An Automata Toolbox

Version of May 25, 2018

Mikołaj Bojańczyk and Wojciech Czerwiński

Preface

These are lecture notes for a course on advanced automata theory, that we gave at the University of Warsaw in the years 2015-2018. The material was chosen to highlight interesting constructions; with a smaller emphasis on the theoretical and bibliographical context. The first part of the book – the lectures – is written by the first author, and the second part – the exercise solutions – is written by the second author. Nevertheless, we consulted each other extensively in the process of both teaching and writing.

Mikołaj Bojańczyk and Wojciech Czerwiński

Contents

1 Determinisation of ω-automata

In this chapter, we discuss automata for ω-words, i.e. infinite words of the form

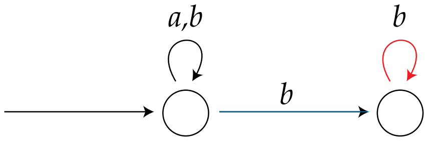
$$a_1 a_2 a_3 \cdots$$

We write Σ^ω for the set of ω words over alphabet Σ. The topic of this chapter is McNaughton's Theorem, which shows that automata over ω-words can be determinised. A more in depth account of automata (and logic) for ω words can be found in [56].

1.1 Automata models for ω-words

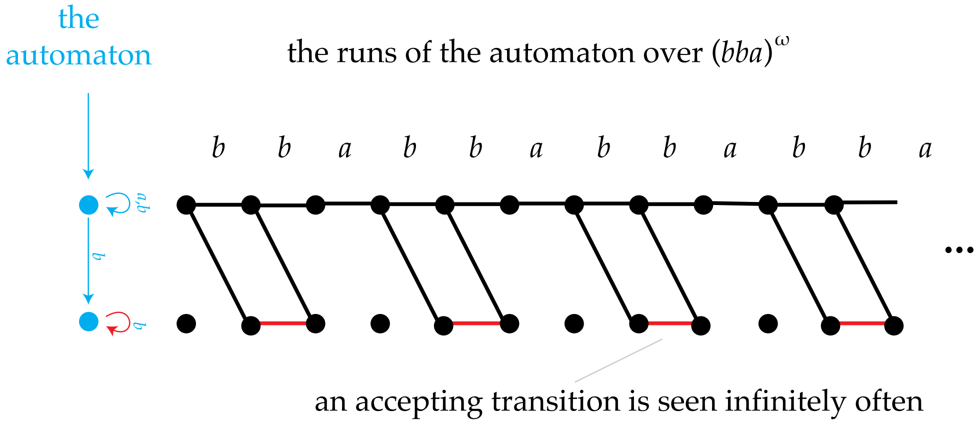
A *nondeterministic Büchi automaton* is a type of automaton for ω-words. Its syntax is typically defined to be the same as that of a nondeterministic finite automaton: a set of states, an input alphabet, initial and accepting subsets of states, and a set of transitions. For our presentation it is more convenient to use accepting transitions, i.e. the accepting set is a set of transitions, not a set of states. An infinite word is accepted by the automaton if there exists a run which begins in one of the initial states, and visits some accepting transition infinitely often.

Example 1. Consider the set of words over alphabet $\{a, b\}$ where the letter a appears finitely often. This language is recognised by a nondeterministic Büchi automaton like this (we adopt the convention that accepting transitions are red edges):



□

This chapter is about determinising Büchi automata. One simple idea would be to use the standard powerset construction, and accept an input word if infinitely often one sees a subset (i.e. a state of the powerset automaton) which contains at least one accepting transition. This idea does not work, as witnessed by the following picture describing a run of the automaton from Example 1:


the automaton
the runs of the automaton over $(bba)^\omega$
b b a b b a b b a b b a
a,b
b
b
...
an accepting transition is seen infinitely often

In fact, Büchi automata cannot be determinised using any construction.

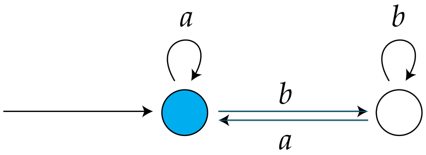
Fact 1.1. *Nondeterministic Büchi automata recognise strictly more languages than deterministic Büchi automata.*

Proof. Take the automaton from Example 1. Suppose that there is a deterministic Büchi automaton that is equivalent, i.e. recognises the same language. Let us view the set of all possible inputs as an infinite tree, where the vertices are prefixes $\{a,b\}^*$. Since the automaton is deterministic, to each edge of this tree one can uniquely assign a transition of the automaton. Every vertex $v \in \{a,b\}^*$ of this tree has an accepting transition in its subtree, because the word vb^ω should have an accepting run. Therefore, we can find an infinite path in this tree which has a infinitely often and uses accepting transitions infinitely often. ■

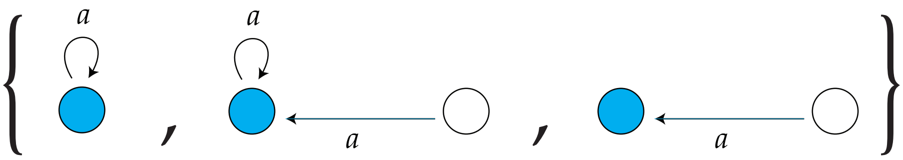
The above fact shows that if we want to determinse automata for ω-words, we need something more powerful than the Büchi condition. One solution is called the Muller condition, and is described below. Later we will see another (equivalent) solution, which is called the parity condition.

Muller automata. The syntax of a Muller automaton is the same as for a Büchi automaton, except that the accepting set is different. Suppose that Δ is the set of transitions. Instead of being a set $F \subseteq \Delta$ of transitions, the accepting set in a Muller automaton is a family $\mathcal{F} \subseteq \mathcal{P}(\Delta)$ of sets of transitions. A run is defined to be *accepting* if the set of transitions visited infinitely often belongs to the family $\mathcal{F}$.

Example 2. Consider this automaton



Suppose that we set $\mathcal{F}$ to be all subsets which contain only transitions that enter the blue state, as in the following picture.



a set of transitions is visualised as the part of the automaton that only uses transitions from that set

it is impossible to see this particular set of transitions (and no others) infinitely often

In this case, the automaton will accept words which contain infinitely many a's and finitely many b's. If we set $\mathcal{F}$ to be all subsets which contain at least one transition that enters the blue state, then the automaton will accept words which contain infinitely many a's. □

Deterministic Muller automata are clearly closed under complement – it suffices to replace the accepting family by $\mathcal{P}(\Delta) - \mathcal{F}$. This lecture is devoted to proving the following determinisation result.

Theorem 1.2 (McNaughton's Theorem)**.** *For every nondeterministic Büchi automaton there exists an equivalent (accepting the same ω-words) deterministic Muller automaton.*

The converse of the theorem, namely that deterministic Muller (even nondeterministic) automata can be transformed into equivalent nondeterministic Büchi automata is more straightforward, see Exercise 7. It follows from the above discussion that

- nondeterministic Büchi automata
- nondeterministic Muller automata
- deterministic Muller automata

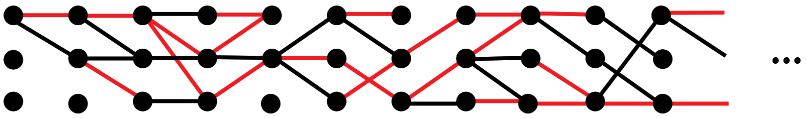
have the same expressive power, but deterministic Büchi automata are weaker. Theorem 1.2 was first proved by McNaughton in [37]. The proof here is similar to one by Muller and Schupp [40]. An alternative proof method is the Safra Construction, see e.g. [56].

The proof strategy is as follows. We first define a family of languages, called universal Büchi languages, and show that the McNaughton's theorem boils down to recognising these languages with deterministic Muller automata. Then we do that.

The universal Büchi language. For $n \in \mathbb{N}$, define a width n dag to be a directed acyclic graph where the nodes are pairs $\{1, \ldots, n\} \times \{1, 2, \ldots\}$ and every edge is of the form

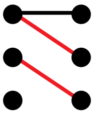
$$(q, i) \to (p, i+1) \qquad \text{for some } p, q \in \{1, \ldots, n\} \text{ and } i \in \{1, 2, \ldots\}.$$

Furthermore, every edge is either red or black, with red meaning "accepting". We assume that there are no multiple edges (i.e. there is at most one edge connecting a given source and target). Here is a picture of a width 3 dag:



In the pictures, we adopt the convention that the i-th column stands for the set of vertices $\{1, \ldots, n\} \times \{i\}$. The top left corner of the picture, namely the vertex $(1, 1)$ is called the *initial vertex*.

The essence of McNaughton's theorem is showing that for every n, there is a deterministic Muller automaton which inputs a width n dag and says if it contains a path that begins in the initial vertex and visits infinitely many red (accepting) edges. In order to write such an automaton, we need to encode as a width n dag as an ω-word over some finite alphabet. This is done using an alphabet, which we denote by $[n]$, where the letters look like this:



Formally speaking, $[n]$ is the set of functions

$$\{1,\ldots,n\} \times \{1,\ldots,n\} \to \{\text{no edge, non-accepting edge, accepting edge}\}.$$

Define the *universal n state Büchi language* to be the set of words $w \in [n]^\omega$ which, when treated as a width n dag, contain a path that starts in the initial vertex and visits accepting edges infinitely often. The key to McNaughton's theorem is the following proposition.

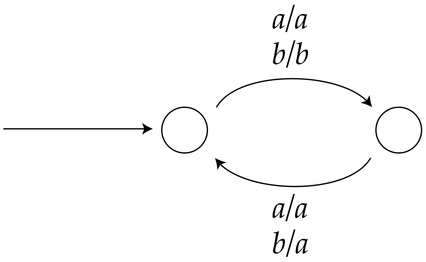
Proposition 1.3. *For every $n \in \mathbb{N}$ there is a deterministic Muller automaton recognising the universal n state Büchi language.*

Before proving the proposition, let us show how it implies McNaughton's theorem. To make this and other proofs more transparent, it will be convenient to use transducers. Define a *sequential transducer* to be a deterministic finite automaton, without accepting states, where each transition is additionally labelled by a word over some output alphabet. In this section, we only care about the special case when the output words have exactly one letter; this is sometimes called a *letter-to-letter* transducer. The name "transducer" refers to an automaton which outputs more than just yes/no; later in this book we will see other (and more powerful) types of transducers, with names like rational transducer or regular transducer. If the input alphabet is Σ and the output alphabet is Γ, then a sequential transducer defines a function

$$f : \Sigma^\omega \to \Gamma^\omega.$$

Example 3. Here is a picture of a sequential transducer which inputs a word over $\{a, b\}$ and replaces letters on even-numbered positions by a.

a transition a/b means that
letter a is input, and letter b is output



□

Lemma 1.4. *Languages recognised by deterministic Muller automata are closed under inverse images of sequential letter-to-letter transducers, i.e. if $\mathcal{A}$ in the diagram below is a deterministic Muller automaton and f is a sequential transducer, there is a deterministic Muller automaton $\mathcal{B}$ which makes the following diagram commute:*

$$\begin{tikzcd}
\Sigma^\omega \arrow[r, "f"] \arrow[dr, "\mathcal{B}"'] & \Gamma^\omega \arrow[d, "\mathcal{A}"] \\
& \{yes, no\}
\end{tikzcd}$$

Proof. A straightforward product construction. The states of automaton $\mathcal{B}$ are pairs (state of the transducer f, state of the automaton $\mathcal{A}$). If the automaton is in state (p, q) and reads letter $a \in \Sigma$, then it does the following. Suppose that the transition of f when in state p and when reading letter a is

$$p \xrightarrow{a/b} p',$$

i.e. the output produced is $b \in \Gamma$ and the new state is p'. Suppose that the transition of $\mathcal{A}$ when in state q and when reading letter b is

$$q \xrightarrow{b} q'.$$

Then the automaton $\mathcal{B}$ has a transition of the form

$$(p, q) \xrightarrow{a} (p', q').$$

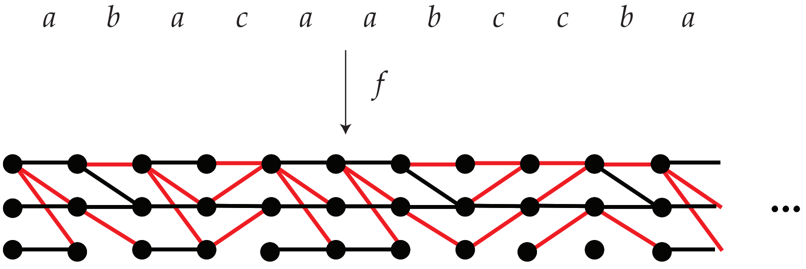
Note how each transition in $\mathcal{B}$ corresponds to two transitions, one in f and one in $\mathcal{A}$. The Muller condition is inherited from the automaton $\mathcal{A}$, i.e. a set of transitions in $\mathcal{B}$ is accepting if the corresponding set of transitions in $\mathcal{A}$ is accepting.

