17 Two-way Tree Automata Solving Pushdown Games

Thierry Cachat

Lehrstuhl für Informatik VII
RWTH Aachen

17.1 Introduction

Parity games (where the winner is determined by the parity of the maximal priority appearing infinitely often) were presented in Chapter 2 and algorithms solving parity games for the case of finite graphs in Chapter 7. In this paper we study parity games on a simple class of infinite graphs: the pushdown (transition) graphs. In [106], Kupferman and Vardi have given a very powerful method for the μ-calculus model checking of these graphs: the formalism of two-way alternating tree automata. This is a generalization of the (one-way) tree automata presented in Chapters 8 and 9.

The transition graph of the pushdown automaton defines the arena: the graph of the play and the partition of the vertex set needed to specify the parity winning condition. We know from Chapter 6 that such games are determined and that each of both players has a memoryless winning strategy on his winning region. The aim of this paper is to show how to compute effectively the winning region of Player 0 and a memoryless winning strategy. The idea of [106] is to simulate the pushdown system in the full W-tree, where W is a finite set of directions, and to use the expressive power of alternating two-way tree automata to answer these questions. Finally it is necessary to translate the 2-way tree automaton into an equivalent nondeterministic one-way tree automaton, with the construction from [190].

In the next section we define two-way alternating automata and the effective construction from [190] of equivalent one-way nondeterministic automata. In Section 17.3 we apply these results to solve parity games over pushdown graphs and to compute winning strategies. Section 17.4 presents an example. Some extensions and modifications are discussed in Section 17.5.

17.2 Reduction 2-way to 1-way

The formalism of alternating two-way parity tree automata is very "powerful", but we cannot handle directly these automata to answer our questions of nonemptiness (for the winning region) and strategy synthesis. We need the reduction presented in this section, which constructs step by step a one-way nondeterministic tree automaton that is equivalent to a given two-way alternating automaton $\mathcal{A}$, in the sense that they accept the same set of trees (finite or infinite).

17.2.1 Definition of Two-way Automata

Given a finite set W of directions, a **W-tree** is a prefix closed set $T \subseteq W^*$, *i.e.*, if $x.d \in T$, where $x \in W^*$ and $d \in W$, then also $x \in T$. We will sometimes forget the "." of the concatenation. The elements of T are called **nodes**, the empty word ϵ is the **root** of T. For every $x.d \in T, d \in W$ the node x is the unique **parent** of $x.d$, and $x.d$ is a **child** of x. The **direction** of a node $x.d$ ($\neq \epsilon$) is d. The **full infinite tree** is $T = W^*$. A **path** (branch) of a tree T is a sequence $\beta \in T^\infty$ such that $\beta = u_0 u_1 \cdots u_n$ or $\beta = u_0 u_1 u_2 \cdots$, $u_0 = \epsilon$ and $\forall i < n, \exists d \in W, u_{i+1} = u_i.d$. A path can be finite or infinite.

Given two finite alphabets W and Σ, a **Σ-labeled W-tree** is a pair $\langle T, l \rangle$ where T is a W-tree and $l : T \longrightarrow \Sigma$ maps each node of T to a letter in Σ. When W and Σ are not important or clear from the context, we call $\langle T, l \rangle$ a labeled tree.

We recall that for a finite set X, $\mathcal{B}^+(X)$ is the set of positive Boolean formulas over X (*i.e.*, Boolean formulas built from elements in X using only $\wedge$ and $\vee$), where we also allow the formulas $\langle \text{true} \rangle$ and $\langle \text{false} \rangle$. For a set $Y \subseteq X$ and a formula $\theta \in \mathcal{B}^+(X)$, we say that Y **satisfies** θ iff assigning $\langle \text{true} \rangle$ to elements in Y and $\langle \text{false} \rangle$ to elements in $X \backslash Y$ makes θ true.

To navigate through the tree let $\text{ext}(W) := W \cup \{\epsilon, \uparrow\}$ be the extension of W: The symbol $\uparrow$ means "go to parent node" and ϵ means "stay on the present node". To simplify the notation we define $\forall u \in W^*, d \in W, u.\epsilon = u$ and $ud\!\uparrow = u$. The node $\epsilon\!\uparrow$ is not defined.

An **alternating two-way automaton** over Σ-labeled W-trees is a tuple $\mathcal{A} := (Q, \Sigma, \delta, q_\mathrm{I}, \mathrm{Acc})$ where

- Q is a finite set of states,
- Σ is the input alphabet,
- $\delta : Q \times \Sigma \longrightarrow \mathcal{B}^+(\text{ext}(W) \times Q)$ is the transition function,
- q_I is the initial state, and
- Acc is the acceptance condition.

The idea of a transition $\delta(q, l_1) = (\uparrow, q') \wedge (d, q'')$ is the following: if the au-

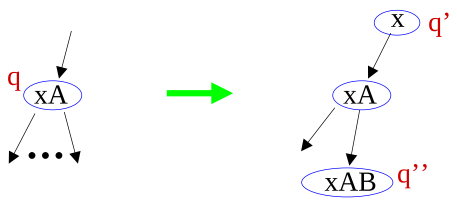


Fig. 17.1. Example of a transition $\delta(q, A) = (\uparrow, q') \wedge (B, q'')$, with the convention that the label is equal to the last letter of the node

tomaton $\mathcal{A}$ is in state q on the node x of the labeled tree $\langle T, l \rangle$ and reads the input $l_1 = l(x)$, it will send a "copy" of $\mathcal{A}$ in state q' to the parent node of x and another copy in state q'' to xd. See Figure 17.1. After that the two copies are running independently. They may come again to the same node with two different states.

