Chapter 1

Basics on Tree Automata

Christof Löding

Informatik 7, RWTH Aachen, Germany

This article is an introduction to the theory of finite tree automata for readers who are familiar with standard automata theory on finite words. It covers basic constructions for ranked tree automata as well as minimization, algorithmic questions, and the connection to monadic second-order logic. Further, we present hedge automata for unranked trees and the model of tree-walking automaton.

1.1. Introduction

The theory of tree automata was established in the late sixties by Thatcher and Wright [53] and Doner [19]. They showed that the basic logical and algorithmic properties of standard automata theory can be transferred from the domain of finite words to the domain of finite trees (or terms), leading to a theory of regular tree languages. Quoting from the introduction of [53], where the theory of tree automata is called generalized finite automata theory: “...the results presented here are easily summarized by saying that conventional finite automata theory goes through for the generalization – and it goes through quite neatly!”

In parallel, Rabin [44] developed a theory of automata on infinite trees, which serves as a basis for the analysis of specification logics in the context of the verification of state based systems with non-terminating behavior. The theory of automata on infinite trees is quite different from the one on finite trees and in this chapter we exclusively consider finite trees.

Besides their application for showing the decidability of certain logical theories [19, 53], tree automata found their first applications in the area of term rewriting, e.g., for the automated termination analysis of certain rewriting systems, which are documented in the electronic book [16]. Nowadays, tree automata are used in various fields, e.g., in verification to model the state space of parametric systems [1] or as the underlying formalism for a logic that allows to specify properties of heap manipulating programs [38].

All these algorithmic applications of tree automata are based on the following two facts: The class of regular tree languages has strong closure properties (e.g., it is closed under boolean operations and projection), and the main algorithmic problems
like emptiness and inclusion are decidable. This article is an introduction to the theory of tree automata, written for readers familiar with conventional automata theory on finite words, where we present basic facts as the ones mentioned above. Of course, we can only cover some aspects of the whole theory in this chapter. The focus is on standard automata for ranked trees (terms) because these are the starting point and the central model of the whole area, but we also mention some other models.

As opposed to ranked trees, where the label of a node determines the number of child nodes, there is no bound on the number of children of a node in an unranked tree. This makes unranked trees suitable to model the structure of XML documents. There are various languages for specifying properties of XML documents, e.g., DTD (document type definition), XML schema, and RELAX NG (see [39]). The core of all these formalisms can be seen as a mechanism for defining languages of unranked trees. Motivated by this new area of application, a theory of automata for unranked trees has been developed (often referred to as hedge automata). Although this model was already mentioned in early papers on tree automata (e.g., [52] and [51]), a systematic study of hedge automata has only been started in the nineties, and is documented in [10]. We present the model of hedge automaton and give a brief overview of their basic properties and their relation to ranked tree automata.

We further present tree-walking automata, which have already been introduced in the seventies in [2] but also have gained new interest in recent years in the context of XML. Tree-walking automata process their input in a sequential way by navigating through the tree, and it turns out that they have rather different properties from standard tree automata.

The structure of the chapter is as follows. In Section 1.2 we present the basic terminology and introduce ranked and unranked trees. Section 1.3 presents the model of ranked tree automata, followed by hedge automata in Section 1.4. In Section 1.5 we briefly mention some results on tree-walking-automata, and we conclude in Section 1.6.

1.2. Trees

We start with some basic notations. By \( \mathbb{N} \) we denote the set of natural numbers, i.e., the set of non-negative integers. For a set \( X \) we denote by \( X^* \) the set of all finite words (or sequences) over \( X \) and by \( X^+ \) all finite nonempty words over \( X \). The empty word is denoted by \( \varepsilon \). A word \( u \) is a prefix of a word \( w \), written as \( u \sqsubseteq w \), if there is some word \( v \) such that \( w = uv \). By \( u \sqsubset w \) we denote the strict prefix relation.

There are two types of trees that we consider in this article: ranked and unranked trees. Ranked trees correspond to terms build up from function symbols with fixed arities. Therefore, in a ranked tree, the number of children of a node is determined by the label of the node. Unranked trees do not have this restriction. They can be
used to model ordered hierarchical structures as, e.g., XML documents or recursive computations.

1.2.1. Ranked trees

A ranked alphabet $\Sigma$ is a finite set of symbols together with an arity $|a| \in \mathbb{N}$ for each $a \in \Sigma$. The set of symbols of arity $i$ is denoted by $\Sigma_i$. The set $T_\Sigma$ of finite trees over $\Sigma$ is the least set containing for each $a \in \Sigma$ and each $t_1, \ldots, t_{|a|} \in T_\Sigma$ also $a(t_1, \ldots, t_{|a|})$. In case of $|a| = 0$ we simply write $a$ instead of $a()$. If we want to make the alphabet of labels explicit we also refer to a tree as a $\Sigma$-tree. Note that in this way we represent trees as words (with a special structure) over the alphabet containing all letters from $\Sigma$ and additionally the parentheses and the comma. An alternative view of trees makes their hierarchical structure more explicit, as explained in the following.

The domain $\text{dom}(t) \subseteq \mathbb{N}^*$ of a tree $t = a(t_1, \ldots, t_{|a|})$ is defined inductively as $\text{dom}(t) = \{\varepsilon\} \cup \bigcup_{i=1}^{|a|} i \cdot \text{dom}(t_i)$, where $\cdot$ denotes the concatenation of words, and we view $t$ as a mapping from its domain to the alphabet $\Sigma$ (see Example 1.1). The elements of the domain are called the nodes of the tree. Nodes that are labeled with symbols of arity 0 are called leaves, the other ones are called inner nodes, and $\varepsilon$ is the root of the tree. For $u \in \mathbb{N}^*$ and $i \in \mathbb{N}$ we call $u \cdot i$ the $i$th child or the $i$th successor of $u$.

Example 1.1. The left-hand side of Figure 1.1 shows a graphical representation of the tree $t = a(b(a(c, d)), c)$ over the ranked alphabet $\Sigma = \{a, b, c, d\}$ with $\Sigma_2 = \{a\}$, $\Sigma_1 = \{b\}$, and $\Sigma_0 = \{c, d\}$, i.e., $|a| = 2$, $|b| = 1$, $|c| = |d| = 0$. The children of a node are ordered from left to right, i.e., the $i$th edge going downwards from a node leads to the $i$th child. This is indicated by the labels on the edges. In future drawings we omit these labels because they can be derived from the left-to-right ordering of the nodes.

On the right-hand side the domain of the tree is shown. One can see that the name of a node corresponds to the concatenation of the edge labels leading to this node. Viewing $t$ as a mapping from its domain to $\Sigma$, we have $t(\varepsilon) = a$, $t(1) = b$, $t(2) = c$, $t(11) = a$, $t(111) = c$ and $t(112) = d$.

\[
\begin{array}{c|c|c|c}
 & a & \varepsilon \\
1/ & \backslash & / & \backslash \\
b & b & c & 1 & 2 \\
1 & a & 11 \\
1/ & \backslash & / & \backslash \\
c & d & 111 & 112
\end{array}
\]

Fig. 1.1. A graphical representation of a tree and its domain.
Given a tree \( t \) and a node \( u \in \text{dom}(t) \), the subtree \( t_u \) of \( t \) at node \( u \) is the tree naturally obtained when removing all nodes of which \( u \) is not a prefix. Formally \( t_u \) is the tree with domain \( \text{dom}(t_u) = \{ v \in N^* \mid uv \in \text{dom}(t) \} \) such that \( t_u(v) = t(uv) \).

In the above example (the tree \( t \) in Figure 1.1) for \( u = 11 \) we get the subtree \( t_{11} = a(c,d) \).

The height of a tree corresponds to the length of a longest path from the root to a leaf. Defined inductively, the height of \( a \) for \( a \in \Sigma \) is 0, and for a tree \( a(t_1, \ldots, t_{|a|}) \) the height is the maximal height of one of the trees \( t_i \) plus 1.

1.2.2. Hedges and unranked trees

Unranked trees are defined in a similar way as ranked ones but over a simple alphabet where the symbols do not have arities. Let \( \Sigma \) be an (unranked) alphabet, i.e., just a finite set of symbols. The set \( T_{\Sigma}^{\text{unr}} \) of unranked trees over \( \Sigma \) is the least set containing for each \( a \in \Sigma \) and each \( t_1, \ldots, t_n \in T_{\Sigma}^{\text{unr}} \) with \( n \geq 0 \) also \( a(t_1, \ldots, t_n) \).

Note that unranked trees are unbounded in two dimensions, vertically and horizontally. While there is only a finite number of ranked trees of a fixed height (for a fixed alphabet), there are infinitely many unranked trees of height one.

All the terminology and definitions from ranked trees, such as domain, nodes, leaves, etc. can be directly transferred to the unranked setting.

Finite sequences of unranked trees are called hedges. We denote the set of all hedges over the alphabet \( \Sigma \) by \( \mathcal{H}_\Sigma \).

Because unranked trees are unbounded vertically and horizontally, it is often useful to encode them by ranked trees. In particular, when dealing with automata, a lot of results for ranked tree automata can be transferred to the unranked setting using such encodings. We present here two such encodings.