(The assumption that the transducer is letter-to-letter is not necessary, but then defining the Muller condition for $\mathcal{B}$ becomes a bit more complicated, because each transition of $\mathcal{B}$ corresponds to several transitions in $\mathcal{A}$.) ■

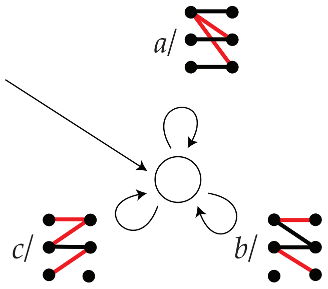
Let us continue with the proof of McNaughton's theorem. We claim that every language recognised by a nondeterministic Büchi automaton reduces to a universal Büchi language via some transducer. Let $\mathcal{A}$ be a nondeterministic Büchi automaton with input alphabet Σ. We assume without loss of generality that the states are numbers $\{1, \ldots, n\}$ and the initial state is 1. By simply copying the transitions of the automaton, one obtains a sequential transducer

$$f : \Sigma^\omega \to [n]^\omega$$

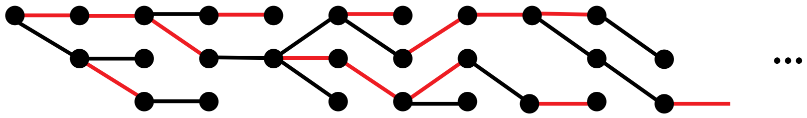
such that a word $w \in \Sigma^\omega$ is accepted by $\mathcal{A}$ if and only if $f(w)$ contains a path from the initial vertex with infinitely many accepting edges. Here is a picture:



The sequential transducer does even need states, i.e. one state is enough:



Using Lemma 1.4, we compose the transducer with the automaton from Proposition 1.3, getting a deterministic Muller automaton equivalent to $\mathcal{A}$. It now remains to show the proposition, i.e. that the n state universal Büchi language can be recognised by a Muller automaton. The proof has two steps. The first step is stated in Lemma 1.5 and says that a deterministic transducer can replace an arbitrary width n dag by an equivalent tree. Here we use the name *tree* for a width n dag, where every non-isolated node other than (1,1) has exactly one incoming edge. Here is a picture of such a tree, with the isolated nodes not drawn:



Lemma 1.5. *There is a sequential transducer*

$$f : [n]^\omega \to [n]^\omega$$

which outputs only trees and is invariant with respect to the universal Büchi language, i.e. if the input contains a path with infinitely many accepting edges, then so does the output and vice versa.

The second step is showing that a deterministic Muller automaton can test if a tree contains an accepting path.

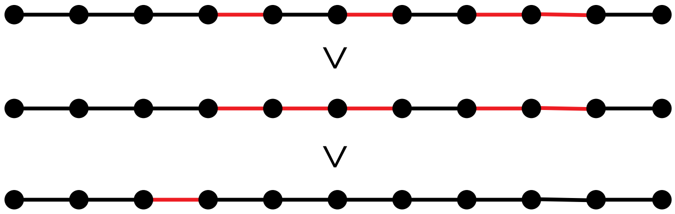
Lemma 1.6. *There exists a deterministic Muller automaton with input alphabet $[n]$ such that for every $w \in [n]^\omega$ that is a tree, the automaton accepts w if and only if w contains a path from the root with infinitely many accepting edges.*

Combining the two lemmas using Lemma 1.4, we get Proposition 1.3, and thus finish the proof of McNaughton's theorem. Lemma 1.5 is proved in Section 1.2 and Lemma 1.6 is proved in Section 1.3.

1.2 *Pruning the graph of runs to a tree*

We begin by proving Lemma 1.5, which says that a sequential transducer can convert a width n dag into a tree, while preserving the existence of a path from the initial vertex with infinitely many accepting edges. The transducer is simply going to remove edges.

Profiles. For a path π in a width n dag, define its *profile* to be the word of same length over the alphabet "accepting" and "non-accepting" which is obtained by replacing each edge with its appropriate type. We order profiles lexicographically, with "accepting" smaller than "non-accepting".



A finite path π in a width n dag is called *profile optimal* if it begins in the initial vertex, and its profile is lexicographically least among profiles of paths in w that begin in the initial vertex and have the same target as π.

Lemma 1.7. *There is a sequential transducer*

$$f : [n]^\omega \to [n]^\omega$$

such that if the input is w, then $f(w)$ is a tree with the same reachable (from the initial vertex) vertices as in w, and such that every finite path in $f(w)$ that begins in the root is a profile optimal path in w.

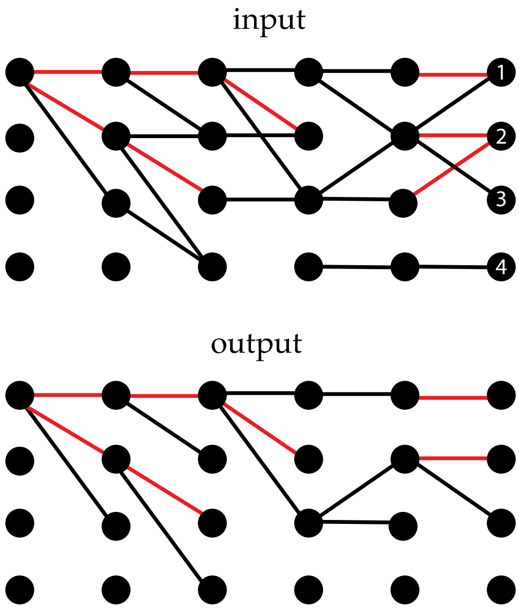
Proof. The key observation is that the prefix of a profile optimal path is also profile optimal. Therefore, if we want to do find a profile optimal path that leads to a vertex (q, i), we need to do the following. Consider all paths from the initial vertex to (q, i), decomposed as $\pi \cdot e$ where e is the last edge of the path and π is the remaining part of the path from the initial vertex to column $i - 1$. Because profile optimal paths are closed under prefixes, if we want $\pi \cdot e$ to be profile optimal, then π should be profile optimal. Since profiles are sorted lexicographically, then the profile of π should be optimal among profiles of paths that go from the initial vertex to some neighbour of (q, i) in the previous column $i - 1$. If there are several candidates for $\pi \cdot e$ with the same profile of π, then we should use those that have a smaller profile for e (i.e. is it "accepting" is preferred over "non-accepting"). In the end there might be several paths $\pi \cdot e$ that meet all of these criteria, and all of them are profile optimal.

Based on the discussion above, we describe a sequential transducer as in the statement of the lemma. After reading the first i letters, the automaton keeps in its memory the following information:

1. which vertices of the form (i, q) are targets of profile optimal paths, i.e. which ones are reachable from the initial vertex;

2. if both (i, q) and (i, p) are targets of profile optimal paths, then how are these profiles ordered.

The above information can be kept in the finite state space of the sequential transducer, since it consists of a subset of $\{1, \ldots, n\}$ together with an ordering on it (a total, transitive, reflexive but not necessarily antisymmetric relation). The information can be maintained by the automaton (i.e. it is enough to know the old information and the new letter to get the new information), and it is also enough to produce the output tree. Here is a picture of the construction:



The state of the tranducer is this information:

The reachable vertices are

❶ ❷ ❸

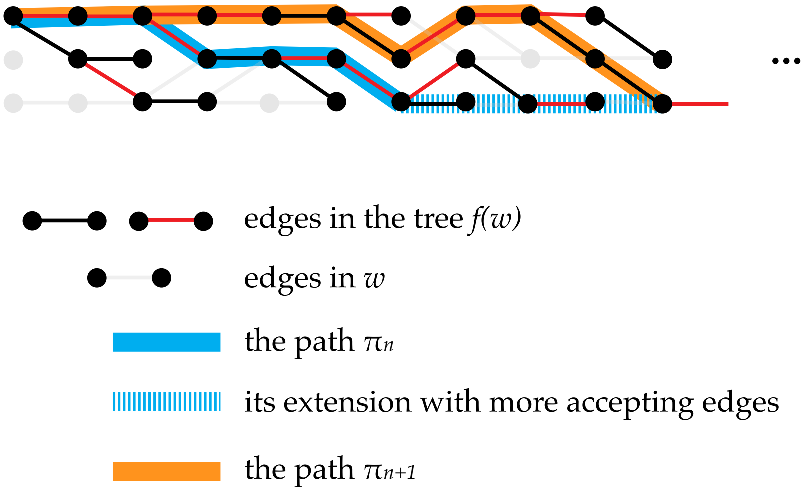
and the least profiles for reaching them are ordered as

❶ = ❷ < ❸

■

Lemma 1.8. *Let f be the sequential transducer from Lemma 1.7. If the input to f contains a path with infinitely many accepting edges, then so does the output.*

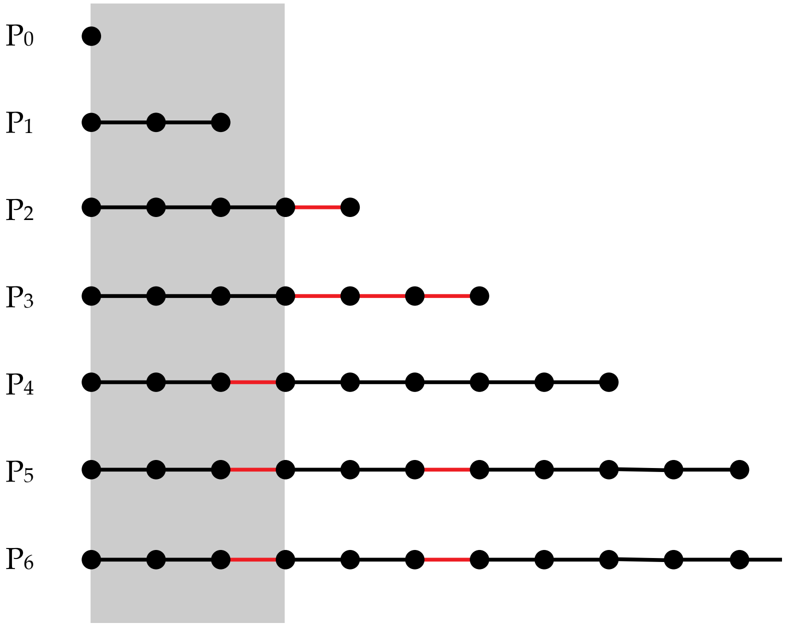
Proof. Assume that the input w to f contains a path with infinitely many accepting edges. Define a sequence $\pi_0, \pi_1, \ldots$ of finite paths in $f(w)$ as follows by induction. In the definition, we preserve the invariant that each path in the sequence $\pi_0, \pi_1, \ldots$ can be extended to an accepting path in the graph w. We begin with π_0 being the edgeless path that begins and ends in the root of the tree $f(w)$. This path π_0 satisfies the invariant, by the assumption that the input w contains a path with infinitely many accepting edges. Suppose that π_n has been defined. By the invariant, we can extend π_n to an infinite accepting path in the graph w, and therefore we can extend π_n to a finite path (call it σ_n) in w that contains at least one more accepting edge. Define π_{n+1} to be the unique path in the tree $f(w)$ which begins in the root of the tree $f(w)$ and has the same target as the new path that extends π_n with at least one accepting edge.



