_k \rho_{k+1} = \rho_1$ reachable from v. Let π be some path from v to a vertex of ρ. There are two cases to be considered:

1. $V(\rho) \subseteq V_{>t}$
   Then there must be $i \in \mathbb{N}$ s.t. $\pi_i = w \in X_t$ by Fact 4.3 and $\alpha^t_\sigma(w) = \bot$ by definition of α^t_σ . From definition of $Link(\alpha^t_\sigma)$ player P_1 has a winning strategy for v in L (she can force play to $v_\bot$ and then loop through this vertex).
2. Otherwise
   Let $j \in \mathbb{N}$ be s.t. $\rho_j \in X_t$ and $\forall i \leq j . \rho_i \in V_{>t}$ (such j must exist). Then ρ_j is also reachable in L (by definition of $Link(\alpha^t_\sigma)$). Moreover, let ρ' be a cycle obtained from ρ by substituting every sequence $u = \rho_i \dots \rho_{i+l} = v$, where $u, v \in X_t$ and $\{\rho_{i+1} \dots \rho_{i+l}\} \subseteq V_{>t}$, by a path $uw_{u,v}v$. Then ρ' is a cycle of L and it is easy to check that P_1 also wins the cycle ρ' of L.

$\Leftarrow$ similar

□

Finally we define the set of all possible outcomes for a node t:

$$Border(t) = \{\alpha^t_\sigma \mid \sigma \in \Sigma_0\}$$

The following important corollary says how we can derive the desired information from $Border(r)$, where r is the root of T.

Corollary 4.1. *Let $(T, \mathcal{X})$ be a tree decomposition of G, r its root node and $v \in X_r$ a vertex of G. Then P_0 has a winning strategy for $\mathcal{G}(v)$ iff there is $\alpha \in Border(r)$ s.t. P_0 has a winning strategy for v in $Link(\alpha)$.*

It should be noted that the test whether P_0 wins in the game $Link(\alpha)(v)$ can be done in constant time, which depends only on the tree-width of G.

4.4.2 Computing $Border(t)$

Having a nice tree decomposition $(T, \mathcal{X})$, we compute the set $Border(t)$ for every node t of T in a bottom-up manner. Here we give an algorithm for each of the four node types.

Start Node Let t be a start node, $X_t = \{v\}$. We put $Border(t) = \{\alpha\}$, where $\alpha(v) = \emptyset$.

Forget Node Let t be a forget node with a single child t' and $X_{t'} = X_t \cup \{v\}$. By definition of tree decompositions we know that there is no edge connecting v with $V \setminus V_t$, since v does not appear anywhere in the part of T yet to be explored. We will modify each $\alpha \in Border(t')$ according to the value of $\alpha(v)$, creating α' (a new function which is defined only for $w \in X_t$). There are three cases to be considered.

$\alpha(v) = \bot$ **or** $\alpha(v, v) = p, p$ **odd**

$$\alpha'(w) = \begin{cases} \bot & \text{if } \alpha(w, v) = q \text{ for some } q \\ \alpha(w) & \text{otherwise} \end{cases}$$

$\alpha(v) = \top$ **or** $\alpha(v) = \{(v, p)\}, p$ **even**

$$\alpha'(w) = \begin{cases} \top & \text{if } \alpha(w) = \{(v, q)\} \text{ for some } q \\ \alpha(w) & \text{otherwise} \end{cases}$$

None of the previous

Let $\beta_p = \{(u, \max\{p,q\}) \mid (u,q) \in \alpha(v), u \neq v\}$, i.e. we take all elements of $\alpha(v)$ except for $\alpha(v,v)$, and replace the original priority with p if p is bigger. Now we put

$$\alpha'(w) = \begin{cases} \min_{\sqsubseteq}(\alpha(w) \cup \beta_p) & \text{if } (v,p) \in \alpha(w) \\ \alpha(w) & \text{otherwise} \end{cases}$$

Let $\alpha''(w)$ be $\alpha'(w)$ minus any pair (v,q) for some q. For all three cases we put $mod(\alpha)(w) = \alpha''(w)$ for $w \in X_t$ and claim that

$$Border(t) = \{mod(\alpha) \mid \alpha \in Border(t')\}$$

The correctness follows from the definition of border.

Introduce Node Let t be an introduce node with a child t', and $X_t = X_{t'} \cup \{v\}$. Let $\alpha \in Border(t)$. We now have to connect all edges between v and $X_{t'}$. We start with the edges going from X_t to v. Let us define the following operation

$$mod(\alpha, U, v)(u) = \begin{cases} \alpha(u) \cup \{(v, \max(\lambda(u), \lambda(v)))\} & \text{if } u \in U \\ \alpha(u) & \text{otherwise} \end{cases}$$

Let U_1 be the set of odd vertices in $X_{t'}$ with edges to v, i.e. $U_1 = \{u \in X_{t'} \cap V_1 \mid (u,v) \in E\}$, and $U_0(\alpha)$ the set of even vertices in $X_{t'}$ which have v as a successor and for which no choice has been made yet, i.e. $U_0(\alpha) = \{u \in X_{t'} \cap V_0 \mid (u,v) \in E \wedge \alpha(u) = \emptyset\}$. Then we define $U(\alpha) = \{mod(\beta, U, v) \mid U \subseteq U_0(\alpha) \wedge \beta = mod(\alpha, U_1, v)\}$. In other words we first create β by considering all choices of P_1, and then player P_0 sets strategy for a subset of the vertices where is he not yet decided.

In the second stage we connect the edges going from v to $X_{t'}$. We use a similar operator to *mod*:

$$mod2(\alpha, v, W)(u) = \begin{cases} \{(w, \max(\lambda(v), \lambda(w))) \mid w \in W\} & \text{if } u = v \\ \alpha(u) & \text{otherwise} \end{cases}$$

Let $W = \{w \in X_{t'} \mid (v,w) \in E\}$. We define $Border(t)$ depending on the player owning v.

$v \in V_1$

$$Border(t) = \{mod2(\gamma, v, W) \mid \gamma \in U(\alpha), \alpha \in Border(t')\}$$

$v \in V_0$

$$Border(t) = \{mod2(\gamma, v, \{w\}) \mid \gamma \in U(\alpha), \alpha \in Border(t'), w \in W\} \cup \{U(\alpha) \mid \alpha \in Border(t')\}$$

(In the first case we added all edges from v to X_t. In the second case we include all possible choices of V_0's strategy for v.) The correctness again follows from the definition of border.

Join Node Let t be a join node with t_1 and t_2 as its children. If we take $\alpha_1 \in Border(t_1)$ and $\alpha_2 \in Border(t_2)$, we are not guaranteed that there is a strategy σ s.t. $\alpha_1 = \alpha_\sigma^{t_1}$ and $\alpha_2 = \alpha_\sigma^{t_2}$. Instead of checking whether this is really the case we actually require a weaker condition:

Definition 4.6. Let $\alpha_1 \in Border(t_1)$ and $\alpha_2 \in Border(t_2)$. We say that α_1 and α_2 are *compatible* if one of the following is satisfied for each $v \in X_t \cap V_0$:

1. $\alpha_1(v) = \emptyset, \alpha_2(v) \neq \emptyset$
2. $\alpha_2(v) = \emptyset, \alpha_1(v) \neq \emptyset$
3. $\alpha_1(v) = \alpha_2(v) = \emptyset$ and there is $w \in V \setminus V_t$ such that $(v, w) \in E$
4. $\alpha_1(v) = \alpha_2(v) \neq \emptyset$

For compatible borders $\alpha_1 \in Border(t_1)$ and $\alpha_2 \in Border(t_2)$ we define the following operator J:

$$\alpha_1 J \alpha_2(v) = \begin{cases} \alpha_1(v) & \text{if } v \in V_0 \text{ and } \alpha_2(v) = \emptyset \\ \alpha_2(v) & \text{if } v \in V_0 \text{ and } \alpha_1(v) = \emptyset \\ \alpha_1(v) & \text{if } v \in V_0 \text{ and } \alpha_1(v) = \alpha_2(v) \\ \min_{\sqsubseteq}(\alpha_1(v), \alpha_2(v)) & \text{if } v \in V_1 \end{cases}$$

Lemma 4.3. *Let t be a join node with t_1 and t_2 as its children. Then*

$$Border(t) = \{\alpha_1 J \alpha_2 \mid \alpha_1 \in Border(t_1), \alpha_2 \in Border(t_2), \alpha_1 \textit{ compatible with } \alpha_2\}$$

Proof. We first show that $Border(t)$ is included in the set on the right – this is the easy inclusion. For each $\alpha_\sigma^t \in Border(t)$ consider $\alpha_\sigma^{t_1}$ and $\alpha_\sigma^{t_2}$, the borders corresponding to σ for the nodes t_1 and t_2. By definition of *Border* we have

$\alpha_\sigma^{t_1} \in Border(t_1)$ and $\alpha_\sigma^{t_2} \in Border(t_2)$. It is easy to check that $\alpha_\sigma^{t_1}$ and $\alpha_\sigma^{t_2}$ are compatible and that $\alpha_\sigma^t = \alpha_\sigma^{t_1} J \alpha_\sigma^{t_2}$.

For the other inclusion take $\alpha_{\sigma_1}^{t_1} \in Border(t_1)$ and $\alpha_{\sigma_2}^{t_2} \in Border(t_2)$ which are compatible, and define the strategy σ by the following prescription:

$$\sigma(x) = \begin{cases} \sigma_1(x) & \text{if } x \in V_{>t_1} \\ \sigma_2(x) & \text{if } x \in V_{>t_2} \\ \sigma_1(x) & \text{if } x \in X_t \text{ and } \alpha_{\sigma_2}^{t_2}(x) = \emptyset \\ \sigma_2(x) & \text{if } x \in X_t \text{ and } \alpha_{\sigma_1}^{t_1}(x) = \emptyset \\ \sigma_1(x) & \text{if } x \in X_t \text{ and } \alpha_{\sigma_1}^{t_1}(x) = \alpha_{\sigma_2}^{t_2}(x) \end{cases}$$

By definition of tree-width $V_{t_1} \cap V_{t_2} = X_t$ (i.e. $G[V_{t_1}]$ and $G[V_{t_2}]$ are disjoint except for their common interface), so we only have to check the choices made for vertices in $V_0 \cap X_t$. Obviously if $\alpha_{\sigma_1}^{t_1}(x) = \emptyset$, then no successor has been chosen for the vertex v in $G[V_{t_1}]$. Similarly for $\alpha_{\sigma_2}^{t_2}(x) = \emptyset$. Finally if $\alpha_{\sigma_1}^{t_1}(x) = \alpha_{\sigma_2}^{t_2}(x)$ it does not matter which strategy we use for the vertex x. Altogether we get $\alpha_{\sigma_1}^{t_1} J \alpha_{\sigma_2}^{t_2} = \alpha_\sigma^t$ for the strategy σ defined above and therefore $\alpha_{\sigma_1}^{t_1} J \alpha_{\sigma_2}^{t_2} \in Border(t)$. □

4.4.3 Main Result

Theorem 4.3. *Let $\mathcal{G} = (V, E, \lambda)$ be a parity game, $(T, \mathcal{X})$ a tree decomposition of G of width k and $v \in V$. Then we can solve $\mathcal{G}(v)$ in time roughly $O(n \cdot (k+1)^2 d^{2(k+1)^2})$, where $n = |V|$ and $d = |\{\lambda(v) \mid v \in V\}|$.*

Proof. We first convert the tree-decomposition $(T, \mathcal{X})$ (assuming it has $O(n)$ nodes) into a nice tree-decomposition using Lemma 4.1, which says that a nice tree decomposition of at most $4n$ (and of the same width) nodes can be constructed from $(T, \mathcal{X})$ in $O(n)$ time.

Let $r \in V(T)$ be a node such that $v \in X_r$ and orient T so r is the root. We now compute the set $Border(r)$ using the algorithm above. Finally we can find the winner of $\mathcal{G}(v)$ applying the Corollary 4.1.

It remains to establish the time needed to compute $Border(t)$ for $t \in V(T)$. The size of $Border(t)$ is roughly $d^{(k+1)^2}$ for each $t \in V(T)$, because $\alpha \in Border(t)$ can be thought of as a table of priorities of size $(k+1) \times (k+1)$. (To get a precise bound we need to consider also the elements $\bot$ and $\top$. That would give us the bound of $(d^{k+1}+2)^{k+1}$.)

The time to compute $Border(t)$ is different for the four different types of nodes. What dominates is the time needed for join nodes. For each join node t we have to consider all pairs $\alpha_1 \in Border(t_1)$, $\alpha_2 \in Border(t_2)$. The time needed for testing compatibility and computing $\alpha_1 J \alpha_2$ is at most $(k+1)^2$. Therefore the time to compute $Border(t)$ can be bounded by $(k+1)^2 \cdot d^{2(k+1)^2}$.

As there are at most $4n$ nodes the time needed to compute $Border(r)$ is roughly in $O(n \cdot (k+1)^2 \cdot d^{2(k+1)^2})$. This is also the bound on the time needed to find a winner for $\mathcal{G}(v)$, as the test in Corollary 4.1 can be performed in time $O((k+1)^2)$. It remains to mention that in the general case the number of priorities d is from the range $\langle 1, n \rangle$, and therefore our algorithm is *polynomial* in n. □

We have been able to identify examples of parity games for which the standard algorithm based on computing approximants needs exponential time, but which are of very low tree-width. In [Mad97] there is an example of such a parity game. This example is parametrised by n – the number of vertices. The game graph in Fig. 4.3 shows an instance of size 10. Note that the tree-width of this game graph is only 2 (this value does not depend on n).

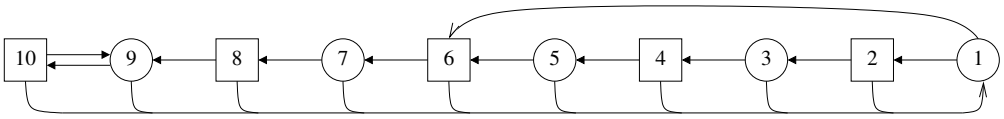


Figure 4.3: Parity game example

4.5 Adaptation to μ-calculus

In this section we explain how to adapt the algorithm for parity games to μ-calculus model checking. As an instance of a model checking problem, we are given a transition system $\mathcal{T}$ of size n and a μ-calculus formula φ of size m, and alternation depth d. Moreover, we assume that $\mathcal{T}$ has a tree decomposition of tree-width k and therefore also a nice tree decomposition $(T, \mathcal{X})$ of the same size.

The most straightforward way would be to translate $\mathcal{T}$ and φ into a parity game $\mathcal{G}$ following the construction given in Section 2.5.3. The game graph of $\mathcal{G}$ is by construction somewhere between synchronous and parallel product of

T and $Sub(\varphi)$, the set of all subformulas of φ. (Both synchronous and parallel product G of G_1 and G_2 have the set of vertices $V(G) = V(G_1) \times V(G_2)$. The set of edges is $E(G) = \{((v,w)(v',w')) \mid (v,v') \in E(G_1), (w,w') \in E(G_2)\}$ in the case of synchronous product, and $E(G) = \{((v,w)(v,w')) \mid v \in V(G_1), (w,w') \in E(G_2)\} \cup E(G) = \{((v,w)(v',w)) \mid w \in V(G_2), (v,v') \in E(G_1)\}$.) However, how can we be sure that the tree-width of G is bounded by k, as the T is? Note that in general the product of G_1 and G_2 can be of much higher tree-width than the graphs G_1 and G_2. An example of this is in Fig. 4.4 for parallel product. The tree-width of both G_1 and G_2 is one, however the tree-width of $G_1 \times G_2$ is 4, as it is a 4×4 grid. (The construction works for a synchronous product as well. We just have to add loops to all vertices of G_1 and G_2.)

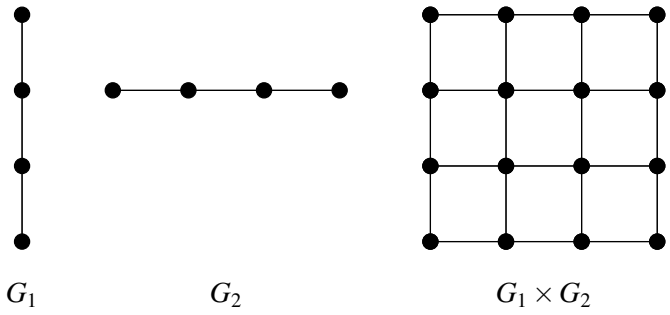


Figure 4.4: Graphs G_1, G_2 and their parallel product

But let us closely look at the construction of the parity game in Section 2.5.3. The vertices of the game are pairs (v,ψ), where $v \in V(\mathcal{T})$ and $\psi \in Sub(\varphi)$. Note that all the edges (except those created in steps 8. and 10.) are of the form $((v,\psi),(v,\psi'))$ – i.e. only the formula component of the vertex changes. In the remaining cases 8. and 10. the edges are of the form $((v,\psi),(w,\psi'))$, where $(v,w) \in E(\mathcal{T})$. This leads us to the following modification of the algorithm for parity games: We compute on the tree-decomposition of T, but for each vertex $v \in V(\mathcal{T})$ we include all pairs (v,ψ), $\psi \in Sub(\varphi)$ in the border. In other words we still play the game $\mathcal{G}$, but let T to tell us which vertices of $V(G)$ we see.

Formally, let $(T,\mathcal{X})$ be the tree-decomposition of $\mathcal{T}$ of width k. Then the border of $t \in V(T)$ will be a function $\beta : X_i \times Sub(\varphi) \to \{\bot, \top, 2^{X_i \times Sub(\varphi) \times P}\}$ (cf. Definition 4.4). In other words, borders will be tables with rows and columns annotated by pairs from the set $X_i \times Sub(\varphi)$. If we put $m = |Sub(\varphi)|$, then the border dimensions will be at most $(k+1) \cdot m \times (k+1) \cdot m$.

Having modified borders, we have to modify the algorithm as well. This is

easy – instead of adding (Introduce nodes) or removing (Forget note) a single vertex v we add/remove all vertices (v, ψ) for $\psi \in Sub(\varphi)$. Start nodes are obvious. Finally when dealing with Join nodes, we just modify the check when two borders correspond to the same strategy. The existence of edges between vertices of G (pairs (v, ψ)) can easily be checked on the fly. That we deal with all the edges is guaranteed by the above-mentioned fact that all the edges (except those created in cases 8. and 10.) are of the form $((v, \psi), (v, \psi'))$, and in cases 8. and 10. the edges are of the form $((v, \psi), (w, \psi'))$, where $(v, w) \in E(T)$.

4.5.1 Complexity

Theorem 4.4. *Let $\mathcal{T}$ be a transition system of n vertices with a tree decomposition of tree-width k, and φ a formula of size m and alternation depth d. Then we can solve the model checking problem for $\mathcal{T}$ and φ in time time $O(n \cdot ((k+1) \cdot m)^2 d^{2((k+1)\cdot m)^2})$.*

Proof. We start with the complexity estimate for parity games. In the μ-calculus case, the size of borders has grown from $k+1$ to $(k+1) \cdot m$. However, we do not increase the number of nodes in tree-decomposition. The number of priorities is equal to d. The rest follows from Theorem 4.3. □

Comparing to the result of [LBC+94], our algorithm is *linear* in the size of the system, no matter what the formula is. It should be also noted, that the estimated running time is really the upper bound and the algorithm may benefit from further optimisation.

4.5.2 Application to Software Model Checking

The algorithm presented above looks suitable for model checking software. Programs written in structured programming languages have a low tree-width and, moreover, we can find the tree decomposition just by performing a simple syntactic analysis [Tho98]. In practice it is usually the case that the size of the system itself is huge, whereas the formula is quite small. This is where the fact that our algorithm is linear in the size of the system may give better results compared to previous algorithms.

Chapter 5

DAG-width

Most of the results presented in this chapter, with the notable exception of the algorithm for parity games (Section 5.6, which has been adapted from [BDHK06]), have been published in slightly different form in [Obd06].

In the previous chapter we have seen that on graphs of bounded tree-width we can solve parity games in polynomial time. One drawback of this approach is that for the purposes of tree decomposition we ignore the orientation of edges. However there are graphs on which it is easy to solve a parity game in polynomial time, but can which can be of tree-width equal to the number of its vertices. A typical example of such a graph is a directed clique of size n, which arises from an undirected clique of the same size by orienting edges in such a way they form a DAG (with one source and one sink). As we have seen in the previous chapter (page 41) K_n, a clique of size n, has tree-width $n-1$ but it is easy to solve parity games on DAGs. (By DAGs in this context we mean directed acyclic graphs. In the case of parity games we allow (and require) self loops at leaves, guaranteeing every vertex has a successor, so that the parity game is well defined.) Similarly it is easy to solve parity games on trees with back edges – see Section 3.6.

Therefore we may look for some decomposition similar to tree decomposition, which is defined on directed graphs. As surprising as it may seem, there are not that many such decompositions around. The main reason for this has been the problem of finding the right 'separation lemma' for directed graphs (more or less a canonical way of dividing a graph using smallest possible cut-sets).

Of the few notions of decomposition which appear in the literature [JRST01,

BG04, Saf05], the notion of *directed tree-width* defined by Johnson, Robertson, Seymour and Thomas in [JRST01] has probably been the most important. (Some background on directed tree-width can also be found in [Ree99].) The decomposition structure of a graph in this case is a tree (as for tree decompositions), but this time directed (i.e. with a designated root). In their paper the authors present an algorithm for solving problems like Hamiltonian cycle in polynomial time on graphs of bounded directed tree-width.

However, the notion of directed tree-width has had less impact than tree-width. We try to identify reasons why this is so. It appears as though in the attempt to capture as broad a class of graphs as possible, the definition is too general. The main problem seems to be that the separator sets are not monotone with respect to decomposition (called arboreal decomposition). This makes reasoning about directed tree-width complicated and error prone (if you want to read the [JRST01] paper, check also the addendum [JRST02]). Also the requirement of non-empty sets in the nodes of the decomposition causes several other issues.

Our goal is therefore to find a measure with nice algorithmic and graph theoretical properties, which is simpler to use and reason about and retains generality. We introduce a new connectivity measure called DAG-width, whose decomposition structure is a DAG.

This measure has been first published at SODA'06 by the author of this thesis. Independently Berwanger, Dawar, Hunter and Kreutzer came up with exactly the same measure, even giving it the same name (this latter fact is not that surprising, since 'DAG-width' is an obvious choice). Their paper [BDHK06] appeared at STACS'06 later than [Obd06], but in addition to the results presented in [Obd06] it also contains the algorithm for solving parity games which is not included in [Obd06]. We are now working together on exploring the DAG-width. A joint journal paper should appear soon.

The rest of this chapter is organised as follows: In the following section we present the notion of directed tree-width as defined in [JRST01]. The next section contains the definition of DAG-width and some of its properties including a normal form. Then we present a variant of cops-and-robber games related to the definition of DAG-width. We continue by proving that this measure is a little stricter than directed tree-width and show also the relationship to tree-width of undirected graphs. Finally in the last section we present a modified

version of the algorithm [BDHK06] for solving parity games in polynomial time on graphs of bounded DAG-width.

5.1 Directed Tree-width

Directed tree-width was introduced by Johnson, Robertson, Seymour and Thomas in [JRST01] (also see [Ree99]) as a counterpart of tree-width for directed graphs. The decomposition structure is still a tree, though this time directed (i.e. with a designated root). We will see that the definition looks quite different from that of tree-width, while being still closely related. Before we present the definition we need to introduce some more notation.

For a directed acyclic graph R we use the following notation: If $r, r' \in V(R)$ we write $r < r'$ iff there is a directed path with initial vertex r and terminal vertex r'. We write $r \leq r'$ iff $r < r'$ or $r = r'$. Finally if $e \in E(R)$ then $e \sim r$ iff e is incident with r. For a graph G a set $S \subseteq V \setminus Z$ is *Z-normal* if there is no directed path in $G \setminus Z$ with first and last vertices in S that uses a vertex of $G \setminus (S \cup Z)$. I.e. no path can leave S and then return back to S without passing through a vertex in Z.

Definition 5.1 (arboreal decomposition)**.** An *arboreal decomposition* of a graph G is a triple $(R, \mathcal{X}, \mathcal{W})$ where R is a directed tree, and $\mathcal{X} = \{X_e \mid e \in E(R)\}$, $\mathcal{W} = \{W_r \mid r \in V(R)\}$ are sets of vertices of G satisfying:

(R1) $\mathcal{W}$ is a partition of $V(G)$ into nonempty sets

(R2) for $e \in E(R)$, $e = (r_1, r_2)$ the set $\bigcup\{W_r \mid r \in V(R) \text{ and } r \geq r_2\}$ is X_e-normal.

The *width* of $(R, \mathcal{X}, \mathcal{W})$ is the least integer w such that for all $r \in V(R)$, the union of W_r and the sets X_e on neighbouring edges has at most w elements (formally $|W_r \cup \bigcup_{e \sim r} X_e| \leq w$). The *directed tree-width* of a graph G (written as $dtw(G)$) is the minimum width over all possible arboreal decompositions of G.

You can see an example of arboreal decomposition of width 1 in Fig. 5.1. Sets W_r are drawn in the nodes and edges are annotated by sets X_e.