The first-child-next-sibling encoding, abbreviated as FCNS encoding, is very natural when unranked trees have to be represented as data structures using pointers (see the textbook [31]). For each node we have one pointer to its first (left-most) child, and one pointer to its next (right) sibling. This is illustrated in Figure 1.2.

The symbols from the unranked alphabet all become binary symbols in the ranked

Fig. 1.2. An unranked tree and its FCNS representation
alphabet, and a new symbol \( \bot \) of arity 0 is added. The first child of a node in the FCNS encoding corresponds to its first child in the unranked tree, and the second child of a node in the FCNS encoding corresponds to its right sibling in the unranked tree. If the corresponding nodes (first child or right sibling) do not exist in the unranked tree, then we put the symbol \( \bot \) in the FCNS encoding. The encoding should be clear from the example and we do not give a formal definition here. For a tree \( t \) we denote its FCNS encoding by \( \text{fcns}(t) \).

The extension encoding is a natural way of representing unranked trees by building them up from the basic trees of height zero, using a binary operation. This operation is called the extension operation. Applied to two trees \( t \) and \( t' \), the operations attaches \( t' \) as the right-most subtree of the roof of \( t \). It appears explicitly in [12] as an encoding, but already in [51] it is used in a proof (where it is denoted “1” and oriented the other way, the left operand is added as left-most subtree of the root of the right operand).

Consider, for example, the tree on the left-hand side of Figure 1.2. On the left-hand side of Figure 1.3 it is shown how it is built up from two other trees by one application of the extension operator \( @ \). These two trees can further be constructed from smaller trees by this operation. This yields a coding as a binary tree where the leafs are trees of height 0, and the inner nodes indicate how to combine them by the extension operation. The right-hand side of Figure 1.3 shows the resulting coding for our example tree.

Looking at the picture it seems to be difficult to extract the unranked tree from the encoding. But there is a rather simple way of doing it: The left most leaf corresponds to the root of the unranked tree \( t \). Walking up from this leaf to the root of \( \text{ext}(t) \) we meet three times \( @ \). The right subtrees of these \( @ \) correspond to the three subtrees of the root in \( t \). Now we can proceed in the same way for decoding these subtrees.

In Section 1.4 we show how these encodings can be used to transfer results for automata on ranked trees to automata on unranked trees.
1.3. Ranked Tree Automata

The model of finite automaton on finite words is one of the many ways to characterize the class of regular word languages. In the following we develop a notion of tree automaton that has many of the good properties making finite automata so attractive in various fields of computer science.

In the whole section we only consider ranked trees and fix a ranked alphabet $\Sigma$. A nondeterministic finite tree automaton (NFTA) over $\Sigma$ is a tuple $A = (Q, \Sigma, \Delta, F)$, where $Q$ is a finite set of states, $F \subseteq Q$ is the set of final states, and $\Delta \subseteq \bigcup_{i=0}^{n} Q^i \times \Sigma_i \times Q$ is the transition relation. Transitions are of the form $(q_1, \ldots, q_{|a|}, a, q)$ with $a \in \Sigma$, and $q_1, \ldots, q_{|a|}, q \in Q$. If $a$ is of arity 0, then the transitions are written as $(a, q)$ and are called leaf transitions.

Intuitively, such an automaton works as follows on an input tree. It starts at the leafs and labels them by states according to the leaf transitions. For an inner node $u$ whose children are already labeled by some state, the transition relation determines which states $q$ can be put at $u$, depending on its label $a$, and the states $q_1, \ldots, q_{|a|}$ at its children. For this reason, these automata are called bottom-up or frontier-to-root automata because they start at the leafs (which are at the bottom in the graphical representations), and work their way upwards to the root. One should note that we can also view the automaton as working top-down. Then $F$ is interpreted as the set of initial states, and the transitions $(a, q)$ for $a$ of arity zero are interpreted as allowed or accepting pairs at the leafs. However, the bottom-up and top-down views lead to two different notions of determinism. We come back to this issue in Section 1.3.1.

Formally, a run of $A$ on a tree $t$ is a $Q$-tree $\rho$ satisfying the following conditions:

- $\text{dom}(\rho) = \text{dom}(t)$.
- For each leaf $u$ with $t(u) = a$ and $\rho(u) = q$ there is a transition $(a, q) \in \Delta$.
- For each inner node $u$ with $t(u) = a$, $\rho(u) = q_i$, and $\rho(u_i) = q_i$ for each $i \in \{1, \ldots, |a|\}$, there is a transition $(q_1, \ldots, q_{|a|}, a, q) \in \Delta$.

A run $\rho$ is accepting if it ends in a final state at the root, i.e., if $\rho(\varepsilon) \in F$. A tree $t$ is accepted by $A$ if there is an accepting run of $A$ on $t$. The set of all trees accepted by $A$ is called the language of $A$ and is denoted by $T(A)$. In general, we call a set of trees a tree language or simply language. A language is called regular if it is the language of some NFTA. We call two NFTAs equivalent if they accept the same language.

Example 1.2. A typical example is an NFTA that evaluates boolean terms built from the constants 0, 1, the binary operations $\land$, $\lor$, and the unary operation $\neg$. The automaton uses two states $q_0$ and $q_1$ with leaf transitions $(0, q_0), (1, q_1)$, and for each inner symbol of the term applies its semantics: $(q_i, q_j, \land, q_{i\land j})$ for $i, j \in \{0, 1\}$, and similarly for $\lor$. For the negation it simply flips the state: $(q_i, \neg, q_{\neg i})$. If we are interested in all terms evaluating to 1, then we let $q_1$ be the only final state. \fi
Example 1.3. As second example we consider a language over the ranked alphabet from Example 1.1. We want to construct an NFTA recognizing all trees with the following property: There is a leaf such that the sequence of labels from this leaf to the root is of the form \(da^*ba^*\), and there is a leaf such that the label sequence to the root is of the form \(ca^*\). The tree depicted in Figure 1.1 satisfies the property because the path from the middle leaf to the root is labeled \(daba\), and the path from the right leaf to the root is labeled \(ca\).

The NFTA that we construct, nondeterministically guesses the two leaves and then verifies the required property on the corresponding paths. We use states \(q_{db}, q_{da}, q_c\), where \(q_{db}\) is used on the path from the \(d\)-leaf before the \(b\) and \(q_{da}\) after the \(b\), and \(q_c\) is used on the path from the \(c\)-leaf. An additional state \(q\) is used on the parts of the tree that do not belong to one of the two paths, and a state \(q_a\) is intended for the positions where the two paths have merged. The transition relation contains the following transitions

- \((c, q_c, d, q_{da})\), \((d, q_{db})\), \((c, q, d, q)\), \((d, q, q_{db})\), \((q, q_{da}), (a, q_{db})\), \((q_{db}, q, a, q_{da})\), \((q_{da}, q, a, q)\), \((q, a, a, q)\), \((q, a, q_{da})\), \((q_{da}, a, q)\), \((q, q_{da}), a, q\), \((q, q_{da}), a, q\).

The final state is \(q_a\). Figure 1.4 shows an accepting run of this NFTA on the tree from Figure 1.1.

![Fig. 1.4. A run of the automaton from Example 1.3.](image)

An NFTA is called complete if for each \(a \in \Sigma\) and all \(q_1, \ldots, q_{|a|} \in Q\) there is at least one \(q \in Q\) such that \((q_1, \ldots, q_{|a|}, a, q) \in \Delta\). One can easily turn each NFTA into an equivalent one that is complete. But one should note that for alphabets of high arity this might require to add many transitions.

1.3.1. Determinization and closure properties

On finite words it is well known that nondeterministic automata can be transformed into equivalent deterministic automata using the subset construction. We show here that the same holds for tree automata.

An NFTA is called deterministic (DFTA) if for each \(a \in \Sigma\) and all \(q_1, \ldots, q_{|a|} \in Q\) there is at most one \(q \in Q\) such that \((q_1, \ldots, q_{|a|}, a, q) \in \Delta\). For this definition it
is essential that we take the bottom-up view of the automata. Given the states at the children of the node and the label of the node there is at most one transition that can be executed. For DFTAs we can view the transition relation as a (partial) function, and we denote it by \( \delta \) instead of \( \Delta \), i.e., \( \delta(q_1, \ldots, q_{|a|}, a) = q \) instead of \( (q_1, \ldots, q_{|a|}, a, q) \in \Delta \).

**Theorem 1.1 ([19, 53]).** For each NFTA one can construct an equivalent complete DFTA. In the worst case, the number of states of the DFTA is exponential in the number of states of the given NFTA.

**Proof.** The proof is done by an adaption of the classical subset construction for word automata from [45] (see also [26]). Let \( \mathcal{A} = (Q, \Sigma, \Delta, F) \) be an NFTA. We construct the subset automaton \( \mathcal{P}(\mathcal{A}) = (2^Q, \Sigma, \delta, F') \), where \( 2^Q \) denotes the power set of \( Q \), and \( \delta \) and \( F' \) are defined as follows:

- \( \delta(R_1, \ldots, R_{|a|}, a) = R \) for every \( R_1, \ldots, R_{|a|} \in 2^Q \) and
  \[
  R = \{ q \in Q \mid \exists q_1 \in R_1, \ldots, q_{|a|} \in R_{|a|} : (q_1, \ldots, q_{|a|}, a, q) \in \Delta \}.
  \]
- \( F' = \{ R \subseteq Q \mid R \cap F \neq \emptyset \} \).