The path π_{n+1} satisfies the invariant, because its target is the same as the target of σ_n, and σ_n is a finite prefix of some accepting path. Define P_n to be the profile of the path π_n. By definition, the paths $\pi_1, \pi_2, \ldots$, and therefore also the corresponding profiles, get longer and longer. Furthermore, if profiles P_n and P_{n+1} have both length at least i, then the first i positions of P_n give a word that is lexicographically smaller than the first i positions of P_{n+1}, this is because the path π_{n+1} was taken from the tree $f(w)$ which had profile optimal paths. We claim that the sequence of profiles $P_0, P_1, P_2, \ldots$ has a well defined limit

$$\lim_{n \to \infty} P_n = P \in \{\text{accepting, non-accepting}\}^\omega.$$

More precisely, we claim that for every position i, the i-th letter of the profiles $P_1, P_2, \ldots$ eventually stabilises. The limit P is defined to be the sequence of these stable values. The limit exists because for every i, if we look at the prefixes of $P_0, P_1, \ldots$ of length i, then they get lexicographically smaller and smaller; and therefore they must eventually stabilise, as in the following picture:



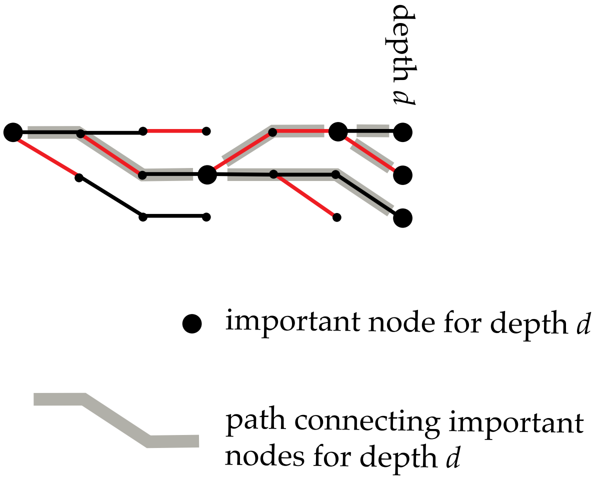
Claim 1.9. *The limit P contains the letter "accepting" infinitely often.*

Proof. Toward a contradiction, suppose that P has the letter "accepting" finitely often, i.e. there is some i such that after i, only the letter "non-accepting" appears in P. Choose n so that $\pi_n, \pi_{n+1}, \ldots$ have profile consistent with P on the first i letters. By construction, the profile P_{n+1} has an accepting letter on some position after i, and this property remains true for all subsequent profiles $P_{n+2}, P_{n+3} \ldots$ and therefore is also true in the limit, contradicting our assumption that P has only "non-accepting" letters after position i. ■

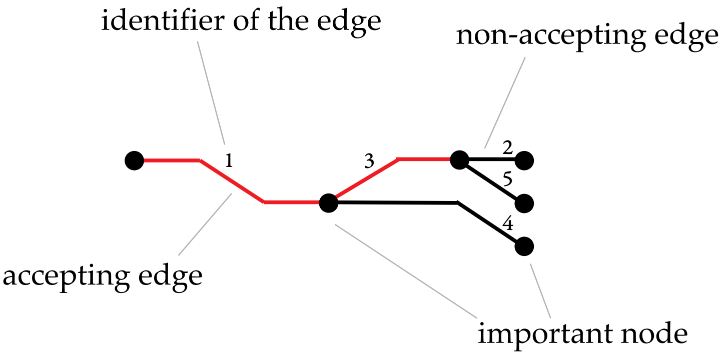
Consider the set of finite paths in the tree $f(w)$ which have profile that is a prefix of P. This set of paths forms a tree (because it is prefix-closed). This tree has bounded degree (assuming the parent of a path is obtained by removing the last edge) and it contain paths of arbitrary finite length (suitable prefixes of the paths $\pi_1, \pi_2, \ldots$). The König lemma says that every finitely branching tree with arbitrarily long paths contains an infinite path. Applying the König lemma to the paths in $f(w)$ with profile P, we get an infinite path with profile P. By Claim 1.9 this path has infinitely many accepting edges. ∎

1.3 *Finding an accepting path in a tree graph*

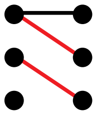
We now show Lemma 1.6, which says that a deterministic Muller automaton can check if a width n tree contains a path with infinitely many accepting edges. Consider a tree $t \in [n]^\omega$, and let $d \in \mathbb{N}$ be some depth. Define an *important node for depth d* to be a node which is either: the root, a node at depth d, or a node which is a closest common ancestor of two nodes at depth d. This definition is illustrated below (with red lines representing accepting edges, and black lines representing non-accepting edges):


depth d
● important node for depth d
path connecting important nodes for depth d

Definition of the Muller automaton. We now describe the Muller automaton for Lemma 1.6. After reading the first d letters of an input tree (i.e. after reading the input tree up to depth d), the automaton keeps in its state a tree, where the nodes correspond to nodes of the input tree that are important for depth d, and the edges correspond to paths in the input tree that connect these nodes. This tree stored by the automaton is a tree with at most n leaves, and therefore it has less than $2n$ edges. The automaton also keeps track of a colouring of the edges, with each edge being marked as accepting or not, where "accepting" means that the corresponding path in the input tree contains at least one accepting edge. Finally, the automaton remembers for each edge an identifiers from the set $\{1, \ldots, 2n-1\}$, with the identifier policy being described below. A typical memory state looks like this:

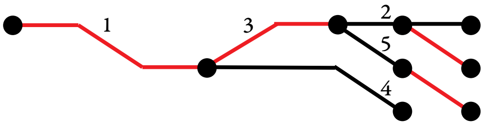


The big black dots correspond to important nodes for the current depth, red edges are accepting, black edges are non-accepting, while the numbers are the identifiers. All identifiers are distinct, i.e. different edges get different identifiers. It might be the case (which is not true for the picture above), that the identifiers used at a given moment have gaps, e.g. identifier 4 is used but not 3. The initial state of the automaton is a tree which has one node, which is the root and a leaf at the same time, and no edges. We now explain how the state is updated. Suppose the automaton reads a new letter, which looks something like this:

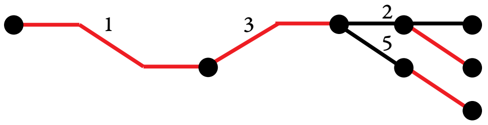


To define the new state, perform the following steps.

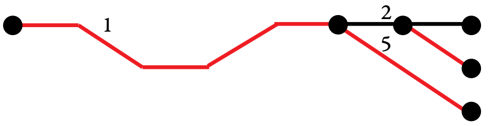
1. Append the new letter to the tree in the state of the automaton. In the example of the tree and letter illustrated above, the result looks like this:



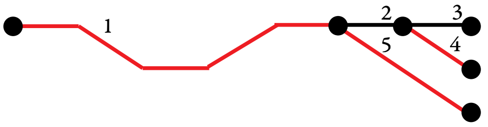
2. Eliminate paths that die out before reaching the new maximal depth. In the above picture, this means eliminating the path with identifier 4:



3. Eliminate unary nodes, thus joining several edges into a single edge. This means that a path which only passes through nodes of degree one gets collapsed to a single edge, the identifier for such a path is inherited from the first edge on the path. In the above picture, this means eliminating the unary nodes that are the targets of edges with identifiers 1 and 5:



4. Finally, if there are edges that do not have identifiers, these edges get assigned arbitrary identifiers that are not currently used. In the above picture, there are two such edges, and the final result looks like this:



This completes the definition of the state update function. We now define the acceptance condition.

The acceptance condition. When executing a transition, the automaton described above goes from one tree with edges labelled by identifiers to another tree with edges labelled by identifiers. For each identifier, a transition can have three possible effects, described below:

1. **Delete.** An edge can be deleted in step 2 or in step 3 (by being merged with an edge closer to the root). The identifier of such an edge is said to be deleted in the transition. Since we reuse identifiers, an identifier can still be present after a transition that deletes it, because it has been added again in step 4, e.g. this happens to identifier 4 in the above example.

2. **Refresh.** In step 3, a whole path $e_1 e_2 \cdots e_n$ can be folded into its first edge e_1. If the part $e_2 \cdots e_n$ contains at least one accepting edge, then we say that the identifier of edge e_1 is refreshed. This happens to identifiers 1 and 5 in the above example.

3. **Nothing.** An identifier might be neither deleted nor refreshed, e.g. this is the case for identifier 2 in the example.

The following lemma describes the key property of the above data structure.

Lemma 1.10. *For every tree in $[n]^\omega$, the following are equivalent:*

(a) the tree contains a path from the root with infinitely many accepting edges;

(b) some identifier is deleted finitely often but refreshed infinitely often.

Before proving the above fact, we show how it completes the proof of Lemma 1.6. We claim that condition (a) can be expressed as a Muller condition on transitions. The accepting family of subsets of transitions is

$$\bigcup_i \mathcal{F}_i$$

where i ranges over possible identifiers, and the family $\mathcal{F}_i$ contains a set X of transitions if

- some transition in X refreshes identifier i; and

- none of the transitions in X delete identifier i.

Identifier i is deleted finitely often but refreshed infinitely often if and only if the set of transitions seen infinitely often belongs to $\mathcal{F}_i$, and therefore, thanks to the fact above, the automaton defined above recognises the language in the statement of Lemma 1.6.

Proof of Lemma 1.10. The implication from (b) to (a) is straightforward. An identifier in the state of the automaton corresponds to a finite path in the input tree. If the identifier is not deleted, then this path stays the same or grows to the right (i.e. something is appended to the path). When the identifier is refreshed, the path grows by at least one accepting edge. Therefore, if the identifier is deleted finitely often and refreshed infinitely often, there is some path that keeps on growing with more and more accepting states, and its limit is a path with infinitely many accepting edges.
Let us now focus on the implication from (a) to (b). Suppose that the tree t contains some infinite path π that begins in the root and has infinitely many accepting edges. Call an identifier *active* in step d if the path described by this identifier in the d-th state of the run corresponds to an infix of the path π. Let I be the set of identifiers that are active in all but finitely many steps, and which are deleted finitely often. This set is nonempty, e.g. the first edge of the path π always has the same identifier. In particular, there is some step d, such that identifiers from I are not deleted after step n. Let $i \in I$ be the identifier that is last on the path π, i.e. all other identifiers in I describe finite paths that are earlier on π. It is not difficult to see that the identifier i must be refreshed infinitely often by prefixes of the path π. ∎

Problem 1. Are the following languages ω-regular (i.e. recognised by nondeterministic Büchi automata)?

1. ω-words which have infinitely many prefixes in a fixed regular language of finite words $L \subseteq \Sigma^*$;

2. ω-words with infinitely many infixes of the form ab^pa, where p is prime;

3. ω-words with infinitely many infixes of the form $ab^n a$, where n is even.

Problem 2. Call an ω-word *ultimately periodic* if it is of the form uv^ω for some finite words u, v. Show that if an ω-regular langauge is nonempty, then it contains an ultimately periodic word.

Problem 3. Let UP be the set of ultimately periodic words. Let K and L be ω-regular languages. Show that if $L \cap UP = K \cap UP$ then $K = L$.

Problem 4. Are the following languages ω-regular?

1. ω-words with arbitrarily long infixes belonging to a fixed regular language of finite words L;
2. ω-words which have infinitely many prefixes in a fixed language of finite words $L \subseteq \Sigma^*$ (not necessarily regular).

Problem 5. Show that the language of words "there exists a letter b" cannot be accepted by a nondeterministic automaton with the Büchi acceptance condition, where all the states are accepting (but possibly transitions over some letters in some states are missing).

Problem 6. Show that the language "finitely many occurrences of letter a" cannot be accepted by a deterministic automaton with the Büchi acceptance condition.

Problem 7. Show that every language accepted by a nondeterministic automaton with the Muller acceptance condition is also accepted by some nondeterministic automaton with the Büchi acceptance condition.

Problem 8. Show that nonemptiness is decidable for automata with the Muller acceptance condition.

Problem 9. Define a metric on ω-words by

$$d(u, v) = \frac{1}{2^{\text{diff}(u,v)}},$$

where $\mathrm{diff}(u, v)$ is the smallest position where u and v have different labels. A language L is called *open* (in this metric) if for every $w \in L$ there exists some open ball centered in w that is included in L (standard definition). Prove that the following conditions are equivalent for an ω-regular language L:

1. is open;
2. is of the form $K\Sigma^\omega$ for some $K \subseteq \Sigma^*$;
3. is of the form $K\Sigma^\omega$ for some regular $K \subseteq \Sigma^*$.