More precisely a run of an alternating two-way automaton $\mathcal{A}$ over a labeled tree $\langle W^*, l\rangle$ is *another* labeled tree $\langle T_r, r\rangle$ in which every node is labeled by an element of $W^* \times Q$. The latter tree is like the unfolding of the run, its structure is quite different from W^*. A node in T_r, labeled by (x, q), describes a "copy" of the automaton that is in state q and is situated at the node x of W^*. Note that many nodes of T_r can correspond to the same node of W^*, because the automaton can come back to a previously visited node. The label of a node and its successors have to satisfy the transition function. Formally, a run $\langle T_r, r\rangle$ is a Σ_r-labeled Γ-tree, for some (almost arbitrary) set Γ of directions, where $\Sigma_r := W^* \times Q$ and $\langle T_r, r\rangle$ satisfies the following conditions:

(a) $\epsilon \in T_r$ and $r(\epsilon) = (\epsilon, q_{\mathrm{I}})$
(b) Consider $y \in T_r$ with $r(y) = (x, q)$ and $\delta(q, l(x)) = \theta$. Then there is a (possibly empty) set $Y \subseteq \mathrm{ext}(W) \times Q$, such that Y satisfies θ, and for all $\langle d, q'\rangle \in Y$, there is $\gamma \in \Gamma$ such that $y.\gamma \in T_r$ and the following holds: $r(y.\gamma) = (x.d, q')$.

Remember that $x.d$ can be $x.\epsilon$ or $x.\uparrow$, and the latter is defined only if $x \neq \epsilon$. So the run cannot go up from the root of the input tree. Note that it cannot use a transition $\delta(q, l(x)) = \langle \text{false}\rangle$ since the formula $\langle\text{false}\rangle$ cannot be satisfied.

A run $\langle T_r, r\rangle$ is accepting if all its infinite paths satisfy the acceptance condition Acc (the finite paths of a run end with a transition $\theta = \langle\text{true}\rangle$, which is viewed as successful termination). We consider here only **parity** acceptance conditions (see previous chapters): Acc is given by a priority function $\Omega : Q \longrightarrow [m]$. An infinite path $\beta \in {T_r}^{\omega}$ satisfies the acceptance condition iff the smallest priority appearing infinitely often in this path is even: $\min \mathrm{Inf}(\Omega(r(\beta)))$ is even. Such a path in the run consists of following only one "copy" of the automaton. An automaton accepts a labeled tree if and only if there exists a run that accepts it. The tree language accepted by an automaton $\mathcal{A}$ is denoted $L(\mathcal{A})$. Two automata are equivalent if they accept the same tree language.

The automaton $\mathcal{A} = (Q, \Sigma, \delta, q_{\mathrm{I}}, \Omega)$ and the input tree $\langle T, l\rangle$ are now fixed for the rest of the Section 17.2. In the next subsections we will study how the automaton $\mathcal{A}$ can accept the given tree. The strategy for $\mathcal{A}$ will give information about the transitions used by $\mathcal{A}$ (because $\mathcal{A}$ is not deterministic). Then the annotation will store the priorities seen during the detours of $\mathcal{A}$. With all these auxiliary tools, it is possible to construct a *one-way* tree automaton that checks whether $\mathcal{A}$ accepts a tree.

17.2.2 Strategy

In the same way as in Chapters 6, 4 and 8 of this book, $\mathcal{A}$ itself (as an alternating automaton) is equivalent to a two-player parity game. The initial configuration of this game is $(\epsilon, q_{\mathrm{I}})$ $(= r(\epsilon))$. From a configuration $(x, q), x \in T, q \in Q$, Player 0 chooses a set $Y \subseteq \mathrm{ext}(W) \times Q$ that satisfies $\delta(q, l(x))$, then Player 1 chooses $\langle d, q' \rangle \in Y$, the new configuration is $(x.d, q')$ and so on. If $x.d$ is not defined or $\delta(q, l(x)) = \langle \text{false} \rangle$ then Player 1 wins immediately. If Y is empty $(\delta(q, l(x)) = \langle \text{true} \rangle)$ then Player 0 wins immediately. If the play is infinite, then Player 0 wins iff the parity condition is satisfied. So Player 0 is trying to show that $\mathcal{A}$ accepts the input tree, and Player 1 is trying to challenge that.

Player 0 has a memoryless winning strategy iff $\mathcal{A}$ has an accepting run (see Chapter 6). In other words, if $\mathcal{A}$ has an accepting run, then it has an accepting run using a memoryless winning strategy: choosing always the same "transitions" from the same node and state. We decompose the run of $\mathcal{A}$ using this strategy.

Definition 17.1. A **strategy** for $\mathcal{A}$ and a given tree is a mapping

$$\tau : W^* \longrightarrow \mathscr{P}(Q \times \mathrm{ext}(W) \times Q).$$

Intuitively $(q, d, q') \in \tau(x)$ means that if $\mathcal{A}$ is in state q on the node x, it has to send a copy in state q' to node xd. It is memoryless because it depends only on x. Note that the strategy does not read the labels, but it is defined for a fixed tree $\langle T, l \rangle$. See an example on Figure 17.2.