5.1.1 Game Characterisation

As in the case of tree-width, the authors of directed tree-width attempted to give a game characterisation of graphs of bounded directed tree-width. The

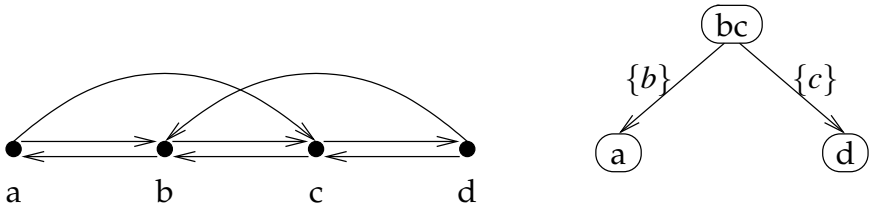


Figure 5.1: Graph and its arboreal decomposition

original informal definition goes like this: The robber stands on a vertex of the graph, and can at any time run at a great speed to any other vertex along an oriented path of the graph. He is not permitted to run through a cop, and must stay in the same strongly connected component of $G \setminus Z$, where Z is the set of vertices currently occupied by the cops. There are k cops, each of whom at any time either stands on a vertex or is in a helicopter. The goal of the player controlling the cops is to land a cop via a helicopter onto a vertex currently occupied by the robber, and the robber's objective is to elude capture. (The point of helicopters is that cops are not constrained to move along the paths of the graph.) The robber can see the helicopter landing and may run to a new vertex before it actually lands.

More formally, the game is played on a graph G by two players: the cop player, and the robber player. It is played according to the following rules: At the beginning the robber player chooses a vertex $u \in V(G)$, giving us an initial game position $(\emptyset, u)$. Given a position (X, v), the cop player chooses a set $X' \subseteq [V]^{\leq k}$, and the robber player a vertex $v' \in V(G) \setminus X'$ such that both v and v' lie in the same strongly connected component of the graph $G \setminus (X \cap X')$, giving us the next position (X', v'). A play is a maximal sequence of positions formed from an initial game position according to the rule above. The play is winning for the cop player if it is finite – i.e. for the final position (X, v) of the play it is true that there is $X' \in [V]^{\leq k}$ such that no vertex of the graph $V(G) \setminus X'$ is in the same strongly connected component of the graph $G \setminus \{X \cap X'\}$ (this immediately implies $v \in X'$). On the other hand the robber player wins if the play is infinite. If k cops can capture the robber in G we say that k cops can *search* G. Moreover if they can do so without revisiting a vertex then they can *monotonely search* G.

If we compare this game to the game characterising tree-width (see Section 4.2), we see that there are two main differences: 1. the robber must re-

spect the orientation of edges, and 2. he must stay in the same strongly connected component of $G \setminus Z$. The first restriction is very natural, as it is the most straightforward generalisation. However restricting the robber to stay in the same strongly connected component is something which makes the game rather different. One possible explanation is that this restriction is closely related to the restrictions we consider when working in the domain of network flows, which is one of the traditional applications of directed graphs and separator sets. At least the paper [Ree99] seems to support this explanation. Another explanation is the restriction to strongly connected components allows to generalise the notion of haven for the cops and robber game.

It has also been shown in [JRST01] that monotone and non-monotone strategies for cops are not necessarily equivalent, this being in sharp contrast with games for the undirected case of tree-width. Their example is in Fig 5.2. Here we use the convention that an undirected edge represents two edges with the same ends, one in each direction. This graph has directed tree-width three, but there is no monotone search strategy for four cops - they have to revisit a previously occupied vertex in order to capture the robber.

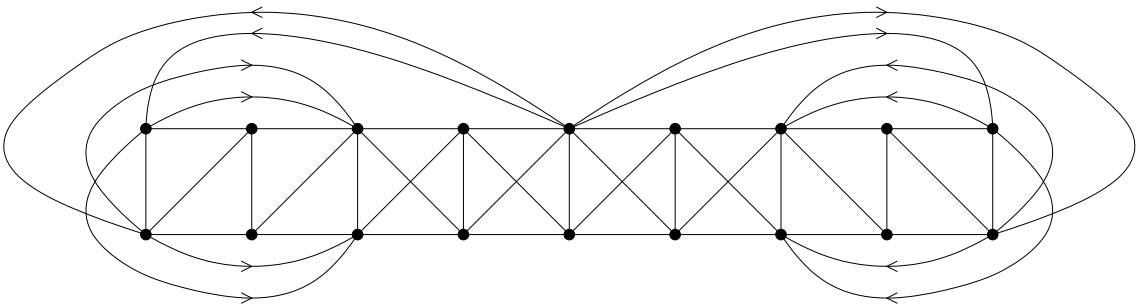


Figure 5.2: Graph for which monotone and non-monotone search strategies are not equivalent

As for the undirected case of the game, there is a notion of haven. Let G be a directed graph and $w > 0$. A *haven* of order w in D is a function β assigning to every set $Z \subseteq V(G)$ with $|Z| < w$ vertex set of a strongly connected component of $G \setminus Z$ in such a way that if $Z' \subseteq Z \subseteq V(G)$ with $|Z| < w$, then $\beta(Z') \subseteq \beta(Z)$. It is easy to see that if there is a haven of order w in the graph G, then the robber can win against $w-1$ cops by staying in $\beta(Z)$, where Z is the set of vertices currently occupied by the cops. As we pointed out earlier, the definition of haven very closely resembles the one for the undirected case (cf. page 44).

Regarding the relationship between directed tree-width and the games defined above, the following is shown in [JRST01]:

Theorem 5.1 ([JRST01])**.** *Let G be directed graph and k an integer. If the robber has a haven of order k, then the directed tree-width of G is at least $k-1$.*

The proof of this theorem is simple and constructive. However the authors were not able to prove the opposite implication. What the authors have actually proved is the following weaker statement:

Theorem 5.2 ([JRST01])**.** *Let G be directed graph and $k > 0$ an integer. Then either the directed tree-width of G is at most $3k-2$, or there is a haven of order k (which implies the robber has a winning strategy against $k-1$ cops).*

We can actually show that the converse of Theorem 5.1 does not hold. See the graph in Fig. 5.3. This graph has four components: K_6, K_{4a}, K_{4b}, and K_2. Double arrow between the components signifies that there is an edge from every vertex of the tail component to every vertex of the head component. (So from each vertex of K_6 you can get to any other vertex, and from each vertex of K_{4a} (K_{4b}) you can get to both vertices of K_2.)

In this graph six cops can capture the robber, even using a monotone strategy. Their strategy is following:

1. occupy K_5

2a. if the robber moves to K_{4a}, occupy the vertex a

3a. occupy K_2 (using the cops from x and y)

4a. occupy the rest of K_{4a}, using the remaining cops from K_5

2b-4b. as 2a-4a, but working on b, K_{4b} instead

On the other hand there is no arboreal decomposition of width five for this graph. The proof of this fact is a tedious analysis, and can be done by an exhaustive search in the space of possible decompositions. The intrinsic reason why we cannot find a decomposition of width 5 is that the definition of arboreal decomposition requires the sets W_r to form a partition of $V(G)$ into *nonempty* sets. If we allowed the sets W_r to be empty, this problem would not arise and the graph in Fig. 5.3 would have an arboreal decomposition of width five.

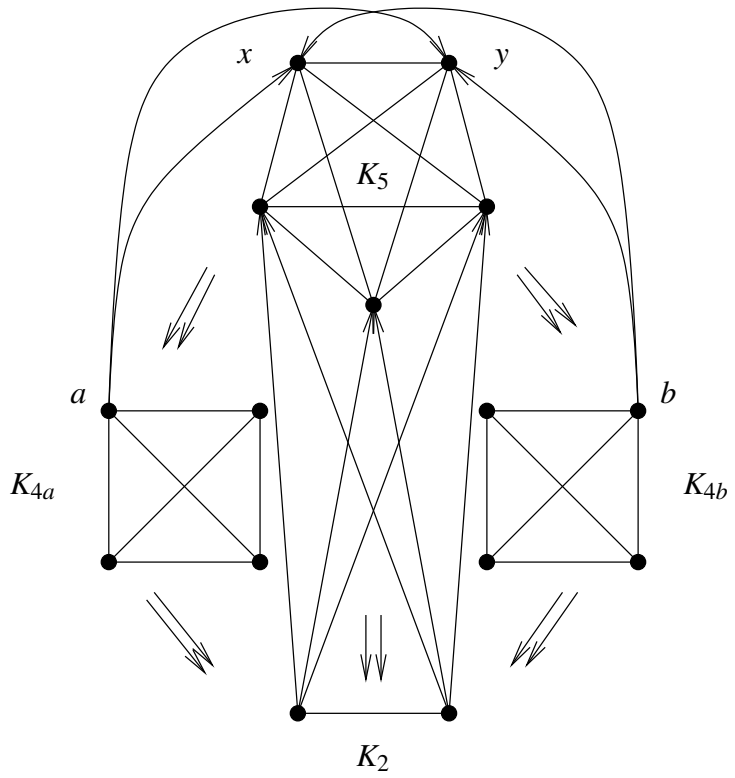


Figure 5.3: Graph which can be searched by 6 cops, but has directed tree-width > 5

Unknown to the author up till few days before submitting this thesis, Adler [Adl] has proved among other things the same result. The paper [Adl] contains a different example together with a full proof.

5.1.2 Algorithms

In their paper [JRST01], the authors present a generic algorithm to solve many NP-hard problems in polynomial time. As in the case of ordinary tree-width, the dynamic programming approach is used, computing a table of partial solutions for each node of the decomposition. Here the authors show a generic requirement for the algorithm to run in polynomial time: For every integer k there is a real number α such that the two following properties are true about the tables of partial solutions (here called *itineraries*).

Axiom 1 Let G be a graph and $A, B \subseteq V(G)$ disjoint sets of vertices such that no edge of G has head in A and tail in B. Then the itinerary for $A \cup B$ can be computed from itineraries for A and B in time $O((|A| + |B|)^{\alpha})$.

Axiom 2 Let G be a graph and $A, B \subseteq V(G)$ disjoint sets of vertices such that A is Z-normal for some $Z \subseteq V(G)$ and $|B| \leq k$. Then the itinerary for $A \cup B$ can be computed from itineraries for A and B in time $O((|A|+1)^{\alpha})$.

The construction of a polynomial algorithm for a given itinerary is obvious: We go through nodes in the order given by $<$. For a node d with multiple successors we compute first the information for $G[W']$, where $W' = \bigcup_{(d,d') \in E(R)} W_{>d'}$. This we can do by iterative application of Axiom 1. Then they apply Axiom 2 to $G[W']$ and W_d. There is however a problem in the original proof, which was first pointed out by the author of this thesis and subsequently fixed in [JRST02].

Using this approach, the paper continues to show that using the generic approach above the following problems can be decided in polynomial time on graphs of bounded directed tree-width: Hamiltonian path and Hamiltonian cycle, even cycle through a specified vertex etc.

5.2 DAG-width

The main issue with the directed tree-width is that arboreal decompositions are not intuitively related to the graphs they decompose. More specifically, the problematic element are the sets X_e. The only structural restriction on these sets is that the elements of X_e cannot belong to the subtree at the head of e. One would expect a restriction like: 'for each vertex $v \in V(G)$, the sets X_e (respective their associated edges) s.t. $v \in X_e$ form a connected subtree of R'. Design of algorithms working on arboreal decompositions is then very complicated, as we cannot exploit any extra structure.

This is probably the reason why there have not been many papers citing [JRST01] and the measure proved to be very difficult to use for designing algorithms[1]. The author of this thesis tried for some time to come up with polynomial algorithm for solving parity games on graphs of bounded directed tree-width, but failed miserably probably because of the issues above.

Since there is still need for a good measure of a directed graph, we would like to propose a new measure called DAG-width, which rectifies some of the problems associated with directed tree-width. The design goals for the new measure are summarised below:

[1] This is the opinion of most of the graph theorists the author has talked to.

- The decomposition must be reasonably intuitive.
- There should exist a straightforward game characterisation.
- The new measure should have a close relationship to both tree-width and directed tree-width.
- And it should be closed under directed unions.

The main difference with both tree-width and directed tree-width is that we use a DAG instead of a (directed) tree as basis for the decomposition. (By DAG in the rest of this chapter we mean directed acyclic graph, without self-loops. Also any vertex with no incoming edges is called *root* of the DAG throughout this chapter. The relation $\sqsubseteq$ on vertices of a DAG is the same as defined in Section 5.1.) This indeed looks natural for the case of directed graphs. The definition of DAG-width is below. Note that the properties (D1)-(D3) closely correspond to (T1)-(T3) in Definition 4.1.

Definition 5.2 (DAG decomposition)**.** A *DAG decomposition* of a (directed) graph G is a pair $(D, \mathcal{X})$ where D is a DAG and $\mathcal{X} = \{X_d \mid d \in V(D)\}$ is a multiset of subsets of $V(G)$ satisfying:

(D1) $V(G) = \bigcup_{d \in V(D)} X_d$

(D2) If $(d, d') \in E(D)$, then for each $(v, w) \in E$ s.t. $v \in X_{\geq d'} \setminus X_d$ we have $w \in X_{\geq d'}$, where $X_{\geq c} = \bigcup_{c' \geq c} X_{c'}$.
If d is a root we require for each $v \in X_{\geq d}$ and $(v, w) \in E$ that also $w \in X_{\geq d}$.

(D3) for all $d, d', d'' \in D$ if d' lies on (some) path from d to d'', then $X_d \cap X_{d''} \subseteq X_{d'}$.

In the rest of this paper we will also use $X_{>d}$ to denote the set $X_{\geq d} \setminus X_d$. The *width* of a DAG decomposition $(D, \mathcal{X})$ is $\max_{d \in D} |X_d| - 1$. The *DAG-width* of a graph G (written $dgw(G)$) is the minimum width over all possible DAG decompositions of G. DAGs have DAG-width zero. To get a better intuition consider Fig. 5.4.

The properties (D1) and (D3) in the definition of DAG-width are self-explanatory, but (D2) deserves some comments. It says that if vertex $v \in X_{\geq d'} \setminus X_d$ ('below' and including the node d', but not in the node d), then all its successors must be in $X_{\geq d'}$. An alternative definition can be given using 'guarding'. We

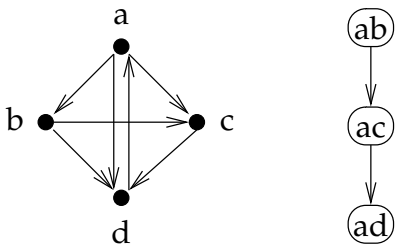


Figure 5.4: Graph G and its DAG decomposition

say that a set $W \subseteq V$ *guards* a set $V' \subseteq V$ if whenever there is an edge $(u,v) \in E$ such that $u \in V'$ and $v \notin V'$, then $v \in W$. Using guarding D2 can be rephrased as:

(D2′) For each $(d,d') \in E(D)$ the set $X_d \cap X_{d'}$ guards $X_{\geq d'} \setminus X_d$. For the root d of D the set $X_{\geq d}$ is guarded by $\emptyset$.

In addition to the root of D there is a different kind of root nodes. Node $d \in V(D)$ is a *root node* of vertex $v \in V(G)$ if there exists $(d',d) \in E(D)$ such that $v \notin X_{d'}$. I.e. at least one of the predecessors of d must not contain v. We denote $X_R(v)$ the set of all root nodes for a vertex v. (Obviously by (D1) for each vertex of G this set contains at least one element.)

From the definition of DAG decomposition it is apparent that root nodes play an important role - and there is an associated normal form. We say that a DAG decomposition $(D,\mathcal{X})$ is in *normal form*, if in addition to (D1)-(D3) the following two properties hold:

(i) for every edge $(d,d') \in E(D)$ there is no other path from d to d' in D.

(ii) every node $d \in D$ is a root node of some vertex $v \in V(G)$

Lemma 5.1 (normal form)**.** *Let G be a graph and $(D,\mathcal{X})$ its DAG decomposition of width k. Then there is an algorithm which converts $(D,\mathcal{X})$ into $(D',\mathcal{X}')$ of width k in normal form in time linear in the size of D.*

Proof. We will construct a sequence $(D,\mathcal{X}) = (D_0,\mathcal{X}_0),\ldots,(D_f,\mathcal{X}_f) = (D',\mathcal{X}')$ of pairs $(D_i,\mathcal{X}_i)$ by iteratively removing (one at a time) edges violating (i) and nodes violating (ii) in the definition of normal form above, while maintaining the invariant that $(D_i,\mathcal{X}_i)$ is a DAG decomposition of G.

$(D_0,\mathcal{X}_0)$ satisfies the DAG decomposition axioms by definition. Let $(D_i,\mathcal{X}_i)$ be the DAG decomposition after the i-th iteration. For case (i) we put $V(D_{i+1}) =$

$V(D_i), E(D_{i+1}) = E(D_i) \setminus \{(d,d')\}, \mathcal{X}_{i+1} = \mathcal{X}_i$. It is clear that removing an edge (d,d') s.t. there is some other path from d to d' in D_i does not violate any of (D1)-(D3).

For case (ii) assume there is a node $d \in V(D_i)$ s.t. d is not a root node for any vertex. Let $s_1, \ldots, s_k$ be its predecessors and $t_1, \ldots, t_l$ its successors. We put $V(D_{i+1}) = V(D_i) \setminus \{d\}$, $\mathcal{X}_{i+1} = \mathcal{X}_i \setminus X_d$ and $E(D_{i+1}) = \{e \in E(D_i) \mid e \not\sim d\} \cup E'$ where $E' = \{(s_i, t_j) \mid 1 \leq i \leq k, 1 \leq j \leq l\}$. I.e. we remove the node d and add an edge from each direct predecessor of d to each direct successor of d.

(D1) is preserved since d is not a root node for any vertex, and therefore each vertex must also be in some other set $X_{d'}$.. To prove (D2) it is sufficient to note that we must have $X_d \subseteq X_{s_i}$ for $1 \leq i \leq k$ as d is not a root node for any vertex. Finally (D3) is also easy, as that property is not affected by removing a node on any path in D.

□

A natural question to ask would be whether we get a normal form for graphs of DAG-width k if (ii) above is replaced by 'every node $d \in D$ is a root node of exactly one vertex $v \in V(G)$', giving us a bijection between vertices of G and nodes of D. However using this restriction we would run into similar difficulties as in the case of directed tree-width – our definition would not correspond to the most natural extension of the cops and robber games characterising tree-width [ST93]. An example to illustrate this is the graph in Fig. 5.3. In any DAG decomposition of this graph of DAG-width five we need to have two roots for the vertices in K_2.

We also could have made the root nodes more prominent in the definition of DAG decomposition. The following property (D2″) can be used instead of (D2) without changing the class of the DAG decompositions:

(D2″) if $(v,w) \in E(G)$ then for each $d \in X_R(v)$ either $w \in X_d$ or there exists $d' \in X_R(w)$ s.t. $d < d'$.

To see that (D2) implies (D2″) consider an edge $(v,w) \in E(G)$ and a node $d \in X_R(v)$. Because d is a root node of v, there must be a node d' s.t. $(d',d) \in E(D)$ and $v \notin X_{d'}$. As $X_{d'} \cap X_d$ guards $X_{\geq d} \setminus X_{d'}$, then either $w \in X_d$, or there is a path in D from d to d'' where w appears for the first time. In the latter case by definition $d'' \in X_R(v)$. The proof that (D2″) implies (D2) is very similar.

In addition to root nodes for a vertex v we also define *root node* for a DAG decomposition $(D, \mathcal{X})$ to be a node d with no predecessor in D (in other words source vertex of D). Since D is a DAG a root node of D must always exist. Moreover, by the following lemma, we can always assume that D has a unique root node. In the text to follow we will usually assume that this is the case.

Lemma 5.2. *Let $(D, \mathcal{X})$ be a DAG decomposition of width k where D has multiple roots $d_1, \ldots, d_m$. Then the decomposition $(D', \mathcal{X}')$, where $V(D') = V(D) \cup \{d_0\}$, $E(D') = E(D) \cup \{(d_0, d_i) \mid 1 \leq i \leq m\}$ and $\mathcal{X}' = \mathcal{X} \cup \{X_{d_0}\}$, where $X_{d_0} = \emptyset$, is a DAG decomposition of G of width k.*

Proof. The process of adding a new root is depicted in Fig. 5.5. As $X_{d_0} = \emptyset$ and we have not changed the rest of the decomposition it is easy to see that (D1)-(D3) still hold. □

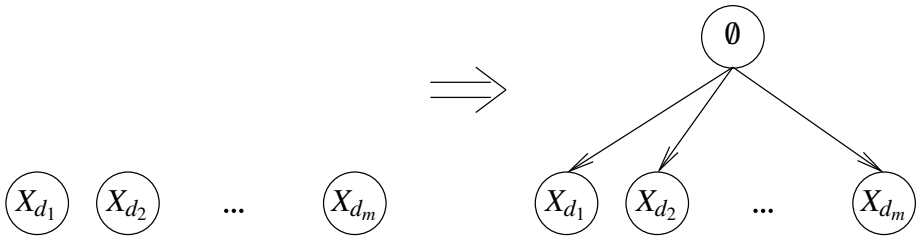


Figure 5.5: Adding a unique root to a DAG decomposition

In [JRST01] the authors mention two other attempts at defining a version of tree-width for directed graphs which did not work because they were not closed under directed unions. (A graph G is a directed union of graphs G_1 and G_2 if G_1 and G_2 are induced subgraphs of G, $V(G_1) \cup V(G_2) = V(G)$ and no edge of G has a head in $V(G_1)$ and tail in $V(G_2)$.) DAG decompositions are indeed closed under directed unions:

Lemma 5.3. *Let G, G_1, G_2 be as above with $dgw(G_1) = k_1$ and $dgw(G_2) = k_2$. Then $dgw(G) = \max\{k_1, k_2\}$.*

Proof. We assume that the set of vertices $V(G_1)$ and $V(G_2)$ are disjunct. Let $(D_1, \mathcal{X}_1)$ $[(D_2, \mathcal{X}_2)]$ be the decomposition of G_1 (G_2) of width k_1 (k_2), and let d_2 be the unique root of D_2. Then $(D, \mathcal{X})$, where $V(D) = V(D_1) \cup V(D_2)$, $E(D) = E(D_1) \cup E(D_2) \cup \{(d, d_2) \mid d \in D_1\}$ and $\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2$, is a DAG decomposition of width $\max(k_1, k_2)$. □

5.3 Games for DAG-width

We are now going to give a game characterisation of graphs of bounded DAG-width. The game we are considering is again a variant of the, by now familiar, cops-and-robber game. There are k cops moving around in a helicopter and a robber who is trying to avoid them. The only difference to the game for tree-width (cf. Sec 4.2) is that the robber must respect the orientation of edges. This means that in contrast to the case of directed tree-width games (cf. Sec 5.1.1) we do not require the robber to stay in the same strongly connected component, meaning we get a more natural generalisation to directed graphs.

Formally, the game is played on a graph G by two players: the cop player, and the robber player. It is played according to the following rules: At the beginning the robber player chooses a vertex $u \in V(G)$, giving us an initial game position $(\emptyset, u)$. Given a position (X, v), the cop player chooses a set $X' \subseteq [V]^{\leq k}$, and the robber player a vertex $v' \in V(G) \setminus X'$ such that there is a directed path from v to v' in the graph $G \setminus (X \cap X')$, giving us the next position (X', v'). A play is a maximal sequence of positions formed from an initial game position according to the rule above. The play is winning for the cop player if it is finite – i.e. for the final position (X, v) of the play it is true that there is $X' \in [V]^{\leq k}$ such that no vertex of the graph $V(G) \setminus X'$ is reachable from v in the graph $G \setminus \{X \cap X'\}$ (this immediately implies $v \in X'$). On the other hand the robber player wins if the play is infinite. If k cops can capture the robber in G we say that k cops can *search* G. Moreover if they can do so without revisiting a vertex then they can *monotonely search* G.

We are now going to show that this game indeed characterises DAG-width. We start with the easier direction:

Theorem 5.3. *Let G be a directed graph of DAG-width k. Then it can be monotonely searched by $k+1$ cops.*

Proof. Let $(D, \mathcal{X})$ be a DAG decomposition of G of width k. We will assume that the DAG D has a single root. If this is not so, we can convert $(D, \mathcal{X})$ into a DAG decomposition of the same width having this property by Lemma 5.2. The winning monotone strategy for the cops is as follows. In the first move cops will occupy X_{d_0}, where d_0 is the root of D. The robber now must be in some vertex $v \in X_{>d_0}$. Since D is a rooted DAG with every node accessible from the root, there must be an edge $(d_0, d_1) \in E(D)$ s.t. some $d \in X_R(v)$ is reachable from

d_1. In the next move the cops in $X_{d_0} \setminus X_{d_1}$ will take off, leaving only $X_{d_0} \cap X_{d_1}$ occupied. By (D2) the robber must stay in $X_{\geq d_1}$. The cops now occupy the remaining vertices of X_{d_1}, forcing the robber into $X_{>d_1}$ and so on. Continuing in this way the robber will be eventually captured. The number of cops is limited by $\max_{d \in D} |X_d| = dgw(G) + 1$. □

Unlike the case of directed tree-width we can prove that the opposite is true as well:

Theorem 5.4. *Let G be a directed graph which can be monotonely searched by $k+1$ cops. Then G has DAG-width at most k.*

Proof. Let π be a monotone search strategy for $k+1$ cops. It is not hard to see that we can always restrict ourselves to the case where at most one cop moves at a time. We will construct a DAG decomposition $(D, \mathcal{X})$ of width k as follows:

The nodes of our decomposition will be pairs (Y, C), where $Y \in [V]^{<k+1}$ and C is a strongly connected component of $G \setminus Y$. Here Y stands for the set of vertices currently occupied by the cops and C is the strongly connected component of $G \setminus Y$ containing the robber. We set $V(D)$ to be the set of all game positions (Y, C) from which the cop player wins using the strategy π.