Problem 10. Which of the following candidates for a Myhill-Nerode congruence inded have the property: $\sim_L$ has finite index if and only if L is ω-regular

1. an equivalence relation $\sim_L$ on Σ^* where $u \sim_L v$ is defined by

$$uw \in L \Leftrightarrow vw \in L \qquad \text{for all } w \in \Sigma^\omega$$

2. an equivalence relation $\sim_L$ on Σ^ω where $u \sim_L v$ is defined by

$$wu \in L \Leftrightarrow wv \in L \qquad \text{for all } w \in \Sigma^*$$

3. an equivalence relation $\sim_L$ on Σ^* where $u \sim_L v$ is defined by

$$\text{and} \begin{cases} uw \in L \Leftrightarrow vw \in L & \text{for all } w \in \Sigma^\omega \\ s(ut)^\omega \in L \Leftrightarrow s(vt)^\omega \in L & \text{for all } s, t \in \Sigma^* \end{cases}$$

2
Infinite duration games

In this chapter, we prove the Büchi-Landweber Theorem [15, Theorem 1], see also [56, Theorem 6.5], which shows how to solve games with ω-regular winning conditions. These are games where two players move a token around a graph, yielding an infinite path, and the winner is decided based on some property of this path that is recognised by an automaton on ω-words. The Büchi-Landweber Theorem gives an algorithm for deciding the winner in such games, thus answering a question posed in [18] and sometimes called "Church's Problem".

2.1 Games

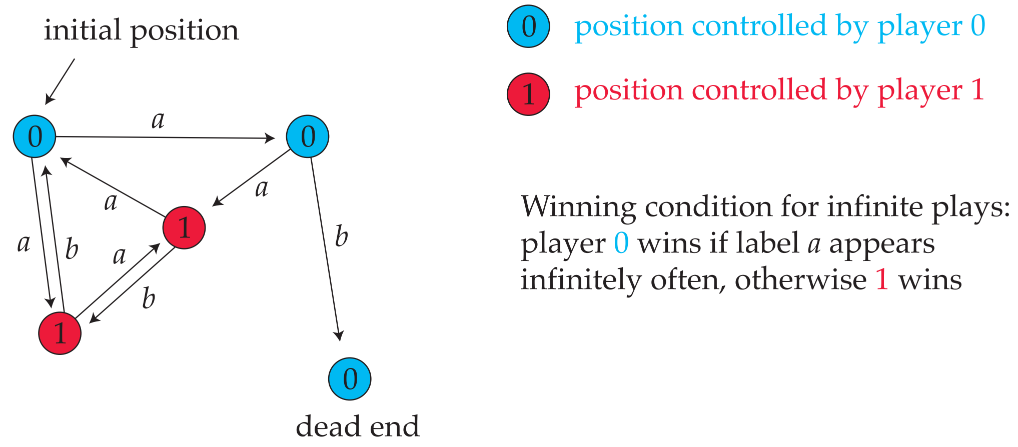
In this chapter, we consider games played by two players (called 0 and 1), which are zero-sum, perfect information, and most importantly, of potentially infinite duration.

Definition 2.1 (Game). *A* game *consists of*

- *a directed graph, not necessarily finite, whose vertices are called positions;*
- *a distinguished initial position;*
- *a partition of the positions into positions controlled by player 0 and positions controlled by player 1;*

- *a labelling of edges by a finite alphabet* Σ*, and a* winning condition*, which is a function from* Σ^ω *to the set of players* $\{0,1\}$*.*

Intuitively speaking, the winning condition inputs a sequence of labels produced in an infinite play, and says which player wins. The definition is written in a way which highlights the symmetry between the two players; this symmetry will play an important role in the analysis. Here is a picture.

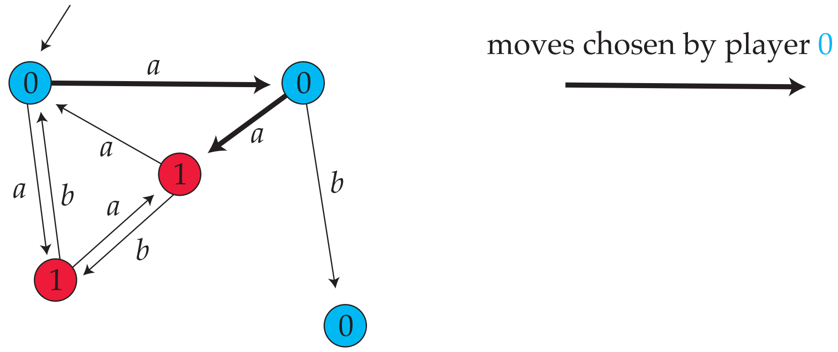

initial position
0
a
0
a
a
1
a
b
a
b
b
1
0
dead end
0 position controlled by player 0
1 position controlled by player 1

Winning condition for infinite plays: player 0 wins if label a appears infinitely often, otherwise 1 wins

The game is played as follows. The game begins in the initial position. The player who controls the initial position chooses an outgoing edge, leading to a new position. The player who controls the new position chooses an outgoing edge, leading to a new position, and so on. If the play reaches a position with no outgoing edges (called a dead end), then the player who controls the dead end loses immediately. Otherwise, the play continues forever, and yields an infinite path and the winner is given by applying the winning condition to the sequence of edge labels seen in the play.

To formalise the notions in the above paragraph, one uses the concept of a strategy. A *strategy* for player $i \in \{0,1\}$ is a function which inputs a history of the play so far (a path, possibly with repetitions, from the initial position to some position controlled by player i), and outputs the new position (consistent with the edge relation in the graph). Given strategies for both players, call these σ_0 and σ_1, a unique play (a path in the graph from the initial position) is obtained, which is either a finite path ending in a dead end, or an infinite path. This play is called winning for player i if it is finite and ends in a dead end controlled by the opposing player; or if it is infinite and winning for player i according to the winning condition. A strategy for player i is defined to be winning if for every every strategy of the opponent, the resulting play is winning for player i.

Example 4. In the game from the picture above, player 0 has a winning strategy, which is to always select the fat arrows in the following picture.



□

Determinacy. A game is called *determined* if one of the players has a winning strategy. Clearly it cannot be the case that both players have winning strategies. One could be tempted to think that, because of the perfect information, one of the players must have a winning strategy. However, because of the infinite duration, one can use the axiom of choice to come up with strange games where neither of the players has a winning strategy.
The goal of this chapter is to show a theorem by Büchi and Landweber: if the winning condition of the game is recognised by an automaton, then the game is determined, and furthermore the winning player has a finite memory winning strategy, in the following sense.

Definition 2.2 (Finite memory strategy)**.** *Consider a game where the positions are V. Let i be one of the players. A strategy for player i with memory M is given by:*

- *a deterministic automaton with states M and input alphabet V; and*
- *for every position $v \in V$ controlled by i, a function f_v from M to the neighbours of v.*

The two ingredients above define a strategy for player i in the following way: the next move chosen by player i in a position v is obtained by applying the function f_v to the state of the automaton after reading the history of the play, including v.

We will apply the above definition to games with possibly infinitely many positions, but we only care about finite memory sets M. An important special case is when the set M has only one element, in which case the strategy is called *memoryless*. For a memoryless strategy, the new position chosen by the player only depends on the current position, and not on the history of the game before that. The strategy in Example 4 is memoryless.

Theorem 2.3 (Büchi-Landweber Theorem). *Let Σ be finite and let*

$$\text{Win} : \Sigma^\omega \to \{0, 1\}$$

be ω-regular, i.e. the inverse image of 0 (and therefore also of 1) is recognised by a deterministic Muller automaton. Then there exists a finite set M such that for every game with winning condition Win*, one of the players has a winning strategy that uses memory M.*

The proof of the above theorem has two parts. The first part is to identify a special case of games with ω-regular winning conditions, called parity conditions, which map a sequence of numbers to the parity $\in \{0, 1\}$ of the smallest number seen infinitely often.

Definition 2.4 (Parity condition). *A parity condition is any function of the form*

$$w \in I^\omega \qquad \mapsto \qquad \begin{cases} 0 & \textit{if the smallest number appearing infinitely often in } w \textit{ is even} \\ 1 & \textit{otherwise} \end{cases}$$

for some finite set $I \subseteq \mathbb{N}$. A parity game *is a game where the winning condition is a parity condition.*

Parity games are important because not only can they be won using finite memory strategies, but even memoryless strategies are enough.

Theorem 2.5 (Memoryless determinacy of parity games)**.** *For every parity game, one of the players has a memoryless winning strategy.*

In fact, for edge labelled games (which is our choice) the parity condition is the only condition that admits memoryless winning strategies regardless of the graph structure of the game, among conditions that are prefix independent, see [20, Theorem 4].
The above theorem is proved in Section 2.2. The second step of the Büchi-Landweber theorem is a reduction to parity games. This essentially boils down to transforming deterministic Muller automata into deterministic parity automata, which are defined as follows: a parity automaton has a ranking function from states to numbers, and a run is considered accepting if the smallest rank appearing infinitely often is even. This is a special case of the Muller condition, but it turns out to be expressively complete in the following sense:

Lemma 2.6. *For every deterministic Muller automaton, there is an equivalent deterministic parity automaton.*

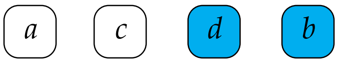
Proof. The lemma can be proved in two ways. One way is to show that, by taking more care in the determinisation construction in McNaughton's Theorem, we can actually produce a parity automaton. Another way is to use a data structure called the later appearance record [31]. The construction is presented in the following claim.

Claim 2.7. *For every finite alphabet Σ, there exists a deterministic automaton with input alphabet Σ, a totally ordered state space Q, and a function*

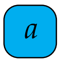
$$g : Q \to \mathcal{P}(\Sigma)$$

with the following property. For every input word, the set of letters appearing infinitely often in the input is obtained by applying g to the smallest state that appears infinitely often in the run.

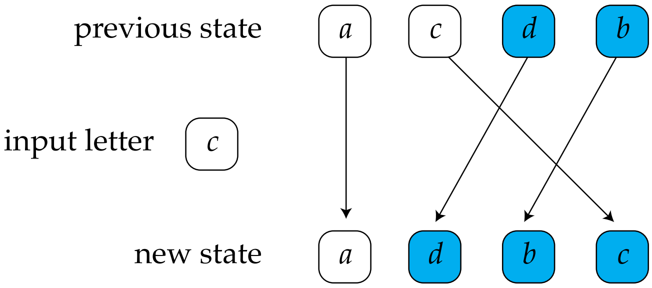
Proof. The state space Q consists of data structures that look like this:



More precisely, a state is a (possibly empty) sequence of distinct letters from Σ, with distinguished blue suffix. The initial state is the empty sequence. After reading the first letter a, the state of the automaton is



When that automaton reads an input letter, it moves the input letter to the end of the sequence (if it was not previously in the sequence, then it is added), and marks as blue all those positions in the sequence which were changed, as in the following picture:



Consider a run of this automaton over some infinite input $w \in \Sigma^\omega$. Take some blue suffix of maximal size that appears infinitely often in the run. Then the letters in this suffix are exactly those that appear in w infinitely often. Therefore, to get the statement of the claim, we order Q first by the number of white (not blue) positions, and in case of the same number of white positions, we use some arbitrary total ordering. The function g returns the set of blue positions. This completes the proof of the claim. ■

The conversion of Muller to parity is a straightforward corollary of the above lemma: one applies the above lemma to the state space of the Muller automaton, and defines the ranks according to the Muller condition. ■

Let us now finish the proof of the Büchi-Landweber theorem. Consider a game with an ω-regular winning condition. By Lemma 2.6, there is a deterministic parity automaton which accepts exactly those sequences of edge labels where player 0 wins. Consider a new game, call it the *product game*, where the positions are pairs (position of the original game, state of the deterministic parity automaton). Edges in the product game are of the form

$$(v, q) \xrightarrow{b} (w, p)$$

such that $v \xrightarrow{a} w$ is an edge of the original game (the label of the edge is on top of the arrow), the deterministic parity automaton goes from state q to state p when reading label a, and b is the number assigned to state q by the parity condition. It is not difficult to see that the following conditions are equivalent for every position v of the original game and every player $i \in \{0, 1\}$:

1. player i wins from position v in the original game;

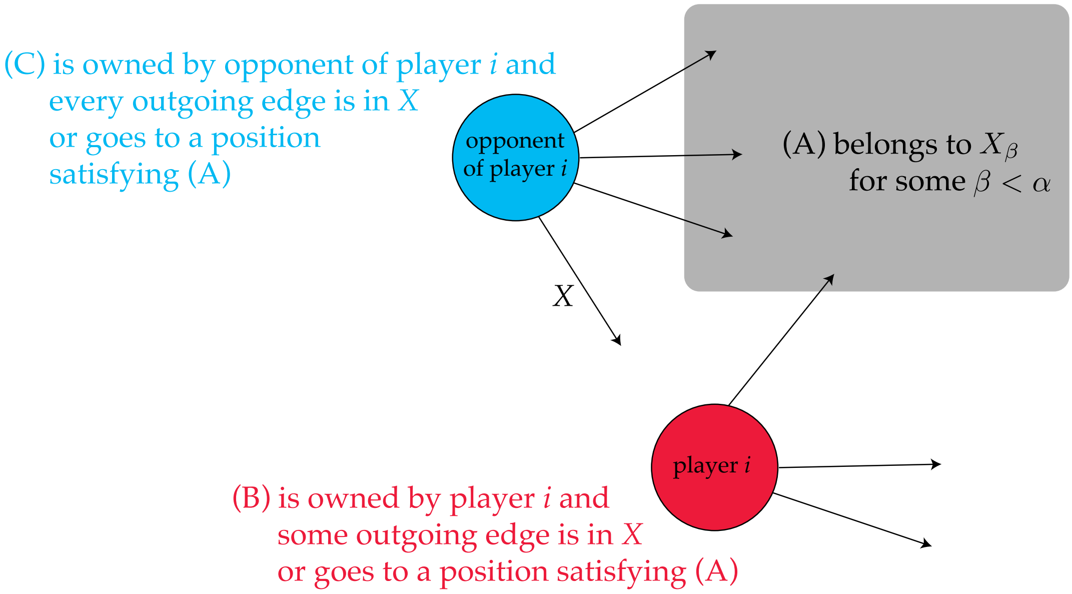
2. player i wins from position (v, q) in the product game, where q is the initial state of the deterministic parity automaton recognising L.