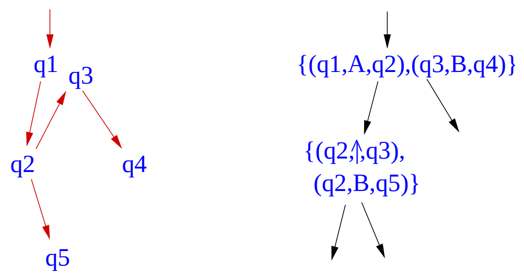


Fig. 17.2. Part of a run and the corresponding strategy

In this subsection we want to verify with a one-way automaton some simple conditions on the strategy τ of an alternating two way tree automaton $\mathcal{A}$. The first condition for a strategy to be correct (at node x) is to satisfy the transition of $\mathcal{A}$. The second condition is that the strategy can be followed: if $(q, d, q') \in \tau(x)$ then the strategy $\tau(xd)$ has to be defined in xd for the state q', such that the run can continue. Formally, both conditions are:

$$\forall x \in W^*, \forall (q, d, q') \in \tau(x):$$

$$\{\, (d_2, q_2) \mid (q, d_2, q_2) \in \tau(x) \,\} \text{ satisfies } \delta(q, l(x)), \text{ and} \tag{17.1}$$

$$\exists d_1, q_1, (q', d_1, q_1) \in \tau(xd) \text{ or } \emptyset \text{ satisfies } \delta(q', l(xd)), \tag{17.2}$$

and for the root:

$$\exists d_1, q_1, (q_{\mathrm{I}}, d_1, q_1) \in \tau(\epsilon) \text{ or } \emptyset \text{ satisfies } \delta(q_{\mathrm{I}}, l(\epsilon)). \tag{17.3}$$

Considering $St := \mathscr{P}(Q \times \mathrm{ext}(W) \times Q)$ as an alphabet, a $(St \times \Sigma)$-labeled tree defines a memoryless strategy on the corresponding Σ-labeled tree. We will construct a *one-way* automaton $\mathcal{B}$ that checks that this strategy is correct according to the previous requirements. For $(q, d, q') \in \tau(x)$, if $d \in W$ it has just to check in the direction d downwards that the strategy is well defined for q', but if $d = \uparrow$, he must have *remembered* that the strategy *was* defined for q' in the parent-node. The states of $\mathcal{B}$ are pairs $\langle Q_1, Q_2 \rangle \in \mathscr{P}(Q) \times \mathscr{P}(Q)$, where $q' \in Q_1$ means that $\mathcal{B}$ has to check (down) that the strategy can be followed for q', and $q'' \in Q_2$ means that q'' is already allowed at the parent node.

$$\mathcal{B} := (\mathscr{P}(Q) \times \mathscr{P}(Q), St \times \Sigma, \delta_{\mathcal{B}}, \langle \{q_{\mathrm{I}}\}, \emptyset \rangle, \langle \mathrm{true} \rangle) \text{ where} \tag{17.4}$$

$$\delta_{\mathcal{B}}(\langle Q_1, Q_2 \rangle, \langle \tau_1, l_1 \rangle) :=$$

$$\text{IF } \forall q \in Q_1,\ \{ (d_2, q_2) \mid (q, d_2, q_2) \in \tau_1 \} \text{ satisfies } \delta(q, l_1), \text{ and} \tag{17.5}$$

$$\forall (q', \epsilon, q) \in \tau_1,\ \{ (d_2, q_2) \mid (q, d_2, q_2) \in \tau_1 \} \text{ satisfies } \delta(q, l_1), \text{ and} \tag{17.6}$$

$$\forall (q, \uparrow, q') \in \tau_1,\ q' \in Q_2 \tag{17.7}$$

$$\text{THEN } \bigwedge_{d \in W} \big(d, \langle \{ q' \mid \exists\ (q, d, q') \in \tau_1 \}, Q_2' \rangle \big) \tag{17.8}$$

$$\text{with } Q_2' := \{ q'' \mid \exists\ d_1, q_1,\ (q'', d_1, q_1) \in \tau_1 \text{ or } \emptyset \text{ satisfies } \delta(q'', l_1) \}, \tag{17.9}$$

$$\text{ELSE } \langle \mathrm{false} \rangle. \tag{17.10}$$

The acceptance condition is easy to enunciate: it just requires that each path of $\mathcal{B}$ is infinite (*i.e.*, the transition is possible at each node). Note that although we have used the formalism of alternating automata, $\mathcal{B}$ is a *deterministic* one-way automaton: $\mathcal{B}$ sends exactly one copy to each son of the current node. It has $4^{|Q|}$ states.

In condition (17.5) there is no requirement on the $q \notin Q_1$, that's why the condition (17.1) above is stronger. This is not a problem for the following, as we are searching *some* winning strategy (one could define the *minimal* valid strategy as in [190]). If $\mathcal{A}$ follows the strategy, its run is "deterministic" on the input tree labeled by $St \times \Sigma$.

A **path** β in a strategy (tree) τ is a sequence $(u_0, q_0), (u_1, q_1), (u_2, q_2), \cdots$ of pairs from $W^* \times Q$ such that $(u_0, q_0) = (\epsilon, q_{\mathrm{I}})$ and for all $i \geqslant 0$, there is some $c_i \in \mathrm{ext}(W)$ such that $(q_i, c_i, q_{i+1}) \in \tau(u_i)$ and $u_{i+1} = u_i c_i$. Thus, β just follows (nondeterministically) the "transitions" of τ. The parity condition for β is defined exactly the same way as for a path of (a run of) $\mathcal{A}$. We say that τ is accepting if all infinite paths in τ are accepting.

Proposition 17.2. *A two-way alternating parity automaton accepts an input tree iff it has an accepting strategy tree over the input tree.*

With the help of a so called annotation, we will check in the following subsections whether a strategy is accepting.

17.2.3 Annotation

The previous automaton $\mathcal{B}$ just checks that the strategy can be followed (ad infinitum) but forgets the priorities of $\mathcal{A}$. To check the acceptance condition, it is necessary to follow each path of $\mathcal{A}$ up and down, and remember the priorities appearing. Such a path can be decomposed into a *downwards* path and several finite *detours* from the path, that come back to their origin (in a loop). Because each node has a unique parent and $\mathcal{A}$ starts at the root, we consider only downwards detour (each move $\uparrow$ is in a detour). That is to say, if a node is visited more than once by a run β, we know that the first time it was visited, the run came from above. To keep track of these finite detours, we use the following annotation.