Now fix a node $(Y, C) \in V(D)$, and let Y' be the next position of the cops given by the strategy π. Then for each strongly connected component C' of $G \setminus Y'$ s.t. C' is reachable from a vertex of C in the graph $G \setminus (Y \cap Y')$ we insert the edge $((Y, C), (Y', C'))$ into $E(D)$. Finally we put $X_{(Y,C)} = Y$ for every node $(Y, C) \in V(D)$.

Obviously the number of vertices in each node is at most $k+1$, the number of cops. It remains to be checked that the properties (D1) to (D3) hold, which is not hard: (D1) is obvious, and (D2) together with (D3) hold since π is monotone. □

Note that the Theorem 5.4 provides us with an upper bound on the minimal number of nodes in a DAG decomposition. The bound is polynomial in n and exponential in k, which is in contrast to both tree and arboreal decompositions. In the case of arboreal decompositions the number of nodes can be at most n, as the sets X_r are by definition non-empty and form a partition of V. In the case of tree decompositions, we have the bound $4n$ for nice tree decompositions (Lemma 4.1), which are already in a very special form.

It remains an open problem whether monotone and general strategies are equivalent - it indeed looks quite likely. As a step in this direction, the next theorem provides us with a way of describing a winning strategy for a robber:

Theorem 5.5. *A graph G cannot be searched by $< k$ cops iff there is a function σ mapping each $X \in [V(G)]^{<k}$ to a nonempty union $\sigma(X)$ of strongly connected components of $G \setminus X$ s.t. if $X \subseteq Y \in [V(G)]^{<k}$ then:*

1. $\forall S \in \sigma(X).\exists T \in \sigma(Y)$ *s.t. there is a directed path from S to T in $G \setminus X$*
2. $\forall S \in \sigma(Y).\exists T \in \sigma(X)$ *s.t. there is a directed path from S to T in $G \setminus X$*

Proof. If such a function σ exists, the robber can remain uncaptured by choosing the corresponding element of $\sigma(X)$ in each step. Conversely, suppose that $< k$ cops cannot search the graph. Then for each $X \in [V(G)]^{<k}$ let $\sigma(X)$ be the union of all strongly connected components of $G \setminus X$ s.t. the robber player can guarantee a win from those components. Then σ clearly satisfies the theorem. (To keep the size of $\sigma(X)$ small we can always chose the greatest subset s.t. there is not a (directed) path between any two strongly connected components in this set.) □

The function σ above plays the same role as havens for graphs of bounded tree-width/directed tree-width. When one compares the definition of σ and either of the two definitions of haven, he may ask whether we really need $\sigma(X)$ to be a union of strongly connected components, and not just a single strongly connected component. The answer is yes and the reason can be seen of Fig. 5.6. The graph G_X there cannot be searched by less than three cops. But it is easy to see that $\sigma(\{b,c\}) = G_a \cup G_d$ (where G_a and G_d are single-vertex components consisting only of a and d respectively), since two cops can force the robber from a to d and back again.

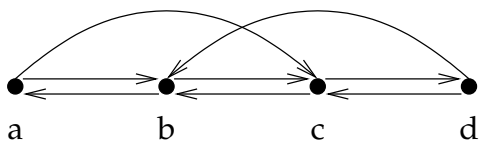


Figure 5.6: Graph G_X

5.4 Nice DAG Decompositions

As in the case of tree decompositions, providing algorithms working on general DAG decompositions would be unnecessarily complicated. Therefore in this section we introduce the notion of *nice DAG decomposition*, closely modelled after the nice tree decompositions for tree-width (cf. page 41).

Definition 5.3 (Nice DAG decomposition)**.** DAG decomposition $(D, \mathcal{X})$ is called a *nice DAG decomposition*, if the following four conditions are satisfied:

1. D has a unique root node
2. every node of D has at most two children,
3. if a node d has two children d_1 and d_2, then $X_d = X_{d_1} = X_{d_2}$, and
4. if a node d has one child d', then either $|X_d| = |X_{d'}| + 1$ and $X_{d'} \subseteq X_d$, or $|X_d| = |X_{d'}| - 1$ and $X_d \subseteq X_{d'}$.

Theorem 5.6. *Every graph G of DAG-width k has a nice DAG decomposition of width k. Furthermore let $(D, \mathcal{X})$ be a DAG decomposition of G of with n nodes. Then we can in time $O(n^2)$ construct a nice DAG decomposition $(D', \mathcal{X}')$ of the same width with $O(n^2)$ nodes.*

Before we present the proof of this theorem, we will need the following lemmas:

Lemma 5.4. *Let $(D, \mathcal{X})$ be a DAG decomposition of G, $(d, d') \in E(D)$ such that $X_{d'} \subset X_d$, and take $X \subset (X_d \setminus X_{d'})$. Let d'' be a new node not in D. We define $(D', \mathcal{X}')$ by putting:*

- $V(D') = V(D) \cup \{d''\}$
- $E(D') = (E(D) \setminus \{(d, d')\}) \cup \{(d, d''), (d'', d')\}$
- $\mathcal{X}' = \mathcal{X} \cup X_{d''}$, *where* $X_{d''} = X_{d'} \cup X$

Then $(D', \mathcal{X}')$ is a DAG decomposition of G of the same width as $(D, \mathcal{X})$.

Proof. The properties (D1) and (D3) are obviously satisfied. (D2) holds for the new edge (d, d'') since $X_{\geq d''} \setminus X_d = X_{\geq d'} \setminus X_d$, and for the other new edge (d'', d') since $X_{d''} \supset X_d$. □

Lemma 5.5. *Let $(D,\mathcal{X})$ be a DAG decomposition of G, $(d,d') \in E(D)$ such that $X_d \subset X_{d'}$, and take $X \subset (X_{d'} \setminus X_d)$. Let d'' be a new node not in D. We define $(D',\mathcal{X}')$ by putting:*

- $V(D') = V(D) \cup \{d''\}$
- $E(D') = (E(D) \setminus \{(d,d')\}) \cup \{(d,d''),(d'',d')\}$
- $\mathcal{X}' = \mathcal{X} \cup X_{d''}$, *where* $X_{d''} = X_d \cup X$

Then $(D',\mathcal{X}')$ is a DAG decomposition of G of the same width as $(D,\mathcal{X})$.

Proof. Similar to the proof of 5.4. □

Proof of Theorem 5.6. Let $(D,\mathcal{X})$ be a DAG decomposition of width k. We are going to transform this decomposition into a nice DAG decomposition $(D',\mathcal{X}')$ in several stages.

First if $(D,\mathcal{X})$ does not have a single root, we can add one using Lemma 5.2. Second for each node d which has at least two successors $d_1,\ldots,d_l$ we replace this node with a binary branching tree with l leaves, such that every leaf has a single successor d_i. For all the new nodes d' we set $X_{d'} = X_d$. An example of this transformation for $l = 3$ is shown on Fig. 5.7. It is clear that the axioms (D1) to (D3) do hold, and therefore we have a DAG decomposition.

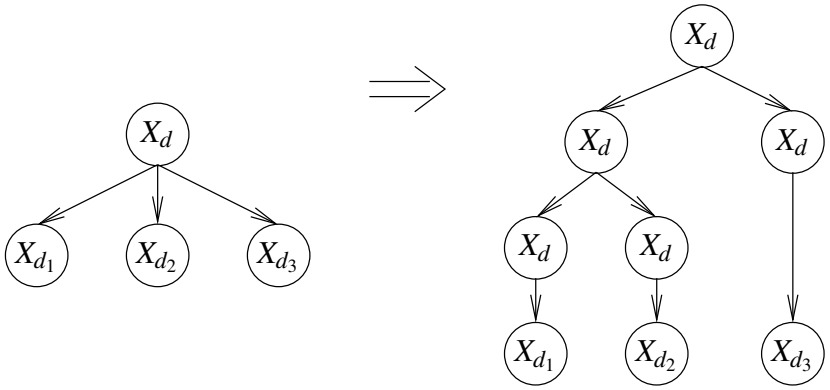


Figure 5.7: Reducing the number of successors to two

Finally we are going to apply Lemmas 5.4 and 5.5. Before we can do so, we need to make sure that for every node d with a single child d' we have either $X_d \subset X_{d'}$ or $X_{d'} \subset X_d$. First note that if $X_d = X_{d'}$, we can contract the edge (d,d'). In all other cases we substitute the edge (d,d') with two edges (d,d'') and (d'',d'), where d'' is a new node with $X_{d''} = X_d \cap X_{d'}$.

Now for every two nodes d, d' such that d' is a child of d and $|X_d| - |X_{d'}| = m > 1$ let $X_d = X_0 \supset X_1 \supset \ldots \supset X_m = X_{d'}$ be a decreasing sequence of sets. We replace the edge (d, d') with a sequence of edges $(d, d_1), (d_1, d_2), \ldots, (d_{m-1}, d')$. By iterative application of Lemma 5.4 the result is again a DAG decomposition of G. If d' is a child of d and $|X_{d'}| - |X_d| = m > 1$ we apply the dual construction, using the increasing sequence $X_d = X_0 \subset X_1 \subset \ldots \subset X_m = X_{d'}$ and Lemma 5.5.

By construction and Lemmas 5.2, 5.4 and 5.5 the decomposition we get is a nice DAG decomposition and we are finished.

For the bound on the size of the decomposition let us examine the number of nodes added in each step. In the first step we add at most one node. In step two the number of new nodes for a node with m successors is at most $2(m-1)$. As any node can have at most n successors, and there are $n+1$ nodes in total, the number of new nodes can be bounded by $2n^2$.

Finally for the step three we introduce at most $2k+1$ new nodes for each non-conforming edge. However all these edges must have been edges in the original decomposition $(D, \mathcal{X})$ and their number is thus bounded by n^2. Altogether we add at most $O(n^2)$ new nodes. □

In a nice DAG decomposition $(D, \mathcal{X})$, as in nice tree decomposition, every node is one of four possible types. These types are:

Start If a node is a leaf, it is called a *start node*.

Join If a node has two children, it is called a *join node*

Forget If a node d has one child d' and $X_d \subset X_{d'}$, the node d is called a *forget node*.

Introduce If a node d has one child d' and $X_d \supset X_{d'}$, the node d is called an *introduce node*.

(The *forget* and *introduce* names relate to the way algorithms on DAG decompositions work – from leaves towards the root.) Moreover, we may assume that *start* nodes contain only a single vertex. If this is not the case, we can transform every nice tree decomposition into one having this property by adding a chain of *introduce* nodes in place of non-conforming *start* nodes. Similarly we may assume that the (unique) root node also contains a single vertex.

5.5 Relationship to Other Measures

Here we present the relationship between DAG-width and both tree-width and directed tree-width. As one could expect, DAG-width falls in between the two measures.

Theorem 5.7. *Let G be a directed graph. Then $dgw(G) \leq tw(G)$. Moreover for each natural number $k > 0$ there is a graph G_k such that $tw(G_k) = k-1$ and $dgw(G_k) = 0$.*

Proof. A tree decomposition $(T, \mathcal{X})$ of a (directed) graph G can be easily turned into a DAG decomposition $(D, \mathcal{X})$ of G, where the DAG D is created from the tree T by selecting one node as a root and orienting all edges away from this root. To prove that $(D, \mathcal{X})$ is a DAG decomposition of D we show that (D1)-(D3) hold. (D1) and (D3) are easy to prove, since they are immediately implied by (T1) and (T3) for $(T, \mathcal{X})$. For (D2) assume that there is $(d, d') \in V(D)$ and $(v, w) \in E(G)$ violating this property. By (T2) there must be a node $c \in V(D)$ such that $\{v, w\} \subseteq X_d$. However by (T3) no node outside of $X_{\geq d'}$ contains v and similarly no node inside $X_{\geq} d'$ contains w, a contradiction. The width is clearly the same for both the decompositions.

For the second proposition it is enough to take for G_k the directed clique of size k, i.e. a graph created by taking an (undirected) clique of k vertices and orienting all edges so they form a DAG (with a single source and single sink). Fig. 5.8 is an example of such a directed clique for $k = 5$. Since cliques of size k have tree-width $k-1$ (see page 41) the result follows. □

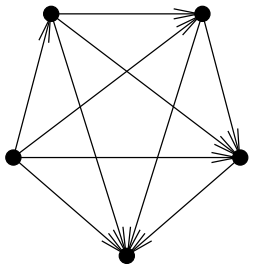


Figure 5.8: Directed 5-clique

The next lemma confirms that DAG-width is 'correct' counterpart of tree-width on directed graphs. (Directed tree-width also enjoys this property.)

Lemma 5.6. *Let G be an undirected graph of tree-width k. Then the directed graph G' created by replacing each edge of G with a pair of oppositely oriented edges has DAG-width k.*

Proof. By Theorem 5.7 we already know that $dgw(G') \leq tw(G)$. Because $tw(G) = k$ there must be a haven σ in G of size $k+1$. But this σ also satisfies the assumptions from Theorem 5.5 (i.e. is a haven) and therefore by Theorem 5.3 $dgw(G') > k$, which finishes the proof. □

The relationship to directed tree-width is a little bit more complicated. An obvious approach is to use the Theorem 5.2, which states that for each G and k either G has tree-width at most $3k-2$, or it has a haven of order k. It remains to observe that a haven of order k in a directed graph gives a winning strategy for a robber against $k-1$ cops in the (directed tree-width version of) cops-and-robber game, which easily translates to a winning strategy against the same number of cops in the DAG-width version of the game (the robber is actually more powerful here). Therefore directed tree-width and DAG-width are within a constant factor of each other. For an example of a graph G s.t. $dtw(G) < dgw(G)$ take the graph G in Fig. 5.9. It is easy to see that $dtw(G) = 1$ and $dgw(G) = 2$.

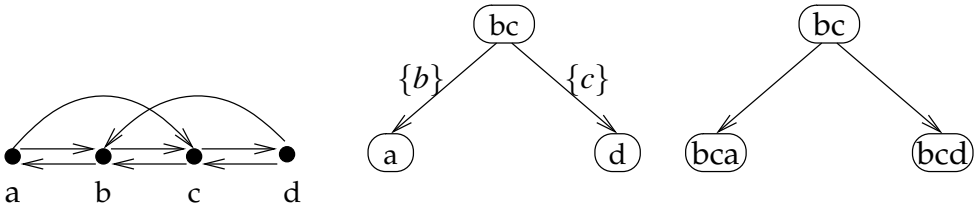


Figure 5.9: Graph G, its arboreal decomposition and DAG decomposition

The natural question is whether we can convert a DAG decomposition into an arboreal decomposition of the same width. We will show how to do that for a subclass of DAG decompositions.

Definition 5.4. A DAG decomposition $(D, \mathcal{X})$ of a graph G is called *simple* if, in addition to (D1) - (D3), it also satisfies the following:

(D4) $\forall v \in V(G)$ we have $|X_R(v)| = 1$ (i.e. each vertex of G has exactly one root node)

Lemma 5.7. *Let $(D,\mathcal{Y})$ be a simple DAG decomposition of graph G of width k in normal form and d_0 its root. Then $(R,\mathcal{X},\mathcal{W})$ computed by the Alg. 3 is an arboreal decomposition of G of the same width as $(D,\mathcal{Y})$.*

Proof. Note that this algorithm is a standard DFS topological sort. We write $d \prec d'$ for $d,d' \in V(R)$ iff $\gamma[d] < \gamma[d']$. Then $\prec$ is a linearisation of $<$ on D.

Algorithm 3: `DAGtoTree`

input : DAG decomposition $(D,\mathcal{Y})$
output: arboreal decomposition $(R,\mathcal{X},\mathcal{W})$
i:=0
$V(R)$:=$V(D)$; $E(R)$:=$\emptyset$
$\mathcal{X}$:=$\emptyset$; $\mathcal{W}$:=$\bigcup_{d\in V(R)} W_d = \emptyset$
W_{d_0}:=Y_{d_0}; `DFS` (d_0)

Procedure `DFS`(d)

for *each d' s.t. $e=(d,d') \in E(D)$* **do**
  if *have not seen d' before* **then**
    $W_{d'}$:=$Y_{d'} \setminus Y_d$
    $E(R)$:=$E(R) \cup \{e\}$
    X_e:=$Y_d \cap Y_{d'}$; $\mathcal{X}$:=$\mathcal{X} \cup \{X_e\}$
    `DFS` (d')
  γ[d]:=i++

To begin $\mathcal{W}$ is clearly a partition of $V(G)$ (the vertices left in each W_d are exactly those vertices of Y_d for which d is the root node). Moreover for $(d,d') \in E(R)$ the set $W_{d'}$ is empty only when $Y_{d'} \subseteq Y_d$, which is not possible since the DAG decomposition is in normal form. We will prove (R2) by induction on $\gamma[d]$. In the rest of the proof let $e_d \in E(R)$ be the only edge of R with head d. We claim that (R2) holds for e_d.

For d with $\gamma[d]=0$ we have that d is a leaf (i.e. vertex with no successors) in D. Then (R2) holds for e_d by construction and (D2). Similarly in the inductive step we are done if all the successors of d in D are also successors of d in R (using the induction hypothesis for these successors). For contradiction assume there is a cycle violating (R2). Therefore there must be some $v \in W_d$, $d' \in V(R)$ s.t. $(d,d') \in E(D), (d,d') \notin E(R)$, and $w \in X_{>d}$ s.t. this cycle starts with the edge (v,w).

Since the edge (d, d') was not included in $E(R)$, then d' must have been finished before d was ($d' \prec d$), and therefore $e_{d'}$ satisfies induction hypothesis. However by (D4) we must have $X_{e_{d'}} \subseteq X_e \cup W_d$ which completes the proof. Finally note that the width of $(R, \mathcal{X}, \mathcal{W})$ is at most k, since $W_d \cup \bigcup_{d \sim e} X_e \subseteq Y_d$. □

For general DAG decompositions we run into technical difficulties. It seems that the wording of (R1) is too restrictive - the main problem being the requirement for the sets W_d to be non-empty. However if we drop this requirement (which looks like the only way forward) the algorithm above can be easily modified to work on general DAG decompositions.

5.6 The Algorithm for Parity Games

In this section we are going to present a polynomial algorithm for solving parity games on graphs of bounded DAG-width. The author of this thesis came up with all the key ideas himself, but at GAMES'06 Berwanger et. al. [BDHK06] presented him with (already submitted) fully written proof of the same result, obtained independently. As author did not have a fully written proof of the result at that time, what is presented here is an adaptation of their (much nicer) proof which later appeared in [BDHK06].

The algorithm is, on a high level, similar to the one for graphs of bounded tree-width, as presented in Section 4.4. However we cannot just simply modify that algorithm to work on DAG decompositions instead of tree decompositions – there is an important conceptual difference. In the case of tree-decomposition, the set X_t for a node t acts as an interface between the vertices in the nodes below t and the rest of the graph. I.e. all paths leaving and entering $X_{>t}$ must pass through the set X_t. In the case of DAG decompositions it is only the paths leaving the set $X_{>d}$ which must pass trough X_d – a path coming from outside can enter $X_{>d}$ at any vertex of this set. Therefore if we have only used the algorithm for tree-width, we would have to consider borders of size $n.k$ (tables with n rows of 'enter' vertices and k columns of 'leave' vertices, cf. page 47) and the running time of such algorithm would be exponential, as the number of possible borders is $p^{n \cdot k}$, where p is the number of priorities. Therefore we must come up with a new solution.

In contrast to the algorithm for tree-width we will consider not just memo-

ryless strategies, but general strategies with access to history. I.e. a strategy of player P_0 (P_1) is a function $\sigma : V^*V_0 \to V$ ($\tau : V^*V_1 \to V$). It is interesting to note that even though we know that parity games are memorylessly determined, using the general strategies allows us to present a simpler algorithm.

For now fix a parity game $\mathcal{G}$ and let $Y \subseteq V(G)$ guarded by the set $S \subseteq V(G) \setminus Y$. Let us first consider the case where the strategies σ of P_0 and τ of P_1 are fixed. Let $\pi = \pi_0\pi_1 \ldots \pi_i \ldots$ be a play of the game $\mathcal{G}$ respecting the strategies σ and τ. Let $\pi[Y]$ be maximal prefix of π when restricted to vertices of Y. We define the result of $\pi[Y]$ to be

$$result^{\tau}_{\sigma}(v,Y) = \begin{cases} \bot & \text{if } \pi[Y] \text{ is infinite and winning for } P_1 \\ \top & \text{if } \pi[Y] \text{ is infinite and winning for } P_0 \\ (w,p) & \text{if } \pi[Y] = \pi_0,\ldots,\pi_j,\ w = \pi_{j+1} \text{ and } p = \max\{\lambda(\pi_i) \mid 0 \le i \le j+1\} \end{cases}$$

The next step is to fix a strategy σ of P_0 and try to find the best results player P_1 can achieve against this strategy. Two cases are simple. If there is a winning cycle of P_1 reachable from v in the subgame $G[Y_\sigma]$, then we know P_1 can win against the strategy σ if starting in v, both in the game restricted to Y_σ and in the whole game $\mathcal{G}_\sigma$. On the other hand if all paths in Y_σ starting in v lead to a winning cycle for P_0, then P_1 loses every play starting in v also in the game $\mathcal{G}_\sigma$.

The third possibility is that there is no winning cycle for P_1 in Y_σ, but P_1 can force the play into a vertex of S. Then the 'value' of such play π is the highest priority of a vertex on this path. However note that there can be more paths starting in v which lead to a given vertex $w \in S$. In that case it is in the player P_1's interest to choose the one with the lowest score w.r.t. the '$\sqsubseteq$' ordering. Moreover, there may be paths starting in v but leading to different vertices of the set S. Then we remember the best achievable result for each of these 'leave' vertices.

To formalise the description above, we need to extend the '$\sqsubseteq$' ordering to pairs (v,p). For two such pairs (v,p),(w,q) we put $(v,p) \sqsubseteq (w,q)$ iff $v = w$ and $p \sqsubseteq q$. Specifically if $v \neq w$ then the two pairs are incomparable. We extend the ordering $\sqsubseteq$ by adding the maximal element $\top$, and the minimal element $\bot$. For a set $X \subseteq (V \times \mathbb{N}) \cup \{\top, \bot\}$ we denote $\min_{\sqsubseteq} X$ to be the set of $\sqsubseteq$-minimal elements of X. Note that $\min_{\sqsubseteq} X = \{\bot\}$ iff $\bot \in X$ and $\min_{\sqsubseteq} X = \{\top\}$ iff $X = \{\top\}$. Moreover for $Z = \min_{\sqsubseteq} X$ if $Z \neq \{\bot\}$ and $Z \neq \{\top\}$, then Z contains at most one pair (v,p) for each vertex v.

With all the machinery in place we now define

$$result_\sigma(v,Y) = \min_{\sqsubseteq}\{(w,p) \mid \exists \tau \text{ s.t. } (w,p) = result^\tau_\sigma(v,Y)\}$$

For the rest of this section we fix a parity game $\mathcal{G}$ and its game graph G, together with a nice DAG decomposition $(D,\mathcal{X})$ of G of width k. For each node $d \in V(D)$ we have the set $X_{>d}$ which is guarded by X_d. What we want to know are the results for all possible strategies σ of P_0 and all vertices of $X_{>d}$. So we define the following structure:

$$Frontier(d) = \{(v,\alpha) \mid v \in X_{>d} \text{ and } \exists \sigma \text{ s.t. } \alpha = result_\sigma(v,X_{>d})\}$$

It is sometimes useful to split the set $Frontier(d)$ according to the first component of the pair (v,α). To this end we define

$$Frontier_v(d) = \{(v,\alpha) \mid (v,\alpha) \in Frontier(d)\}$$

Note that $Frontier(d) = \{Frontier_v(d) \mid v \in X_{>d}\}$.

Before we explain the algorithm, let us look at the sizes of the sets we have defined. First note that the set $Y = \min_\sqsubseteq X$ for $X \subseteq ([V]^{\leq d} \times \mathbb{N})$ can have at most k elements, each chosen from the set of available priorities, which can be bounded by n. Therefore $Frontier_v(d)$ can contain at most $(n+1)^k+2$ different elements. Finally $Frontier(d)$ can be of size at most $n.((n+1)^k+2)$.

We will use dynamic programming on D to compute the set $Frontier(d)$ for each node $d \in V(D)$ from the frontiers of its successors. There are four cases, depending on the type of the node in nice DAG decomposition.

Start node We put $Frontier(d) = \emptyset$, because for each start node d we have $X_{>d} = \emptyset$,

Join node Let d be a join node with two children d_1 and d_2. We are going to prove that $Frontier(d) = Frontier(d_1) \cup Frontier(d_2)$.

First let $(v,\alpha) \in Frontier(d_1) \cup Frontier(d_2)$. If $v \in X_{>d_1} \setminus X_{>d_2}$, then clearly $(v,\alpha) \in Frontier(d)$ as $X_d = X_{d_1}$ by definition of join node. The case $v \in X_{>d_2} \setminus X_{>d_1}$ is symmetrical. The last case is $v \in X_{>d_1} \cap X_{>d_2}$. Since $X_{d_1} = X_{d_2}$, by the axiom (D2) we get $(v,\alpha) \in Frontier(d_1) \iff (v,\alpha) \in Frontier(d_2)$. Therefore also $(v,\alpha) \in Frontier(d)$.

For $(v,\alpha) \in Frontier(d)$ we can use similar reasoning to show that $(v,\alpha) \in Frontier(d_1)$ or $(v,\alpha) \in Frontier(d_2)$.

Introduce node Let d be an introduce node with a child d', and $X_d = X_{d'} \cup \{v\}$. By definition of DAG decompositions $X_{>d} = X_{>d'}$. Moreover all paths from a vertex of $X_{d'}$ to v must pass through a vertex of $X_{d'}$ and $X_{d'} \subset X_d$. Therefore $\mathit{Frontier}(d) = \mathit{Frontier}(d')$.