The implication from 1 to 2 crucially uses determinism of the automaton and would fail if a nondeterministic automaton were used (under an appropriate definition of a product game). Since the product game is a parity game, Theorem 2.5 says that for every position v, condition 2 must hold for one of the players; furthermore, a positional strategy in the product game corresponds to a finite memory strategy in the original game, where the memory is the states of the deterministic parity automaton.
This completes the proof of the Büchi-Landweber Theorem. It remains to show memoryless determinacy of parity games, which is done below.

2.2 *Memoryless determinacy of parity games*

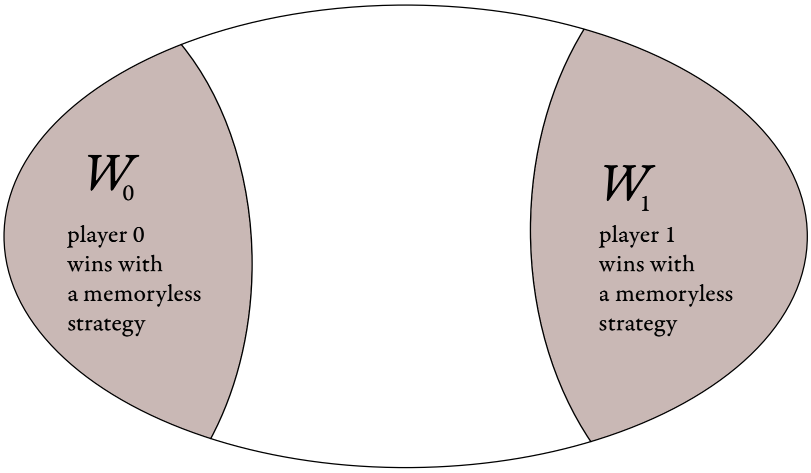
In this section, we prove Theorem 2.5 on memoryless determinacy of parity games. The proof we use is based in [61] and [56]. Recall that in a parity game, the positions are assigned numbers (called ranks from now on) from a finite set of natural numbers, and the goal of player i is to ensure that for infinite plays, the minimal number appearing infinitely often has parity i. Our goal is to show that one of the players has a winning strategy, and furthermore this strategy is memoryless. The proof of the theorem is by induction on the number ranks used in the game. We choose the induction base to be the case when there are no ranks at all, and hence the theorem is vacuously true. For the induction step, we use the notion of attractors, which is defined below.

Attractors. Consider a set of edges X in a parity game (actually the winning condition and labelling of edges are irrelevant for the definition). For a player $i \in \{0,1\}$, we define below the i-attractor of X, which intuitively represents positions where player i can force a visit to an edge from X. The attractor is approximated using ordinal numbers. (For a reader unfamiliar with ordinal numbers, just think of natural numbers, which are enough to treat the case of games with finitely many positions.) Define X_0 to be empty. For an ordinal number $\alpha > 0$, define X_α to be all positions which satisfy one of the conditions (A), (B) or (C) depicted below:


(C) is owned by opponent of player i and every outgoing edge is in X or goes to a position satisfying (A)
opponent of player i
X
(A) belongs to X_β for some $\beta < \alpha$
player i
(B) is owned by player i and some outgoing edge is in X or goes to a position satisfying (A)

The set X_α grows as the ordinal number α grows, and therefore at some point it stabilises. If the game has finitely many positions – or, more generally, finite outdegree – then it stabilises after a finite number of steps and ordinals are not needed. This stable set is called the *i-attractor of X*. Over positions in the i-attractor, player i has a memoryless strategy which guarantees that after a finite number of steps, the game will use an edge from X, or end up in a dead end owned by the opponent of player i. This strategy, called the attractor strategy, is to choose the neighbour that belongs to X_α with the smallest possible index α.

Induction step. Consider a parity game. By symmetry, we assume that the minimal rank used in the game is an even number. By shifting the ranks, we assume that the minimal rank is 0. For $i \in \{0,1\}$ define W_i to be the set of positions v such that if the initial position is replaced by v, then player i has a memoryless winning strategy. Define U to be the vertices that are in neither W_0 nor in W_1. Our goal is to prove that U is empty. Here is the picture:

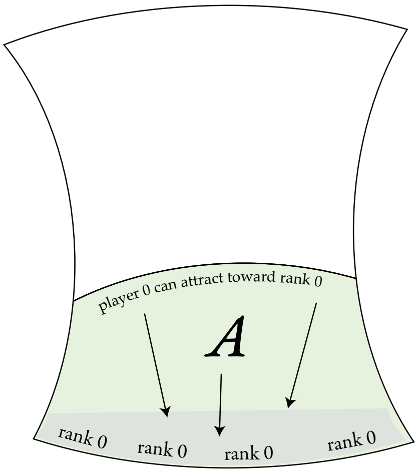


By definition, for every position in $w \in W_i$, player i has a memoryless winning strategy that wins when starting in position w. In principle, the memoryless strategy might depend on the choice of w, but the following lemma shows that this is not the case.

Lemma 2.8. *Let $i \in \{0,1\}$ be one of the players. There is a memoryless strategy σ_i for player i, such that if the game starts in W_i, then player i wins by playing σ_i.*

Proof. By definition, for every position $w \in W_i$ there is a memoryless winning strategy, which we call the *strategy of w*. We want to consolidate these strategies into a single one that does not depend on w. Choose some well-ordering of the vertices from W_i, i.e. a total ordering which is well-founded. Such a well-ordering exists by the axiom of choice. For a position $w \in W_i$, define its *companion* to be the least position v such that the strategy of v wins when starting in w. The companion is well defined because we take the least element, under a well-founded ordering, of some set that is nonempty (because it contains w). Define a consolidated strategy as follows: when in position w, play according to the strategy of the companion of w. The key observation is that for every play using this consolidated strategy, the sequence of companions is non-increasing in the well-ordering, and therefore it must stabilise at some companion v; and therefore the play must be winning for player i, since from some point on it is consistent with the strategy of v. ■

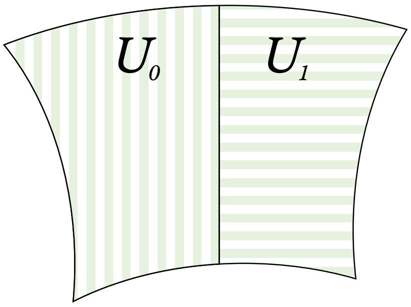
Define the game restricted to U to be the same as the original game, except that we only keep positions from U. In general restricting a game to a subset of positions might create new dead ends. However, in this particular case, no new dead ends will be created: if a position controlled by player i has all of its outgoing edges to $W_0 \cup W_1$, then a short analysis shows that the position is already in either $W_0 \cup W_1$. Define A to be the 0-attractor, inside the game limited to U, of the rank 0 edges in U (i.e. both endpoints are in U). Here is a picture of the game restricted to U:



Consider a position in A that is controlled by player 1. In the original game, all outgoing edges from the position go to $A \cup W_0$; because if there would be an edge to W_1 then the position would also be in W_1. It follows that:

(1) In the original game, if the play begins in a position from A and player 0 plays the attractor strategy on the set A, then the play is bound to either use an edge inside U that has minimal rank 0, or in the set W_0.

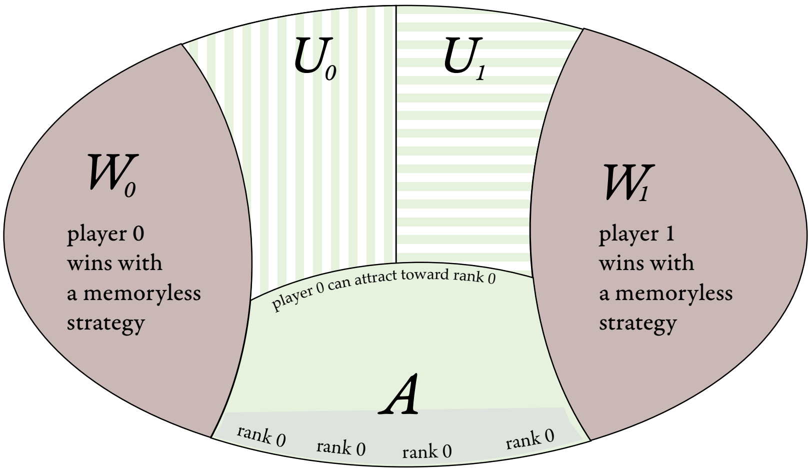
Consider the following game H: we restrict the original game to positions from $U - A$, and remove all edges which have minimal rank 0 (these edges necessarily originate in positions controlled by player 1, since otherwise they would be in A). Since this game does not use rank 0, the induction assumption can be applied to get a partition of $U - A$ into two sets of positions U_0 and U_1, such that on each U_i player i has a memoryless winning strategy in the game H:



Here is how the sets U_0, U_1 can be interpreted in terms of the bigger original game.

(2) In the original game, for every $i \in \{0,1\}$, if the play begins in a position from U_i and player i uses the memoryless winning strategy corresponding to U_i, then either (a) the play eventually visits a position from $A \cup W_0 \cup W_1$ or an edge with rank 0; or (b) player i wins.

Here is a picture of the original game with all sets:


U0
U1
W0
player 0 wins with a memoryless strategy
W1
player 1 wins with a memoryless strategy
player 0 can attract toward rank 0
A
rank 0
rank 0
rank 0
rank 0

Lemma 2.9. *U_1 is empty.*

Proof. Consider this memoryless strategy for player 1 in the original game:

- in U_1 use the winning memoryless strategy inherited from the game restricted to $U - A$;
- in W_1 use the winning memoryless strategy from Lemma 2.8;
- in other positions do whatever.

We claim that the above memoryless strategy is winning for all positions from U_1, and therefore U_1 must be empty by assumption on W_1 being all positions where player 1 can win in a memoryless way. Suppose player 1 plays the above strategy, and the play begins in U_1. If the play uses only edges that are in the game H, then player 1 wins by assumption on the strategy. The play cannot use an edge of rank 0 that has both endpoints in U, because these were removed in the game H. The play cannot enter the sets W_0 or A, because this would have to be a choice of player 0, and positions with such a choice already belong to W_0 or A. Therefore, if the play leaves $U - A$, then it enters W_1, where player 1 wins as well. ■

In the original game, consider the following memoryless strategy for player 0:

- in U_0 use the winning memoryless strategy from the game H;
- in W_0 use the winning memoryless strategy from Lemma 2.8;
- in A use the attractor strategy to reach a rank 0 edge inside U;
- on other positions, i.e. on W_1, do whatever.

We claim that the above strategy wins on all positions except for W_1, and therefore the theorem is proved. We first observe that the play can never enter W_1, because this would have to be a choice of player 1, and such choices are only possible in W_1. If the play enters W_0, then player 0 wins by assumption on W_0. Other plays will reach positions of rank 0 infinitely often, or will stay in U_0 from some point on. In the first case, player 0 will win by the assumption on 0 being the minimal rank. In the second case, player 0 will win by the assumption on U_0 being winning for the game restricted to $U - A$.
This completes the proof of memoryless determinacy for parity games, and also of the Büchi-Landweber Theorem.