Definition 17.3. An **annotation** for $\mathcal{A}$ and a given tree is a mapping

$$\eta : W^* \longrightarrow \mathscr{P}(Q \times [m] \times Q). \tag{17.11}$$

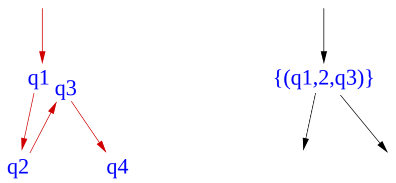


Fig. 17.3. Part of a run and the corresponding annotation, assuming that $\Omega(q_2) = 2, \Omega(q_3) = 3$

Intuitively $(q, f, q') \in \eta(x)$ means that from node x and state q there is a detour that comes back to x with state q' and the smallest priority seen along this detour is f. Figure 17.3 presents an example. By definition, the following conditions are required for the annotation η of a given strategy τ:

$$\forall\ q, q' \in Q,\ x \in W^*,\ d \in W,\ f, f' \in [m]:$$

$$(q, \epsilon, q') \in \tau(x) \Rightarrow (q, \Omega(q'), q') \in \eta(x), \tag{17.12}$$

$$(q_1, f, q_2) \in \eta(x), (q_2, f', q_3) \in \eta(x) \Rightarrow (q_1, \min(f, f'), q_3) \in \eta(x), \tag{17.13}$$

$$(q, d, q_1) \in \tau(x), (q_1, \uparrow, q') \in \tau(xd) \Rightarrow (q, \min(\Omega(q_1), \Omega(q')), q') \in \eta(x), \tag{17.14}$$

$$(q, d, q_1) \in \tau(x), (q_1, f, q_2) \in \eta(xd),\ (q_2, \uparrow, q') \in \tau(xd) \Rightarrow (q, \min(\Omega(q_1), f, \Omega(q')), q') \in \eta(x). \tag{17.15}$$

Considering $An := \mathscr{P}(Q \times [m] \times Q)$ as an alphabet, the aim is to construct a *one-way* automaton $\mathcal{C}$ on $(An \times St)$-labeled trees that checks that the annotation satisfies these requirements. The conditions 17.12 and 17.13 above can be checked in each node (independently) without memory. For the last two, the automaton has to remember the whole $\eta(x)$ from the parent node x, and the part of $\tau(x)$ leading to the current node.

$$\mathcal{C} := (An \times \mathscr{P}(Q \times Q), An \times St, \delta_{\mathcal{C}}, \langle \emptyset, \emptyset \rangle, \langle \text{true} \rangle),$$

where

$$\begin{aligned}
&\delta_{\mathcal{C}}(\langle \eta_0, \alpha \rangle, \langle \eta_1, \tau_1 \rangle) := \\
&\text{IF conditions 17.12 and 17.13 hold for } \eta_1 \text{ and } \tau_1 \text{ AND} \\
&\forall (q, q_1) \in \alpha, (q_1, \uparrow, q') \in \tau_1 \;\Rightarrow\; (q, \min(\Omega(q_1), \Omega(q')), q') \in \eta_0 \\
&\forall (q, q_1) \in \alpha, (q_1, f, q_2) \in \eta_1, (q_2, \uparrow, q') \in \tau_1 \\
&\qquad\qquad \Rightarrow\; (q, \min(\Omega(q_1), f, \Omega(q')), q') \in \eta_0 \\
&\text{THEN } \bigwedge_{d \in W} \big(d, \langle \eta_1, \{ (q, q_1) \mid \exists\, (q, d, q_1) \in \tau_1 \} \rangle\big) \\
&\text{ELSE } \langle \text{false} \rangle.
\end{aligned}$$

Similarly to $\mathcal{B}$, $\mathcal{C}$ is a *deterministic* one-way automaton with $2^{|Q|^2 m} \cdot 2^{|Q|^2} = 2^{|Q|^2(m+1)}$ states, and the acceptance condition is very simple: each path has to be infinite. Note that if a part of the tree is not visited by the original automaton $\mathcal{A}$, the strategy and annotation can be empty on this part. The automaton $\mathcal{C}$ does not check that the annotation is minimal, but this is not a problem. With the help of the annotation one can determine if a path of $\mathcal{A}$ respects the acceptance condition or not, as showed in the next subsection.

17.2.4 Parity Acceptance

Up to now the automata $\mathcal{B}$ and $\mathcal{C}$ together just check that the strategy and annotation for the run of $\mathcal{A}$ are correct, but do not verify that the run of $\mathcal{A}$ is accepting, *i.e.*, that each path is valid. With the help of the annotation we can "simulate" (follow) a path of $\mathcal{A}$ with a one-way automaton, and determine the parity condition for this path. This one-way automaton does not go into the detours, but reads the smallest priority appearing in them.

$$\begin{aligned}
&\mathcal{D} := (Q \times [m], An \times St, \delta_{\mathcal{D}}, \langle q_{\mathrm{I}}, 0 \rangle, \Omega_0), \\
&\delta_{\mathcal{D}}(\langle q, i \rangle, \langle \eta_1, \tau_1 \rangle) := \bigvee_{(q,d,q') \in \tau_1, d \in W} (d, \langle q', \Omega(q') \rangle) \;\;\vee \bigvee_{(q,f,q') \in \eta_1} (\epsilon, \langle q', f \rangle).
\end{aligned}$$

At each step $\mathcal{D}$ either goes *down* following the strategy, or simulates a detour with an ϵ-move and the corresponding priority. The second component ($[m]$) of the states of $\mathcal{D}$ just remembers the last priority seen. We can transform $\mathcal{D}$ into a nondeterministic one-way automaton $\mathcal{D}'$ without ϵ-moves with the same state space. Note that $\mathcal{D}$ can possibly stay forever in the same node by using ϵ-transitions, either in an accepting run or not. This possibility can be checked by $\mathcal{D}'$ just by reading the current annotation, with a transition $\langle$true$\rangle$ or $\langle$false$\rangle$.