Forget node This is the most difficult case. Let d be a forget node with a single child d' and let $X_d = X_{d'} \setminus \{v\}$. We are now going to remove the vertex v from the set of separating vertices. To get the set $\mathit{Frontier}(d)$, we need to first compute its subset, the set $\mathit{Frontier}_v(d)$. Assuming we have already computed this set, it is comparatively easy to compute $\mathit{Frontier}_w(d)$ for the remaining vertices $w \in X_{>d}$.

To do so, we start with an element $(w,\alpha) \in \mathit{Frontier}_w(d')$. If α contains a pair (v,p) for some p we need to consider all possible ways of extending the path from w to v with an element of $\mathit{Frontier}_v(d)$. The way of combining the two frontiers is formally defined by the operator *weld*:

Definition 5.5 (*weld*)**.** Let $(w,\alpha) \in \mathit{Frontier}(d')$, $(v,\beta) \in \mathit{Frontier}(d)$, and $w \neq v$. Then

$$\mathit{weld}(\alpha,v,\beta) = \begin{cases} \min_{\sqsubseteq}((\alpha \cup \beta_p) \setminus \{(v,p)\}) & \text{if } (v,p) \in \alpha \text{ for some } p \\ \alpha & \text{otherwise} \end{cases}$$

where $\beta_p = \{(u, \max\{p,q\}) \mid (u,q) \in \beta\}$.

Lemma 5.8. $\mathit{Frontier}_w(d) = \{\mathit{weld}(\alpha,v,\beta) \mid \alpha \in \mathit{Frontier}_w(d'), \beta \in \mathit{Frontier}_v(d)\}$

Proof. If $(w,\gamma) \in \mathit{Frontier}_w(d)$, then $\gamma = \mathit{result}_\sigma(w, X_{>d})$ for some σ. It is not hard to see $\gamma = \mathit{weld}(\alpha,v,\beta)$, where $\alpha = \mathit{result}_\sigma(w, X_{>d'})$ and $\beta = \mathit{result}_\sigma(v, X_{>d})$.

For the other direction let $\alpha = \mathit{result}_{\sigma_1}(w, X_{>d'})$ and $\beta = \mathit{result}_{\sigma_2}(v, X_{>d})$. There are two cases to consider. First if no pair of the form (v,p) is included in α, then the player P_1 cannot, starting from w, reach v in $X_{>d}$ when P_0 is using the strategy σ_1. Therefore $\mathit{weld}(\alpha,v,\sigma) = \alpha = \mathit{result}_{\sigma_1}(w, X_{>d})$. For the other case ($\alpha$ contains a pair (v,p)) take the strategy σ which behaves like σ_1 till a play reaches v and then behaves like σ_2. Clearly $\mathit{weld}(\alpha,v,\beta) = \mathit{result}_\sigma(w, X_{>d})$, as we consider all possible extensions of the 'best' path leading to v (the set β_p), and then take the minimal elements from $\beta_p \cup \alpha$ minus the pair (v,p) (as $v \notin X_d$). □

We are now going to show how to compute the set $\mathit{Frontier}_v(d)$ in the first place. Take all successors of the vertex v. Let $u_1, \ldots, u_k$ be the successors which

are in the set X_d, and $v_1, \ldots, v_l$ the successors in the set $X_{>d}$. (Note that by (D2) all successors of v must be in the set $X_{\geq d} = X_d \cup X_{>d}$.)

Let us compute the effect of choosing each of the successors of v. The simpler case is for the vertices $u_1, \ldots, u_k$. Here the game starts in v and in one step moves to u_i, where it stops since $u_i \in X_d$. Let $mod(u_i) = (u_i, \max(\lambda(u_i), \lambda(v))$. Then we put $\overline{M_0} = \{mod(u_i) \mid 1 \leq i \leq k\}$ and $M_0 = \{\{\alpha\} \mid \alpha \in \overline{M_0}\}$. In other words, $\overline{M_0}$ contains the results of each two-vertex path (v, u_i), and M_0 contains these results as one element sets (having both the sets $\overline{M_0}$ and M_0 simplifies the proof a bit).

On the other hand, if the game starting in v moves in one step to some v_i, we must consider all possible results for a game starting in v_i and not leaving $X_{>d'}$ – i.e. the elements of $Frontier_{v_i}(d')$. For an $\alpha \in Frontier_{v_i}(d')$ we will construct α' as follows: First we detect the cycles in $X_{>d}$ going through v: If there is a pair $(v, p) \in \alpha$, we put $\bot$ into α' if $\max(p, \lambda(v))$ is odd, and $\top$ if it is even. Then for each other pair $(w, p) \in \alpha$ (with $w \neq v$) we insert the pair $(w, \max(p, \lambda(v))$. We then choose the minimal elements of α', the set $\min_{\sqsubseteq} \alpha'$, and call the result $mod(\alpha)$ (here we overload the notation a bit). For all possible choices of $\alpha \in Frontier_{v_i}(d')$ this gives us the set

$$M_i = \{mod(\alpha) \mid \alpha \in Frontier_{v_i}(d')\}$$

We now split our analysis into two cases, depending whether $v \in V_0$, or $v \in V_1$. We start with the simpler case, which is $v \in V_0$.

Lemma 5.9. *If $v \in V_0$, then $Frontier_v(d) = \bigcup_{i=0}^{l} M_i$.*

Proof. Clearly the strategy σ of player P_0 at vertex v can choose any of the successors $u_1, \ldots, u_k, v_1, \ldots, v_l$. If $\sigma(v) = u_i$ then $mod(u_i) = result_\sigma(v, X_{>d})$ belongs to $Frontier_v(d)$. If $\sigma(v) = v_i$ then let $\alpha = result_\sigma(v_i, X_{>d'}) \in Frontier_{v_i}(d')$. Then clearly $mod(\alpha) \in Frontier_v(d)$.

Now to the other direction. The case $\alpha \in M_0$ is clear. So let $\alpha \in M_i$ for some $1 \leq i \leq l$. Then $\alpha = mod(\beta)$, where $\beta = result_\sigma(v_i, X_{>d'})$ for some strategy σ of P_0. Let $\sigma' = \sigma[v \to v_i]$ (the strategy which behaves as σ on all elements except v, where $\sigma'(v) = v_i$). Then $mod(\beta) = result_{\sigma'}(v, X_{>d})$. □

The rest of the proof will be concerned with the case $v \in V_1$.

Lemma 5.10. *If $v \in V_1$, then $\alpha \in Frontier_v(d)$ iff there are $\alpha_1, \ldots, \alpha_l$, with $\alpha_i \in M_i$, such that $\alpha = \min_{\sqsubseteq}(\overline{M_0} \cup \bigcup_{i=1}^{l} \alpha_i)$.*

Proof. Let $\alpha \in Frontier_v(d)$. Then $\alpha = result_\sigma(v, X_{>d})$ for some σ. We put $\alpha_i = result_\sigma(v_i, X_{>d'})$. Then clearly $\alpha = \min_s qe(\overline{M_0} \cup \bigcup_{i=1}^{l} \alpha_i)$, as the player P_1 can freely choose among the vertices $u_1, \ldots, u_k$ (the relevant results are in $\overline{M_0}$), and $v_1, \ldots, v_l$ (the relevant results are $\alpha_1, \ldots, \alpha_l$).

For the other direction each $\alpha_i = mod(result_{\sigma_i}(v, X_{>d'}))$. Then for strategy σ which behaves like σ_i on all paths starting v, v_i. Clearly $\alpha = result_\sigma(v, X_{>d}) \in Frontier_v(d)$. □

However we cannot apply the Lemma 5.10 directly by trying all combinations of α_i's. The reason is that there are $((n+1)^k+2)^l$ many possible combinations and our algorithm would become exponential (as any vertex can have at most n successors). Note however that the size of $Frontier_v(d)$ is bounded by $(n+1)^k+2$. If we could check in polynomial time for each potential pair (w, p) whether it does belong to $Frontier_v(d)$, then we could compute the set $Frontier_v(d)$ also in polynomial time. Fortunately there is a way to do exactly that.

The trick is in noticing that if $\alpha \in Frontier_v(d)$ has m elements, then, since each of them has to come from some α_i, there are at most m such indices i such that α_i contributes to α. This is formalised by the following lemma (cf. Lemma 5.10):

Lemma 5.11. *If $v \in V_1$, then $\alpha \in Frontier_v(d)$ iff there is a set $I \subseteq 1, 2, \ldots, l$ and $\alpha_1, \ldots, \alpha_l$, with $\alpha_i \in M_i$, such that*

1. $\alpha = \min \sqsubseteq (\overline{M_0} \cup \bigcup_{i \in I} \alpha_i)$, *and*
2. *for each $i \notin I$ we have $\alpha \sqsubseteq \alpha_i$.*

Proof. The forward direction is simple. Take the α_i's as defined in the proof of Lemma 5.10. As we mentioned earlier, at most $|\alpha|$ of them contribute to the set α. Put indices of these contributing α_i's into the set I. The remaining α_i's now serve to satisfy the condition (ii). In the other direction take the α_i's from both (i) and (ii). Then they clearly satisfy the conditions of Lemma 5.10. □

Now since $|\alpha| \leq k$ we can check all possible subsets of $\{1, 2, \ldots, l\}$ of size $\leq |\alpha|$ as candidates for I – there are at most n^k such subsets. As size of the sets $|M_i|$ is bounded by $(n+1)^k+2$, for each such I there are at most $((n+1)^k+2)^{|I|}$ combinations of α_i's to be tried. For each such combination we must check that

every set M_i, $i \notin I$, contains an element α_i such that $\alpha \sqsubseteq \alpha_i$. This can be done in time $n.((n+1)^k + 2)$. Altogether the time needed for each forget node is in $O(n^{k^2})$.

Theorem 5.8. *Let $\mathcal{G}$ be a parity game, $n = V(G)$, and $(D, \mathcal{X})$ a DAG decomposition of its game graph G of width k. Then there is an algorithm which solves the parity game $\mathcal{G}$ in time polynomial in n.*

Proof. We start by converting the DAG decomposition $(D, \mathcal{X})$ into a nice DAG decomposition $(D', \mathcal{X}')$. By Theorem 5.6 this can be done in time $O(m^2)$, where m is the number of nodes of D. This number can in turn be bounded by n^k by Theorem 5.4. Therefore D' has at most n^{2k} nodes. Therefore we can compute the set $\mathit{Frontier}(d)$ for each node in D' in polynomial time as was shown above. Now D' must have a unique root d_0. As this root is not guarded, for each vertex $v \in V(G)$ we must have $\mathit{Frontier}_v(d_0)$ equal either to $\bot$ or to $\top$. By definition of the set $\mathit{Frontier}_v(d_0)$ we get that P_0 wins the parity game $\mathcal{G}(v)$ iff $\mathit{Frontier}_v(d_0) = \top$. Note that this gives us the answer for all possible choices of the starting vertex, as $\mathit{Frontier}_v(d_0)$ is defined for each vertex $v \in V(G)$ (the reason being $X_{\geq d_0} = V$). □

Chapter 6

Strategy Improvement

The new results in this chapter are joint work with Colin Stirling.

Among the different algorithms for parity games, the strategy improvement algorithm of Vöge and Jurdziński [VJ00] is somewhat special. There are several reasons for this. For example there is no known example of a parity game where this algorithm needs more than a linear number of stages (each running in time cubic in the number of vertices). This is not the case with most of the other known algorithms, for which we have examples of parity games where the respective algorithms need an exponential number of steps. Secondly, it can be proved that at each stage there is a choice available which guarantees the strategy improvement algorithm to finish by going through a linear number of stages.

The strategy improvement algorithm of [VJ00] is based on the algorithm of Puri [Pur95] for discounted payoff games, which in turn is an adaptation of a well known strategy improvement algorithm for stochastic games of Hoffman and Karp [HK66]. The main contribution of Vöge and Jurdziński is that their algorithm is discrete and using graph theoretical arguments, whereas the algorithms of [Pur95] and [HK66] are based on continuous methods, and involve manipulating real numbers and solving non-trivial linear programming instances.

The first part of this chapter is a self-contained description of the discrete strategy improvement algorithm. Nevertheless we omit most of the proofs – the interested reader can find them in the original paper [VJ00]. The reader should note that the description of the algorithm and its notation sometime differs from the one provided in [VJ00]. The reason is that we want to give

here a simple and readable account, as the original paper is considered quite technical (which is necessitated by the need to present all the proofs).

In the second part of the chapter we analyse the strategy improvement algorithm. We start with a section which examines the structure of strategy space. Here we contribute several new results. Next we shift our interest to the choice of improvement policy and discuss some particular policies in more detail. We conclude the chapter by presenting an example of a parity game which has an improvement policy of exponential length. Note that the policy used is not the maximal policy given in [VJ00]. This example has been found by Serre [Ser], but it has not been published before.

In this chapter we assume that parity games are in normal form (Definition 2.9) – i.e. we consider games with a maximum number of priorities where each player owns vertices of 'her' priority. (This allows us to identify the game $\mathcal{G}$ with its game graph G.) Also one of the issues in the analysis of the strategy improvement algorithm is that there may be more than one improvement on a given vertex available at any time. This can happen whenever the number of successors is greater than two. In this chapter we therefore restrict the graphs of parity games to at most two outgoing edges per vertex (which is possible by Lemma 2.1 with at most quadratic growth in the number of states).

6.1 Discrete Strategy Improvement Algorithm

The algorithm is modelled after an optimisation problem. Assume we have a pre-order $\sqsubseteq$[1] on Σ_0, the set of strategies of player P_0, satisfying the following two axioms:

P1. There is a maximal element in the pre-order $(\Sigma_0, \sqsubseteq)$, i.e. there is a strategy $\kappa \in \Sigma_0$ such that $\sigma \sqsubseteq \kappa$ for all $\sigma \in \Sigma_0$.

P2. If κ is a maximal element in the pre-order $(\Sigma_0, \sqsubseteq)$, then κ is a winning strategy of P_0 for all elements in W_0.

Assume we also have a function $\mathit{Improve} : \Sigma_0 \to \Sigma_0$ satisfying the following two axioms:

[1] A word of warning: In this chapter we overload the relation symbol $\sqsubseteq$ a lot, but the meaning should always be clear from the context. We think this is better than introducing several different symbols for almost the same relation based on the 'reward' ordering introduced in Section 2.1.1.

I1. If σ is not a maximal element in the pre-order $(\Sigma_0, \sqsubseteq)$, then $\sigma \sqsubset \mathit{Improve}(\sigma)$.

I2. If κ is a maximal element in the pre-order $(\Sigma_0, \sqsubseteq)$, then $\mathit{Improve}(\kappa) = \kappa$.

A generic strategy improvement algorithm is then given by the following procedure:

Algorithm 5: Strategy Improvement Algorithm

choose $\sigma \in \Sigma_0$ at random
while $\sigma \neq \mathit{Improve}(\sigma)$ **do**
  σ:=$\mathit{Improve}(\sigma)$

Termination is guaranteed by I1. and the fact that there are only finitely many strategies. I2. then guarantees that when the algorithm stops, we have a maximal element. Altogether we have the following:

Theorem 6.1. *If pre-order $(\sqsubseteq, \Sigma_0)$ satisfies P1. and P2., and the Improve operator satisfies I1. and I2., then the strategy improvement algorithm stops and returns a winning strategy for the player P_0 from each of the vertices in W_0.*

The problem is to come up with a good definition of the pre-order $\sqsubseteq$ and of the operator *Improve*.

6.2 Ordering on Strategies

In this section we show how to define a pre-order on strategies of one player, in this case the player P_0. The pre-order on strategies we are going to present is induced by a measure for vertices under a strategy σ. We therefore start by defining this measure.

The intuition behind the definition of measure which is given below is this: Player P_0 fixes a strategy σ. Now it is player P_1 who wants to reply to σ in the best way possible. We define the value for a vertex $v \in V$ and a strategy $\sigma \in \Sigma_0$ to be the outcome for P_1 if he plays optimally against σ.

Of course the best thing from the point of view of P_1 is to reach an odd cycle in G_σ. If this is not possible (all the cycles reachable from that initial vertex are even), then a cycle with the smallest winning priority. To describe the highest priority of the winning cycle we define the notion of *pivot*.

Definition 6.1. We say that $p \in V$ is the *pivot* of $v \in V$ under strategy σ if $p = \min_{\sqsubseteq}\{w \in V \mid \exists w' \in R_\sigma(w, V^{\leq w})$ s.t. $(w,w') \in E_\sigma$ and $v \in R_\sigma(w)\}$. We also define function $\Lambda_\sigma : V \to V$ such that $\Lambda_\sigma(v) = p$ if p is the pivot of v. Finally we put $Pivots(\sigma) = \{p \mid \exists v \in V.\Lambda_\sigma(v) = p\}$ to be the set of all pivots in the game G, and $V_\sigma^p = \{v \in V \mid \Lambda_\sigma(v) = p\}$, the set of all vertices with pivot p.

Lemma 6.1. *For a strategy σ the set $\mathcal{V}_\sigma = \bigcup_{p \in Pivots(\sigma)} V_\sigma^p$ forms a partition of V such that for every $p \in Pivots(\sigma)$*

1. $V_\sigma^p = R_\sigma(p, V_\sigma^p)$
2. *$G[V_\sigma^p]$ is a subgame of G*
3. *P_i wins $G[V_\sigma^p]$ iff $p \in V_i$*

Proof. We first show that the lemma holds for $p = \min_{\sqsubseteq} Pivots(\sigma)$. Then clearly $V_\sigma^p = R_\sigma(p) = R_\sigma(p, V_\sigma^p)$, which proves 1. By definition of force set and the fact that there is $(v,w) \in E_\sigma$ such that $w \in R_\sigma(p, V_\sigma^p)$ by definition of p we have also 2. The last claim is obvious, as p is the highest vertex in V_σ^p and there is a loop on p.

Now put $W = V \setminus V_\sigma^p$. By definition of force set and by 1. there is no vertex in W with all successors in V_σ^p, and therefore $G_\sigma[W]$ is a subgame of G_σ. Also, since we have taken p to be the minimal element (w.r.t. $\sqsubseteq$) of $Pivots(\sigma)$, we also have $V_\sigma^p \cap Pivots(\sigma) = \{p\}$. Therefore we can take $p' = \min_{\sqsubseteq} Pivots(\sigma) \setminus \{p\}$ and repeat our argument. □

To rephrase the previous lemma, in the subgame V_σ^p the player P_1 can force the play to a cycle with the priority p and stay in such a cycle. Now we need to define the ordering inside each set V_σ^p. A key ingredient here is a priority profile, a set of vertices $B \subseteq \{v \in V \mid v > p\}$ such that player P_1 can force play from v to p and pass through all elements of B and avoid all elements in $\{w \in V \mid w > p\} - B$. Another ingredient is the distance between v and p.

Definition 6.2. A path $v_1 \to \ldots \to v_k$ is *simple* if all v_i are pairwise distinct. For a simple path $\pi = v_1 \to \ldots \to v_m$ and $p \in \mathbb{N}$ let $B(\pi, p) = \{v_1, \ldots, v_m\} \cap \{v \mid v > p\}$ be its *priority profile* and $d(\pi) = m$ its *distance*.

Remember that we have defined the order '$\sqsubseteq$' on sets in Section 2.1.1. Using this order we can extend $\sqsubseteq$ to pairs (B,d).

Definition 6.3. The *value ordering*, $\sqsubset$, is defined as follows: $(B,d) \sqsubset (B',d')$ if $B \sqsubset B'$, or $B = B'$ and $d < d'$. We also write $(B,d) \sqsubseteq (B',d')$ if $(B,d) \sqsubset (B',d')$, or $B = B'$ and $d = d'$.

We can now define the measure $\Pi_\sigma(v)$ for vertices v that belong to the same set V_σ^p:

Definition 6.4. If $\Lambda_\sigma(v) = p$ then

$$\Pi_\sigma(v) = \min_{\sqsubseteq}\{(B(\pi,p), d(\pi)) \mid \pi \text{ is a simple path } v \to^* p \in V_\sigma^p\}$$

If $\Pi_\sigma(v) = (B,d)$, then the projections $\Pi_\sigma^1(v) = B$ and $\Pi_\sigma^2(v) = d$. We call $\Pi_\sigma(v)$ the *value* of vertex v for the strategy σ.

For instance, for the pivot p it follows that $\Pi_\sigma(p) = (\emptyset, 0)$. Combining the value inside the set V_σ^p and the pivot p, we get Ω_σ.

Definition 6.5. For $v \in V$ and $\sigma \in \Sigma_0$ we define

$$\Omega_\sigma(v) = (\Lambda_\sigma(v), \Pi_\sigma(v))$$

and call Ω_σ the *measure* of strategy σ and $\Omega_\sigma(v)$ the *value* of v under the strategy σ.

We also extend the ordering $\sqsubseteq$ to pairs $(p,(B,d))$ in the obvious way:

Definition 6.6. The *value ordering*, $\sqsubseteq$, is defined as follows: $(p,(B,d)) \sqsubset (p',(B',d'))$ if $p \sqsubset p'$, or $p = p'$ and $(B,d) \sqsubset (B',d')$. We put $(p,(B,d)) \sqsubseteq (p',(B',d'))$ if $(p,(B,d)) \sqsubset (p',(B',d'))$, or $p = p', B = B'$ and $d = d'$.

Having defined a value for each vertex, we can finally define the pre-order on strategies by comparing the values for each vertex point-wise:

Definition 6.7. For $\sigma, \sigma' \in \Sigma_0$ we write $\sigma \sqsubseteq \sigma'$ if $\forall v \in V. \Omega_\sigma(v) \sqsubseteq \Omega_{\sigma'}(v)$. We also write $\sigma \sqsubset \sigma'$ if $\sigma \sqsubseteq \sigma'$ and $\sigma \neq \sigma'$.

6.2.1 Optimal Counter-strategy, Value Tree

Let us look at the definition of Π_σ in more detail. We are interested in the structure of $G_\sigma[V_\sigma^p]$.

Definition 6.8. Let $\sigma \in \Sigma_0$ and $p \in \mathit{Pivots}(\sigma)$. We construct the strategy $\widehat{\sigma} \in \Sigma_1$ as follows: Let $v \in V_\sigma^p$. If v has only one successor w in the set V_σ^p, then $\widehat{\sigma}(v) = w$. If v has two successors w_1, w_2 in the set V_σ^p, we put $\widehat{\sigma}(v) = w_1$ if $\Pi_\sigma(w_1) \sqsubseteq \Pi_\sigma(w_2)$, and $\widehat{\sigma}(v) = w_2$ otherwise. We call such $\widehat{\sigma}$ the *optimal counter-strategy* of P_1 for the strategy σ.

Definition 6.9. Let $\sigma \in \Sigma_0$ and $p \in \mathit{Pivots}(\sigma)$. Then we define $T_\sigma^p = G_\sigma^{\widehat{\sigma}}[V_\sigma^p]$ and call T_σ^p the *value tree* of σ and p.

If we look at the graph T_σ^p for $p \in \mathit{Pivots}(\sigma)$ then by definition of Π_σ and the best counter-strategy $\widehat{\sigma}$ it must be a tree (if we ignore the edge with tail p) with the root p and all edges oriented towards the root. Moreover take an edge $(v,w) \in E(T_\sigma^p)$. Then

$$\Pi_\sigma(v) = \begin{cases} (\Pi_\sigma^1(w) \cup \{v\}, \Pi_\sigma^2(w) + 1) & \text{if } v > p \\ (\Pi_\sigma^1(w), \Pi_\sigma^2(w) + 1) & \text{otherwise} \end{cases}$$

Note that the above implies for each $v \in V_\sigma^p$ that $\Pi_\sigma(v) = (B(\pi, p), d(\pi))$, where $\pi : v \to^+ p$ is the unique path from v to p in the value tree T_σ^p. To sum up, the value tree of σ and p is a structural representation of the measure Π_σ, and will be useful later when discussing the different types of switches. We will also use the relation $\prec_\sigma$.

Definition 6.10. Let $\sigma \in \Sigma_0$ and $u, v \in V$. Then $v \prec_\sigma u$ if $\Lambda_\sigma(u) = \Lambda_\sigma(v) = p$ and there is a path $u \to^+ v$ in T_σ^p. In that case we say that u is *above* v in T_σ^p.

An example of a value tree can be seen is in Fig. 6.1. Dotted lines show those edges of G which are not in σ or $\widehat{\sigma}$, and thick lines show the strategy σ.

6.2.2 Short Priority Profiles

In the definitions above we considered the priority profile to be a set of vertices on a simple path. Here we give an alternative definition of priority profile, equivalent to the original one. One advantage of this definition is that the new priority profiles are usually of at most the same length (and usually shorter) as they provide only the information we need to keep the vertices in the same order.

Definition 6.11. For a simple path $\pi = u_1 \to \ldots \to u_{m-1} \to u_m$ let $C(\pi, p) = \{u_{i_1}, \ldots, u_{i_k}\}$ be the maximal subset of $\{u_1, \ldots, u_m\} \cap \{v \mid v > p\}$ such that

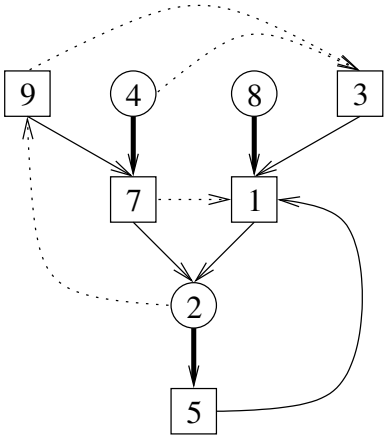


Figure 6.1: Value tree T_σ^5

- $1 \leq i_1 < \ldots < i_k \leq m$, and
- $\forall j : 1 \leq j < k . \forall n : 1 \leq n < i_{j+1} . u_n \leq u_{i_j}$.
- $\forall n : 1 \leq n \leq m . u_n \leq u_{i_k}$

Then we call $C(\pi, p)$ the *short priority profile* of π.