Problem 11. We say that a game is *finite* if it has no infinite plays, i.e. every play eventually reaches a dead end. Prove that every finite game is determined, i.e. exactly one of the players has a winning strategy.

Problem 12. Show that reachability games played on finite game graphs can be solved in time proportional to the number of edges.

Problem 13. Show one player parity games can be solved in PTIME.

Problem 14. Show that solving parity games is in NP $\cap$ coNP.

Problem 15. Consider the following game on a finite game graph V together with function $rank : V \to \mathbb{N}$. At every moment of the play, the owner of the current vertex chooses a next vertex among current vertex successors. This continues until some vertex repeats on the play, i.e. till the first loop is closed. Then depending on the parity of the smallest rank on the loop the winning player is determined. Prove that player i in the described game wins iff player i wins in the parity game on the same arena.

Problem 16. Are Muller games positionally determined?

Problem 17. Show that Büchi games are positionally determined without direct use of the same result for parity games.

Problem 18. Show that the winning condition Muller games is a Borel set, and therefore Muller games are determined by Martin's theorem. (Most of this problem is looking up what Borel sets and Martin's theorem are.)

Problem 19. Show that Muller games on finite arenas are not positionally determined.

Problem 20. Construct an infinite game played on a finite game graph, in which player 0 has a winning strategy, but not a winning finite memory strategy. *Remark:* Notice that by Büchi-Landweber theorem the winning condition in that game cannot be ω-regular.

Problem 21. Consider the following riddle. There are infinitely many dwarfs (countably many). Every dwarf is given a hat, which is either red or green. Every dwarf sees the color of every hat beside his own one. Every dwarf is supposed to tell what is the color of his hat, such that only finitely many dwarfs make a mistake. They can fix a strategy in advance, before getting their hats, but they cannot communicate after getting their hats. Find a winning strategy for dwarfs. *Remark:* Problems 21, 22 and 23 serve as a preparation for the Problem 24.

Problem 22. Show that there is a function inf-xor : $\{0,1\}^\omega \to \{0,1\}$, such that changing one bit of an argument always changes the result. (The solution uses the axiom of choice.)

Problem 23. Consider the following two player game, called Chomp. There is a rectangular chocolate in a shape of $n \times k$ grid. The right upper corner piece is rotten. Players move in an alternating manner, the first one moves first. Any player in his move picks square of the chocolate that is not yet eaten, and eats all pieces that are to the left and to the bottom from the picked piece. The player who eats the rotten piece loses. Determine who has a winning strategy.

Problem 24. Show a game that is not determined.

Problem 25. Consider the following *bisimilarity game* played on a finite game graph with vertices V equipped with a function $rank : V \to \mathbb{N}$. Two players, *Spoiler* and *Duplicator* start from a position $(u, v) \in V \times V$. The play proceeds in rounds. If at the beginning of a round $rank(u) \neq rank(v)$ or u and v belong to different players then Spoiler immediately wins. Otherwise Spoiler makes a move to (u', v) or (u, v') such that $u \to u'$ or $v \to v'$, respectively. Then Duplicator makes a move to (u', v') such that $v \to v'$ or $u \to u'$, respectively. Next round starts from (u', v'). If play continues infinitely long then Duplicator wins. Show that if Duplicator has a winning strategy from position (u, v) then the same player has a winning strategy in the parity game starting from u and in the parity game starting in v.

3 Parity games in quasipolynomial time

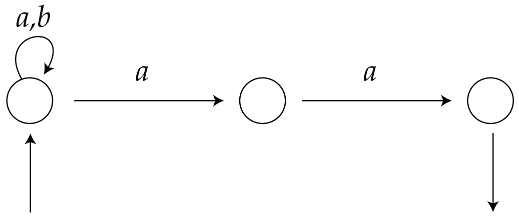
In this chapter, we show the following result.

Theorem 3.1. *Parity games with n positions and d ranks can be solved in time* $n^{\mathcal{O}(\log d)}$.

The time in the above theorem is a special case of *quasipolynomial time* mentioned in the title of the chapter. Whether or not parity games can be solved in time which is polynomial in both n and d is an important open problem. The presentation here is based on the original paper [16], with some new terminology (notably, the use of separation).
Define a *reachability* game to be a game where the objective of player 0 is to visit an edge from a designated subset. (We assume that the designated subset contains all edges pointing to dead ends of player 1, so that winning by reaching a dead end is subsumed by reaching designated edges.) Reachability games can be solved in time linear in the number of edges, as is shown in Exercise 11. Our proof strategy for Theorem 3.1 is to reduce parity games to reachability games of appropriate size.

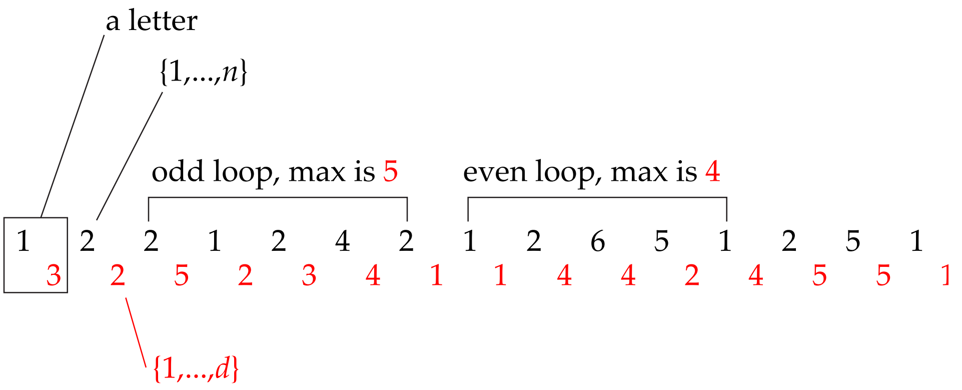
3.1 Reduction to reachability games

The syntax of a *reachability automaton* is exactly the same as the syntax of an NFA. The semantics, however, is different: the automaton inputs an infinite word, and accepts if a final state can be reached (in other words, there is a prefix which is accepted by the automaton when viewed as an NFA). For example, the following reachability automaton



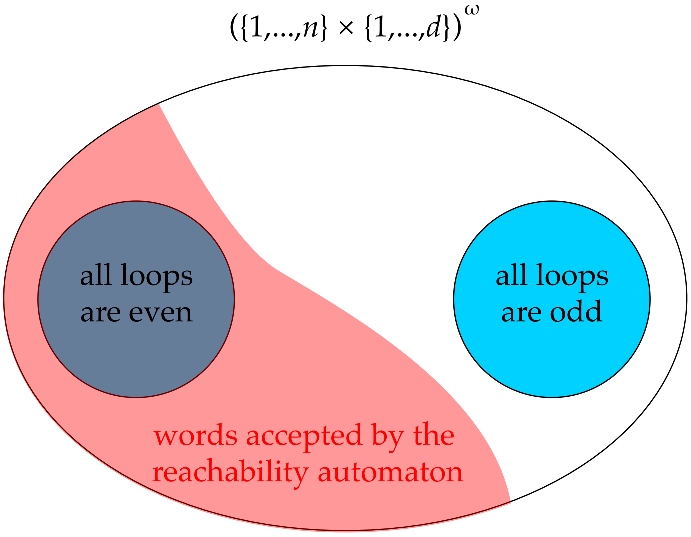
accepts all ω-words over alphabet $\{a, b\}$ which contain two consecutive a's. A reachability automaton is called deterministic if its transition relation is a function.

Consider an infinite word over an alphabet $\{1, \dots, n\} \times \{1, \dots, d\}$. We view this word as an infinite path in a game, where the positions are $\{1, \dots, n\}$ and each edge is labelled by a *rank* from $\{1, \dots, d\}$. Each letter describes a position and the rank of an outgoing edge. An infix of such a path is called an *even loop* if it begins and ends in the same vertex from $\{1, \dots, n\}$ and the maximal rank in the infix is even. Likewise we define odd loops. Here is a picture:



The following lemma shows that to quickly solve parity games, it suffices to find a small deterministic reachability automaton which separates the properties "all loops are even" and "all loops are odd".

Lemma 3.2. *Let $n, d \in \{1, 2, \ldots\}$. Assume that one can compute a deterministic reachability automaton $\mathcal{D}$ with input alphabet $\{1, \ldots, n\} \times \{1, \ldots, d\}$ that accepts every ω-word where all loops are even, and rejects every ω-word where all loops are odd, as in the following picture:*


$(\{1,...,n\} \times \{1,...,d\})^\omega$
all loops are even
all loops are odd
words accepted by the reachability automaton

Then a parity game $\mathcal{G}$ with n positions and d ranks can be solved in time

$$\mathcal{O}((\textit{number of edges in } \mathcal{G}) \times (\textit{number of states in } \mathcal{D})) + \textit{time to compute } \mathcal{D}$$

Proof. Let G be a parity game with vertices $\{1, \ldots, n\}$ and edges labelled by parity ranks $\{1, \ldots, d\}$. Let $\mathcal{D}$ be an automaton as in the assumption of the lemma. Consider a product game $G \times \mathcal{D}$, as defined on page 31, i.e. the positions are pairs (position of v, state of $\mathcal{A}$) and the structure of the game is inherited from G with only the states being updated according to the parity ranks on edges. Player 0 wins the product game $G \times \mathcal{D}$ if a dead end of player 1 is reached, or if the play is infinite and accepted by $\mathcal{D}$ (in the latter case, by the assumption that $\mathcal{D}$ is a reachability automaton, this is done by reaching an accepting state of $\mathcal{D}$ at some point during the play).

Claim 3.3. *If player $i \in \{0, 1\}$ wins G, then player i also wins $G \times \mathcal{D}$.*

Proof. By symmetry, take $i = 0$. Let σ_0 be a winning strategy for player i in the game G. By memoryless determinacy of parity games, we assume that σ_0 is memoryless. Let G_0 be the graph obtained from the graph underlying the game G by fixing the memoryless strategy σ_0, i.e. by removing every edge that originates in a position owned by player 0 and is not used by the strategy σ_0. Paths in the graph G_0 correspond to plays in the game G that are consistent with strategy σ_0. Because σ_0 was winning in the game G, all infinite paths in G satisfy the parity condition. In particular, every loop in G_0 that is accessible from the initial vertex has even maximum. This means that every infinite path in G_0 is accepted by the automaton $\mathcal{D}$. Therefore, the same strategy σ_0 also wins in the game $G \times \mathcal{D}$. ∎

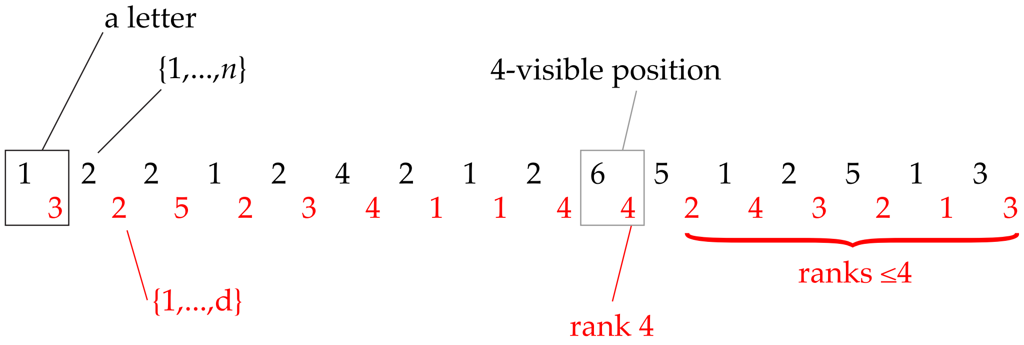
Because $\mathcal{D}$ is a reachability automaton, the product game $G \times \mathcal{D}$ can be solved in time proportional to the number of its edges, which is consistent with the bound in the lemma. ∎

3.2 *A small reachability automaton for loop parity*

By Lemma 3.2, to prove Theorem 3.1, it suffices to find a deterministic automaton which separates "all loops even" from "all loops odd", and which has a quasipolynomial state space (and time to compute the automaton). As a warm-up, we present a simpler construction which has $n^{d/2}$ states.