We will use $\mathcal{D}$ and $\mathcal{D}'$ to find the *invalid* paths of the run of $\mathcal{A}$, just by changing the acceptance condition: $\Omega_0(\langle q, i \rangle) := i + 1$.

Proposition 17.4. *The one-way tree automaton $\mathcal{D}'$ accepts a $(An \times St)$-labeled tree iff the corresponding run of $\mathcal{A}$ is* not *accepting.*

But $\mathcal{D}'$ is not deterministic, and accepts a tree if $\mathcal{D}'$ has *some* accepting run. We can view $\mathcal{D}'$ as a word automaton: it follows just a branch of the tree. For this reason it is possible to co-determinize it: determinize and complement it in a singly exponential construction (see Chapter 8 and 9) to construct the automaton $\overline{\mathcal{D}}$ that accepts those of the $(An \times St)$-labeled trees that represent the accepting runs of $\mathcal{A}$.

We will define the product $\mathcal{E} := \mathcal{B} \times \mathcal{C} \times \overline{\mathcal{D}}$ of the previous automata, that accepts a $(An \times St \times \Sigma)$-labeled input tree iff the corresponding run of $\mathcal{A}$ is accepting. Let

$$\mathcal{E} := (Q_{\mathcal{B}} \times Q_{\mathcal{C}} \times Q_{\overline{\mathcal{D}}}, An \times St \times \Sigma, \delta_{\mathcal{E}}, q_{\mathrm{I},\mathcal{E}}, \mathrm{Acc}),$$
$$\delta_{\mathcal{E}}(\langle q_{\mathcal{B}}, q_{\mathcal{C}}, q_{\overline{\mathcal{D}}}\rangle, \langle \eta_1, \tau_1, l_1\rangle) := \langle \delta_{\mathcal{B}}(q_{\mathcal{B}}, \langle \tau_1, l_1\rangle), \delta_{\mathcal{C}}(q_{\mathcal{C}}, \langle \eta_1, \tau_1\rangle), \delta_{\overline{\mathcal{D}}}(q_{\overline{\mathcal{D}}}, \langle \eta_1, \tau_1\rangle)\rangle\,,$$

where $Q_{\mathcal{B}}$ is the state space of $\mathcal{B}$, and so on. The acceptance condition of $\mathcal{E}$ is then exactly the one of $\overline{\mathcal{D}}$.

We define the automaton $\mathcal{E}'$ to be the "projection" of $\mathcal{E}$ on the input alphabet Σ: $\mathcal{E}'$ nondeterministically guesses the labels from $An \times St$. Finally $\mathcal{E}'$ is a nondeterministic one-way tree-automaton on Σ-labeled trees that is equivalent to $\mathcal{A}$: it accepts the same trees. The strategy and annotation depended on the input tree, now after the projection, $\mathcal{E}'$ can search the run of $\mathcal{A}$ for each input tree. The automaton $\mathcal{E}$ is deterministic and has (like $\mathcal{E}'$) $4^{|Q|} \cdot 2^{|Q|^2(m+1)} \cdot 2^{c|Q|m} = 2^{|Q|^2(m+1)} \cdot 2^{|Q|(2+cm)}$ states.

Theorem 17.5 ([190])**.** *To every alternating two-way parity tree automaton $\mathcal{A}$ there exists an equivalent nondeterministic one-way tree automaton $\mathcal{E}$, in the sense that they recognize the same tree language: $L(\mathcal{A}) = L(\mathcal{E})$.*

Corollary 17.6 ([190])**.** *The emptiness problem for alternating two-way tree automata is in* Exptime.

17.3 Application: Pushdown Games

We use alternating two-way automata to solve parity games on pushdown graphs. Thanks to the previous section the results are effective.

17.3.1 Definition of the Game

We first recall the definition of two player parity games. The **arena** $\mathcal{A} := (V_0, V_1, E)$ is a graph, composed of two disjoint sets of vertices, V_0 and V_1, and a set of edges $E \subseteq V \times V$, where $V = V_0 \uplus V_1$. To define a **parity game** $\mathcal{G} := (\mathcal{A}, \Omega_{\mathcal{G}})$ we need a mapping $\Omega_{\mathcal{G}} : V \longrightarrow [m], m < \omega$ which assigns a priority to each vertex. Then the **initialized game** $(\mathcal{G}, v_{\mathrm{I}})$ is given with an initial vertex $v_{\mathrm{I}} \in V$.

A **play** of $(\mathcal{G}, v_{\mathrm{I}})$ proceeds as follows:

(a) $v_0 = v_{\mathrm{I}}$ is the first "current state" of the play,
(b) from state $v_i, i \geqslant 0$, if $v_i \in V_0$ (resp. $v_i \in V_1$) then Player 0 (resp. Player 1) chooses a successor $v_{i+1} \in v_i E$, which is the new current state.

The play is then the sequence $\pi = v_0 v_1 \cdots \in V^\infty$. This sequence is maximal: it is finite only if no more move is possible. We consider min-parity games:

$$\text{Player 0 wins } \pi \text{ iff } \min \mathrm{Inf}(\Omega_{\mathcal{G}}(\pi)) \text{ is even.}$$

These definitions are essentially the same for finite and infinite arena. We consider now pushdown graphs: (V, E) is the (possibly infinite) transition graph of a pushdown system, which is an unlabeled pushdown automaton.

Definition 17.7. A **pushdown system** (PDS) is a tuple $\mathcal{Z} := (P, W, \Delta)$ where:

(a) P is a finite set of (control) states,
(b) W is a finite (stack) alphabet,
(c) $\Delta \subseteq P \times W \times P \times W^*$ is a *finite* transition relation.