Note that the definition above implies we can create a simple priority profile $C(\pi, p)$ by following the path π from u_1 to u_m and adding to $C(\pi, p)$ each vertex greater than p which is also greater then any of the vertices already in the set.

Fact 6.1. *Let π be a simple path, $B(\pi, p)$ its priority profile and $C(\pi, p)$ its short priority profile. Then $C(\pi, p) \subseteq B(\pi, p)$.*

The next lemma implies that we can replace priority profiles with short priority profiles in the definition of Π_σ. From now on let $W \div W'$ abbreviate $(W \setminus W') \cup (W' \setminus W)$, the symmetric difference of W and W'.

Lemma 6.2. *Let $\sigma \in \Sigma_0$, $p \in Pivots(\sigma)$ and $v, w \in V_\sigma^p$. Let $\pi : v \rightarrow^+ p$ and $\pi' : w \rightarrow^+ p$ be the unique paths from v and w to p in the value tree T_σ^p. Then*

$$B(\pi, p) \sqsubseteq B(\pi', p) \iff C(\pi, p) \sqsubseteq C(\pi', p)$$

Proof. Obviously if $B(\pi', p) = B(\pi', p)$ then also $C(\pi, p) = C(\pi', p)$. On the other hand let $u = \min C(\pi, p)$. Since we have a game with maximum number of priorities, the vertex u is common to both π and π' (as well as the suffixes of π and π' from u onwards). Also (by definition of $C(\pi, p)$) all vertices in the

prefixes of π and π' before the vertex u must be smaller than p, which gives us $B(\pi, p) = B(\pi', p)$.

So let $B(\pi, p) \sqsubset B(\pi', p)$ and be u the highest separator $u = \max(B(\pi, p) \div B(\pi', p))$. By definition $D = B(\pi, p) \cap \{v \in V \mid v > u\} = B(\pi', p) \cap \{v \in V \mid v > u\}$ and therefore all vertices in D must lie in the common suffix of π and π'. As u is the next highest vertex and lies in only one of the two paths π and π', we get $C(\pi, p) \sqsubset C(\pi', p)$. The opposite direction is even simpler. □

6.3 Operator *Improve*, Switching

In this section we deal with switching and define the operator *Improve*. Remember we consider only games where every vertex has at most two successors. We can therefore introduce the following notation:

Definition 6.12. Let σ be a strategy of P_0 and $v \in V$. Then we use $\overline{\sigma}(v)$ to denote the successor of v which is not $\sigma(v)$. Moreover $\sigma[v]$ is the strategy defined as

$$\sigma[v](u) = \begin{cases} \overline{\sigma}(u) & \text{iff } u = v \\ \sigma(u) & \text{otherwise} \end{cases}$$

By a *switch* of σ on a vertex $v \in V_0$ we mean the change of strategy from σ to $\sigma[v]$. This notation extends naturally to sets $X \subseteq V_0$ (i.e. $\sigma[X]$). Similarly by a *switch* on $X \subseteq V_0$ we mean the change of strategy from σ to $\sigma[X]$.

Now we can finally define the *Improve* operator. We allow P_0 to change his strategy σ for all vertices such that $\Omega_\sigma(\sigma(x)) \sqsubset \Omega_\sigma(\overline{\sigma}(x))$.

Definition 6.13. Let σ be a strategy. We define a set of *enabled* vertices as

$$\mathit{Enabled}(\sigma) = \{v \in V \mid \Omega_\sigma(\sigma(x)) \sqsubset \Omega_\sigma(\overline{\sigma}(x))\}$$

The operator *Improve* is then defined as

$$\mathit{Improve}(\sigma) = \sigma[X] \text{ where } X \subseteq \mathit{Enabled}(\sigma)$$

Note that in [VJ00] the authors suggest using $\mathit{Improve}(\sigma) = \sigma[X]$ where $X = \mathit{Enabled}(\sigma)$ (i.e. take all possible improvements).

The following theorems shows that the operator *Improve* is well defined. We do not give the proof here and refer the reader to [VJ00] for details. Nevertheless we would like to mention that the proof is easy to come up with for the case that $|X| = 1$.

Theorem 6.2 ([VJ00]). *Let* $\sigma \in \Sigma_0$ *and* $X \subseteq \mathit{Enabled}(\sigma)$. *Then* $\sigma \sqsubseteq \sigma[X]$.

It is also possible to show the converse of the previous theorem, i.e. if none of the switches is enabled, then we do not improve the strategy σ.

Theorem 6.3 ([VJ00]). *Let* $\sigma \in \Sigma_0$ *and* $X \subseteq V$ *such that* $X \cap \mathit{Enabled}(\sigma) = \emptyset$. *Then* $\sigma[X] \sqsubseteq \sigma$.

This theorem has also an important consequence.

Corollary 6.1 (maximal strategy). *There is a strategy* $\kappa \in \Sigma_0$ *which is the maximal element in the pre-order* $(\sqsubseteq, \Sigma_0)$, *i.e.* $\sigma \sqsubseteq \kappa$ *holds for each* $\sigma \in \Sigma_0$.

We will use κ to denote this maximal strategy in this chapter. As we can see, the definition of *Improve* gives us a free hand in choosing the subset X of $\mathit{Enabled}(\sigma)$. In general *strategy improvement policy* is a function $P : \Sigma_0 \to 2^V$ such that for each $\sigma \in \Sigma_0$ we have $P(\sigma) \subseteq \mathit{Enabled}(\sigma)$. In the concrete case of fixed $\sigma \in \Sigma_0$, a *strategy improvement policy for* σ is a sequence $X_1, \ldots, X_k$ of subsets of V such that if we put $\sigma_0 = \sigma$ and define

$$\sigma_{i+1} = \begin{cases} \sigma_i[X_{i+1}] & \text{if } i < k \\ \sigma_i & \text{if } i \geq k \end{cases}$$

then $X_i \subseteq \mathit{Enabled}(\sigma_i)$ and for σ_{k+1} we have $\mathit{Enabled}(\sigma_{k+1}) = \emptyset$. Finally for a policy P we call the sequence $\sigma = \sigma_0\sigma_1 \ldots \sigma_k$ where $P(\sigma_i) = \sigma_{i+1}$ the *improvement sequence*.

The proof of the following theorem can be found in [VJ00].

Theorem 6.4. *The pre-order* $\sqsubseteq$ *on strategies given by Definition 6.7 satisfies the properties P1 and P2 and the operator Improve given by Definition 6.13 satisfies the properties I1 and I2 from Section 6.1.*

6.3.1 Different Types of Switches

If we look at the definition of Ω_σ, we can distinguish three different kinds of switches. Let $\sigma \in \Sigma_0, v \in \mathit{Enabled}(\sigma)$, and $\Omega_\sigma(\sigma(v)) = (p, (B, d))$ and $\Omega_\sigma(\overline{\sigma}(v)) = (p', (B', d'))$. Then the switch on v is either

pivot switch if $p \sqsubset p'$,

priority switch if $p = p'$ and $B \sqsubset B'$, or

distance switch if $p = p'$, $B = B'$ and $d < d'$.

We are now going to examine the different types of switches in more detail. Before we do so we will need a variant of the force set:

Definition 6.14. Let $\sigma \in \Sigma_0$, $W \subseteq V$ and $w \in W$. We define $F_\sigma(w,W)$, the *force set* of w under a strategy σ with respect to W, as a fixed point of the following:

$$\begin{aligned} F^0_\sigma(w,W) &= \{w\} \\ F^{k+1}_\sigma(w,W) &= F^k_\sigma(w,W) \cup \\ &\quad \{u \in V_0 \cap W \mid \exists v \in F^k_\sigma(v,W) \text{ s.t. } (u,v) \in E \text{ and } \sigma(u) = v\} \cup \\ &\quad \{u \in V_1 \cap W \mid \forall v \in W.(u,v) \in E \implies v \in F^k_\sigma(v,W)\} \end{aligned}$$

In other words, $F_\sigma(w,W)$ is the set of vertices from which P_1 cannot avoid ending-up in the vertex w in the graph $G_\sigma[W]$.

6.3.2 Changing the Pivot

Let $\sigma \in \Sigma_0$, $v \in V_0$ and $p \in \mathit{Pivots}(\sigma)$. The first question to ask is in which case it is true that p is no longer a pivot after a switch $\sigma[v]$. This can only happen if there is no cycle on p in $G_{\sigma[v]}$, as by Theorem 6.2 the value of p must increase. So how could we break the cycle on p? Note that if p is a pivot there must be a successor q of p with the score $\Omega_\sigma(q) = (p,(\emptyset,d))$. Let us assume that $p \in V_1$ (if $p \in V_0$, P_0 is already winning that set) and there is only one such successor of p. (If there are two, the reasoning is similar.) To break the cycle on p we have to switch some vertex v on the path from q to p in T^p_σ. Let us deal with the different types of switches. Let $X = \{u \mid \Omega_\sigma(u) = (p,(\emptyset,d)) \text{ for some } d \in \mathbb{N}\}$.

- If $\sigma[v]$ is a *pivot switch*, then we break the cycle on p if $p \in F_\sigma(v,X)$.
- If $\sigma[v]$ is a *priority switch*, then we always break the cycle on p.
- If $\sigma[v]$ is a *distance switch*, then we break the cycle only if $\overline{\sigma}(v) \in F_\sigma(v, X \setminus \{p\})$ (note that this implies $v \prec_\sigma \overline{\sigma}(v)$) and $p \in F_\sigma(v,X)$.

6.3.3 Pivot-neutral Switches

In the previous part we showed under which circumstances a pivot p stops being a pivot. On the other hand a switch in V_σ^p which keeps p as a pivot will only change the values of vertices above v in the value tree T_σ^p.

Proposition 6.1. *Let $\sigma \in \Sigma_0$, and $v \in Enabled(\sigma)$ with $\Lambda_\sigma(v) = p$ such that $p \in Pivots(\sigma[v])$. Then for all $w \in V_\sigma^p$ such that $v \not\prec_\sigma w$ we have*

$$\Omega_\sigma(w) = \Omega_{\sigma[v]}(w)$$

Pivot Switch This kind of switch is the most straightforward one. Let $v \in V_\sigma^p$ for $p \in Pivots(\sigma)$. With a pivot switch $\sigma[v]$ we remove from V_σ^p the vertices from which P_1 is forced to v. I.e. $V_{\sigma[v]}^p = V_\sigma^p \setminus F_\sigma(v, V_\sigma^p)$. This means that if $p \in F_\sigma(v, V_\sigma^p)$ there is no cycle on p in $G_{\sigma[v]}$. Obviously for each vertex v the number of pivot switches on this vertex is bounded by n, as every time we do such a switch we increase the value of $\Lambda_\sigma(v)$.

Priority and Distance Switches There are two cases we want to distinguish here. Either $v \prec_\sigma \overline{\sigma}(v)$ (i.e. $\overline{\sigma}(v)$ is above v), or not. We start with the latter case first. Then in addition to Proposition 6.1 we can prove the following:

Proposition 6.2. *Let $\sigma \in \Sigma_0$, and $v \in Enabled(\sigma)$ with $\Lambda_\sigma(v) = p$, and $v \not\prec_\sigma \overline{\sigma}(v)$ such that $p \in Pivots(\sigma[v])$. Then for all $w \in V_\sigma^p$ we have*

$$\Lambda_\sigma(w) = p = \Lambda_{\sigma[v]}(w)$$

The second kind of switch, where $\overline{\sigma}(v)$ is above v, is more complicated. By definition v must be enabled. In the case of priority switch the highest vertex w on the path $\overline{\sigma}(v) \to^* v$ in the value tree T_σ^p must belong to V_0. In the case of distance switch the highest vertex on this path is smaller than p. That in both cases implies $p \sqsubset w$ and, if P_1 kept the counter-strategy $\widehat{\sigma}$ after the switch, at least the cycle $v \to \sigma(v) \to^* v$ would be removed from V_σ^p. The next lemma shows in which circumstances we do not change the set V_σ^p:

Proposition 6.3. *Let $\sigma \in \Sigma_0$, and $v \in Enabled(\sigma)$ with $\Lambda_\sigma(v) = p$, and $v \prec_\sigma \overline{\sigma}(v)$ such that $p \in Pivots(\sigma[v])$. Then*

$$V_{\sigma[v]}^p = \begin{cases} V_\sigma^p & \text{if } \overline{\sigma}(v) \notin F_\sigma(v, V_\sigma^p) \\ V_\sigma^p \setminus F_\sigma(v, V_\sigma^p) & \text{otherwise} \end{cases}$$

6.3.4 Complexity Analysis Restrictions

In addition to classifying switches by the component of measure they improve on, we can classify them by the effect on the $\sqsubseteq$ relation among vertices.

Definition 6.15. Let $\sigma \in \Sigma_0$ and $X \sqsubseteq \mathit{Enabled}(\sigma)$. We say that the switch on X is *substantial*, if for some $v \in V$ it is true that $\Lambda_\sigma(v) \sqsubset \Lambda_{\sigma[X]}(v)$.

From what we have seen earlier, a single vertex substantial switch is each pivot switch, and some priority and distance switches on v where $v \prec \overline{\sigma}(v)$. Obviously in any improvement policy there can be at most $O(n^2)$ substantial improvement steps – at most n for each vertex.

So to have an exponential improvement sequence there must be exponentially many non-substantial improvement steps. That means in analysing the strategy improvement algorithm we can restrict ourselves to analysing switches in a single set V_σ^p. In that case we ask how many non-substantial switches we have to make before a substantial switch becomes available. Moreover we can assume that there are no pivot switches available in the set V_σ^p, as these do not depend on the inner structure of $G_\sigma[V_\sigma^p]$ and can be performed at any time.

Finally we can assume that each vertex $p \in \mathit{Pivots}(\sigma)$ is odd, i.e. $p \in V_1$. If this is not the case, than σ is already a winning strategy for all vertices in $F_0(X) = X$, where $X = \bigcup_{p \in \mathit{Pivots}_\sigma \cap V_0} V_\sigma^p$. We can therefore remove the set X from the game by Theorem 2.4.

6.4 On the Structure of Strategy Space

In this section we are going to present some interesting facts about the structure of the pre-order $(\Sigma_0, \sqsubseteq)$. The motivation here is that by understanding this structure, we may either be able to bound the number of iterations more tightly, find examples of parity games where an exponential number of iterations is needed for a given strategy improvement policy, find an effective way of choosing the initial strategy etc. All the results in this section are ours, unless indicated otherwise.

We start with a simple lemma with interesting consequences, some of which appeared already in [VJ00].

Lemma 6.3. *Let σ be a strategy and $Z \subseteq V_0$ s.t. all vertices of Z are disabled in $\sigma[Z]$. Then some vertex of Z is enabled in σ.*

Proof. Assume not. By Theorem 6.3 we have both $\sigma \sqsubset \sigma[Z]$ (all vertices of Z are disabled in $\sigma[Z]$) and $\sigma[Z] \sqsubset \sigma$ (all vertices of Z are disabled in σ), which is a contradiction. □

Let $\kappa \in \Sigma_0$ be the maximal strategy. Then for each strategy $\sigma \in \Sigma_0$ s.t. $\sigma \neq \kappa$ we have that $\kappa = \sigma[Z]$ for some $Z \subseteq V$, $Z \neq \emptyset$. Together with the fact that by definition $\mathit{Enabled}(\kappa) = \emptyset$ we get the following:

Corollary 6.2. *For every strategy $\sigma \in \Sigma_0$ where $\kappa = \sigma[Z]$ for some $Z \neq \emptyset$ there is $z \in Z$ which is enabled in σ.*

Corollary 6.3 ([VJ00])**.** *For every initial strategy there is a strategy improvement policy of length at most n. Moreover there is such a policy switching exactly one vertex in every improvement step.*

When games are restricted to at most two successors per each vertex we can however prove a stronger proposition: The elements of Z can be ordered in such a way that none of the switches is enabled after it has been done.

Theorem 6.5. *Let σ_0 be a strategy and $Z \subseteq V_0$ s.t. $\sigma_0[Z] = \kappa$, where κ is the maximal strategy. Then the elements of Z can be ordered in a sequence $z_1, \ldots, z_k$ such that in the sequence $\rho : \sigma_0 \xrightarrow{z_1} \sigma_1 \xrightarrow{z_2} \ldots \xrightarrow{z_k} \sigma_k = \kappa$ the following holds for all $1 \leq i \leq k$:*

1. $z_{i+1} \in \mathit{Enabled}(\sigma_i)$
2. $\forall j \leq i . z_j \notin \mathit{Enabled}(\sigma_i)$

Proof. We will prove the theorem by induction, more specifically by constructing the sequence ρ in reverse, starting with $\sigma_k = \kappa$. The base case is clear, as κ is the maximal strategy and therefore $\mathit{Enabled}(\kappa) = \emptyset$, implying 2.

For the inductive step assume we already know the vertices $z_{i+2} \ldots z_k$ and the strategy σ_{i+1}. Let $Y = Z \setminus \{z_{i+2} \ldots z_k\}$ – then $Y \cap \mathit{Enabled}(\sigma_{i+1}) = \emptyset$ (by 2.). Now chose $y \in Y$ to be a vertex s.t. for any other $z \in Y$ we have $\sigma_{i+1}[y] \not\sqsubseteq \sigma_{i+1}[z]$. Obviously there must be such an element. We argue that then $\mathit{Enabled}(\sigma_{i+1}[y]) \cap Y = \{y\}$. For contradiction assume that there is an element $x \in Y$, which is enabled in $\sigma_{i+1}[y]$.

Then $\sigma_{i+1}[y] \sqsubset \sigma_{i+1}[\{y,x\}]$, since x is enabled. Moreover, x cannot be enabled in $\sigma_{i+1}[\{y,x\}]$ (that would imply $\sigma_{i+1}[y] \sqsubset \sigma_{i+1}[\{y,x\}] \sqsubset \sigma_{i+1}[y]$, which is impossible). By Corollary 6.2 y must then be enabled. Therefore $\sigma_{i+1}[\{y,x\}] \sqsubset \sigma_{i+1}[x]$, which gives us $\sigma_{i+1}[y] \sqsubset \sigma_{i+1}[x]$, a contradiction with the maximality of $\sigma_{i+1}[y]$. Therefore we put $z_{i+1} = y$ and $\sigma_i = \sigma_{i+1}[y]$ and we are done. □

6.4.1 Restrictions on Improvement Sequences

The following theorem tells us something about the structure of the strategy space. It subsumes several special cases, which we will discuss later.

Theorem 6.6. *Let* $\sigma, \sigma' \in \Sigma_0$, $\sigma \sqsubset \sigma'$, $Z \subseteq V$ *such that* $\sigma' = \sigma[Z]$, $X = \mathit{Enabled}(\sigma)$ *and* $X' = \mathit{Enabled}(\sigma')$. *Then* $Z \cap X \not\subseteq X'$.

Proof. By contradiction – assume that $Z \cap X \subseteq X'$, and let $Z' = Z \setminus X$. We have that $\sigma \sqsubseteq \sigma'$ and it also must be the case that $\sigma' \sqsubseteq \sigma'[Z \cap X]$, as $Z \cap X \subseteq X'$ and therefore $Z \cap X$ is enabled in σ'. However $\sigma'[Z \cap X] = \sigma[Z']$, and by definition none of the vertices in Z' is enabled in σ. Therefore $\sigma[Z'] \sqsubseteq \sigma$, a contradiction. □

Corollary 6.4. *Let* $\sigma_0\sigma_1 \ldots \sigma_k$ *be an improvement sequence of strategies for some improvement policy P. Take* i, j *such that* $0 \leq i < j \leq k$, $X_i = \mathit{Enabled}(\sigma_i)$ *and* $X_j = \mathit{Enabled}(\sigma_j)$, *and* $Z \subseteq V$ *such that* σ_j=$\sigma_i[Z]$. *Then* $Z \cap X_i \not\subseteq X_j$.

The corollary above tells us more about the possible improvement sequences, ruling out some impossible ones. The hope was that we could show there is a sub-exponential bound on the length of such a sequence. That this is not true is shown in Sec. 6.6.

A simpler version of the corollary above, the Corollary 6.5, appeared first in in [MS99] as a property of general strategy improvement algorithm (the paper deals with strategy improvement for Markov Decision Processes). This result is not mentioned in [VJ00], and was unknown to the author of this thesis at the time we proved Theorem 6.6. We also present it here, as it is not widely known in the community, and is highly relevant to understanding the strategy improvement algorithm.

Corollary 6.5. *Let* $\sigma_0\sigma_1 \ldots \sigma_k$ *be a improving sequence of strategies for some policy P. Then for every* $0 \leq i < j \leq k$: $\mathit{Enabled}(\sigma_i) \not\subseteq \mathit{Enabled}(\sigma_j)$

Finally we have this easy consequence:

Corollary 6.6. *Let* $\sigma \in \Sigma_0$ *such that* $|\mathit{Enabled}(\sigma)| = 1|$. *Then* $\kappa(v) = \overline{\sigma}(v)$.

The corollary suggests a useful heuristic: Whenever (during a run of strategy improvement algorithm) we encounter a strategy σ with a single enabled vertex v, we know the value of $\kappa(v)$. This can be used in several ways. One

possibility is to search for strategies with only a single improvement available. Another one is to run several parallel strategy improvement algorithms (with different starting strategies), updating the others whenever we reach a strategy with a single enabled vertex in one of the parallel runs.

6.4.2 Inverse and Minimal Strategies

Here we introduce the concepts of inverse and minimal strategies, which may be useful in understanding the structure of strategy space.

Definition 6.16. Let $\sigma \in \Sigma_0$ be a strategy. We define $\widetilde{\sigma} = \sigma[V]$ and call $\widetilde{\sigma}$ the *inverse strategy* to σ.

Let us have a look at the inverse strategy to κ, where κ is the maximal strategy. This strategy, $\widetilde{\kappa}$, is the strategy for which there is the longest minimal improvement sequence – we need to make at least n switches if we switch one vertex at a time.

On the other hand the *minimal strategy* κ' is the minimal element in the pre-order $(\Sigma_0, \sqsubseteq)$. Again, as for the case of κ, we can prove that that there is only one minimal element – i.e. the dual of Corollary 6.1. The relationship between the minimal strategy κ' and the inverse to the maximal strategy $\widetilde{\kappa}$ is that $\kappa' \sqsubseteq \widetilde{\kappa} \sqsubseteq \kappa$.

An interesting way to use the inverse strategies would be, for example, to perform improvements on both the strategy σ and its inverse σ' in parallel and try to correlate the outcomes.

6.5 Choice of the Improvement Policy

There are two important choices to be made when running the strategy improvement algorithm. The first choice is which initial strategy to start with. The second one is which improvement policy to use, and we will concentrate on this aspect. The most obvious is the **maximal policy**, which for a strategy σ switches all vertices in $\mathit{Enabled}(\sigma)$. Unfortunately this policy is also the hardest one to analyse. More suitable for analysis are policies which always switch only one vertex at a time (single-vertex improvement policies). In this section we will deal only with the latter policies.

A short discourse. Strategy improvement has long been studied in the AI community. In the case of simple stochastic games Melekopoglou and Condon showed in [MC94] examples of games for which several very natural single-vertex improvement policies have an exponential length. However whether the strategy improvement algorithm (for simple stochastic games) using the maximal policy finishes in sub-exponential number of steps is still an open question.

This contrasts with the SI algorithm for parity games, where we have not been able to find examples of games with improvement sequences of exponential length for any of the natural single-vertex improvement policies shown below.

Minimal Distance policy The idea behind this policy comes from the fact that a switch on vertex $v \in V^p_\sigma$, can potentially change the values for all vertices above v (i.e. the vertices $w \in V^p_\sigma$ s.t. $v \prec w$). Therefore switching a vertex with the shortest distance to the pivot should increase the value of the greatest number of vertices. Also the following lemma provides us with some hope.

Lemma 6.4. *Let $\sigma \in \Sigma_0$ and $v \in V^p_\sigma$ such that $\Pi^2_\sigma(v) = 1$ (i.e. $\sigma(v) = p$). Then in any improvement sequence $\sigma = \sigma_0\sigma_1 \dots \sigma_k$ where p remains being a pivot there is at most one switch on v.*

Proof. Easily follows from the fact that the value of the pivot p cannot increase, and at the time of the switch (say for in σ_i) it must be the case that $\Omega_{\overline{\sigma}_i}(p) \sqsubset \Omega_{\overline{\sigma}_i}(p)$. □

Maximal Set policy This is a variant of the previous policy. Instead of choosing a vertex with the shortest distance to pivot, we choose the vertex $v \in V^p_\sigma$ maximising cardinality of the set $\{w \in V^p_\sigma \mid v \prec w\}$.

Maximal Separator policy In this policy we try to maximise the increase in the value of a vertex v. For a strategy σ and a vertex $v \in \mathit{Enabled}(\sigma)$ we define the *maximal separator* as $\mathit{MaxSep}(v) = \Pi^1_\sigma(v) \div \Pi^2_\sigma(v)$ (the highest priority in the symmetric difference of the respective priority profiles). Now in every step we chose the vertex v such that $\mathit{MaxSep}(v) > \mathit{MaxSep}(w)$ for all $w \in \mathit{Enabled}(\sigma) \cap V^p_\sigma$. The idea here is that bigger increases in value get us closer to the maximal value in shorter time.