It is straightforward to prove that \( \mathcal{P}(\mathcal{A}) \) is equivalent to \( \mathcal{A} \). \( \square \)

For this generic construction presented in the proof of Theorem 1.1 the resulting DFTA always has exponentially many states compared to the given NFTA. But in many cases it is not necessary to consider the whole power set \( 2^Q \) because not all of these sets are reachable in the DFTA. In Section 1.3.2 we show how to compute the set of reachable states of a tree automaton. However, there are cases where the exponential blow-up cannot be avoided. This follows from the lower bound for determinization of word automata (see [26]).

It is also possible to consider top-down determinism. We call an NFTA **top-down deterministic** if \( |F| = 1 \) (\( F \) is interpreted as set of initial states and there should be only one such initial state), and for each \( a \in \Sigma \) and each \( q \in Q \) there is at most one transition of the form \( (q_1, \ldots, q_{|a|}, a, q) \in \Delta \). It is rather easy to see that top-down determinism is a strong restriction. Not even all finite tree languages can be accepted by top-down deterministic automata. For example, each top-down deterministic automaton accepting the trees \( a(b, c) \) and \( a(c, b) \) will also accept the trees \( a(b, b) \) and \( a(c, c) \). One can show that top-down deterministic automata can only define so-called path-closed languages. Intuitively, this means that if we take several trees accepted by a top-down deterministic automaton, decompose these trees into their paths, and build a new tree out of (some of) these paths, then the resulting tree is also accepted by the automaton. In the above example we can use the left-most path of \( a(b, c) \) and the right-most path of \( a(c, b) \) to obtain \( a(b, b) \). For more results and references on top-down deterministic tree automata we refer the reader to [36].
For the closure properties of regular tree languages we observe that complete deterministic automata can be complemented by exchanging final and non-final states. For the intersection we can apply a standard product construction simulating both automata, and accepting if both of them reach a final state. For the union we can simply take the disjoint union of the components of two NFTAs. In case we want to take the union of two DFTAs and want to obtain a DFTA again we also use a product construction.

**Theorem 1.2.** The class of regular tree languages is closed under union, intersection, and complement.

1.3.2. **Decision problems and algorithms**

We now turn to algorithmic questions for tree automata. For estimating the complexity of algorithms we have to define the size of a tree automaton. A natural and common way of doing this is to take the number of states and for each transition the number of its entries. More precisely, let $A = (Q, \Sigma, \Delta, F)$ be an NFTA. The size of a transition $\tau = (q_1, \ldots, q_n, a, q)$ $\in \Delta$ is $|\tau| = n + 2$, and the size of $A$ is

$$|A| = |Q| + \sum_{\tau \in \Delta} |\tau|.$$  

We consider the following decision problems.

- **The membership problem**

  **Given:** An NFTA $A$ and a tree $t$
  **Question:** Is $t \in T(A)$?

- **The emptiness problem**

  **Given:** An NFTA $A$
  **Question:** Is $T(A) = \emptyset$?

- **The universality problem**

  **Given:** An NFTA $A$
  **Question:** Is $T(A) = T_\Sigma$?

- **The inclusion problem**

  **Given:** NFTAs $A_1$ and $A_2$
  **Question:** Is $T(A_1) \subseteq T(A_2)$?

---

*aSometimes, the membership problem is defined for a fixed automaton and the problem where the tree and the automaton are part of the input is called the uniform membership problem.*
The equivalence problem

Given: NFTAs $A_1$ and $A_2$

Question: Is $T(A_1) = T(A_2)$?

A detailed complexity analysis of the membership problem is given in [35]. Here we just show how to solve the problem in polynomial time.

**Theorem 1.3.** The membership problem can be solved in polynomial time.

**Proof.** Let $A = (Q, \Sigma, \Delta, F)$ be an NFTA and $t \in T_{\Sigma}$. The algorithm computes for each node $u$ of $t$ the set $Q_u$ of states that the automaton can reach in this node. For a leaf $u$ this set is given by $Q_u = \{q \mid (t(u), q) \in \Delta\}$. For an inner node $u$ with $k$ children the set $Q_u$ can easily be computed from the transitions and the sets $Q_{u1}, \ldots, Q_{uk}$:

$$Q_u = \{q \mid \exists q_1 \in Q_{u1}, \ldots, q_k \in Q_{uk} : (q_1, \ldots, q_k, t(u), q) \in \Delta\}.$$ 

Each set $Q_u$ can be computed in polynomial time and hence the whole algorithm is polynomial in the size of $A$ and $t$. □

The decidability of the emptiness problem is already shown in [53] and [19]. In both papers the argument is the following: If an NFTA accepts a tree, then there is also an accepted tree whose height can be bounded by the number of states of the NFTA. Hence, one can do the membership test for all the trees up to this height. Of course this algorithm is not very efficient.

To obtain efficient algorithms, the emptiness problem for tree automata can be reduced to many other problems that are efficiently solvable. For example, it can easily be transformed into the satisfiability problem for Horn-formulas as described in [16]. The latter problem can be solved in linear time\textsuperscript{b} by so-called unit propagation. Another closely related problem is alternating graph reachability, a problem known to be P-complete [25]. To be self contained we present an algorithm directly working on tree automata. The algorithm is explained in the proof of Theorem 1.4.

**Theorem 1.4.** The emptiness problem for NFTA can be solved in linear time.

**Proof.** We solve the problem by computing the set of reachable states, i.e., the set of states that can be reached at the root in a run on some tree. If one the final states is reachable, then the language of the automaton is not empty, otherwise it is empty.

Figure 1.5 shows an algorithm to compute the set $R$ of reachable states. The underlying idea is a simple fixpoint computation: All states that occur in a leaf transition are reachable, and if there is a transition $(q_1, \ldots, q_{|a|}, a, q)$ such that $q_1, \ldots, q_{|a|}$ are already in $R$, then $q$ can be added to $R$. This is exactly what is done

\textsuperscript{b}For the complexity estimations we refer to the RAM model of computation.
Input: NFTA $\mathcal{A} = (Q, \Sigma, \Delta, F)$

1: $R = \{ q \in Q \mid \exists a \in \Sigma : (a, q) \in \Delta \}$

2: for all $\tau = (q_1, \ldots, q_{|a|}, a, q) \in \Delta$ do

3: $\text{pre}(\tau) = \{ q_1, \ldots, q_{|a|} \}$

4: $\text{dest}(\tau) = q$

5: end for

6: $M := R$

7: while $M \neq \emptyset$ do

8: Choose and remove $p$ from $M$

9: for all $\tau$ with $p \in \text{pre}(\tau)$ do

10: Remove $p$ from $\text{pre}(\tau)$

11: if $\text{pre}(\tau) = \emptyset$ then

12: Add $\text{dest}(\tau)$ to $M$ and to $R$

13: end if

14: end for

15: end while

Output: $R$

Fig. 1.5. Algorithm for computing the reachable states of a tree automaton.

in the algorithm in Figure 1.5 but in such a way that it can be implemented to run in linear time by choosing the appropriate data structures. As described above, we have to check for a transition $\tau = (q_1, \ldots, q_{|a|}, a, q)$ whether the states $q_1, \ldots, q_{|a|}$ are already in $R$. We do this by first defining the set $\text{pre}(\tau) = \{ q_1, \ldots, q_{|a|} \}$, and whenever we add a state $q$ to $R$, we remove it from all $\text{pre}$-sets it occurs in. If $\text{pre}(\tau)$ becomes empty, we add $q = \text{dest}(\tau)$ to $R$.

The set $M$ is used to keep track of which states still have to be removed from the $\text{pre}$-sets. To obtain a linear time algorithm, the set $M$ can be implemented as a FIFO-queue, and to quickly access the transitions in the loop of line 9 one has to store for each state $q$ a list of the transitions $\tau$ with $q \in \text{pre}(\tau)$.

Similar to the case of word automata, where emptiness is easy to decide and universality is PSPACE-hard, we obtain a much higher complexity for the universality problem. It is difficult to attribute the following result to a specific paper. But [50] shows a similar result for the equivalence problem, and those techniques can also be used for the universality problem.

**Theorem 1.5.** The universality problem for tree automata is EXPTIME-complete.

**Proof.** Checking universality can be done in EXPTIME by determinizing the automaton and then checking the complement for emptiness.

A standard way to show EXPTIME-hardness is to use a reduction from the word problem for alternating polynomial space bounded Turing machines. Alternating machines are natural in connection with tree automata because computations of
these machines are trees. Proofs based on alternating machines can, e.g., be found in [50] and [55] (not for the universality problem but related decision problems).

Besides alternating machines there is another formalism closely related to tree automata: Strategies in games can be represented by trees (the branching coming from the different options of the opponent). Hence, it is natural to express problems concerning the existence of winning strategies by means of tree automata. We use here the two-person corridor tiling problem, known to be Exptime-hard [15].