A stack content is a word from W^*. Unlike standard notation we write the top of the stack at the right of the word (we are considering suffix rewriting as in Chapter 15). A **configuration** is a stack content and a control state: (w, p), shortly wp, where $w \in W^*, p \in P$. The **transition graph** of $\mathcal{Z}$ is (V, E) where $V = W^*P$ is the whole set of configurations and $\forall u, w \in W^*, \forall a \in W, \forall p, p' \in P$

$$(uap)E(uwp') \Leftrightarrow (p, a, p', w) \in \Delta.$$

This defines a vertex labeled graph: each vertex is labeled by his name, the edges have no label. We use the name pushdown system, like in [61] because the transitions are not labeled: we are not interested in the language recognized by the pushdown automaton but in the underlying transition graph. To obtain a parity game, it remains to define the sets V_0 and V_1, associating the vertices to the two players, and the priorities of the configurations. One fixes a disjoint union $P = P_0 \uplus P_1$, then $V_0 = W^*P_0$ and $V_1 = W^*P_1$. The mapping $\Omega_{\mathcal{G}}$ is first defined on P, then $\Omega_{\mathcal{G}}(wp) = \Omega_{\mathcal{G}}(p), \forall w \in W^*$ and $p \in P$. So the player and the priority only depend on the control states of $\mathcal{Z}$, like in [196] and [198]. These restrictions will be discussed later in Section 17.5.1.

The **pushdown game** is completely defined if we also fix an initial configuration $v_{\mathrm{I}} \in V$: $v_{\mathrm{I}} = w_{\mathrm{I}} p_{\mathrm{I}}$.

17.3.2 Winning Region

We construct an alternating two-way automaton $\mathcal{A}$ that determines if Player 0 can win the game $(\mathcal{G}, v_{\mathrm{I}})$, *i.e.*, wins every play, whatever Player 1 does. The automaton $\mathcal{A}$ will simulate the transitions of the pushdown system $\mathcal{Z}$ on the full W-tree, guess nondeterministically the best moves of Player 0 and follow each possible move of Player 1 using alternation.

As an example, the transition $(p, a, p', bc) \in \Delta$ from a configuration uap of the pushdown system can be simulated by a two-way automaton over the full W-tree from the node ua by the following sequence of moves: $\uparrow, b, c$ because $ua{\uparrow}bc = ubc$. We have chosen suffix rewriting rather than prefix to conform with the notation of the tree. The control states of $\mathcal{Z}$ are represented in the states of $\mathcal{A}$.

For our particular application, we simplify the definition of two-way automata a little. The full W-tree will not be labeled by an input alphabet Σ, and the automaton will "read" the last letter of the node, almost the same way as a pushdown automaton (as remarked in [106], another solution is to check that each label is equal to the last letter of its node).

To simulate with many steps a transition of $\mathcal{Z}$, $\mathcal{A}$ has to remember in its states the letters it has to write (see Figure 17.4). Let

$$\mathrm{tails}(\Delta) := \{\, u \in W^* \mid \exists v, a, p, p' \; (p, a, p', vu) \in \Delta \vee vu = v_{\mathrm{I}} \,\},$$

$$\mathcal{A} := (P \times \mathrm{tails}(\Delta), W, \delta, \langle p_{\mathrm{I}}, v_{\mathrm{I}} \rangle, \Omega),$$

$$\forall\, b, l_1 \in W, x \in W^*, p \in P:$$

$$\delta_{\mathcal{A}}(\langle p, b.x\rangle, l_1) := (b, \langle p, x\rangle), \tag{17.16}$$

$$\delta_{\mathcal{A}}(\langle p, \epsilon\rangle, l_1) := \bigvee_{(p,l_1,p',w)\in\Delta} (\uparrow, \langle p', w\rangle) \quad \text{if } p \in P_0, \tag{17.17}$$

$$\delta_{\mathcal{A}}(\langle p, \epsilon\rangle, l_1) := \bigwedge_{(p,l_1,p',w)\in\Delta} (\uparrow, \langle p', w\rangle) \quad \text{if } p \in P_1. \tag{17.18}$$

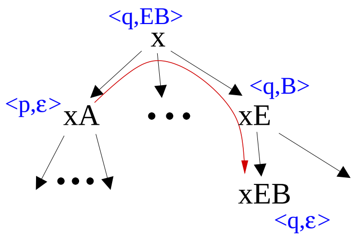


Fig. 17.4. Transition (p, A, q, EB) of the PDS (long arrow) simulated by the two-way automaton

From a state $\langle p, bx\rangle$ ($bx \neq \epsilon$ because $b \in W$) the automaton goes down in direction b, that is to say writes b, whatever it reads, and remembers the (sub)word x it still has to write and the state p of the pushdown system. These intermediate states just simulate a transition of $\mathcal{Z}$, they do not correspond to a particular configuration of the game. Only a state $\langle p, \epsilon\rangle$ on a node $w \in W^*$ corresponds to a configuration wp of the game. If $p \in P_1$ then $wp \in V_1$ and $\mathcal{A}$ executes all the possible moves of Player 1, to ensure that Player 0 can win after each of these moves. But if $p \in P_0$, $\mathcal{A}$ chooses nondeterministically a move of Player 0 and tries to make Player 0 win.

The "winning" condition of $\mathcal{A}$ is almost the same as the one of $\mathcal{G}$: $\Omega(\langle p, x\rangle) = \Omega_{\mathcal{G}}(p)$. The initial state of $\mathcal{A}$ causes it to go "deterministically" to the initial configuration of the game.

Theorem 17.8. *Player 0 has a winning strategy in $(\mathcal{G}, v_{\mathrm{I}})$ iff $\mathcal{A}$ accepts the full infinite tree W^*.*

Proof. By definition, $\mathcal{A}$ accepts W^* iff there exists an accepting run $\langle T_r, r\rangle$. Each path in $\langle T_r, r\rangle$ describes a play of the game, with the same winning condition. If each path is accepting, then each play is winning for Player 0, and every possible answer of Player 1 is in $\langle T_r, r\rangle$. That describes a winning strategy for Player 0. Conversely a winning strategy for Player 0 determines a tree of all the plays that follow it, which is an accepting run for $\mathcal{A}$.