Fact 3.4. *Let $n, d \in \{1, 2, \ldots\}$. There is a deterministic reachability automaton with $n^{d/2}$ states which satisfies the properties in Lemma 3.2.*

Proof. Consider a finite word over the alphabet $\{1, \ldots, n\} \times \{1, \ldots, d\}$. For a rank $a \in \{1, \ldots, d\}$, a position in the word is called *a-visible* if its letter has rank exactly a, and all later positions have ranks $\leq a$, as in the following picture



After reading a word, for each even rank a, the automaton stores the number of a-visible positions up to threshold $n-1$, i.e. the state space is a function

$$\text{even numbers in } \{1,\dots,d\} \quad \to \quad \{0,1,\dots,n\}.$$

Whenever the threshold is exceeded, i.e. the number of a-visible positions exceeds n for some a, the automaton accepts. If this happens, then the pigeonhole principle says that the input word contains two a-visible positions with the same label in $\{1,\dots,n\}$, and therefore the infix connecting these positions forms an even loop with maximum exactly a. Therefore, if the automaton accepts, then there is an even loop. Contrapositively: if there are only odd loops, then the automaton rejects. On the other hand, if the input word satisfies the parity condition, i.e. the maximal rank seen infinitely often is an even number a, then at some point there will be at least n positions that are a-visible. Therefore if the input satisfies the parity condition (in particular, if the input has all loops even), then the automaton must accept. ■

Note that in the above construction, the automaton satisfies a stronger property than required by Lemma 3.2, namely it accepts all words satisfying the parity condition (instead of only those where all loops are even).

Lemma 3.5. *Let $n, d \in \{1,2,\dots\}$. There is deterministic reachability automaton with $n^{\mathcal{O}(\log d)}$ states which satisfies the assumptions of Lemma 3.2.*

The rest of Section 3.2 is devoted to proving the above lemma. Like in the construction with $n^{d/2}$ states, the automaton will reject all words which violate the parity condition, and not just those where all loops are odd. This stronger property, however, is not used in the proof of Theorem 3.1.

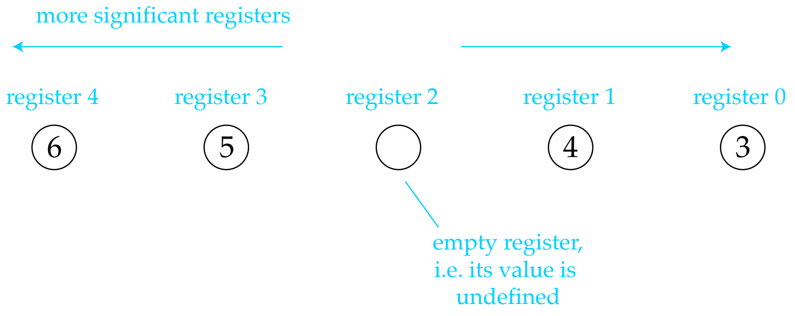
We begin with a *nondeterministic* reachability automaton $\mathcal{A}$ which satisfies the properties in the lemma in the following sense: if all loops are even, then at least one run reaches an accepting state, and if all loops are odd, then all runs avoid accepting states.

Choose the smallest k so that $n < 2^k$. The nondeterministic automaton uses k registers with names $\{0, \ldots, k-1\}$. Each register stores a number from $\{1, \ldots, d\}$, or it is undefined. A state of the automaton is a valuation of these registers or an accepting sink state, i.e. the number of states is at most $(1+d)^k + 1$. By choice of k, we have

$$\begin{aligned} (d+1)^k &\leq \\ (d+1)^{\log(n+1)} &= \\ 2^{\log(n+1) \cdot \log(d+1)} &= \\ (n+1)^{\log(d+1)} & \end{aligned}$$

and therefore the number of states in $\mathcal{A}$ is at most $n^{\mathcal{O}(\log d)}$. Our final automaton $\mathcal{D}$ will be obtained by keeping the same states as $\mathcal{A}$ and removing transitions so as to make the automaton deterministic, and hence the size of $\mathcal{D}$ will be as required to make Theorem 3.1 true.

Here is a picture of a state of the automaton $\mathcal{A}$:

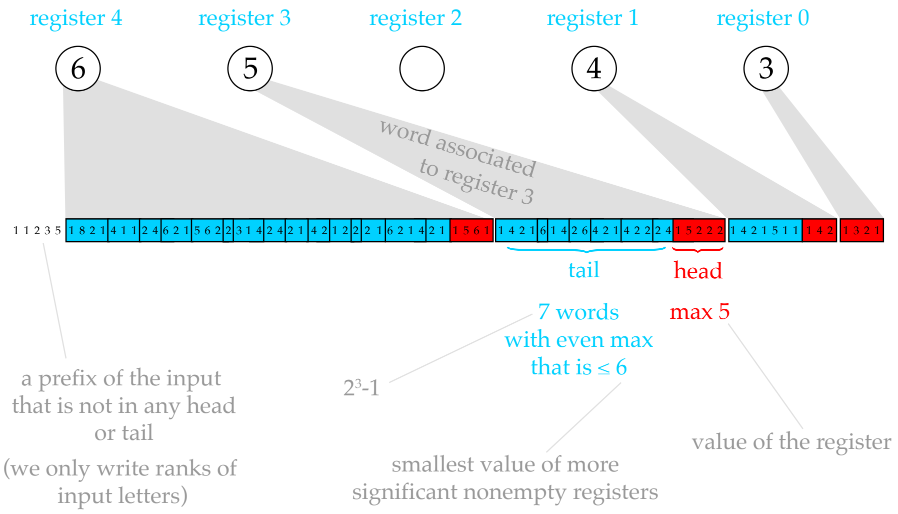

more significant registers
register 4
register 3
register 2
register 1
register 0
6
5
4
3
empty register, i.e. its value is undefined

We design the transition relation to respect following invariant.

(*) Suppose that the automaton has read a finite word, and has not accepted yet. Then the register valuation is nondecreasing on nonempty registers. Furthermore, one can associate to each register r a word w_r so that:

1. if r is empty then w_r is empty; and
2. the word $w_{k-1}w_{k-2}\cdots w_1 w_0$ is a suffix of the input read so far; and
3. if a register r is nonempty and stores $i \in \{1,\dots,d\}$, then:
   (a) all words associated to nonempty registers $< r$ use ranks $\leq i$;
   (b) the word w_r associated to r is a concatenation of two words:
   - TAIL: $2^r - 1$ words with even maximal rank;
   - HEAD: a word with maximal rank exactly i.

Here is a picture of the invariant. In the picture, we only draw the ranks of the input letters, and not their labels in $\{1,\dots,n\}$. One reason is that the automaton completely ignores the labels in its transition relation.



In the initial state, all registers are empty; this state clearly satisfies the invariant. Before giving the state update function, we explain two properties of the invariant.

Lemma 3.6. *Assume that the invariant is satisfied, all registers are nonempty and store even ranks, and the input letter has even rank. Then the input contains an even loop.*

Proof. If register r stores an even rank, then the associated word w_r is a concatenation of 2^r words with even maximal rank: one for the head, and $2^r - 1$ for the tail. Therefore, if all registers are nonempty and store even ranks, and a letter of even rank appears in the input, then a suffix of the input – including the new input letter – can be factorised as a concatenation of

$$\underbrace{2^{k-1} + 2^{k-2} + \cdots 2^0}_{\text{the registers}} + \underbrace{1}_{\text{the input letter}} = 2^k > n$$

words with even maximal rank. For each of these words, choose the position which achieves the maximal rank. The pigeonhole principle says that two positions achieving the maximal rank must have the same label. The infix connecting these two positions is an even loop. ■

The above lemma justifies the following acceptance criterion of the automaton: if all of its registers are nonempty and store even ranks, and it reads an even rank, then it accepts.

Lemma 3.7. *Emptying any subset of the registers preserves the invariant.*

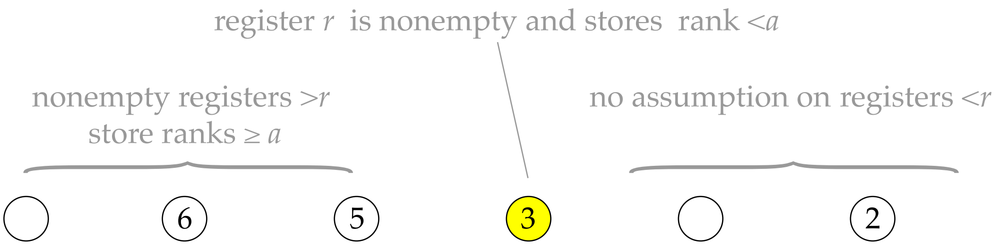
Proof. It is enough to show that emptying any single register r preserves the invariant. If r is the most significant nonempty register, then the word associated to r is put into the prefix of the input that is not assigned to any register. Otherwise, the word associated to r is appended to the head of the closest more significant register. ■

Transitions of the automaton. We now describe the transitions of the automaton and justify that they preserve the invariant. Suppose that the automaton reads a letter with rank $a \in \{1, \ldots, d\}$. Then the automaton allows three types of transitions A, B and C, as described below.

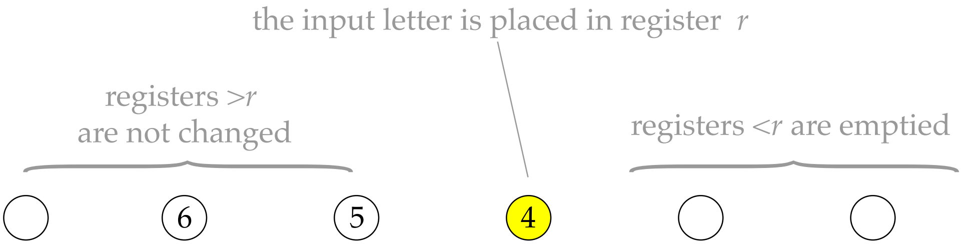
A. Assume that in the current state, all registers store values $\geq a$, which includes the special case when all registers are empty. Under this assumption, the automaton is allowed to do nothing, i.e. not change the state when reading a.

Why the invariant is preserved. If all registers are empty, then the new input letter a becomes part of the input that is not associated to any register. Otherwise, a is appended to the head of the least significant nonempty register.

B. Let r be any register which satisfies conditions written in grey below:

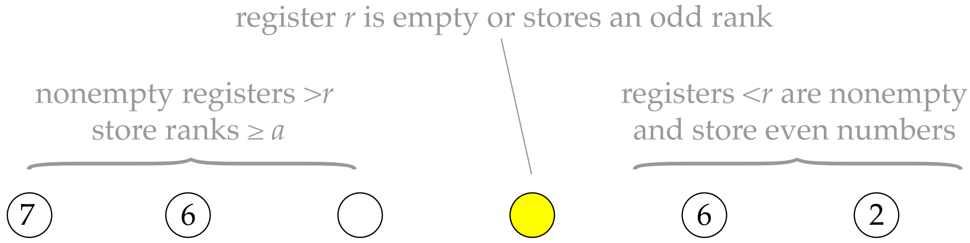

register r is nonempty and stores rank $<a$
nonempty registers $>r$ store ranks $\geq a$
no assumption on registers $<r$
6
5
3
2

Then the automaton can do the following update (the picture uses $a = 4$):


the input letter is placed in register r
registers $>r$ are not changed
registers $<r$ are emptied
6
5
4

Why the invariant is preserved. We view this transition as a two-step process. First, all registers $< r$ are made empty, which preserves the invariant by Lemma 3.7. Next, the input letter a is appended to the head of the register r (which is now the least significant nonempty register), and therefore becomes the new maximum in this head.