A corridor tiling system is of the form $D = (D, H, V, b, f, n)$, where $D$ is a finite set of tiles (or dominos), $H, V \subseteq D \times D$ are horizontal and vertical constraints, $b, f \in D^n$ are $n$-tuples of tiles corresponding to the initial row and the final row of the corridor tiling, and $n \in \mathbb{N}$ is the width of the corridor. Intuitively, the goal is to cover a corridor of width $n$ (and arbitrary length $m$) with the tiles such that the horizontal and vertical constraints are respected, and the corridor starts with $b$ and ends with $f$.

Formally, a corridor tiling is a mapping $C : \{1, \ldots, m\} \times \{1, \ldots, n\} \rightarrow D$ for some $m \in \mathbb{N}$ such that

1. $(C(1, 1), \ldots, C(1, n)) = b$ (it begins with the initial row),
2. $(C(i, j), C(i, j+1)) \in H$ for all $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, n-1\}$ (it respects the horizontal constraints),
3. $(C(i, j), C(i+1, j)) \in V$ for all $i \in \{1, \ldots, m-1\}$ and $j \in \{1, \ldots, n\}$ (it respects the vertical constraints),
4. $(C(m, 1), \ldots, C(m, n)) = f$ (it ends with the final row).

The problem of deciding for a given corridor tiling system whether there exists a corridor tiling for it is PSPACE-hard. We consider the game variant of the problem. The game board is the infinite corridor $\mathbb{N}_1 \times \{1, \ldots, n\}$, where $\mathbb{N}_1$ denotes the natural numbers without 0. The starting configuration is the one where the first row of the corridor is covered with the initial row $b$. Now the two players Eva and Adam play tiles in turn on successive positions, Eva starting in position $(2, 1)$. Both players have to respect the horizontal and vertical constraints.

Thus, a configuration in the game corresponds to a partial function $C : \mathbb{N}_1 \times \{1, \ldots, n\} \rightarrow D$ whose domain is an initial segment of $\mathbb{N}_1 \times \{1, \ldots, n\}$, i.e., if $C(i, j)$ is defined, then also $C(i', j')$ is defined if $i' < i$, or $i' = i$ and $j' < j$.

Eva wants to construct a corridor tiling, i.e., she wins if at some point the mapping $C$ is a corridor tiling. Note that the rules of the game ensure that conditions 1–3 are always satisfied. Hence, her goal is to construct the final row. Adam wins otherwise, i.e., either if the play gets stuck because there is no next move respecting the constraints, or if the game goes on forever and no corridor tiling is ever reached.

The problem of two-player corridor tiling is to decide for a given tiling system $D$ whether Eva has a winning strategy in the game described above. This problem is Exptime-hard [15].

The basic idea for the reduction to the universality of NFTAs is simple: We
code strategies of Eva as trees and then construct an NFTA of polynomial size that accepts all trees that do not code a winning strategy for Eva.

The alphabet that we use is \( \Sigma = D \times \{E, A, \bot, !\} \), where the elements of \( D \times \{E\} \) are of arity 1, those of \( D \times \{A\} \) are of arity \(|D|\), and those of \( D \times \{\bot, !\} \) are of arity 0. A path through such a tree corresponds in a natural way to a mapping \( C : N_1 \times \{1, \ldots, n\} \to D \) whose domain is an initial segment of \( N_1 \times \{1, \ldots, n\} \) (just place the tiles in the order they appear, starting from the position \((2, 1)\) as in a play). The idea is that the first component corresponds to the tile that has been played, and the second component indicates whose turn it is (\(E\) for Eva, \(A\) for Adam, \(\bot\) in case the play is over because a constraint was not respected, and \(!\) in the case a corridor tiling has been completed).

A tree of the above shape represents a winning strategy for Eva if the following conditions are satisfied

1. The root is labeled by \((d, A)\) for some \(d \in D\) (\(d\) is the first tile that is played by Eva, and \(A\) indicates that it is now Adam’s turn).
2. The nodes from \(D \times \{A\}\) and \(D \times \{E\}\) alternate on each path (Eva and Adam play their tiles in turn).
3. There is no node that has only \(\bot\) successors (the play never gets stuck).
4. At each node not labeled \(\bot\) the tile respects the constraints associated to its position.
5. At each node labeled \(\bot\) the tile does not respect one of the constraints associated to its position.
6. A node is labeled \(!\) if the mapping \(C\) corresponding to the path to the root ends with the row \(\bar{f}\) (and hence corresponds to a corridor tiling in combination with the previous conditions).

It is not difficult to see that for each of the above conditions one can construct a small NFTA checking whether the condition does not hold. For the last three items the automata need to guess the nodes where the condition is not satisfied. For verifying that a constraint is not satisfied it is sufficient that the automaton can remember a tile and can count \(n\) steps. The details of these constructions are left to the reader.

Taking the union of these NFTAs results in an NFTA of polynomial size that accepts all trees iff Eva does not have a winning strategy. \(\square\)

Since universality is a special case of the inclusion and equivalence problems, we can easily transfer the lower bound.

**Theorem 1.6** ([50]). The inclusion and the equivalence problem for tree automata are EXPTime-complete.

**Proof.** Since \(T(A_1) \subseteq T(A_2)\) iff \(L(A_1) \cap (T_\Sigma \setminus L(A_2)) = \emptyset\), we can reduce the inclusion problem to the emptiness problem with an exponential cost. Because emptiness is decidable in polynomial time (Theorem 1.4), we obtain membership in
Exptime for the inclusion problem. Equivalence can be decided by checking both inclusions.

For the Exptime-hardness we observe that $L(A)$ is universal iff $T_\Sigma \subseteq T(A)$ and apply Theorem 1.5. The same argument works for equivalence with $=$ instead of $\subseteq$.

It is easy to see that the problems of inclusion and equivalence are decidable in polynomial time for DFTAs (the exponential step for NFTAs is the complementation). In [50] it is shown that equivalence can also be decided in polynomial time if the automata are $m$-ambiguous for some constant $m$, where $m$-ambiguous means that there are at most $m$ accepting runs of the automaton for each input.

1.3.3. Congruences and minimization

On finite words deterministic automata can be efficiently minimized. The minimization algorithm identifies equivalent states of the automaton and merges them. In Section 1.3.1 we have already seen that the determinization technique for automata on finite words can be generalized to (bottom-up) tree automata. In this section we show that this also works for minimization.

The background for minimization of automata on finite words is the right-congruence that defines two words $w_1, w_2$ to be equivalent w.r.t. some language $L$ of finite words if every word $w$ either leads both $w_1$ and $w_2$ into the language when appended as a suffix, or leads both outside of the language: $w_1w \in L$ iff $w_2w \not\in L$. This defines an equivalence relation on the set of finite words that is compatible with concatenation to the right. The index of the equivalence, i.e., the number of its classes is finite iff the language $L$ is regular, and it naturally induces an automaton for $L$ whose states are the equivalence classes. This relation is often referred to as Myhill/Nerode-congruence referring to the first papers where it has been used [41, 42].

The goal of this section is to lift these results to ranked trees. The first step is to define a concatenation operation for trees. For this we need the notion of context. In our setting a context is a tree in which exactly one of the leafs is labeled by a variable. This leaf is then used for the concatenation operation.

Let $X \notin \Sigma$ be a variable. A context $C$ is a tree in $T_{\Sigma \cup \{X\}}$ (where the arity of $X$ is 0), in which exactly one of the leafs is labeled by a variable $X$. We denote the set of contexts by $C_\Sigma$.

For a context $C \in C_\Sigma$ and a tree $t \in T_\Sigma$ we write $C[t]$ for the tree that is obtained by replacing the $X$-labeled leaf in $C$ by $t$. This is illustrated in Figure 1.6.

Based on this concatenation operation we can now define an equivalence relation on trees. Let $T \subseteq T_\Sigma$ be a tree language. We say that two trees $t_1, t_2 \in T_\Sigma$ are equivalent w.r.t. $T$ if

$$\forall C \in C_\Sigma : C[t_1] \in T \Leftrightarrow C[t_2] \in T.$$ 

In general, a context can have several different variables at the leafs. A context with $n$ variables is called an $n$-context. We are using 1-contexts here.
We denote this equivalence by $\sim_T$. As for finite words there is a close correspondence between this relation and deterministic automata. In [33] it is mentioned that it is difficult to attribute the following result to a specific paper. We refer the reader to [33] for references.

**Theorem 1.7.** The index of $\sim_T$ is finite iff $T$ is regular, and the number of equivalence classes of $\sim_T$ corresponds to the number of states of a minimal DFTA accepting $T$.

**Proof.** Assume that $T$ is regular and let $A = (Q, \Sigma, \delta, F)$ be a complete DFTA for $T$. Denote by $\delta^*$ the function $\delta^*: T_\Sigma \to Q$ that assigns to each tree the state that is at the root of the unique run of $A$ on $t$. If $\delta^*(t_1) = \delta^*(t_2)$ for two trees $t_1$ and $t_2$, then it is easy to see that $t_1 \sim_T t_2$. Since $Q$ is finite the index of $\sim_T$ is finite.