These strategies are not necessarily memoryless as presented, but the result of Chapter 6 holds for both formalisms: there is a memoryless winning strategy if there is a winning strategy.

17.3.3 Winning Strategy

Once the automaton $\mathcal{E}$ of Theorem 17.5 is defined, we know from Chapter 8, Theorem 8.19, that we can solve the emptiness problem and generate a regular tree in $L(\mathcal{E})$ if $L(\mathcal{E}) \neq \emptyset$. Implicitly in that tree the states of $\mathcal{E}$ describe a strategy for $\mathcal{A}$ (Section 17.2.2), *i.e.*, for the game $(\mathcal{G}, v_{\mathrm{I}})$. If we follow a path (branch) of that tree, $\mathcal{E}$ corresponds to a deterministic word automaton $\mathcal{F}$ that can output the moves of Player 0. Finally $\mathcal{F}$ defines a memoryless winning strategy for Player 0 in $(\mathcal{G}, v_{\mathrm{I}})$ under the assumption that $L(\mathcal{E}) \neq \emptyset$, *i.e.*, if Player 0 wins the game.

More precisely $\mathcal{F}$ accepts all configurations in the winning region connected to v_{I} and each final state of $\mathcal{F}$ is labeled by a move of player 0, such that the strategy defined in this way is winning. This result from [106] is stronger (and more general, see Section 17.5.2) than the result of [196] that prove the existence of a pushdown strategy. The finite automaton $\mathcal{F}$ can easily be simulated with a pushdown automaton that defines a strategy like in [196].

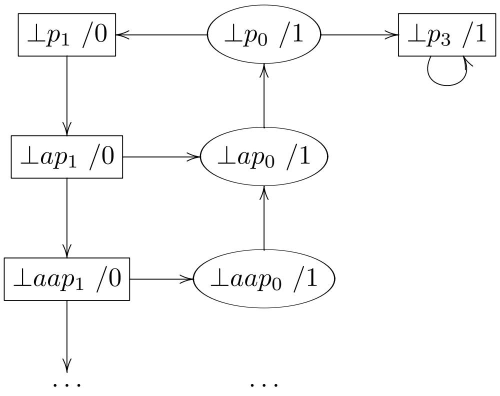
Since we have considered an initial configuration v_{I}, the previous results do not define the memoryless winning strategy over the whole winning region of Player 0, but only over the nodes that can be reached by the play starting from v_{I}.

17.4 Example

We present here a simple example of pushdown game to illustrate the results of this chapter. Using notations of section 17.3, let

$$\begin{aligned}
&W = \{a, \perp\}, P_0 = \{p_0\}, P_1 = \{p_1, p_3\},\\
&\Delta = \{(p_1, \perp, p_1, \perp a), (p_1, a, p_1, aa), (p_1, a, p_0, a), (p_0, a, p_0, \epsilon), (p_0, \perp, p_1, \perp),\\
&\qquad (p_0, \perp, p_3, \perp), (p_3, \perp, p_3, \perp)\},\\
&\Omega_G(p_1) = 0, \Omega_G(p_0) = \Omega_G(p_3) = 1.
\end{aligned}$$

The game graph looks as follows:



We consider the initial position $q_{\mathrm{I}} = \langle p_{\mathrm{I}}, v_{\mathrm{I}} \rangle = \langle p_1, \bot \rangle$. For the automaton $\mathcal{A}$ we get tails$(\Delta) = \{\epsilon, \bot, \bot a, a, aa\}$. Transitions are

$$\delta_{\mathcal{A}}(\langle p_1, \epsilon \rangle, a) := (\uparrow, \langle p_1, aa \rangle) \wedge (\uparrow, \langle p_0, a \rangle), \qquad \text{(Player 1)}$$
$$\delta_{\mathcal{A}}(\langle p_0, \epsilon \rangle, \bot) := (\uparrow, \langle p_3, \bot \rangle) \vee (\uparrow, \langle p_1, \bot \rangle). \qquad \text{(Player 0)}$$

As a shortcut we consider

$$\delta_{\mathcal{A}}(\langle p_1, \epsilon \rangle, a) := (\epsilon, \langle p_1, a \rangle) \wedge (\epsilon, \langle p_0, \epsilon \rangle),$$
$$\delta_{\mathcal{A}}(\langle p_0, \epsilon \rangle, \bot) := (\epsilon, \langle p_3, \epsilon \rangle) \vee (\epsilon, \langle p_1, \epsilon \rangle).$$

Other transition rules are straightforward. The strategy τ is only relevant at the node $\bot p_0$, where Player 0 has a real choice: he must go to $\bot p_1$, otherwise Player 1 wins. We put

$$\tau(\bot) = \{(p_0, \epsilon, p_1), (p_1, a, p_1), (p_3, \epsilon, p_3)\}.$$

Other values are forced:

$$\forall i > 0, \tau(\bot a^i) = \{(p_1, a, p_1), (p_1, \epsilon, p_0), (p_0, \uparrow, p_0)\}.$$

Following the rules for the annotation we get for all $i \geq 0$:

$$(p_0, \epsilon, p_1) \in \tau(\bot) \Rightarrow (p_0, 0, p_1) \in \eta(\bot) \qquad \text{cf 17.12}$$
$$(p_1, \epsilon, p_0) \in \tau(\bot a^{i+1}) \Rightarrow (p_1, 1, p_0) \in \eta(\bot a^{i+1}) \qquad \text{cf 17.12}$$
$$(p_1, a, p_1) \in \tau(\bot a^i), (p_1, 1, p_0) \in \eta(\bot a^{i+1}), (p_0, \uparrow, p_0) \in \tau(\bot a^{i+1})$$
$$\Rightarrow (p_1, \min(0,1,1), p_0) \in \eta(\bot a^i) \Rightarrow (p_1, 0, p_0) \in \eta(\bot a^i) \qquad \text{cf 17.15}$$
$$(p_1, 0, p_0) \in \eta(\bot), (p_0, 0, p_1) \in \eta(\bot)$$
$$\Rightarrow (p_0, 0, p_0) \in \eta(\bot), (p_1, 0, p_1) \in \eta(\bot) \qquad \text{cf 17.13}$$