Fair policies In a fair policy we do not switch a vertex again if we have a choice. More precisely for each vertex v we keep a value $\rho(v)$, which is a time-stamp of a last switch on this vertex, and select always some vertex of $\mathit{Enabled}(\sigma)$ with a minimal value of $\rho(v)$. Any of the policies above has of course its fair version.

The idea behind using fair policies is that we can avoid repeatedly switching a vertex with seemingly better properties, while missing on an important switch.

6.5.1 Experimental Results

We tried to analyse the policies above to see whether using such a policy can allow us to derive a sub-exponential bound on the number of improvement steps. However we have not been successful in this quest.

We therefore tried to experimentally evaluate these policies on a large test set of parity games. This set contained games on random graphs, games on standard regular graphs (grids, trees with back edges) as well as games on which some known algorithm needs exponential time.

The results were not much surprising. All the 'reasonable' (or 'natural') policies above finished in quite a small number of iterations (linear in the number of vertices). Their 'stupid' versions (like Maximal Distance or Minimal Separator) did much worse. Finally we would like to mention that none of the (single-vertex) improvement policies was able to beat the standard maximal policy, switching all enabled vertices at each time.

6.6 Improvement Policy of Exponential Length

In this section we are going to present a family of parity games G_{2n} parametrised by $n \in \mathbb{N}$, for each of which there exists an initial strategy and an improvement policy such that the number of iterations of the strategy improvement algorithm is exponential in n. This example was first discovered by Serre [Ser], however it has never been published. Since it gives an insight into the strategy improvement algorithm, we present it here. To get a general idea, you can have look at the graph G_6 in Fig. 6.2. The initial strategy is given by the full edges.

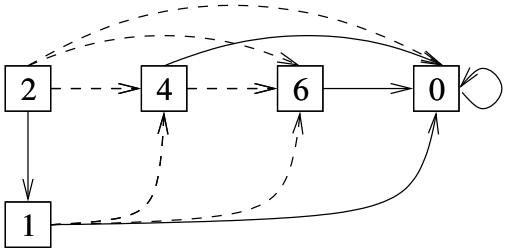


Figure 6.2: Graph G_6

The game graph of G_{2n} is defined recursively. All vertices in the game graph graph belong to P_0, and their priority will be given by their subscript - i.e. $\lambda(v_i) = i$. We start with the game graph G_2 with three vertices v_0, v_1 and v_2, which is in Fig. 6.3.

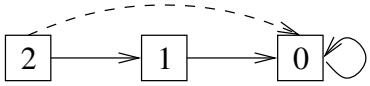


Figure 6.3: Graph G_2

The graph G_{2n+2} is then constructed from the graph G_{2n} by adding a new vertex v_{2n+2}. We proceed by adding an edge (v, v_{2n+2}) for every vertex of v of G_{2n} except the vertex v_0. The new vertex v_{2n+2} has a single successor v_0. The construction is schematically shown on Fig. 6.4. Formally:

$$\begin{aligned} V(G_{2n+2}) &= V(G_{2n}) \cup \{v_{2n+2}\} \\ E(G_{2n+2}) &= E(G_{2n}) \cup \{(v, v_{2n+2}) \mid v \in V(G_{2n}), v \neq v_0\} \cup \{(v_{2n+2}, v_0)\} \end{aligned}$$

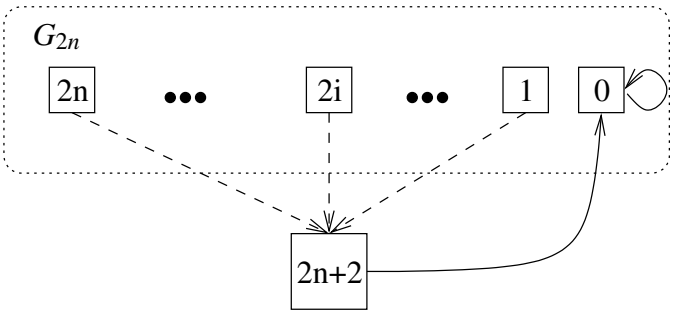


Figure 6.4: Construction of G_{2n+2}

Note that there is only one cycle in each graph G_{2n} - the cycle on v_0. Therefore all our improvement steps will be non-substantial. The initial strategy σ_0

satisfies:

$$\sigma_0(v_i) = \begin{cases} 1 & \text{if } i = 2 \\ 0 & \text{otherwise} \end{cases}$$

Now we are going to give the improvement sequence. We again use an inductive definition. For G_2 we have only one improvement step, $(v_2 \mapsto v_0)$. For the graph G_{2n+2} the sequence is as follows:

1. Perform the improvement sequence on G_{2n}
2. $(v_1 \mapsto v_{2n+2})$
3. Make n improvement steps $(v_{2i} \mapsto v_{2n+2})$ for $i = 1..n$ in an increasing order (i.e. starting with v_2 and finishing with v_{2n}).
4. Perform the improvement sequence on G_{2n}.

Next we have to show that each step is an improvement. For the graph G_2 it is obvious. For G_{2n+2} we go by the improvement sequence given above:

1. all steps are improvements by induction hypothesis
2. is improvement, as v_{2n+2} has the highest priority and no edges currently point to it (so it cannot be in the priority profile of v_1)
3. is improvement as for each edge $(v_{2i}, v_{2j}) \in E(G_{2n})$ we have that $i < j$ for all $1 \leq i, j \leq n$
4. all steps are improvements, the reason being that σ now looks almost like the initial strategy, except that all vertices point to v_{2n+2} instead of 0.

It remains to compute the number of steps. Let $C(n)$ be the number of improvement steps for the game G_{2n}. Then from the construction we get the following system of equations, from which we can conclude that $C(n)$ is exponential in n.

$$\begin{aligned} C(2) &= 1 \\ C(2n+2) &= C(n) + 1 + n + C(n) \end{aligned}$$

Chapter 7

Spine

The work described in this chapter has been done in collaboration with Colin Stirling.

In this chapter we present a new algorithm (actually two algorithms, but they have much in common) for solving parity games. The algorithm is based on the notion of spine, a structural way of capturing the (possible) winning sets and counter-strategies. The definition of spine and the algorithms were inspired by the strategy improvement algorithm (described in the previous chapter), but there are important differences. For one, we do not start with an arbitrary strategy for one of the players, but with computing the 'obvious' starting (partial) strategies for both players. Second, in our algorithm we do not perform arbitrary improvement steps. Instead we try to get rid of winning cycles by (hopefully temporarily) making the associated measure worse. Third, we tried to give an algorithm which is symmetric, i.e. which allows us to make improvement steps for both the players in alternation. We succeeded only partially in this respect, but the issues encountered were stimulating.

The hope behind the structure of spine and algorithms working on it were that it could provide us with a polynomial time algorithm for solving parity games. This has not been achieved, at least in the sense that we were not able to obtain a polynomial bound on the running time of the algorithm. As is customary for the problem of solving parity games, the only estimate we have is the trivial exponential bound. On the other hand, neither were we able to produce a counterexample on which the number of iterations needed is exponential. We hope that this new algorithm can provide further insight into the complexity of solving parity games.

Parity games we consider in this chapter are in the special normal form as defined in Section 2.3. In this normal form the vertices are identified with their priorities (each vertex has a unique priority) and each player owns the vertices with 'her' priorities (i.e. P_0 owns all even vertices). This restriction to normal form allows us to present our algorithm in a concise way, without introducing unnecessary complexity. However it can be modified, if needed, to deal with the case where the number of priorities is significantly smaller then the number of vertices, giving us 'better' complexity bounds.

7.1 Definitions

Here we introduce the fundamental definition of this chapter, a structure called a spine. A *spine* for a game G is a structural representation of the underlying game and has several nice properties, which will be useful later.

Definition 7.1 (Spine)**.** Let $\mathcal{G} = (V, E)$ be a parity game and $X : V \to V$ a function. For such an X we define $B(X) = \{b \mid \exists v \in V \text{ s.t. } X(v) = b\}$ and $X_b = \{v \in V \mid X(v) = b\}$. Moreover we can assume that $B = \{b_1, \ldots, b_k\}$, where $b_1 > b_2 > \ldots > b_k$. We say that X is a *spine* of a game $\mathcal{G} = (V, E)$ if the following axioms are true for each $b \in B$:

I1. $\forall v \in X_b . v \leq b$

I2. $\forall v \in X_b$ if $v \neq b$ then there exists $w \in X_b$ s.t. $(v, w) \in E$

I3. if $b \in X_b$, then there exists $b' \in B(X)$ and $w \in X_{b'}$ s.t. $b' \leq b$ and $(v, w) \in E$.

I4. $\forall v \in V_0 \cap X_b . (v, w) \in E \implies X(w) \sqsubseteq X(v)$ (E4).
$\forall v \in V_1 \cap X_b . (v, w) \in E \implies X(v) \sqsubseteq X(w)$ (O4).

I5. If $b \in V_0$ ($b \in V_1$) then $G[X_b]$ is won by P_0 (P_1).

For a spine X we call a vertex $b \in B(X)$ a *base vertex* of the set X_b and the set X_b the *upper set* of the vertex b. Note that we do not require for the base vertices themselves to be elements of their upper sets – i.e. it is not necessarily true that $b \in X_b$. Finally we say $X(v)$ is the *score* of vertex $v \in V$ in spine X.

Let us look at the definitions of I1-I5 to see what they mean. The first property, I1, says that every upper set can contain only vertices smaller than or

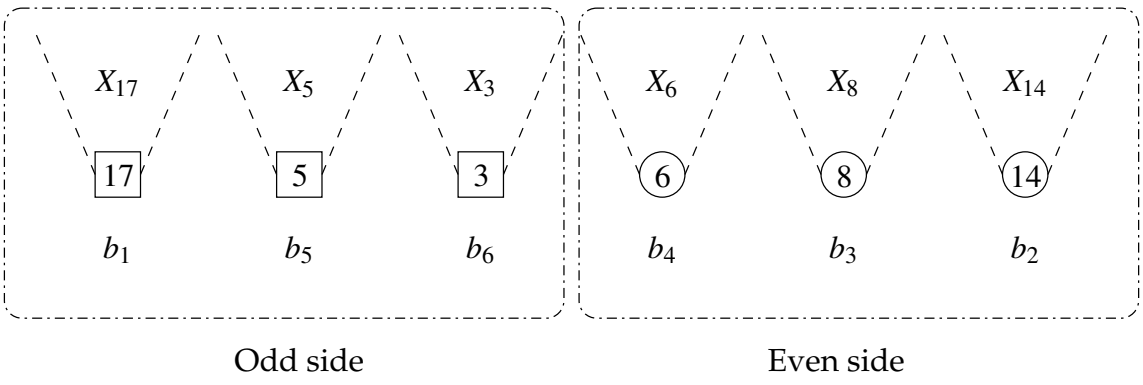


Figure 7.1: Spine

equal to the base vertex. Properties I2 and I3 then require the existence of at least one successor in the same upper set for each vertex of that set, with a possible exception of the base vertex. In that case the base vertex is constrained specifically by I3.

To understand the property I4, have a look at Fig. 7.1. This figure shows how to think about the structure of a spine pictorially. Here the base vertices $b_1, \ldots, b_k$ are ordered from left to right by $\sqsubseteq$ (therefore we can talk about 'even side' and 'odd side' of a spine), and their upper sets are drawn above them. In this representation the property I4 basically says even vertices have their successors 'to the left' (or in the same set; E4), whereas odd vertices have theirs 'to the right' (or in the same set; O4).

Finally I5 states that each player wins any of the sets on his side when they are considered as separate games. Note that from I2 it follows that every game $G[X_b]$ has at most one extra edge which is not in E (the edge (b,b)); and the extra edge is present only in the case when the base vertex b is a member of the set, but does not have any successors in this set. From I5 it follows that there must be a partial strategy σ_X such that σ_X is defined only on vertices in $\bigcup_{b \in B(X) \cap V_0} X_b$ and σ_X restricted to X_b is a winning strategy of P_0 in each of the games X_b for $b \in B(X) \cap V_0$. Similarly we can define a strategy τ_X for player P_1.

As is possible to see from the definition, each spine defines a decomposition of a parity game $\mathcal{G}$ into a number of upper sets X_b with some specific properties. Before we proceed further we need to show that every parity game has at least one spine, so we have our starting point.

Lemma 7.1. *For every game $\mathcal{G}$, the function X=`InitialSpine`($\mathcal{G}$) is a spine of the game $\mathcal{G}$. We call this X* initial spine *in the rest of the paper.*

Proof. The algorithm works as follows. In U it keeps the list of vertices not yet processed, starting with $U = V(G)$. Now in each iteration it selects the highest vertex not already in some upper set and makes it a new base vertex b. Now the player who owns b tries to force as many vertices from U to b and the resulting set is taken to be the upper set X_b. This set is then removed from U and the algorithm continues with a new iteration.

To prove that X is indeed a spine we need to check that the properties I1-I5 hold, and that X is defined for all vertices in V. The latter fact is obvious from the algorithm. I1 holds since we always select b to be the highest vertex not yet present in any of the already created upper sets, and I2 holds from the definition of force set. I5 must also hold by definition of the force set, as the player owning the base vertex b can force the play to this vertex. Moreover if b has a successor in X_b he wins since b is the highest vertex in X_b and he can force a cycle on this vertex. The other case, that b does not have a successor in X_b, is even easier, as there is the winning edge (b,b) in the game $\mathcal{G}[X_b]$.

Finally the properties I3 and I4 are implied by the following three invariants (I3 by i, I4 by ii and iii), which are easy to prove by induction. In these invariants U^i and X^i mean the value of U and X at the beginning of i-th iteration of the **while** cycle.

i. $\forall v \in U^i . \exists w \in U^i . (v,w) \in E$

ii. $v \in U^i \cap V_0 \implies \forall (v,w) \in E . w \in U^i \vee X^i(v) \in V_1$

iii. $v \in U^i \cap V_1 \implies \forall (v,w) \in E . w \in U^i \vee X^i(v) \in V_0$

□

Procedure `InitialSpine(`U`)`

U:=$V(G)$; X:= empty function
while $U \neq \emptyset$ **do**
  $b := \max(U)$
  $Y := F_{o(b)}(\{b\}, U)$
  foreach $v \in Y$ **do** $X(v) = b$
  $U := U \setminus Y$
return X

As is possible to see from the definition, there are two different types of upper sets. We say that upper set X_b is a *cyclic upper set*, if $b \in X_b$ and b has some successor in the set X_b. Otherwise the upper set is called *acyclic upper set* and $b \in X_b$ by definition. From now on we will use 'cyclic set' and 'acyclic set' to mean cyclic upper set and acyclic upper sets (so dropping the adjective 'upper'). The important property of spines is that it is not possible to have all upper sets acyclic.

Lemma 7.2. *In every spine there is at least one upper set which is cyclic.*

Proof. Let X be a spine and b be the minimal base vertex (with respect to the $'\leq'$ ordering). Then X_b must be cyclic, otherwise I3 would contradict the minimality of b. □

7.2 Switching

Consider a cyclic set X_b (w.l.o.g. we assume $b \in V_0$). Then, according to the definition of a spine, $\mathcal{G}[X_b]$ is a proper subgame of $\mathcal{G}$ ($G[X_b]$ does not contain any extra edges not already present in G). From I5 we have that P_0 wins $\mathcal{G}[X_b]$. Therefore it is in player P_1's interest not to stay in the subgame $\mathcal{G}[X_b]$ when playing on the full graph G. In other words he wants to 'switch' his strategy out of the set X_b. This motivates the following definition:

Definition 7.2 (switch)**.** Let X be a spine of $\mathcal{G}$ and $b \in B(X)$ s.t X_b is a cyclic set in X. We say that the pair (v,w) is a *b-switch* in spine X if $v \in X_b, o(v) \neq o(b), (v,w) \in E(G)$ and $b = X(v) < X(w) = b'$. We say that a pair (v,w) is a *switch* in spine X, if it is a b-switch for some $b \in B(X)$. We also define for $U \subseteq V$ and spine X the set $Switches(U,X) = \{(v,w) \mid v \in U, (v,w)$ is a switch in $X\}$. Finally if $Switches(U,X) \neq \emptyset$, then we define $MinSwitch(U,X) = (v,w)$ where $(v,w) \in Switches(U,X)$ and $\forall (v',w') \in Switches(U,X).w \leq w'$.

An example is in Fig. 7.2. Here P_1 has a 4-switch $(3,6)$ (the dotted arrow), which allows him to escape from the losing cycle on 4. Notice that the definition of switch implies that if $b \in V_0$ and (v,w) is a b-switch, then $v \in V_1$ and also $X(v) \sqsubset X(w)$, since X must satisfy O4 for b. Now that we have the definition of b-switch, we can formalise the paragraph at the beginning of this section.

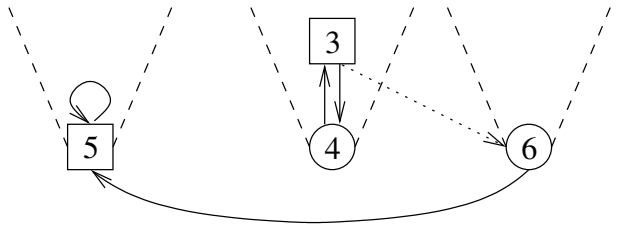


Figure 7.2: Switch example

Lemma 7.3. *If X_b is a cyclic set in spine X such that there is no b-switch available, then $X_b \subseteq W_{o(b)}$.*

Proof. W.l.o.g we can assume that $v \in V_0$. By I5 P_0 must have a winning strategy σ in the game $\mathcal{G}[X_b]$. By definition of b-switch and $O4$ we have that for each $w \in V_1 \cap X_b$ all successors of w are included X_b. Therefore σ is also a winning strategy for the vertices of X_b in the full game $\mathcal{G}$. □

Now observe that if we had an algorithm which is always able to find a spine with a cyclic set Y such that there are no switches available for Y, then we could easily solve the parity game $\mathcal{G}$. The goal of our algorithm will be exactly that: Given a spine X we want to find a new spine X' (possibly with many steps in between) containing a cyclic set with no available switches.

On the other hand, if there are switches available, we would like to have a transformation which, given a spine X and a switch (v,w), would create a new spine X' which reflects the effect of the switch on X. Moreover, we would like to do it in a way which guarantees that our algorithm finishes after a finite number of iterations. One way is to define a measure which can compare spines and show that every time we apply a switch the measure grows.

The most straightforward way of capturing the effect of a switch (v,w) (with $X(v) = b$ and $X(w) = b'$) on the spine X is to move the vertex v over to the set $X_{b'}$. However that would definitely break at least the axiom I4 with no easy way of fixing it later. The next best thing we can do is to move the whole set $R(v,X_b)$. Now if we try to recompute the spine in a similar way as computing the initial spine, we would succeed if X_b was a strongly connected component and therefore $X_b = R(v,X_b)$.

The solution to this problem is not only to make one switch out of a cyclic set X_b , but let the player P_1 make switches out of this set until no cycle remains, or there are no more switches available. We leave the recomputation of

the spine till this point, i.e. we do not recompute after a single switch. This is what we call *breaking cycles in* X_b and what is described by the algorithm `BreakEven` for the case $b \in V_0$. The case $b \in V_1$ is symmetric, and the corresponding algorithm is called `BreakOdd`.

Procedure `BreakEven`(U,X)

X':=X
while $U \neq \emptyset$ **do**
  if $Switches(U,X) = \emptyset$ **then**
    return (U, X')
  else
    b:=$MinSwitch(U,X)$
    Y:=$R(b,U)$
    foreach $v \in Y$ **do** $X'(v) = X(w)$
    U:=$U \setminus Y$
return $(\emptyset, X')$

Before we show how to recompute the spine after breaking the upper set U we need to prove two lemmas about the procedure `BreakEven`. The first one shows the existence of several invariants which will be useful later, while the second says what happens if we cannot break all the cycles in X_b.

Lemma 7.4. *Let X be a spine, $b \in V_0$ and X_b a cyclic set. If* `BreakEven`*(X_b, X)= $(\emptyset,X')$ then the following holds:*

A1) X' satisfies I1, I2, O4

A2) $\forall v \in V_0 \setminus B(X).X'(v) \sqsubseteq X'(\sigma_X(v))$

A3) $\forall v \in V(G).X(v) \sqsubseteq X'(v)$, and $\forall v \in X_b.X(v) \sqsubset X'(v)$

Proof. The algorithm `BreakEven` works as follows: We start with U being the cyclic set X_b. Then we select the smallest switch available (say (v,w)) and move all vertices in U which can reach v to the set $X_{b'}$, where $b' = X(w)$. If $U \neq \emptyset$ then we repeat this process. However since we take the minimal available switch in each iteration, we know that for the next selected switch (v',w') it must be the case $X(w) \sqsubseteq X(w')$. We continue this way until $U = \emptyset$.

Before we proceed further note, that $X'(v)$ differs from $X(v)$ only on the vertices originally in U (i.e. the set X_b). Then A3 holds, since for each switch

(v,w) used in some iteration we know $X(v) \sqsubset X(w)$ by $O4$ and the definition of switch.

Next we prove A2. Again we have to show this only for vertices which were originally in X_b. This can be done by a simple induction, using the fact that every time some $u \in R(v,U) \cap V_0$, then either also $\sigma_X(u) \in R(v,U)$ (and therefore $X'(u) = X(\sigma_X(u))$), or $\sigma(u) \in (U \setminus R(v,U))$ and then $X(u) \sqsubseteq X(\sigma_X(u))$ follows from the fact that when $\sigma_X(u)$ is actually moved, it goes to a higher set (in the '$\sqsubseteq$' ordering).

Finally I1 holds because for each switch (v,w) used in the algorithm $X(v) < X'(w)$, and I2 by definition of switch and $R(v,U)$. O4 can be possibly broken only for vertices in X_b, but this cannot happen as we always select the minimal switch and compute a reachability set for this switch. Altogether we have that A1 also holds. □

Lemma 7.5. *Let X be a spine, $b \in V_0$ and X_b a cyclic set. If* `BreakEven`*(X_b, X)= (Y,X') and $Y \neq \emptyset$ then $Y \subseteq W_0$.*

Proof. If $Y \neq \emptyset$, then there is no switch available for the set Y and by O4 all successors of vertices in $V_1 \cap Y$ must also be in Y. Then σ_X is a winning strategy for P_0 in the game $\mathcal{G}$ for all vertices in Y. □

7.2.1 Recomputing the Spine

If we have successfully managed to break all cycles in X_b, what we get back from the procedure `BreakEven` is a structure Y. But this Y does not have to be a spine. What we need to do next is to bring Y into a consistent state – a new spine X'. The algorithm `RecomputeEven` behind this is in some sense similar to the algorithm `InitialSpine` for computing the initial spine, but tries to preserve the existing upper even sets. The following important lemma says that `RecomputeEven` does exactly what we want it to do.

Lemma 7.6. *Let X be a spine of a game $\mathcal{G}$, $b \in V_0$, and X_b a cyclic set such that* `BreakEven`*(X_b, X)=($\emptyset$,Y). Then X'=*`RecomputeEven`*(Y) is a spine of $\mathcal{G}$ and $\forall v \in V(G).X(v) \sqsubseteq X'(v)$, and $\forall v \in X_b.X(v) \sqsubset X'(v)$.*

Proof. The algorithm works in a similar way as the algorithm `InitialSpine` with the following exception: Throughout the algorithm we keep in c the highest even base vertex in Y which has not been yet processed (by construction this

Procedure `RecomputeEven(`Y`)`

U:=$\bigcup_{b\in B(Y)} Y_b$; X':=empty function
while $U \neq \emptyset$ **do**
  c:=max$\{(B(Y) \setminus B(X')) \cap V_0\}$
  if $c \geq \max(U)$ **then**
    $b := c$
    $Z := F_0(U \cap Y_c, U)$
  else
    $b := \max(U)$
    $Z := F_{o(b)}(\{b\}, U)$
  foreach $v \in Z$ **do** $X'(v) = b$
  $U := U \setminus Z$
return X'

is equivalent to c not being a base vertex of X'). If $c > \max(U)$ at any iteration, we select c as the new base vertex and compute the force set of $Y_c \cap U$, thus preserving the existing even upper sets.

The following invariants are easy to prove by induction. In these invariants U_i, X^i, and c_i are the values of U, X, and c at the beginning of i-th iteration of the **while** cycle.

i. $\forall v \in U_i . \exists w \in U_i . (v,w) \in E$

ii. $v \in U_i \cap V_0 \implies \forall (v,w) \in E . w \in U_i \vee X^i(v) \in V_1$

iii. $v \in U_i \cap V_1 \implies \forall (v,w) \in E . w \in U_i \vee X^i(v) \in V_0$

iv. $\forall v \in V . c_i \sqsubseteq Y(v) \implies v \notin U_i$

v. $\forall v \notin U_i . Y(v) \sqsubseteq X'(v)$

Note that from the last two properties we automatically get $\forall v \in V . Y(v) \sqsubseteq X'(v)$. By applying Lemma 7.4 also $\forall v \in V . X(v) \sqsubseteq X'(v)$ and $\forall v \in X_b . Y(v) \sqsubset X'(v)$. We now split our analysis into three separate cases.

case $b = c_i (b \in V_0)$**:** We show that I1-I5 hold for $U_i \cap Y_c$. The fact that they hold also for $F_0(U_i \cap Y_c)$ can be shown in the same way as in the proof of Lemma 7.1.

I1: By Lemma 7.4 I1 holds for Y_c. For a contradiction assume there is a vertex $v \in U_i \cap Y_c$ s.t. v has no successor in U. Let w be any of the successors of

v in Y_c - by I1 for Y_c there must be at least one. If $v \in V_0$ then because $w \notin U_i$, by ii and v we get a contradiction with $v \in U$. Similarly for $v \in V_1$ we get a contradiction with i and O4 for Y.