Now assume that $\sim_T$ has only finitely many equivalence classes. By $[t]_T$ we denote the $\sim_T$ equivalence class of $t$. We define the complete DFTA $A_T = (Q_T, \Sigma, \delta_T, F_T)$ as follows:

- $Q_T = \{[t]_T \mid t \in T_\Sigma\}$,
- $\delta_T([t_1]_T, \ldots, [t_{|a|}]_T, a) = [a(t_1, \ldots, t_{|a|})]_T$,
- $F_T = \{[t]_T \mid t \in T\}$.

Note that this definition of the transition function is possible because the equivalence class of $a(t_1, \ldots, t_{|a|})$ does not depend on the choice of the representatives $t_1, \ldots, t_{|a|}$.

A straightforward induction shows that $\delta_T^*(t) = [t]_T$, and hence $T(A_T) = T$. □

To compute from a given DFTA an equivalent one that is minimal we introduce an equivalence relation on states. Let $A = (Q, \Sigma, \delta, F)$ be a DFTA. The relation $\sim_A \subseteq Q \times Q$ is the smallest relation satisfying the following:

1. If $p \sim_A q$, then $p \in F \iff q \in F$.
2. If $p \sim_A q$, then
   $$\delta(q_1, \ldots, q_{i-1}, p, q_{i+1}, \ldots, q_{|a|}, a) \sim_A \delta(q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_{|a|}, a)$$
   for all $q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_{|a|} \in Q$ and $a \in \Sigma$. 

---

**Basics on Tree Automata**

![Fig. 1.6. Concatenation of a context and a tree](image-url)
The first condition ensures that the equivalence classes are either contained in $F$ or have an empty intersection with $F$. The second condition ensures that the equivalence relation is compatible with the transition function. Hence, one can built the quotient $DFTA \ A/\sim$ that uses the equivalence classes as states. It is left as an exercise for the reader to show that $A/\sim$ is isomorphic to $A_T$ from the proof of Theorem 1.7.

Computing the relation $\sim_A$ can be done by successively marking those pairs of states that do not satisfy one of the two conditions. Concerning the complexity of minimization, [13] presents an implementation of the algorithm that runs in quadratic time in the size of the given tree automaton.

1.3.4. Logic

Instead of taking the view of trees as mappings from a domain to a finite set of labels we can also view them as relational structures and use predicate logic to express properties of such structures. Using first-order logic extended with quantification over monadic predicates (sets of elements) one obtains a formalism that is expressively equivalent to tree automata. Such a relation has first been established for finite words [11, 20, 54], and was lifted to trees in [19, 53].

For a ranked alphabet $\Sigma$ with symbols of maximal arity $k$ we consider the signature consisting of the relational symbols $S_1, \ldots, S_k$ for the binary successor (child) relations, $\sqsubseteq$ for the prefix relation on nodes, and $(P_a)_{a \in \Sigma}$ for the unary relations “the node is labeled $a$”. A tree $t \in T_\Sigma$ corresponds to a relational structure $t = (\text{dom}(t), S_1^t, \ldots, S_k^t, \sqsubseteq^t, (P_a^t)_{a \in \Sigma})$ over the universe $\text{dom}(t)$ and the relations defined in the natural way as

- $S_i^t(u, v)$ if $v$ is the $i$th child of $u$, i.e., $v = u_i$,
- $u \sqsubseteq^t v$ if $u \sqsubseteq v$, and
- $P_a^t(u)$ if $t(u) = a$.

In monadic second-order logic (MSO) we use two types of variables: first-order variables interpreted by elements and denoted by small letters (e.g., $x, y$), and monadic second-order variables interpreted by sets of elements and denoted by capital letters (e.g., $X, Y$).

MSO-formulas are built up from atomic formulas $S_i(x, y), x \sqsubseteq y, P_a(x), x \in X$, and $x = y$ by boolean combinations and quantifications over the two types of variables. Let us consider two simple examples.

- $\text{leaf}(x) = \neg \exists y : S_1(x, y)$ expresses that $x$ does not have a first child, and hence is a leaf (if it does not have a first child it has no children).
- The following formula expresses that the set $X$ is a cut through the tree, i.e., a
maximal set of nodes that are pairwise incomparable w.r.t. $\sqsubseteq$:

$$\text{cut}(X) = \forall y \exists x : x \in X \land (y \sqsubseteq x \lor x \sqsubseteq y) \land \\
\forall x, y : x \sqsubseteq y \rightarrow (x \notin X \lor y \notin X)$$

- Using the above formula one can easily express that the nodes labeled $a$ form a cut: $\exists X : \text{cut}(X) \land \forall x : (P_a(x) \leftrightarrow x \in X)$

For a sentence $\varphi$, i.e., a formula without free variables we write $t \models \varphi$ if $\varphi$ holds in $t$. Given such a sentence $\varphi$ it defines the tree language

$$T(\varphi) = \{ t \in T_\Sigma | t \models \varphi \}.$$

We call a tree language $T \subseteq T_\Sigma$ $\text{MSO-definable}$ if it is the language of some $\text{MSO}$-sentence $\varphi$. Now we can state the equivalence theorem.

**Theorem 1.8 ([19, 53]).** A tree language $T \subseteq T_\Sigma$ is $\text{MSO-definable}$ iff it is regular.

**Proof.** We only sketch the idea: For the direction from logic to automata one proceeds by induction on the structure of the formula. To deal with free variables, one annotates the nodes of the tree with tuples of 0 and 1 to encode the interpretations of the free variables. For example, an automaton for the atomic formula $S_1(x, y)$ accepts trees over the alphabet $\Sigma \times \{0, 1\} \times \{0, 1\}$ where exactly one node $u$ has label 1 in the second component (the interpretation of $x$), and exactly one node $v$ has label 1 in the third component (the interpretation of $y$), and $v$ is the first child of $u$. It is easy to see that one can build such automata for all atomic formulas. Then one uses the closure properties of tree automata to deal with boolean combinations (union, intersection, and complement), and quantifications (projection).

For the direction from automata to logic one constructs a formula describing an accepting run of the given automaton. For each state $q$ one introduces a set variable $X_q$ that is interpreted by the positions in the run in which the automaton is in state $q$. If the state set is $\{q_1, \ldots, q_n\}$ then the formula has the shape

$$\exists X_{q_1} \cdots \exists X_{q_n} : \text{AccRun}(X_{q_1}, \ldots, X_{q_n})$$

where the formula $\text{AccRun}(X_{q_1}, \ldots, X_{q_n})$ expresses that the sets indeed code a run (each node is in exactly one set, the root is in a set $X_q$ for a final state $q$, and the local constraints of the transition relation are respected). □

The translation between MSO-formulas and automata is effective. As a consequence, decision problems for $\text{MSO}$ logic on finite trees can be reduced to decision problems for tree automata. For example, the satisfiability problem “Given an $\text{MSO}$-sentence, does it have a model?” reduces to the emptiness problem for tree automata, which is decidable according to Theorem 1.4.

**Corollary 1.1.** The satisfiability problem for $\text{MSO}$ logic over finite trees is decidable.
One should note however that the translation from formulas to automata is non-elementary because each negation requires complementation of the automaton, which is exponential. In the worst case this cannot be avoided (see [46] for a proof in the case of words). Nevertheless, the construction has been implemented in the tool MONA [29, 30] and finds applications, e.g., in program verification [38].

On finite words there is also a nice characterization of languages that can be described in first-order logic (FO), i.e., MSO without set variables. These are exactly the star-free languages (see [18] for an overview). Over trees such a characterization is still missing (see [5] for a recent paper on the subject).

1.4. Hedge automata

In this section we adapt the automaton model such that we can handle unranked trees (and hedges). We keep the principle that the state at a node should only depend on the node label and the states at the children. For this purpose we have to define the transitions in such a way that they can deal with child sequences of arbitrary length. Since we have to deal with state sequences, we can rely on formalisms for defining word languages. To obtain an automaton model that has good closure and algorithmic properties we use regular languages of words to specify the transitions.

A \textit{nondeterministic finite hedge automaton} (NFHA) over $\Sigma$ is a tuple $A = (Q, \Sigma, \Delta, F)$, where $Q$ is a finite set of states, $F \subseteq Q$ is the set of final states, and $\Delta$ is a finite set of transitions of the form $(L, a, q)$ with $a \in \Sigma$, $q \in Q$, and $L \subseteq Q^*$ a regular language over the alphabet $Q$.

NFHAs work on unranked trees in a similar way as NFTAs work on ranked trees: Each node is labeled by a state, where the possible states at a node are determined by its label from $\Sigma$, and by the states at the children. Formally, a run of $A$ on a tree $t$ is a $Q$-tree $\rho$ satisfying the following conditions:

- $\text{dom}(\rho) = \text{dom}(t)$.
- For each node $u$ with $t(u) = a$, $\rho(u) = q$, and $\rho(ui) = q_i$ for each $i \in \{1, \ldots, k\}$, where $k$ is the number of children of $u$, there is a transition $(L, a, q) \in \Delta$ with $q_1 \cdots q_k \in L$. In particular, if $k = 0$ then the empty words has to be in $L$.