Now we can see that $\mathcal{D}$ can not find an accepting path, *i.e.*, a winning path for Player 1. Player 0 win the game from q_{I}, provided he never moves to the vertex $\bot p_3$. Unlike finite graphs, we can have here arbitrary long paths with (minimal) priority 1, but no infinite path winning for Player 1.

Exercise 17.1. Complete the solution of this example, and compute the strategy according to section 17.3.3.

17.5 Discussion, Extension

We discuss here some conventions and hypotheses we have made, sometimes implicitly.

17.5.1 Discussion on the Conventions

We have assumed that the priority of a configuration depends only on the control state. Another possibility is to define regular set of states for each priority, or equivalently a finite automaton with output (over the alphabet $W \uplus P$) that accepts each configuration and outputs its priority. That wouldn't be more general: this automaton can be simulated by the states of the one-way automaton $\mathcal{E}$ (or by $\mathcal{A}$ with new labels on the tree). Otherwise it can be simulated by $\mathcal{Z}$ by extending the stack alphabet. The same ideas apply for the definition of V_0 and V_1 in V.

A usual convention for an arena (V_0, V_1, E) is that $E \subseteq V_0 \times V_1 \cup V_1 \times V_0$, *i.e.*, Player 0 and 1 alternate. This convention may clarify the situation but is not essential for us. If a pushdown system $\mathcal{Z}$ does not satisfy it, we can add "dummy" states to obtain a new pushdown game $\mathcal{Z}'$ which is equivalent to $\mathcal{Z}$ and satisfies the condition that in the new states there is only one possible move (choice).

The usual convention is also that a player who cannot move has lost. This is convenient with our formalism if we consider (see equations 17.17 and 17.18) that an empty disjunction is false and an empty conjunction is true (analogously it agrees with the definitions of $\Box$ and $\Diamond$ in μ-calculus). With pushdown games we can simulate another convention. We know in which configuration no transition is possible: if the stack is empty, or if there are no q', u with $(q, a, q', u) \in \Delta$. We can add new transitions to a particular state for the second case, and for the first case we can use a new symbol as the "bottom" of the stack, that can neither be put nor removed, and new transitions for this symbol.

17.5.2 Extensions

One can easily extend the results presented in this paper to any suffix (resp. prefix) rewrite system, either by simulating it with a pushdown automaton (up to bisimilarity) or by adapting our construction to allow $\mathcal{A}$ to go up along a fixed word (stored in its memory). In contrast one could restrict the pushdown system so that a transition consists just of pushing *or* popping one letter, which is equivalent to the general model.

In [106] other results are obtained with the help of two-way automata: the model checking procedure is extended to any μ-calculus formula (Theorem 2) over any context-free or even prefix recognizable graph (Theorem 5). In the present paper we have just considered the problem of solving parity games. On the other hand, each μ-calculus formula on a pushdown system is equivalent to a parity game. To simulate the prefix recognizable rewrite rules (see Chapter 15), the two-way automaton simulates the finite automata that recognize the different parts of the word (the prefix, the old suffix and the new suffix) using alternation and guessing the correct rule.

17.6 Conclusion

After some technical work to make the two-way automata "usable", it was possible to compute winning regions and winning strategies. This formalism is very powerful and hopefully comprehensible.

Its expressive power is the same as the μ-calculus on trees and on the transition systems that can be simulated on trees: pushdown systems and prefix recognizable graphs.

Chapter 15 of this book deals with another result about model checking: it is shown that Monadic Second Order logic (MSO) is decidable on prefix recognizable graphs. It is well known (see Chapter 14) that MSO is at least as expressive as μ-calculus, so implicitly the model-checking problem for μ-calculus on prefix-recognizable graphs was already solved by Caucal in [28]. It is natural to define parity games on prefix-recognizable graphs the same way as we have done: a configuration (node of the game graph) is a word, for clarity we suppose that the priority and the player (V_0 and V_1) are given by the first letter of the word. In fact we can define in MSO the winning region of Player 0 (resp. Player 1).

But if we compare both approaches in more detail, important differences show up: the MSO formula describes the whole winning region: the decision procedure gives a finite automaton that recognizes the whole set of winning vertices of Player 0 (resp. 1). On the contrary, the construction presented in the present chapter just checks one at a time if a given "initial" configuration is in the winning region. On the other hand, it is proved in [106] that his technique generates a winning strategy for this initialized game, represented by a finite automaton.

A similar result could be obtained by the methods introduced in Chapter 15, if a strategy could be defined in MSO. Unfortunately, this is not possible over the given arena. Indeed, a strategy is a binary relation, or a function from the vertices to the vertices, and it is not allowed in MSO to quantify about (non monadic) relations. Note that a strategy provides more information than a winning region does. It is possible to stay forever in the winning region and not win (never reach the "goal"). One cannot quantify about paths: they are not uniquely defined by their set of nodes. Finally, if the prefix-recognizable graph is a directed tree, and the game played from the root (top-down), the situation is much simpler: the strategy is a subtree with some good conditions, and is MSO-definable. (This gives an answer to a question of [180].) But in general the unraveling tree of a prefix-recognizable graph from a given vertex is not a prefix-recognizable graph (it is an algebraic tree, but this is outside the scope of this book).