I2 holds since $b \geq \max U_i$, and I3 can be proved by using ii. I4 follows from ii and iii. Finally we show that I5 has to hold. Assume this is not the case. Then there must be some $x \in V_0 \cap Y_c$ such that $\sigma_X(x) \notin Y_c$. But by property 2 of Lemma 7.4 $c \sqsubset X'(\sigma_X(x))$ and therefore by ii $x \notin U_i$, a contradiction.

case $b = \max(U), b > c (b \in V_0)$**:** Proof of I1-I5 is done in the same way as in Lemma 7.1.

case $b \in V_1$**:** Again, the proof of I1-I5 is done in the same way as in Lemma 7.1. The only thing we have to check is that for all vertices in the set $v \in Z = F_1(\{b\}, U_i)$ we have $Y(z) \sqsubseteq b$. This follows from iv, v and the fact that $b = \max(U_i)$. □

Note that both new even and odd sets, cyclic or not, can appear in the recomputed spine X'. On Fig. 7.3 you can see a spine with only one switch available: $(3,6)$. After running `BreakEven` and `RecomputeEven` we get a new spine with an odd cyclic set $X'_1 = \{1\}$. Similarly for spine in Fig. 7.4 we get a new even cyclic set $X'_4 = \{1,4\}$.

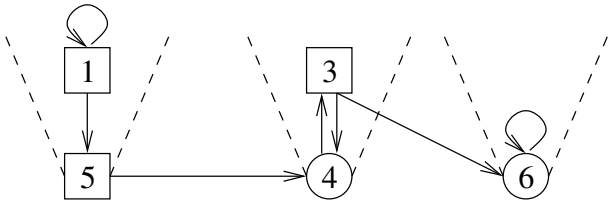


Figure 7.3: New odd cyclic set after the switch $(3,6)$

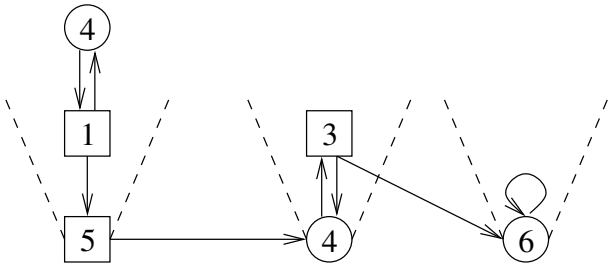


Figure 7.4: New even cyclic set after the switch $(3,6)$

7.3 Symmetric Algorithm

We are now ready to present the first of the two algorithms for solving parity games. The idea of this algorithm is quite simple. We start by computing an initial spine X, using the algorithm `InitialSpine`. Now we proceed in iterations. In each iteration we choose a cyclic set X_b, which must exist by Lemma 7.2. Assuming $b \in V_0$ let (U, X')=`BreakEven`(X_b, X). There are two possibilities as to the value of U. 1) If $U \neq \emptyset$, then $F_0(U) \subseteq W_0$ by Lemma 7.5. We can remove the set $F_0(U)$ from the game and restart the algorithm on the game $\mathcal{G} \setminus F_0(U)$. 2) Otherwise $U = \emptyset$ and X' is a spine by Lemma 7.6. We put $X := X'$ and start with a new iteration. In the case that $b \in V_1$ the reasoning is symmetric (we call `BreakOdd`, `RecomputeOdd` and compute $F_1(U)$).

The question is whether we are able to always guarantee that this algorithm terminates. Before providing the answer, we will first consider a specific case. Let us assume that every time there is a cyclic set, there is also an even cyclic set. This allows us to choose an even cyclic set in each iteration. In that case the convergence is implied by the fact that for all vertices $v \in V(G)$ we have $X(v) \sqsubseteq X'(v)$, and moreover $\forall v \in X_b . X(v) \sqsubset X'(v)$. Since there are at most n upper sets through which a vertex can pass (and there are n vertices), the number of iterations is bounded by n^2 and therefore we would have a polynomial algorithm for solving parity games.

Unfortunately it is not hard to find a game $\mathcal{G}$ and spine X of $\mathcal{G}$ such that 1) there is no even cyclic upper set and 2) W_0 is not an empty set. An example of such a spine is in Fig. 7.5. Here P_0 can always win the cycle on 4 by using the strategy $\sigma(4) = 4$, but X_6 is not cyclic.

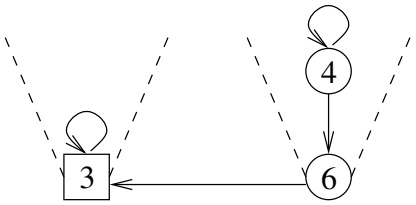


Figure 7.5: No even cyclic set, but $W_0 \neq \emptyset$

As we have just seen, unless we modify the notion of spine we cannot keep switching on one side of the spine only. To be able to keep the notion of spine unchanged we therefore need to find a way of guaranteeing termination of

the algorithm, as doing both even and odd switches can both increase and decrease the score of a vertex in both the '$\sqsubseteq$' and '$\leq$' orderings. To get a convergence in a reasonably straightforward way we will use a modified version of the algorithm `RecomputeEven`. The new algorithm, called `KRecomputeEven`, is exactly the same as `RecomputeEven` except for one modification. Let c be the highest element of $B(X) = B(Y)$ such that $X_c \neq Y_c$ (by Lemma `BreakEven` actually $X_c \subset Y_c$). Then we verbatim copy from X all upper sets X_b such that $b > c$, i.e. we lay $X'(v) = Y(v) = X(v)$ for all vertices $v \in V.X(v) > c$. Also instead of putting $U = \bigcup_{b \in B(Y)} Y_b$ we start with $U = \bigcup_{b \in B(Y), b > c} Y_b$. We leave the rest of the algorithm (including the whole **while** cycle) unchanged.

We claim that the variant of Lemma 7.6 where we replace `RecomputeEven` with `KRecomputeEven` holds, i.e. X' is a spine and the score does not decrease for any of the vertices of G, while strictly increasing for the vertices in X_b.

Using the procedure `KRecomputeEven` we can now finally present the new algorithm, called `SpineSymmetric`. The way it works is very simple. Given a game $\mathcal{G}$ it starts by computing the initial spine. Then it selects some cyclic set X_b in the spine, and tries to break cycles in this set. If it succeeds, then the spine is recomputed. Otherwise the force set of the unbreakable cycle is removed from the game and the whole algorithm is restarted.

Procedure `SpineSymmetric(`$\mathcal{G}$`)`

W_0:=$\emptyset$;W_1:=$\emptyset$;V=$V(G)$
l1 **while** $V \neq \emptyset$ **do**
  X:=`InitialSpine` (V)
  while *true* **do**
    choose $b \in B(X)$ s.t. X_b is cyclic
    if $b \in V_0$ **then**
      (U,Y):=`BreakEven` (X_b, X)
      if $U = \emptyset$ **then** X:=`KRecomputeEven` (Y,X)
      else W_0:=$W_0 \cup F_0(U)$; V:=$V \setminus F_0(U)$; **break** l1
    else
      (U,Y):=`BreakOdd` (X_b, X)
      if $U = \emptyset$ **then** X:=`KRecomputeOdd` (Y,X)
      else W_1:=$W_1 \cup F_1(U)$; V:=$V \setminus F_1(U)$; **break** l1
return W_0,W_1

Theorem 7.1. *Let $\mathcal{G}$ be a parity game and $n = |V(G)|$. Then* `SpineSymmetric`*($\mathcal{G}$)=(W_0, W_1) and* `SpineSymmetric` *stops after at most $O(n^n)$ iterations.*

Proof. The correctness of the algorithm follows from the Lemmas 7.4, 7.5, 7.6, and Theorem 2.4. The bound on the number of iterations follows from the fact that we use the modified version of recomputing (procedures `KRecomputeEven` and `KRecomputeOdd`). This modification guarantees that the sets with base vertices higher than the one currently used as a target for the switch will stay the same, and the target set increases. It is easy to check that a vertex can be removed from an upper set only if actually some greater (w.r.t. $>$) increases in size. □

7.3.1 Optimisations

As in the case of strategy improvement, the performance of the algorithm strongly depends on how we select the upper sets X_b we are going to break. Let $B' \subseteq B(X)$ be the set of bases of cyclic upper sets. There are several obvious choices:

1. Choose $\max B'$.
2. Choose $b \in B'$ such that if we put $mod(b)$ to be the base of the highest modified set, then $\forall b' \in B'.mod(b') \leq mod(b)$.
3. Always prefer a base from $B' \cap V_0$ if this set is non-empty.

We do not know if any of these choices will provide us with a better (possibly polynomial) estimate on the number of iterations of the algorithm `SpineSymmetric`. The complexity analysis for each of these cases is very interesting, but complicated and hard to pinpoint.

7.4 Recursive Spine

Although the algorithm presented in the previous section is not too complicated, it is hard to get any decent complexity estimate. The reason behind this is that the algorithm is not monotone, in the sense that the scores of vertices do not grow monotonely in at least one of the $\sqsubseteq$ and $\leq$ orderings. On the other hand, we have seen that if we do only switches for one of the players (and

in this section we will assume it is the player P_0), we get monotonicity with respect to the 'reward' ordering $\sqsubseteq$.

Let us focus on the problematic case of the spine on Fig. 7.5, where there is no cyclic set for P_0, even though $W_0 \neq \emptyset$. The reason why there is no even cyclic set is because the spine does not tell us the full story. There is a cycle on '4', but it is overshadowed by the 'more profitable' upper set of '6'. We therefore need some refinement of acyclic upper even sets.

Consider an acyclic set X_b in a spine X, where $b \in V_0$. Then, from the definition of spine, we know that P_0 wins all vertices in $\mathcal{G}[X_b]$. But this is true at least partly because of adding the edge (b,b), which is not present in $E(G)$. The plays which go through b in $\mathcal{G}[X_b]$ are not plays of $\mathcal{G}$. In other words, in $\mathcal{G}[X_b]$ we *assume* that P_0 wins the vertex b.

If we want to analyse the game $\mathcal{G}[X_b]$ independently, we need to remove this assumption. We do it by going to the opposite extreme, i.e. by assuming that it is the player P_1 who wins b. In that case it is in his interest to force the play into b, and we put $Y = X_b \smallsetminus F_1(\{b\}, X_b)$. Note that now it must be true that $\mathcal{G}[Y]$ is a subgame of $\mathcal{G}$, since all vertices of Y have at least one successor in Y. Therefore the game $\mathcal{G}[Y]$ must have a spine X_Y, which may or may not contain even cycles. But to all acyclic sets in X_Y we can again apply the same reasoning as to X_b. This more or less gives us a recursive refinement of the notion of spine. Pictorially the situation is described in Fig. 7.6.

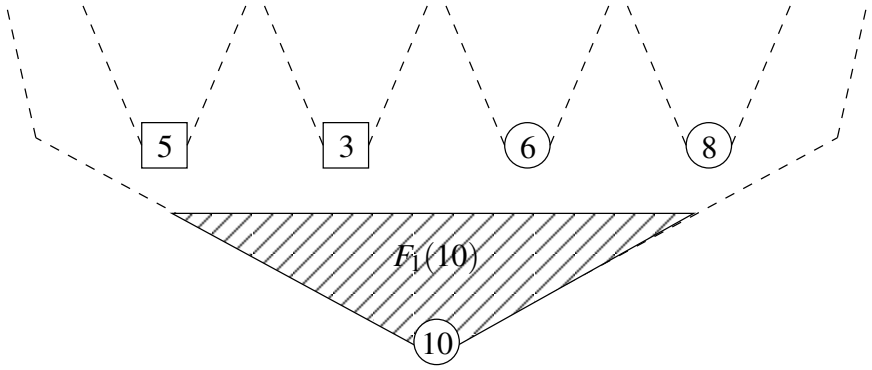


Figure 7.6: Recursive spine

Before we present the formal definition, let us consider the following. In the case of standard (non-recursive) spine X, every vertex v is associated to a single base vertex $b = X(v)$ such that v is in the upper set of this base vertex. In other words, the spine is completely defined by giving the value $X(v)$ for each

vertex. For the recursive case the values $X(v)$ will not be single vertices, but ordered sequences of base vertices whose associated upper sets contain v. We will also need to extend the 'reward' ordering '$\sqsubseteq$' to these ordered sequences of vertices. This can be done in almost the same way as in the case of strategy improvement (see Definition 2.3 on page 9):

Definition 7.3. Let $\alpha, \beta \in V^*$, $\alpha_1 = v_1 \dots v_k$ and $\beta = w_1 \dots w_l$. Then we put $\alpha \sqsubset \beta$ iff one of the following two propositions holds:

- $\exists i. 1 \leq i \leq \min(k,l)$ s.t. $v_i \sqsubset w_i$ and $\forall j. 1 \leq j < i. v_j = w_j$
- $k < l$ and $\forall i. 1 \leq i \leq k. v_i = w_i$

We say that $\alpha \sqsubseteq \beta$ iff either $\alpha \sqsubset \beta$ or $\alpha = \beta$.

For practical reasons we need to extend the order $\sqsubseteq$ to $V \cup \{\bot, \top\}$, where $\bot$ and $\top$ are new symbols such that $\forall v \in V. \bot \sqsubset v \sqsubset \top$ (also $v > \top$ and $v > \bot$). We also define new sets $V_\bot = V \cup \{\bot\}$ and $V_\top = V \cup \{\top\}$.

We are now going to give the definition of recursive spine.

Definition 7.4 (Recursive spine)**.** Let $\mathcal{G} = (V, E)$ be a parity game and $X : V \to V_0^*.V_\top$ a function such that for each $v \in V$ the sequence $X(v) = b_1 \dots b_k$ is a nonempty sequence of vertices satisfying $b_1 > \dots > b_k$. Using X we define the following sets and functions:

$Prefix(X) = \{\alpha \mid \exists v \in V, \beta \in V_\top^+ \setminus \{\top\} \text{ s.t. } X(v) = \alpha.\beta\}$,

$X_{=\alpha} = \{v \in V \mid X(v) = \alpha\}$,

$X_{\geq\alpha} = \{v \in V \mid \exists \beta \in V^* \text{ s.t. } X(v) = \alpha.\beta\}$, the *upper set* of α,

$X_{>\alpha} = X_{\geq\alpha} \setminus X_{=\alpha}$,

$\overline{X_\alpha} : X_{>\alpha} \to V$, where $\alpha \in Prefix(X)$, is defined as $\overline{X_\alpha}(v) = b \iff v \in X_{\geq\alpha.b}$, and

$X_\alpha : X_{>\alpha} \to V_0^*.V_\top$, where $\alpha \in Prefix(X)$, is defined as $X_\alpha(v) = \beta \iff X(v) = \alpha.\beta$.

We say that X is a *recursive spine* of a game $\mathcal{G}$ if the following axioms are true:

J1) $\overline{X_\alpha}$ is a spine for each $\alpha \in Prefix(X)$

J2) $b \in V_0$ and $X_{\geq\alpha.b}$ acyclic in $\overline{X_\alpha}$ implies $X_{=\alpha.b} = F_1(\{b\}, X_{\geq\alpha.b})$

J3) $b \in V_0$ and $X_{\geq\alpha.b}$ cyclic in $\overline{X_\alpha}$ **iff** $\forall v \in X_{\geq\alpha.b}. X(v) = \alpha.b.\top$

The notions of cyclic and acyclic upper sets can be easily adapted to recursive spine. We say that $X_{\geq\alpha.b}$ is cyclic in X if it is a cyclic set in $\overline{X_\alpha}$. (By definition $X_{\geq\alpha.b} = (\overline{X_\alpha})_b$.) Similarly we call $X(v)$ the *score* of a vertex v. Also note that for $\alpha \in \mathit{Prefix}(X)$ the function X_α is again a recursive spine. The '$\top$' element is added for technical convenience, to allow us easily distinguish cyclic sets. Finally we assume that $\mathit{Prefix}(X)$ always contains the empty sequence ε.

In the text to follow we will also need two functions defined on the set $V_\top^+$. Let $\alpha = v_1 \dots v_k$. Then $\mathit{tail}(\alpha) = v_k$, and $\mathit{head}(\alpha) = v_1 \dots v_{k-1}$. In the case of spine X and a vertex v s.t. $X(v) = \alpha$ the function $\mathit{tail}(\alpha)$ returns the topmost base set v is in, and $\mathit{head}(\alpha)$ returns the prefix of the topmost spine in which v is included.

For computing the initial recursive spine we can reuse the procedure `InitialSpine`, performing a recursive descent on acyclic even upper sets. This is implemented by the procedure `RInitialSpine`. The reason why we do not go recursive on acyclic odd sets is that in the algorithm we will only be breaking even cyclic sets – in other words only switches for player P_1 will be considered. Of course the algorithm may be equally well presented for the player P_1 by taking the dual definition of recursive spine.

Procedure `RInitialSpine`(V, α)

X := `InitialSpine` (V)
foreach $b \in B(X) \cap V_0$ *s.t.* X_b *is cyclic* **do**
  foreach $v \in X_b$ **do** $X'(v)$:=$\alpha.b.\top$
foreach $b \in B(X) \cap V_0$ *s.t.* X_b *is acyclic* **do**
  Y:=$F_1(b, X_b)$
  foreach $v \in Y$ **do** $X'(v)$:=$\alpha.b$
  Z:=$X_b \setminus Y$
  `RInitialSpine` $(Z, \alpha.b)$
foreach $b \in B(X) \cap V_1$ **do**
  foreach $v \in X_b$ **do** $X'(v)$:=$\alpha.b$

Lemma 7.7. *For every game $\mathcal{G}$, the function $X =$ `RInitialSpine`$(V(G),\varepsilon)$ is a recursive spine of $\mathcal{G}$ (called the* initial recursive spine *in the rest of the paper).*

Proof. The algorithm is self explanatory, both R2 and R3 being obviously fulfilled. R1 follows from the construction and Lemma 7.1. □

In exactly the same way as for non-recursive spine, we can always guarantee that there is at least one cyclic set in any recursive spine X. To prove that one needs only to take $\overline{X_\varepsilon}$ and apply the Lemma 7.2. However in the case of recursive spines we can also prove that if there are no cyclic sets for one of the players, then the opponent wins all vertices in the game. To prove this, we will need a lemma about winning for non-recursive spines.

Before presenting this lemma we introduce some more notation. Remember that the game $\mathcal{G}[Y]$ for $Y \subseteq V(G)$ is the game $\mathcal{G}$ restricted to vertices of Y, such that if there is $v \in Y$ with no successor in Y, then we add an edge (v,v). Similarly we define the game $\mathcal{G}\langle Y\rangle$ for $Y \subseteq V(G)$ as the the game $\mathcal{G}$ restricted to vertices of Y, but for a vertex $v \in V_i$ with no successor we add a new vertex $w \in V_{1-i}$ and edges (v,w) and (w,w). In other word reaching a vertex with no successor is a win for the opponent. Now we are ready to present the lemma:

Lemma 7.8. *Let X be a (non-recursive) spine of $\mathcal{G}$ such that P_1 wins the game $\mathcal{G}\langle X_b\rangle$ for each $b \in V_0 \cap B(X)$. Then $W_0 = \emptyset$.*

Proof. The proof goes by induction on the size of $B(X)$. The base case is $B(X) = \{b\}$. As X_b must be cyclic by Lemma 7.2, we must have $b \in V_1$ and by I5 P_1 wins all vertices.

For the inductive step let $B(X) = \{b_1, b_2, \ldots, b_{k+1}\}$. Take the game $\mathcal{G}' = \mathcal{G} \setminus X_{b_1}$. By I2 and I3 all vertices in $V(G')$ have at least one successor in $V(G')$ and therefore $\mathcal{G}'$ is really a game. We define function X' by putting $X'(v) = X(v)$ for all $v \in V(G')$. Clearly X' is a spine of $\mathcal{G}'$ with base vertices $B(X') = \{b_2, \ldots, b_{k+1}\}$. By induction hypothesis P_1 has a winning strategy τ for the game $\mathcal{G}'$. The analysis now splits into two cases.

If $b_1 \in V_0$, then by I4 there is no edge $(v,w) \in E(G)$ s.t. $v \in V(G') \cap V_0$ and $w \in X_{b_1}$. By the premise of this lemma P_0 cannot win by staying in X_{b_1} and cannot create a new even cycle of P_1 keeps to strategy τ once the play reaches $V(G')$. Therefore $W_0 = \emptyset$.

The second case is $b_1 \in V_1$. By similar reasoning the only way for P_0 to win for some vertex in $\mathcal{G}$ is to play from some vertex of $\mathcal{G}'$ to X_1. However P_1 using his winning strategy for the game $\mathcal{G}[X_1]$ can either win without leaving the set X_1, or the play must pass through the vertex b_1. If this is a winning play for P_0, it cannot stay in $V(G')$ and must infinitely often pass again through b_1. As $b_1 \in V_1$ and $b_1 = \max(V)$, P_1 wins such a play, thus we get a contradiction. □

Now we can prove the property we mention earlier in this section.

Lemma 7.9. *Let $\mathcal{G}$ be a game and X its recursive spine, such that there is no even cyclic set in X. Then $W_1 = V(G)$.*

Proof. We will prove that the player P_1 will win every game $\mathcal{G}[X_{>\alpha}]$ for $\alpha \in \mathit{Prefix}(X) \cap V_0^*$. As $X_{>\varepsilon} = V(G)$ this would prove the proposition. The proof goes by induction on the depth of spine nesting (length of α), starting with the innermost spine.

Let $\alpha \in \mathit{Prefix}(X) \cap V_0^*$ such that α is not a prefix of any other $\alpha' \in \mathit{Prefix}(X)$. Let $Y = \overline{X}_\alpha$. Then for all $b \in B(Y) \cap V_0$ we must have $Y_b = F_1(b, Y_b)$ (since Y_b cannot be cyclic). So Y satisfies the requirements of Lemma 7.8 and P_1 wins all vertices in $X_{>\alpha}$.

For the inductive case take $\alpha \in \mathit{Prefix}(X) \cap V_0^*$ and assume we have proved the proposition for all prefixes $\alpha.\beta$, $\beta \in V_0^+$. Let $Y = \overline{X}_\alpha$. Then for all $b \in B(Y) \cap V_0$ we must have Y_b is acyclic, and P_1 wins the game $\mathcal{G}[Y_b \setminus F_1(b, Y_b)]$. However it is easy to see that P_1 also must win the game $\mathcal{G}\langle Y_b \rangle$, as there is no edge $(v, w) \in E(G)$ s.t. $v \in (F_1(b, Y_b) \cap V_0)$ and $w \in (Y_b \setminus F_1(b, Y_b))$ (by definition of $F_1(b, Y_b)$). By application of Lemma 7.8 we get P_1 wins the game $\mathcal{G}[X_{>\alpha}]$. □

7.5 Recursive Switching

So far we have given an algorithm for constructing an initial spine, and studied what happens if there is no cyclic even upper set available. But if there is an even cyclic set, we need to adapt the algorithms for breaking cyclic sets and recomputing the spine to work on recursive spines, and show that we preserve monotonicity w.r.t. the extended '$\sqsubseteq$' ordering. Fortunately most of the hard work has already been done in the Section 7.2. In addition to the results shown there we will need the following generalisation of Lemma 7.6. In short it states, that the result of `RecomputeEven` on a modified spine X will again be a spine, even if we add vertices P_1 cannot reach from V' and remove $F_0(S)$ for some set $S \subseteq V$.

Lemma 7.10. *Let $\mathcal{G}$ be a parity game, $V' \subseteq V$ s.t. $\mathcal{G}' = \mathcal{G}[V']$ is a subgame of $\mathcal{G}$ and X spine of the game $\mathcal{G}'$ (or a result of running `RecomputeEven` on some spine of $\mathcal{G}'$).*
Moreover let $Y \subseteq V \setminus V'$ such that

1. *each vertex of Y has a successor in $Y \cup V$, and*

2. *no vertex of $V' \cap V_1$ has a successor in Y.*

Finally let $U \subseteq (V' \cup Y)$ such that $U = F_0(U, (V' \cup Y))$.
And define

$$Z(v) = \begin{cases} X(v) & \text{if } v \in V' \setminus U \\ \bot & \text{if } v \in Y \setminus U \\ \textit{undef.} & \textit{otherwise} \end{cases}$$

Then X'=`RecomputeEven`(Z) is a spine of $G[(V' \cup Y) \setminus U]$ and $\forall v \in (V' \setminus U).X(v) \sqsubseteq X'(v)$.

Proof. The lemma is sketched in Fig. 7.7. The proof itself is very similar to the one of the Lemma 7.6. I5 is guaranteed by the following: If we fix a strategy σ s.t. σ is a winning strategy for each even upper set in X, then by definition of F_0 we have $\sigma(x) \in U \implies x \in U$. The fact $X(v) \sqsubseteq X'(v)$ holds since there is no edge P_1 can choose going from from V' to Y. The proofs of other properties are either obvious, or can be modified similarly. □

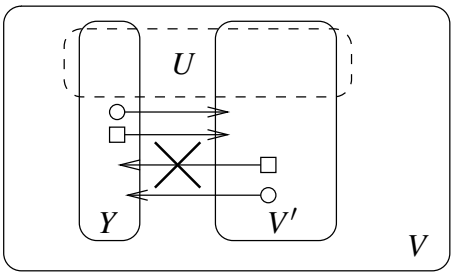


Figure 7.7: Lemma 7.10

Procedure `RBreakEven`(U, α, X)

β:=*head*(α); b:=*tail*(α)
(U, Y):=`BreakEven` $(U, \overline{X_\beta})$
foreach $v \in X_{>\beta}$ *s.t.* $\overline{X_\beta}(v) \neq Y(v)$ **do**
  $X(v)$:=$\beta.Y(v)$
if $U = \emptyset$ **then return** $(\emptyset, \beta, X)$
else if $\beta = \varepsilon$ **then return** (U, β, X)
else return `RBreakEven` (U, β, X)

Breaking even cycles in a recursive spine is done by the algorithm `RBreakEven`, which works as follows: We start with a recursive spine X and even cyclic set $U = X_{=\alpha.b.\top}$, where $\alpha \in \mathit{Prefix}(X)$ and $b \in V_0$. First we try to break cycles in U by running the procedure `BreakEven` on U in the context of $\overline{X_\alpha}$, the (non-recursive) spine where U is the upper set of b. Note that the spine $\overline{X_{\mathit{head}(\alpha)}}$ has not changed in this process.