We use the same terminology as for ranked trees: A run $\rho$ is accepting if it ends in a final state at the root, i.e., if $\rho(\varepsilon) \in F$. A tree $t$ is accepted by $A$ if there is an accepting run of $A$ on $t$. The set of all trees accepted by $A$ is called the language of $A$ and is denoted by $T(A)$.

\textbf{Example 1.4.} We construct an NFHA over $\Sigma = \{a, b, c, d\}$ that accepts the trees that contain the following pattern: there is an $a$ labeled node and two levels below this node there is a $d$ labeled node. The NFHA guesses the $d$ labeled node and verifies that the node two levels above is labeled $a$. For this purpose we use the states $q_d$ for guessing the $d$ labeled node, $q'_d$ at the node above $q_d$, $q_\top$ if the pattern...
has been identified, and \( q \) for the other parts of the tree. The transitions are given below, where \( e \) denotes an arbitrary symbol from \( \Sigma \):

\[
(q^*, e, q), (q^*, d, q_d), (q^* q_d q^*, e, q_d'), (q^* q_d q^*, a, q_T), (q^* q_T q^*, e, q_T).
\]

The only final state is \( q_T \).

1.4.1. Relation to ranked tree automata

In Section 1.2 we have shown two possibilities how to code unranked trees by ranked ones. It is of course desirable that such encodings preserve recognizability of languages by the automaton models that we have presented. This allows to transfer results for ranked tree automata to hedge automata.

The following equivalence theorem can be shown by simple manipulations of automata. Therefore we do not attribute the results to specific papers but consider them to be folklore.

**Theorem 1.9.** Given a language \( T \subseteq T^{\text{ unr}}_\Sigma \), the following conditions are equivalent:

- \( T \) is recognizable by a hedge automaton.
- \( \text{fcns}(T) \) is regular.
- \( \text{ext}(T) \) is regular.

**Proof.** The equivalence can be shown by direct automaton constructions. We illustrate this by showing how to transform a hedge automaton into an automaton for \( \text{ext}(T) \). The other constructions follow a similar principle and are left to the reader.

Let \( A = (Q, \Sigma, \Delta, F) \) be an NFHA. We assume that for each \( a \) and \( q \) there is exactly one rule \( (L_{a,q}, a, q) \). This is no restriction because several rules can be merged by taking the union of the horizontal languages. Furthermore, assume that each language \( L_{a,q} \) is given by a deterministic finite (word) automaton \( B_{a,q} = (P_{a,q}, Q, p_{a,q}, \delta_{a,q}, F_{a,q}) \) over the input alphabet \( Q \). We assume that all state sets are pairwise disjoint.

For simulating a run of \( A \) on an unranked tree \( t \) by a run on \( \text{ext}(t) \) we only use the states of the automata \( B_{a,q} \). A subtree of \( \text{ext}(t) \) that is rooted at a node \( u \) that is the right child of some other node corresponds to a subtree in \( t \), say at node \( u' \). The simulation is implemented such that on \( \text{ext}(t) \) at \( u \) there is a state from \( F_{a,q} \) iff in the corresponding run on \( t \) the state \( q \) is at node \( u' \), and \( u' \) is labeled \( a \).

The NFTA \( A' = (Q', \Sigma', \Delta', F') \) for \( \text{ext}(T) \) is defined as follows:

- \( Q' = \bigcup_{a,q} P_{a,q} \).
- \( \Delta' \) contains the following transitions:
  - \((a, p_{a,q})\) for each \( a \in \Sigma \) and \( q \in Q \). These transitions are used to guess at each leaf the state \( q \) that is used in a run on the unranked tree at the node corresponding to this leaf.
\( (p, p', \emptyset, p'') \) if \( p, p'' \) are both in some \( P_{b,q} \), \( p' \in F_{a,q'} \) for some \( q' \), and \( \delta_{b,q}(p, q') = p' \).

- \( F' = \bigcup_{a \in \Sigma, q \in F} F_{a,q} \).

The construction is illustrated in Figure 1.7. On the left hand side a run on the unranked tree \( b(c, d) \) is shown. The states of the form \( p \) are the states of the automata \( B_{a,q} \) for the horizontal languages. In the picture it is shown how they are used to process the states of the form \( q \). Note that the initial states \( p_{c,q_1} \) and \( p_{d,q_2} \) of \( B_{c,q_1} \) and \( B_{d,q_2} \) must also be final states because \( q_1 \) and \( q_2 \) are assigned to the two leafs and hence \( \varepsilon \in L_{c,q_1} \) and \( \varepsilon \in L_{d,q_2} \). Furthermore \( p_2 \) is a final state of \( B_{b,q_3} \). If \( q_3 \in F \), then both runs are accepting because \( p_2 \) is a final state of \( A' \). □

Based on this theorem we call a language of unranked trees regular if it is the language of some NFHA.

**Theorem 1.10.** The class of regular unranked tree languages is closed under union, intersection, and complement.

**Proof.** Let \( T_1, T_2 \subseteq T^{\text{unr}}_\Sigma \) be two regular languages of unranked trees. According to Theorem 1.9 we obtain that \( \text{ext}(T_1) \) and \( \text{ext}(T_2) \) are regular. Since \( \text{ext} \) is a bijection between unranked trees and ranked trees we obtain \( \text{ext}(T_1) \cap \text{ext}(T_2) = \text{ext}(T_1 \cap T_2) \). By Theorem 1.2 \( \text{ext}(T_1) \cap \text{ext}(T_2) \) is regular and therefore also \( \text{ext}(T_1 \cap T_2) \). Another application of Theorem 1.9 yields that \( T_1 \cap T_2 \) is regular.

Similar arguments work for union and complement. □

It is also possible to show Theorem 1.10 directly by giving automaton constructions that are similar to the ones for the ranked case. But because of the horizontal languages these constructions are more technical to write.

Finally, we mention that we can also use the encodings to solve decision problems for hedge automata using the results from Section 1.3.2. The translations forth and back from hedge automata to binary tree automata are polynomial, and thus we obtain the same complexity bounds as in Section 1.3.2. Note however, that there is a variety of formalisms for representing the horizontal languages in hedge automata which depend on the application at hand. Of course, this has an influence on the
complexity. The above statement on the transfer of complexity bounds from NFTAs assumes that the horizontal languages are represented by regular expressions or nondeterministic finite automata. A more detailed analysis of algorithms for hedge automata can be found in [16].

1.4.2. Grammar based formalisms

Document type definitions for XML documents can be seen on an abstract level as definitions of regular tree languages. However, the common formalisms for such specifications are not based on automata but rather on grammars. Intuitively, a grammar generates trees instead of processing and then rejecting or accepting them. The mechanism is the same as for grammars generating word languages: A set of production rules specifies how nonterminal symbols can be replaced, where the new object that is substituted for a nonterminal can contain further nonterminals. This process is repeated until (after finitely many steps) an object without nonterminal symbols is reached.

Depending on the kind of production rules that are allowed, one obtains classes of grammars with different expressive power. On words this leads to the well-known Chomsky hierarchy. The lowest level consists of the regular languages generated, e.g., by right linear grammars, where the production rules are of the form $X \rightarrow wY$ for nonterminals $X, Y$, and a terminal word $w$, i.e., the nonterminals are only allowed to appear at the end of the word. In the tree setting this corresponds to the nonterminals only occurring at the leaves. This leads to rules of the form $X \rightarrow t$, where $X$ is a nonterminal, and $t$ is a tree in which nonterminal symbols only occur at leaves. Because we are interested in unranked trees, we also need a mechanism for horizontal recursion allowing to generate trees of unbounded width.

One possibility is to allow hedges on the right-hand sides of the rules. For example, the rules $X_1 \rightarrow a(X_2)$ and $X_2 \rightarrow b(c) X_2 \mid \varepsilon$ would allow to generated trees of the form $a(b(c) \cdots b(c))$. The hedge $b(c) X_2$ on the right-hand side of the second rule allows to produce an unbounded number of $b(c)$ below the root. To ensure that we only obtain regular languages one has to restrict this usage of nonterminals in hedges to right linear rules (as indicated above for words). We do not give formal definitions here because we use another popular variant of regular tree grammars that directly allow regular expressions in the rules and thus avoid the recursive usage of nonterminals on the horizontal level.

A regular tree grammar is of the form $G = (N, \Sigma, S, P)$, where $\Sigma$ is the unranked alphabet (the letter are also called terminal symbols), $N$ is a finite set of nonterminals, $S \in N$ is the start symbol, and $P$ is a finite set of rules of the form $X \rightarrow a(r)$, where $a \in \Sigma$ and $r$ is a regular expression over $N$.

The right-hand side of each rule defines a set of trees of height 1 or height 0 in case $\varepsilon$ is in the language defined by the regular expression. A derivation of such a grammar starts with the nonterminal $S$ and in each step replaces a nonterminal $X$ with a tree $a(w)$ if there is a rule $X \rightarrow a(r)$, and $w \in N^*$ is in the language defined
The language $T(G)$ defined by $G$ is the set of all trees in $T^\text{unr}_\Sigma$ that can be generated in this way (in finitely many steps).

**Example 1.5.** The language from Example 1.4 is generated by the grammar with start symbol $S_\top$ and the following rules:

\[
S_\top \rightarrow e(X^*S_\top X^*) \\
Y_d \rightarrow e(X^*X_dX^*) \\
X \rightarrow e(X^*) \\
S_\top \rightarrow a(X^*Y_dX^*) \\
X_d \rightarrow d(X^*)
\]

for each $e \in \Sigma$. \hfill \triangleright

The translation between regular tree grammars and hedge automata is rather simple. In particular, the definition for regular tree grammars that we have given here is very close to the definition of hedge automata: the nonterminals are in correspondence to the states of the automaton, and the production rules correspond to the transitions (compare Examples 1.4 and 1.5).

**Remark 1.1.** The languages that can be generated by regular tree grammars are exactly the regular tree languages.

By imposing restrictions on the grammars one can obtain common languages that are used for defining types of XML documents. Such an analysis is given in [39] (see also [16]). *Document type definitions* (DTD) correspond to local grammars, in which nonterminals can be identified with the symbols from $\Sigma$, and the rules are of the form $a \rightarrow a(r)$, i.e., each nonterminal generates the terminal symbol it corresponds to.

In *XML Schema* the nonterminals are typed versions of the terminals, and the rules are of the form $a^{(i)} \rightarrow a(r)$, where the superscript on the left-hand side of the rule indicates the type of the nonterminal. This itself is not yet a true restriction for regular tree grammars, only a naming convention for nonterminals. Regular grammars following this convention are also referred to as extended DTDs (EDTDs). XML Schema corresponds to so-called single type EDTDs: the regular expressions on the right-hand side of the rules are restricted such that they do not produce words that contain two different types of the same symbol. For example, the rule $a^{(1)} \rightarrow a(b^{(1)}b^{(2)})$ is not allowed because the word $b^{(1)}b^{(2)}$ contains two different types of $b$. The reason for imposing these restrictions is to allow efficient membership tests. A detailed analysis of XML Schema and single type EDTDs is given in [37] (where extended DTDs are called specialized DTDs).

### 1.5. Tree-walking automata

In the previous sections we have seen automata models that work in parallel spread over the whole input tree. The transitions define how to merge the information computed on different subtrees. This section deals with a sequential automaton model introduced in [2]. A run of such an automaton (on a ranked tree) starts by $r$. The language $T(G)$ defined by $G$ is the set of all trees in $T^\text{unr}_\Sigma$ that can be generated in this way (in finitely many steps).
at the root. Based on the current state, the label of the current node, and the information on the position of the current node among its siblings, the automaton can move to one of its children or to its parent node (and change its state at the same time). In general, a run can cross a node of a tree several times. The automaton accepts by switching to a final state.

Let \( \Sigma \) be a ranked alphabet and let \( k \) be the maximal arity of a symbol from \( \Sigma \). A tree-walking automaton (TWA) is of the form \( A = (Q, \Sigma, q^i, \Delta, F) \), where \( Q \) is a finite set of states, \( q^i \in Q \) is the initial state, \( F \subseteq Q \) is the set of final states, and \( \Delta \) is the transition relation. A transition depends on the current state as well as the label and type of the current node. The type of the current node is either root or a number in \( \{1, \ldots, k\} \), where number \( i \) means that the node is the \( i \)th child of its parent. We let \( \text{Types} = \{\text{root}, 1, \ldots, k\} \). Based on this information, a transition determines a new state and a direction in which to move. In general, the possible directions are \( \uparrow \) (move up to the parent node), \( \varepsilon \) (stay at the current node), and \( 1, \ldots, k \) (move to the corresponding child). The set of directions is denoted by \( \text{Dir} = \{\uparrow, \varepsilon, 1, \ldots, k\} \).

The transition relation is of the form \( \Delta \subseteq Q \times \text{Types} \times \Sigma \times Q \times \text{Dir} \), where of course the direction specified in a transition has to be compatible with the other information: If \( (q, \text{type}, a, q', d) \in \Delta \), then \( d \in \{\uparrow, \varepsilon, 1, \ldots, |a|\} \), and if \( \text{type} = \text{root} \) then \( d \neq \uparrow \).

A configuration of a TWA is a pair \((u, q)\) of a node and a state. A run of a TWA \( A \) on a tree \( t \) is a sequence \((u_0, q_0)(u_1, q_1)\cdots\) of configurations that starts in the initial state at the root, i.e., \( u_0 = \varepsilon \) and \( q_0 = q^i \), and each two successive configurations are related by a transition in \( \Delta \). A formal definition of the latter statement requires a lot of case distinctions. We only give two typical examples:

- If \( u_{i+1} \) is the \( j \)th child of \( u_i \), i.e., \( u_{i+1} = u_{i,j} \), and \( u_i \) is the root, then there must be a transition \((q_i, \text{root}, t(u_i), q_{i+1}, j) \in \Delta \). This is shown on the left-hand side of Figure 1.8.

- If \( u_i \) is not the root but the \( \ell \)th child of some other node, then the type root in the transition must be replaced by \( \ell \).

If \( u_i \) is the \( j \)th child of \( u_{i+1} \), i.e., \( u_i = u_{i+1,j} \), then there must be a transition \((q_i, j, t(u_i), q_{i+1}, \uparrow) \in \Delta \). This is shown on the right-hand side of Figure 1.8.

![Fig. 1.8. Illustration of transitions of a TWA](image-url)
A run is accepting if it ends in a final state at the root. Note that runs of a TWA might be infinite because they start looping at some point. These runs are of course non-accepting. The language $T(A)$ of a TWA is defined as usual as the set of trees for which there exists an accepting run of $A$ on $t$.

The possibility of circular runs that do not terminate are one of the reasons that make TWAs difficult to analyze. For example, this prevents easy complementation of deterministic TWAs by exchanging final and non-final states. However, in [40] it is shown that every deterministic TWA can be turned into a deterministic TWA that does not admit circular runs (it follows that deterministic TWAs can be complemented).

**Example 1.6.** The most basic example is a TWA performing a depth-first left-to-right search in the tree. For simplicity we assume that we are working on an alphabet with one binary symbol $a$, and one constant $c$. We use states $q_l, q_r, q_u$ (for left, right, and up), and one final state $q_f$. The initial state is $q_l$, and the transitions are

- $(q_l, \text{type}, a, q_l, 1)$ for all $\text{type} \in \text{Types}$(left)
- $(q_l, 1, c, q_r, \uparrow)$
- $(q_l, 2, c, q_u, \uparrow)$
- $(q_r, \text{type}, a, q_l, 2)$ for all $\text{type} \in \text{Types}$(left)
- $(q_u, 1, a, q_r, \uparrow)$
- $(q_u, 2, a, q_u, \uparrow)$
- $(q_u, \text{root}, a, q_f, \varepsilon)$

An run of this automaton is depicted in Figure 1.9. Note that the automaton from this example simply traverses the whole tree and then accepts at the root, i.e., its language is the set of all trees. But now one can combine this generic automaton with other automata that test certain properties while traversing the tree.

![Fig. 1.9. Run of a TWA doing a depth-first search (Example 1.6)](image)

The way TWAs process an input tree is completely different from the way it is done by NFTAs. A first question that comes up is whether the ability of visiting parts of the tree several times gives more power to TWAs. But similar to the case of two-way automata on finite words this is not the case.

**Theorem 1.11.** TWAs only recognize regular languages.
Proof. It is not difficult to show that \( \sim_T \) has finite index for a TWA language \( T \). Fix a TWA \( A \) with the usual components for the language \( T \) and consider for each tree \( t \) and each type \( \text{type} \) the possible behaviors of \( A \) on \( t \) if \( t \) is a subtree of type \( \text{type} \). For the type root the behavior is simply the information whether \( t \) is accepted by \( A \) or not. If \( t \) is of type \( i \) (i.e., it is the \( i \)th subtree of some node), then the behavior is a relation containing all pairs \((p,q)\) of states such that \( A \) started at the root of \( t \) in state \( p \) has a computation that remains on \( t \) and reaches \( q \) when exiting \( t \).

If the behavior of \( A \) on two trees \( t_1 \) and \( t_2 \) is the same, then \( t_1 \sim_T t_2 \) because

- for the trivial context \( C = X \) it follows that \( C[t_1] \in T \iff C[t_2] \in T \) because the behavior of \( A \) includes the information whether the trees are accepted or not, and
- for a nontrivial context \( C \), an accepting run of \( A \) on \( C[t_1] \) can be turned into an accepting run of \( A \) on \( C[t_2] \) (and vice versa): whenever the run enters the subtree \( t_1 \) with state \( p \) and exists with state \( q \) at the parent node, then we know that this is also possible on \( t_2 \).

Since there are only finitely many possible behaviors, \( T \) is regular according to Theorem 1.7. \( \square \)

It has been open for a long time whether TWAs can recognize all regular languages. Recently, it has been shown that this is not the case. Before that it has been shown for some restricted models of TWA that they are weaker than NFTAs (e.g. [43])

\textbf{Theorem 1.12 ([9])}. There exists a regular tree language that cannot be recognized by a TWA.

As opposed to parallel tree automata, where the nondeterministic and the deterministic model have the same expressive power, determinism is a true restriction for TWAs.