If the new U is non-empty, we have to recursively repeat the process, this time allowing more switches by running `BreakEven` on (now smaller) U in $\overline{X_{\mathit{head}(\alpha)}}$. This is repeated until either $U \neq \emptyset \wedge \alpha = \varepsilon$ or $U = \emptyset$. In the first case we have $U \subseteq W_0$, as for every $v \in U \cap V_1$ and $(v, w) \in E$ also $w \in U$, and for each $v \in U \cap V_0$ we have $\sigma(v) \in U$, where σ is the winning strategy of P_0 in $\mathcal{G}[U]$. (The existence of such σ is implied by the fact that U is an even cyclic set.) In the second case ($U = \emptyset$) we are at the point at which we have to recompute the spine X with α being the current prefix. Note that the algorithm did not affect any vertices outside $X_{>\alpha}$ and also for each prefix α' of α, the spine $\overline{X_{\alpha'}}$ is unchanged.

After breaking the cycles on U we have to recompute the recursive spine. This is handled by the algorithm `RRecomputeEven`.

Lemma 7.11. *Let X be a recursive spine, $\alpha \in \mathit{Prefix}(X)$, and $U = X_{=\alpha.b.\top}$ a cyclic set in $\overline{X_\alpha}$ such that `RBreakEven`(U, α, X)=($\emptyset$, α, Y). Then X'=`RRecomputeEven`(Y, α) is a recursive spine and $\forall v \in V(G).X(v) \sqsubseteq X'(v)$, and $\forall v \in U.X(v) \sqsubset X'(v)$.*

Proof. During the run of the algorithm we will modify the function Y, keeping the current state in the function X'. We will show that the following invariant holds throughout the execution of `RRecomputeEven`: $\forall v \in V.X(v) \sqsubseteq X'(v)$. At the beginning this is guaranteed as $X' = Y$ and $\forall v \in V.X(v) \sqsubseteq Y(v)$ by (a recursive application of) Lemma 7.4.

The procedure `RRecomputeEven` takes over at the prefix α, where $\overline{X'_\alpha}$ is the outermost spine modified by the procedure `RBreakEven`. Since $\overline{X'_\alpha}$ a normal spine, we can call the procedure `RecomputeEven` on it. Let $S =$ `RecomputeEven`$(\overline{X'_\alpha})$. Then by the Lemma 7.6 $\forall v \in X'_{>\alpha}.\overline{X'_\alpha}(v) \sqsubseteq S(v)$. The base vertices of S will give us the first element of the score suffix: if $S(v) = b$, then $X'(v)$ will from now on contain the prefix $\alpha.b$.

The analysis now splits into three cases, according to the base vertices of the sets S_b. If $b \in V_1$, then we put $X'(v) = \alpha.b$ for all $v \in S_b$ and we are done. If

Procedure `RRecomputeEven` (X', α)

S:=`RecomputeEven` $(\overline{X'_\alpha})$
foreach $b \in B(S)$ **do**
  if $b \in V_1$ **then**
    foreach $v \in S_b$ **do** $X'(v)$=$\alpha.b$
  else if $b \in V_0$, *and* S_b *is cyclic* **then**
    foreach $v \in S_b$ **do** $X'(v)$=$\alpha.b.\top$
  else if $b \in V_0$, *and* S_b *is acyclic* **then**
    if $\alpha.b \notin Prefix(X)$ **then**
      Z:=$S_b \setminus F_1(\{b\}, S_b)$
      foreach $v \in Z$ **do** $X'(v)$=$\alpha.b$
      `RInitialSpine` (Z, $\alpha.b$)
    else
      foreach $v \in S_b \setminus X_{\geq \alpha.b}$ **do** $X'(v)$=$\alpha.b.\bot$
      Z:=$S_b \setminus F_1(\{b\}, S_b)$
      foreach $v \in Z$ **do** $X'(v)$=$\alpha.b$
      `RRecomputeEven` (X',$\alpha.b$)
return (X')

$b \in V_0$, and S_b is cyclic, we are finished as well, by putting $X'(v) = \alpha.b.\top$ for all vertices of S_b. In both cases we must have $\forall v \in S_b.X(v) \sqsubseteq X'(v)$ by construction (in the second case this is implied by the fact that $\forall v \in V.v \sqsubset \top$).

The remaining case $b \in V_0$ and S_b acyclic is the crucial one. We first check whether $X_{\geq \alpha.b}$ is an upper set in X. If not, then we know that $\forall v \in S_b.X(v) \sqsubset S_b(v)$, as the score of all vertices in S_b must have increased by Lemma 7.6. So in this case we compute the set $Z = F_1(v, S_b)$, put $X'(v) = \alpha.b$ for each vertex of Z, and simply run `RInitialSpine`$(S_b \setminus Z, \alpha.b)$ to compute the precise value of X' for $v \in Z$. As $\forall \beta \in V_\top^*.\alpha.b \sqsubseteq \alpha.b.\beta$, we keep all the time the invariant $X(v) \sqsubset X'(v)$ for $v \in Z$.

On the other hand if there is an upper set $X_{\geq \alpha.b}$ in X, we also start with computing the set $Z = F_1(v, S_b)$ and put $X'(v) = \alpha.b$ for each $v \in Z$. In this case we first have to show that $X(v) \sqsubseteq X'(v)$ for all vertices in Z. Let $v \in Z$. If $v \notin X_{\geq \alpha.b}$ we are finished, as $X(v) \sqsubset X'(v)$. Therefore for contradiction take $v \in X_{>\alpha.b}$ with the lowest rank in $F_1(v, S_b)$. If $v \in V_1$, we get a contradiction with O4 for the spine $\overline{X'_\alpha}$, as there must be $w \in F_1(v, S_b)$ s.t. $w \notin X_{\geq \alpha.b}$. This contradicts v being in $X_{>\alpha.b}$. The case for $v \in V_0$ is similar. Therefore $X(v) \sqsubseteq X'(v)$ for all $v \in Z = F_1(v, S_b)$.

Now we get to the important moment. By construction there are two classes of vertices in S_b: Those which were in the set $X_{\geq \alpha.b}$ and the ones which came to this set during cycle breaking or recomputation. For the latter ones we set their score in X' to $\alpha.b.\bot$ (so that now $S_b \setminus Z = X'_{>\alpha.b}$). By construction $X(v) \sqsubseteq X'(v)$ also for those vertices. We now recursively apply run the algorithm `RRecomputeEven` on $X'_{>\alpha.b}$. Again we start by running the algorithm `RecomputeEven`. This time to prove that S is a spine of $X'_{>\alpha.b}$ we need to apply a generalised version of Lemma 7.6, the Lemma 7.10. □

7.6 Asymmetric Algorithm

We are now ready to present the second, asymmetric, algorithm based on the structure of a spine. This time we work with with recursive spines. The algorithm works as follows: Given a game $\mathcal{G}$ it starts by computing an initial recursive spine X. Providing there are some even cyclic sets in X it tries to break cycles by running the procedure `RBreakEven`. If this is successful, the recursive spine is recomputed (algorithm `RRecomputeEven`). Otherwise the force set of the unbreakable cycles is removed and we start again. Finally if there are

not any even cyclic sets, all the remaining vertices are won by P_1.

Procedure `SpineAsymmetric(`$\mathcal{G}$`)`

W_0:=$\emptyset$;W_1:=$\emptyset$;V=$V(G)$
while $V \neq \emptyset$ **do**
  X:=`RInitialSpine` (V, ε)
  while *there is even cyclic set in* X **do**
    choose $\alpha.b \in Prefix(X)$ s.t. X_b is cyclic and $b \in V_0$
    (U, β, Y):=`RBreakEven` $(X_b, \alpha.b, X)$
    if $U = \emptyset$ **then**
      X:=`RRecomputeEven` (Y, α)
    else
      W_0:=$W_0 \cup F_0(U)$
      V:=$V \setminus F_0(U)$
      break
  W_1:=$W_1 \cup \{U\}$; **break**
return W_0,W_1

Theorem 7.2. *Let* $\mathcal{G}$ *be a parity game and* $n = |V(G)|$. *Then* `SpineAsymmetric(`$\mathcal{G}$`)`=(W_0, W_1) *and* `SpineAsymmetric` *stops after at most* $O(n^n)$ *iterations.*

Proof. The correctness of the algorithm follows from the Lemmas 7.4, 7.10, and 7.11. The bound on the number of iterations is given by the fact that in every switch we do not decrease the score for any of the vertices, and increase the score for at least one vertex. □

The remaining question is the complexity of a single iteration. The running time of the procedure `RBreakEven` is governed by the by the number of calls to the procedure `BreakEven` in the first part of the algorithm and `RecomputeEven` it the procedure `RRecomputeEven`. Obviously `BreakEven` is called at most n times as $U \subseteq V$. `RecomputeEven` is called at most n times as each vertex of V can be a base vertex only once. Therefore the complexity of a single iteration is $O(n^5)$.

Chapter 8

Concluding Remarks

The original goal the author of this thesis set out to reach was to come up with a polynomial algorithm for solving parity games. Even though this has not been achieved, he still thinks the odds that there is a polynomial algorithm are high. One reason to feel this way is the existence of the strategy improvement algorithm of Vöge and Jurdziński [VJ00]. Even though the understanding of this algorithm has advanced a bit, we are still a long way from showing it is actually polynomial or finding an example of a parity game on which it would need an exponential number of improvement steps. But even experimenting with large sets of examples neither we, nor others, have not been able to find one on which the algorithm needs significantly more than n steps, where n is the number of vertices in the parity game. That suggests that we can learn a great deal more by continuing research in this direction.

We think that one way forward in the search for a polynomial-time algorithm (or proving its non-existence) is to study the effect of the structure of the game graph on the shape of the winning regions. In this thesis we pursued this idea in Chapter 7. But the question is more general. Take the specific case of strategy improvement. We still do not know how does the structure of the game graph affect the set of available improvements steps. For example is it true that on graphs of bounded tree-width the number of iterations is guaranteed to be polynomial?

While we were not able to prove the (non-)existence of polynomial-time algorithm for solving parity games, we may study parity games on restricted classes of graphs. One of the best known and well studied classes of graphs are graphs of bounded tree-width, and in this thesis we gave a polynomial

time algorithm for solving parity games on this class. As tree-width is defined for undirected graphs only, it is often too coarse for describing the structure of directed graphs. When we tried to find a corresponding counterpart of tree-width for directed graphs we realized there is not any suitable available. We therefore defined the DAG-width and presented the result of [BDHK06] that we can indeed solve parity games in polynomial time also on this class of graphs.

Both tree-width and DAG-width are measures which measure how little the graph is connected – graphs of low tree-width are 'almost trees', while graphs of low DAG-width are 'almost DAGs'. On the opposite end of the spectrum is a measure called clique-width (introduced in [ER97] and studied in detail in [CO00]), which measures how close a given graph is to being a clique or, more precisely, to being a complete bipartite graph. This measure comes in two flavours, both for directed and undirected graphs. An interesting question is whether we can solve parity games in polynomial time on graphs of bounded (directed) clique-width. At first sight this seems not to be much difficult – one would just apply the techniques used for parity games on graphs of bounded tree-width and DAG-width. However they are not directly applicable. The problem here is that the clique-width of a graph is defined, instead by giving a decomposition, as the minimum number of labels needed to construct G using the following operations: creation of a new vertex v with label i (denoted $v(i)$), disjoint union ($\oplus$), connecting vertices with specified labels ($\eta_{i,j}$) and renaming labels ($\rho_{i,j}$). Unfortunately the operation $\eta_{i,j}$ can connect a great number of edges at a time, which makes any naïve algorithm exponential. However we have some plausible ideas how to construct a polynomial-time algorithm, which we have not pursued for a lack of time.

Bibliography

[ACP87] S. Arnborg, D.G. Corneil, and A. Proskurowski. Complexity of finding embedings in a k-tree. *SIAM J. Alg. Disc. Meth.*, 8:277–284, 1987.

[Adl] I. Adler. Directed tree-width example. to appear in Journal of Combinatorial Theory, Series B.

[AKS04] M. Agrawal, N. Kayal, and N. Saxena. Primes is in p. *Annals of Mathematics*, 160(2):781–793, 2004.

[BD02] P. Bellenbaum and R. Diestel. Two short proofs concerning tree-decompositions. *Combinatoric, Probability, Computing*, 11:1–7, 2002.

[BDHK06] D. Berwanger, A. Dawar, P. Hunter, and S. Kreutzer. DAG-width and parity games. In *STACS'06*, volume 3884 of *LNCS*, pages 524–536. Springer-Verlag, 2006.

[BG04] D. Berwanger and E. Grädel. Entanglement – a measure for the complexity of directed graphs with applications to logic and games. In *LPAR 2004*, volume 3452 of *LNCS*, pages 209–223. Springer-Verlag, 2004.

[Bod93] H. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In *Proc. 25th STOC*, pages 226–234, 1993.

[Bod97] H. Bodlaender. Treewidth: Algorithmic techniques and results. In *MFCS'97*, volume 1295 of *LNCS*, pages 19–36, 1997.

[BSV03] H. Björklund, S. Sandberg, and S. Vorobyov. A discrete subexponential algorithm for parity games. In *STACS 2003*, volume 2607 of *LNCS*, pages 663–674. Springer-Verlag, 2003.

[BSV04] H. Björklund, S. Sandberg, and S. Vorobyov. Memoryless determinacy of parity and mean payoff games: a simple proof. *Theoretical Computer Science*, 310:365–378, 2004.

[Büc60] J. R. Büchi. Weak second-order arithmetic and finite automata. *Zitschrift für matematische Logik and Grundlagen der Mathematik*, 6:66–92, 1960.

[Büc62] J. R. Büchi. On a decision method in restricted second order aithmetic. In *International Congress on Logic, Methodology and Philosophy of Science*, pages 1–11. Stanford University Press, 1962.

[Büc77] J. R. Büchi. *Fundamentals of Computation Theory*, volume 56 of *LNCS*, chapter Using determinacy to elminate quantifiers, pages 367–378. Springer-Verlag, 1977.

[CO00] B. Courcelle and S. Olariu. Upper bounds to the clique width of graphs. *Discrete Appl. Math.*, 101(1-3):77–114, 2000.

[Con92] A. Condon. The complexity of stochastic games. *Information and Computation*, 96(2):203–224, 1992.

[Cou90] B. Courcelle. Graph rewriting: An algebraic and logic approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, pages 193–242. Elsevier, Amsterdam, 1990.

[EJ88] E. A. Emerson and C. S. Jutla. The complexity of tree automata and logics of programs. In *Proceedings of FoCS'88*, pages 328–337, 1988.

[EJ91] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy. In *Proc. 5th IEEE Foundations of Computer Science*, pages 368–377, 1991.

[EJS93] E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model-checking for fragments of mu-calculus. In *CAV 93*, volume 697 of *LNCS*, pages 385–396. Springer-Verlag, 1993.

[EL86] E. A. Emerson and C. L. Lei. Efficient model checking in fragments of the propositional μ–calculus. In *Symposion on Logic in Computer Science*, pages 267–278. IEEE Computer Society Press, June 1986.

[Elg61] C. C. Elgot. Decision problems of finite automata design and related arithmetics. *Transactions of the American Mathematical Society*, 98:21–51, 1961.

[EM79] A. Ehrenfeucht and J. Mycielski. Positional strategies for mean payoff games. *International Journal of Game Theory*, 8:109–113, 1979.

[ER97] J. Engelfriet and G. Rozenberg. *Handbook of Graph Grammars*, chapter Node Replacement Graph Grammars, pages 1–94. World Scientific, 1997.

[FG02] M. Frick and M. Grohe. The complexity of first-order and monadic second-order logic revisited. In *LICS 2002*, pages 215–224. IEEE Computer Society, 2002.

[GH82] Y. Gurevich and L. Harrington. Trees, automata and games. In *Proceedings of STOC'82*, pages 60–65. ACM Press, 1982.

[GMT02] J. Gustedt, O. Mæhle, and J. A. Telle. The treewidth of Java programs. In *Proceedings of ALENEX'02- 4th Workshop on Algorithm Engineering and Experiments*, volume 2409 of *LNCS*. Springer-Verlag, 2002.

[GTW02] E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games*, volume 2500 of *LNCS*. Springer-Verlag, 2002.

[Hal04] N. Halman. *Discrete and Lexicographic Helly Theorems and Their Relations to LP-type Problems*. PhD thesis, Tel-Aviv University, 2004.

[Her89] B. Herwig. *Zur Modelltheorie von L_μ*. PhD thesis, Universität Freiburg, Germany, 1989.

[HK66] A.J. Hoffman and R.M. Karp. On nonterminating stochastic games. *Management Science*, 12(5):359–370, 1966.

[JPZ06] M. Jurdziński, M. Paterson, and U. Zwick. A deterministic subexponential algorithm for solving parity games. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, pages 117–123. ACM-SIAM, 2006.

[JRST01] T. Johnson, N. Robertson, P. D. Seymour, and R. Thomas. Directed tree-width. *Journal of Combinatorial Theory, Series B*, 82(1):138–154, 2001.

[JRST02] T. Johnson, N. Robertson, P. D. Seymour, and R. Thomas. Addendum to "Directed Tree-Width". `http://www.math.gatech.edu/~thomas/PAP/diradd.pdf`, 2002.

[Jur98] M. Jurdziński. Deciding the winner in parity games is in UP ∩ co-UP. *Information Processing Letters*, 68(3):119–124, 1998.

[Jur00] M. Jurdziński. Small progress measures for solving parity games. In *STACS 2000*, volume 1770 of *LNCS*, pages 290–301. Springer-Verlag, 2000.

[Kal92] G. Kalai. A subexponential randomized simplex algorithm. In 24^{th} *ACM STOC*, pages 475–482, 1992.

[KK91] N. Klarund and D. Kozen. Rabin measures and their applications to fairness and automata theory. In *LICS 1991*, pages 256–265, 1991.

[Kla91] N. Klarund. Progress measures for complementation of ω-automata with applications to temporal logic. In *FOCS 1991*, pages 358–367, 1991.

[Kla94] N. Klarund. Progress measures, immediate determinacy and a subset construction for tree automata. *Annals of Pure and Applied Logic*, 69(2-3):243–268, 1994.

[Klo94] T. Kloks. *Treewidth – computations and approximations*, volume 842 of *LNCS*. Springer-Verlag, 1994.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[KST93] J. Kbler, U. Schning, and J. Torn. *The Graph Isomorphism Problem: Its Structural Complexity*. Birkhuser, 1993.

[KvM99] A. Klivans and D. van Melkebeek. Graph nonisomorphism has subexponential size proofs unless the polynomial hierarchy collapses. In *Proceedings of ACM STOC'99*, pages 659–667, 1999.

[LBC+94] D. E. Long, A. Browne, E. M. Clarke, S. Jha, and W. R. Marrero. An improved algorithm for the evaluation of fixpoint expressions. In *CAV '94*, volume 818 of *LNCS*, pages 338–350. Springer-Verlag, 1994.

[Lud95] W. Ludwig. A subexponential randomized algorithm for the simple stochastic game problem. *Information and Computation*, 117:151–155, 1995.

[Mad97] A. Mader. *Verification of Modal Properties Using Boolean Equation Systems*. Edition versal 8, Bertz Verlag, Berlin, 1997.

[Mar75] D.A. Martin. Borel determinacy. *Annals of Mathematics*, 102:363–371, 1975.

[MC94] M. Melekopoglou and A. Condon. On the complexity of the policy improvement algorithm for markov decision processes. *ORSA Journal of Computing*, 6(2):188–192, 1994.

[McN66] R. McNaughton. Testing and generating sequences by a finite automaton. *Information and Control*, 9(5):521–530, 1966.

[McN93] R. McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65:149–184, 1993.

[Mos84] A. W. Mostowski. *Computation Theory*, chapter Regular expressions for infinite trees and standard form of automata, pages 157–168. LNCS. Springer-Verlag, 1984.

[Mos91] A. W. Mostowski. Games with forbidden positions. Technical Report 78, Instytut Matematyki, Uniwersytet Gdański, Poland, 1991.

[MS99] Y. Mansour and S. Singh. On the complexity of policy iteration. In *UAI'99*, pages 401–408, 1999.

[MSW96] J. Matoušek, M. Sharir, and E. Welzl. A subexponential bound for linear programming. *Algorithmica*, 16(4/5):498–516, 1996.

[Mul63] D. E. Muller. Infinite sequences and finite machines. In *Proceedings of the 4th IEEE Symposium on Switching Circuit Theory and Logical Design*, pages 3–16, 1963.

[MV99] P. B. Miltersen and N. V. Vinodchandran. Derandomizing arthur-merlin games using hitting sets. In *Proceedings of IEEE FOCS'99*, pages 71–80, 1999.

[Niw] D. Niwiński. Personnal communication.

[Niw86] D. Niwiński. On fixed-point clones (extended abstract). In *ICALP'86*, volume 226 of *LNCS*, pages 464–473. Springer-Verlag, 1986.

[Niw97] D. Niwiński. Fixed point characterization of infinite behavior of finite-state systems. *Theoretical Computer Science*, 189(1-2):1–69, 1997.

[Obd03] J. Obdržálek. Fast mu-calculus model checking when tree-width is bounded. In *CAV 2003*, volume 2725 of *LNCS*, pages 80–92. Springer-Verlag, 2003.

[Obd06] J. Obdržálek. DAG-width – connectivity measure for directed graphs. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, pages 814–821. ACM-SIAM, 2006.

[Pap94] C. H. Papadimitriou. *Complexity Theory*. Addison-Wesley, 1994.

[Pra75] V. R. Pratt. Every prime has a succinct certificat. *SIAM Journal on Computing*, 4:214–220, 1975.

[Pur95] Anuj Puri. *Theory of Hybrid Systems and Discrete Event Systems*. PhD thesis, University of California at Berkeley, 1995.

[PV01] V. Petersson and S. Vorobyov. A randomized subexponential algorithm for parity games. *Nordic Journal of Computing*, 8:324–345, 2001.

[Rab69] Michael O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.

[Rab72] M. Rabin. *Automata on infinite objects and Church's problem*. American Mathematical Society, 1972.

[Ree99] B. Reed. Introducing directed tree width. *Electronic Notes in Discrete Mathematics*, 3:222–229, May 1999.

[RS84] N. Robertson and P. D. Seymour. Graph Minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B*, 36:49–63, 1984.

[Saf05] M. Safari. D-width: A more natural measure for directed tree-width. In *MFCS'05*, volume 3618 of *LNCS*, pages 745–756. Springer-Verlag, 2005.

[SE89] R. S. Streett and E. A. Emerson. An automata theoretic decision procedure for the propositional mu-calculus. *Information and Computation*, 81(3):249–264, 1989.

[Sei96] H. Seidl. Fast and simple nested fixpoints. *Information Processing Letters*, 59(3):303–308, 1996.

[Ser] O. Serre. Personnal communication.

[Ser03] O. Serre. Personnal communication. 2003.

[Sha53] L. S. Shapley. Stochastic games. *Proc. Nat. Acad Sci. U.S.A.*, 39:1095–1100, 1953.

[ST93] P. D. Seymour and R. Thomas. Graph searching and a min-max theorem for tree-width. *Journal of Combinatorial Theory, Series B*, 58(1):22–33, 1993.

[Sti95] C. Stirling. Local model checking games. In *CONCUR '95*, volume 962 of *LNCS*, pages 1–11. Springer-Verlag, 1995.

[Sti01] C. Stirling. *Modal and Temporal Properties of Processes*. Texts in Computer Science. Springer-Verlag, 2001.

[Str82] R. S. Streett. Propositional dynamic logic of looping and converse is elementary decidable. *Information and Control*, 54(1–2):121–141, 1982.

[Tho95] W. Thomas. On the synthesis of strategies in infinite games. In *STACS 1995*, volume 900 of *LNCS*, pages 1–13. Springer-Verlag, 1995.

[Tho97] W. Thomas. *Languages, Automata and Logic*, volume 3, chapter 7, pages 389–456. Springer-Verlag, 1997.

[Tho98] M. Thorup. All structured programs have small tree-width and good register allocation. *Information and Computation*, 142(2):159–181, 1998.

[VJ00] J. Vöge and M. Jurdziński. A discrete strategy improvement algorithm for solving parity games. In *CAV 2000*, volume 1855 of *LNCS*, pages 202–215. Springer-Verlag, 2000.

[Wal96] I. Walukiewicz. Pushdown processes: games and model checking. In *Proceedings of CAV 96*, volume 1256 of *LNCS*, 1996.

[Zie98] W. Zielonka. Infinite games on finitely coloured graphs. *Theoretical Computer Science*, 200:135–183, 1998.

[ZP96] U. Zwick and M. S. Paterson. The complexity of mean payoff games on graphs. *Theoretical Computer Science*, 158(1–2):343–359, 1996.