\textbf{Theorem 1.13 ([8])}. There exists a tree language that can be accepted by a TWA but not by a deterministic TWA.

This already illustrates the difference between NFTAs and TWAs. The two models also differ w.r.t. algorithmic questions, as witnessed, e.g., by the following theorem.

\textbf{Theorem 1.14}. The emptiness problem for TWAs is \( \text{Exptime-complete} \).

The reason for this cost is the ability of TWAs to move up and down in the tree. This allows a TWA to sequentially check certain properties that have to be checked in one pass by an NFTA. The \( \text{Exptime} \) lower bound can be shown by a reduction from the two-person corridor tiling problem used in the proof of Theorem 1.5: One can construct a TWA of polynomial size (even a deterministic one) that verifies that
a given tree represents a winning strategy for Eva in a given two-person corridor tiling system. The upper bound can be derived from an exponential translation of TWAs into standard tree automata and the polynomial time emptiness test for the latter (Theorem 1.4). Such an exponential translation can, e.g., be derived from a corresponding result for the more general model of two-way alternating tree automata [17]. That this exponential blow-up cannot be avoided, in general, easily follows from results on the succinctness of two-way automata on words (see [27]).

For more references on TWAs (in particular on extensions of TWAs by pebbles) we refer the reader to the abstract\footnote{An extended version of this abstract can be found on the web page of the author.} [7].

1.5.1. Streams

Because of their sequential behavior, tree-walking automata can be seen as automata working on a linearized version of the tree. However, for navigating inside the linear representation they still use the tree structure. In the context of XML document processing there is a natural setting in which the automaton cannot work on the document as a tree but has to work on a linearized version: when documents are exchanged over a network they arrive as a stream. If large documents are exchanged, then it is desirable to process them online such that it is not necessary to store them completely before analyzing them.

Given a tree, there are various ways of presenting it as a linear structure. In XML documents opening and closing tags are used. We use the symbol itself as opening tag and an over-lined version of the symbol as closing tag. The coding is obtained by a depth-first traversal of the tree, putting an opening tag when entering a subtree, and a closing tag when leaving the subtree.

For example, the tree from the left-hand side of Figure 1.1 and the tree from Figure 1.2 are, respectively, coded as

\[ \text{abac} \bar{c} \text{abc} \bar{d} \text{ab} \bar{c} \bar{d} \text{abc} \bar{a} \bar{b} \bar{a} \]

For a tree \( t \) (ranked or unranked) we refer to the above linearization as \( \text{lin}_{XML}(t) \), and for a tree language \( T \) we denote the set of all linearizations obtained from trees in \( T \) by \( \text{lin}_{XML}(T) \).

Languages of this kind have already been analyzed in [51] under the name of nest sets, and languages of the form \( \text{lin}_{XML}(T) \) for regular tree languages \( T \) are characterized by a specific type of context-free grammars. These grammars are studied in detail in [6].

It is not difficult to show that \( \text{lin}_{XML}(T) \) is not a regular word language in general for regular \( T \). The reason is that a finite automaton cannot check whether all opening tags are closed correctly. In the setting where we are interested in verifying streaming objects, it is often reasonable to assume that the object is well-formed (i.e., all opening tags are closed correctly). The question whether a finite automaton can test for the membership in \( \text{lin}_{XML}(T) \) when the inputs are assumed
to be well-formed is much more interesting. Consider, for example, the (unranked) tree language $T$ that contains all trees whose right-most branch is completely labeled by $a$. Obviously, $\text{linXML}(T)$ is not regular but it contains exactly those words from $\Sigma^* a a^+$ that correspond to well-formed inputs.

The problem of deciding for a regular tree language whether $\text{linXML}(T)$ can be verified by a finite automaton in the above sense is still open in its full generality. Partial results have been obtained in [48, 49].

The problem becomes different when considering another coding with only one type of parenthesis: We again traverse the tree by a depth first search. When entering a subtree we put the symbol and an opening parenthesis, when leaving the subtree we put a closing parenthesis. For the same example trees used above we obtain the following linearizations

$$a(b(a(c())d()))c() \text{ and } a(b(c())d())a(a(b())).$$

Because this notation corresponds to the standard way how trees are written as terms, we refer to this coding of $t$ by $\text{linterm}(t)$ and $\text{linterm}(T)$ for languages $T$. One can easily see that $\text{linterm}(T)$ can be obtained from $\text{linXML}(T)$ by the simple morphism that maps each $a$ to $a(\cdot$ and each $\bar{a}$ to $\bar{a})$. Therefore, an automaton accepting $\text{linXML}(T)$ when restricted to well-formed inputs, can easily be turned into an automaton for $\text{linterm}(T)$ on well-formed inputs.

In the other direction such a translation does not work. For the above example language $T$ containing those trees whose right-most branch is completely labeled by $a$, there is no regular language of words that yields $\text{linterm}(T)$ when restricted to well-formed inputs (this can be shown, e.g., using the techniques developed in [4]).

It turns out that this difference makes the corresponding decision problem simpler: In [4] it is shown that it is decidable for a regular tree language $T$ whether $\text{linterm}(T)$ can be recognized by a finite automaton when the inputs are restricted to be well-formed. But for the application to processing streaming XML documents this result is less interesting because it does not use the standard coding.

Coming back to the first encoding, an automaton model that captures the languages $\text{linXML}(T)$ for regular $T$ can be derived from extended versions of tree-walking automata that are powerful enough to capture the regular tree languages. In [21] and [22] such extensions are mentioned. In the first paper it is shown that TWAs with colored marbles that can be dropped when entering a subtree and have to be picked up when leaving the subtree recognize all regular languages. In the second paper marbles are called invisible pebbles. Basically, for these extended TWAs it is enough to make one depth-first traversal of the tree. The marbles are used to give the TWA access to the parts of the run it has already computed. These automata are very closely related to specific kinds of pushdown automata as they are used, e.g., in [32] and [3].
1.6. Conclusion

In this chapter we have presented basic definitions and results in the field of tree automata. The material presented in Section 1.3 shows that a lot of concepts for the theory of automata over finite words can be adapted to the setting of ranked trees, in particular automata constructions for boolean operations on languages, minimization of automata, and the relation to monadic second-order logic. Also from an algorithmic point of view there are a lot of similarities to word automata, where in most cases the cost for algorithms on tree automata is one exponential higher than for word automata. Section 1.4 illustrates that it is possible to develop a theory of automata for unranked trees along the same lines as for ranked trees. Using encodings of unranked trees by ranked ones it is possible to lift many of the results from ranked tree automata. Tree-walking automata as presented in Section 1.5 process the input tree in a sequential way by navigating inside the tree. We have seen that this leads to a weaker expressive power and different algorithmic properties. Nevertheless, extensions of this model are useful, e.g., when dealing with logical formalisms like (Core)XPath [24] or first-order logic with transitive closure (see [22] and [14]).

There are various active research areas on different new aspects of tree automata, some of which are treated in other chapters of this volume. We have already mentioned the problem of processing streaming XML documents in Section 1.5.1. Another current topic that is also motivated from the XML world is the extension of automata to deal with data values from infinite domains (see Chapter ?? of this volume). Related to this are automaton models that allow the comparison of subtrees for equality and disequality (see [16] for an overview and [23, 28] for more recent developments).

Various applications do not just require to classify trees into accepted and rejected but are concerned with transformations of trees. There is a wide range of literature on tree transducers and we cannot survey all the suggested models here. We refer the reader to Chapter ?? of this volume, which deals with a specific kind of transducers with tree-walking automata as underlying mechanism.

Finally, for further reading we mention some recent surveys on different aspects of tree automata: The electronic book [16] covers a lot of material related to ranked tree automata and hedge automata. Automata related formalisms that are interesting in the context of XML are presented in [47] and logics for unranked trees are surveyed in [34].

Acknowledgements

I thank the editors for the invitation to contribute to this volume, and the anonymous referee for the helpful remarks and suggestions.
Basics on Tree Automata

References


Basics on Tree Automata


Index

[α], 3
@, 5
∼_T, 15
C_S, 14
T_S, 3
T_Σ^atr, 4
dom(t), 3
ext(t), 5
fcns(t), 5

child, 3
complete, 7
context, 14
corridor tiling system, 12

DFTA, 7
document type definitions, 22
domain, 3
DTD, 22

emptiness problem, 9
encoding
   extension, 5
   first-child-next-sibling, 4
equivalence problem, 10
equivalent, 6

FCNS, 4

hedge, 4
hedge automaton
   language of, 18
   nondeterministic, 18
height, 4

inclusion problem, 9

leaf, 3
   transitions, 6

membership problem, 9
monadic second-order logic, 16
MSO, 16
   -definable, 17

NFHA, 18
NFTA, 6
node, 3
   inner, 3

ranked alphabet, 3
regular tree grammar, 21
root, 3
run
   of an NFHA, 18
   of an NFTA, 6
   of a TWA, 23

sentence, 17
subtree, 4
successor, 3

tree, 3
tree-walking automaton, 23
tree automaton
   deterministic, 7
   language of, 6
   nondeterministic, 6
   top-down deterministic, 8
tree language, 6
   regular, 6
TWA, 23

universality problem, 9

XML Schema, 22