C. Let r be any register which satisfies the conditions written in grey below:

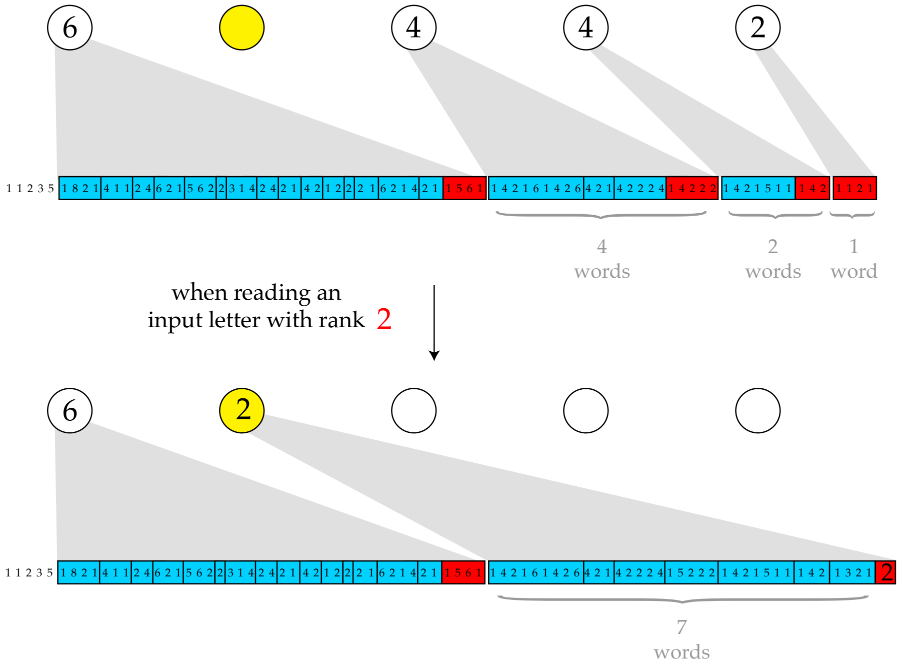

register r is empty or stores an odd rank
nonempty registers $>r$ store ranks $\geq a$
registers $<r$ are nonempty and store even numbers
7
6
6
2

Under these conditions, and assuming that a is even, the automaton can do the same update as in transitions of type B., i.e. it put a into register r and empty all registers $< r$. Apart from the assumption that a is even, there is no assumption that a is bigger than the contents of registers $< r$, e.g. in the above picture a could be 2 or 4.

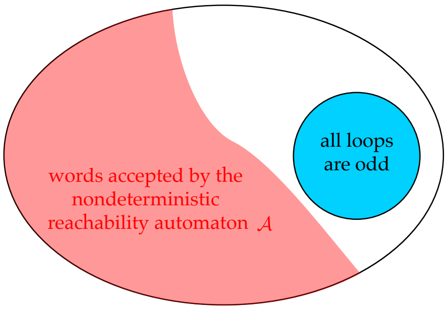
Why the invariant is preserved. We also view this transition as a two-step process. First, we empty register r (but not the smaller ones), which preserves the invariant by Lemma 3.7. Next, all of the words associated to registers $< r$ are concatenated and put into the tail of register r. As explained in the proof of Lemma 3.6, after the update the tail of register r consists of

$$2^{r-1} + 2^{r-2} + \cdots 2^0 = 2^r - 1$$

words with even maximum, as required by the invariant. Finally, the head of register r is set to the one letter word consisting of the new input letter a. Here is a picture:



Since every transition of $\mathcal{A}$ preserves the invariant, we can use Lemma 3.6 to conclude that if the invariant is preserved and the automaton accepts (which happens when all registers store even ranks and a new even rank is read), then the input contains at least one even loop. This gives the following inclusion:


all loops are odd
words accepted by the nondeterministic reachability automaton $\mathcal{A}$

Define $\mathcal{D}$ to be the deterministic reachability automaton which is obtained from $\mathcal{A}$ as follows: if there are several applicable transitions, then choose any transition that maximises the most significant register that is modified. The automaton $\mathcal{D}$ has fewer accepting runs than $\mathcal{A}$, and therefore it still rejects all words that have only odd loops. Therefore, the proof of Lemma 3.5 is completed by the following lemma.

Lemma 3.8. *If the input has only even loops, then $\mathcal{D}$ accepts.*

Proof. For $i \in \{1, 2, \ldots\}$, define $\mathcal{D}_i$ to be a variant of the automaton $\mathcal{D}$ where the number of registers is i instead of k. In particular, $\mathcal{D} = \mathcal{D}_k$. By induction on i, we prove the following generalisation (*) of the lemma. The generalisation is twofold: we allow any number of registers, and we weaken the assumption from "only even loops" to "satisfies the parity condition".

(*) Suppose that $\mathcal{D}_i$ is initialised in an arbitrary state (not necessarily the initial state with all registers empty). If the input satisfies the parity condition, then $\mathcal{D}_i$ accepts, i.e. it reaches a configuration where all registers store even ranks and the input letter has even rank.

Suppose that we have already proved (*) for $i - 1$, or $i = 1$ and there is nothing to prove. We now prove (*) for i. Consider a run of $\mathcal{D}_i$ on an input which satisfies the parity condition, i.e. the maximal rank that appears infinitely often is some even $a \in \{1, \ldots, d\}$. By the induction assumption, the most significant register i must eventually become nonempty, because transitions that do not affect the most significant register are transitions of the automaton $\mathcal{D}_{i-1}$. Once the most significant register becomes nonempty, then it stays nonempty. Wait until the most significant rank a is seen again; either the automaton accepts before this time, or otherwise it puts a into the most significant register. Once the most significant register stores a, and the input contains only values with rank $\leq a$, then the most significant register will keep on containing a. Again by induction assumption, the automaton will eventually fill all registers $< i$ with even ranks and read an even letter, thus accepting. ∎

Problem 26. Consider the following variant of the automaton from Lemma 3.5. Only odd numbers are kept in the registers, and the update function is the same as in Lemma 3.5 when reading an odd number. When reading an even number a, the automaton erases all registers, which store values $< a$. Show that this automaton does not satisfy the properties required in Lemma 3.5.

Problem 27. Show that there is no safety automaton which:

- accepts all ultimately periodic words that satisfy the parity condition;
- rejects all ultimately periodic words that violate the parity condition.

Problem 28. Show that there is no safety automaton with $< \lfloor n/2 \rfloor$ states which satisfies the properties required in Lemma 3.5.

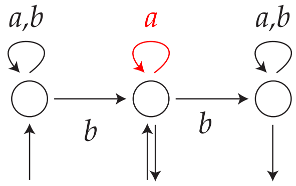
Problem 29. A probabilistic reachability automaton is defined like a finite automaton, except that each transition is assigned a probability – a number in the unit interval – such that for every state, the sum of probabilities for outgoing transitions is 1. The value assigned by such an automaton to an ω-word is the probability that an accepting state is seen at least once. Show that there is a probabilistic reachability automaton over the alphabet $\{1, \ldots, n\}^\omega$, with state space polynomial in n, that:

- assigns value 1 to words that have only even loops;
- assigns value 0 to words that have only odd loops.

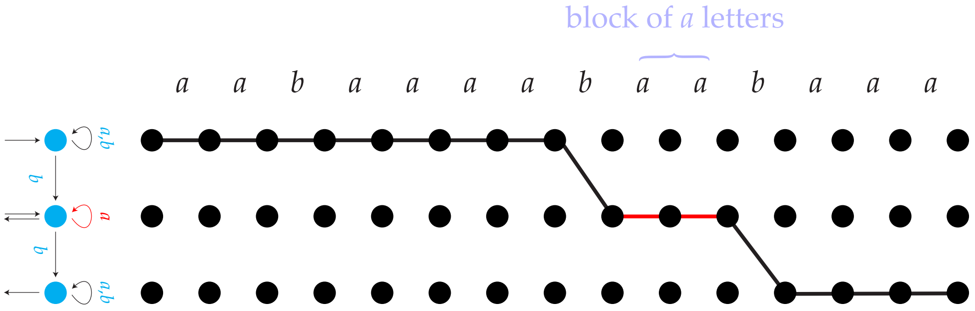
4
Distance automata

The syntax of a distance automaton is the same as for a nondeterministic finite automaton, except that it has a distinguished subset of transitions, called the *costly transitions*. The *cost* of a run is defined to be the number of costly transitions that it uses.

Example 5. Here is a cost automaton, with the costly transitions (one transition, in this particular example) depicted in red.



The nondeterminism of the automaton consists of: choosing the initial state (first or second), and in case the first state was chosen as initial, then choosing the moment when the second horizontal transition is used. This nondeterminism corresponds to selecting a block of a letters, and the cost of a run is the length of such a block, as in the following picture:



□

In this chapter, we prove the following theorem, originally proved by Hashiguchi in [32]. The theorem was part of Hashiguchi's solution [33] to the star height problem, i.e. the problem of determining what is the least number of nested Kleene stars that is needed to define a given regular language.

Theorem 4.1. *The following problem is decidable:*

- **Input.** *A distance automaton.*
- **Question.** *Is the automaton bounded in the following sense: there is some $m \in \mathbb{N}$ such that every input word admits an accepting run of cost $< m$.*

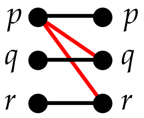
The problem in the above theorem was called *limitedness* in [32]. The algorithm we use, based on [10], uses the Büchi-Landweber Theorem [15] discussed in Chapter 2. The algorithm leads to an ExpTime upper bound on the limitedness problem; the optimal complexity is PSpace, which follows as a special case of [35, Theorem 2.2].

The limitedness game. Fix a distance automaton. For a number $m \in \{1, 2, \ldots, \omega\}$, consider the following game, call it the *limitedness game with bound m*. The game is played in infinitely many rounds $1, 2, 3, \ldots$, by two players called Input and Automaton. In each round:

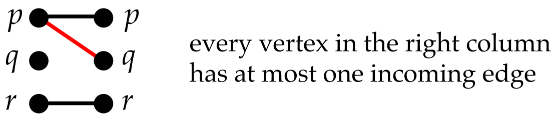
- player Input chooses a letter of the input alphabet;

- player Automaton responds with a set of transitions over this letter.

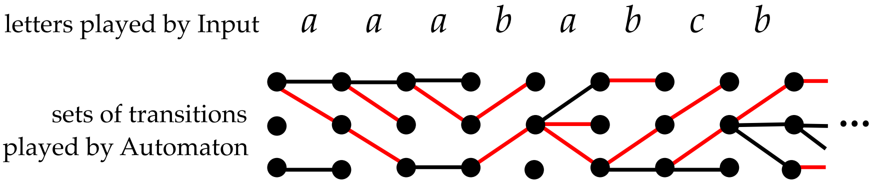
A move of player Automaton in a given round, which is a set of transitions, can be visualised as a bipartite graph, which says how the letter can take a state to a new state, with costly transitions being red and non-costly transitions being black, like below:



For the definition of the game, it is important that player Automaton does not need to choose all possible transitions over the letter played by player Input, only a subset. Actually, as we will later see, in order to win, player Automaton need only use tree-shaped sets like this:


p
q
r
p
q
r
every vertex in the right column
has at most one incoming edge

After all rounds have been played, the situation looks like this:



The winning condition for player Automaton is the following:

1. In every column, at least one accepting state must be reachable from some initial state in the first column; and

2. Every path contains $< m$ costly edges. In case of $m = \omega$, this means that every path contains finitely many costly edges.

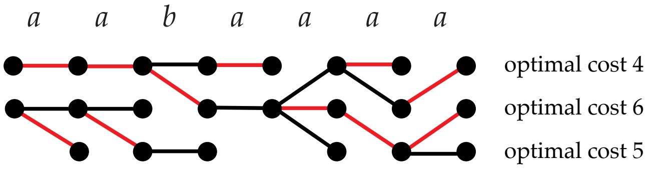
If either of the conditions above is violated, then player Input wins. The following lemma implies the decidability of the limitedness problem.

Lemma 4.2. *For a distance automaton, the following conditions are equivalent, and furthermore one can decide if they hold:*

1. *the automaton is limited;*
2. *there is some $m \in \{1, 2, \dots\}$ such that player Automaton wins the limitedness game with bound m;*
3. *player Automaton wins the limitedness game with bound $m = \omega$*

Proof. The implications from 2 to 1 and from 2 to 3 are immediate. For the other implications and the decidability part, the key is the observation that for every choice of $m \in \{1, 2, \dots, \omega\}$, the limitedness game is a special case of a game with a finite arena and an ω-regular condition. In particular, one can apply the Büchi-Landweber theorem, yielding that a) the winner can be decided; b) the winner needs finite memory. Condition a) shows that item 3 in the lemma is decidable, while condition b) will be used in the implication from 3 to 2.

Implication from 1 to 2. We want to prove that if the automaton is limited, then player Automaton has a winning strategy for some finite m, which will turn out to be the same m as in the definition of limitedness. Define a run ρ of the distance automaton over an input word w to be *optimal* if it has minimal cost among runs that have the same input word, same source state and same target state. The strategy of player Automaton is as follows. Suppose that player Input has played a sequence of letters. Then the sets of transitions chosen by Automaton are so that the transitions form a forest, consisting only of optimal runs, where all reachable configurations (i.e. reachable by some run from an initial state) are covered, as in the following picture:



When player Input gives a new letter, player Automaton responds with a set of transitions which connect the new configurations to the previous ones in a cost-minimising way.

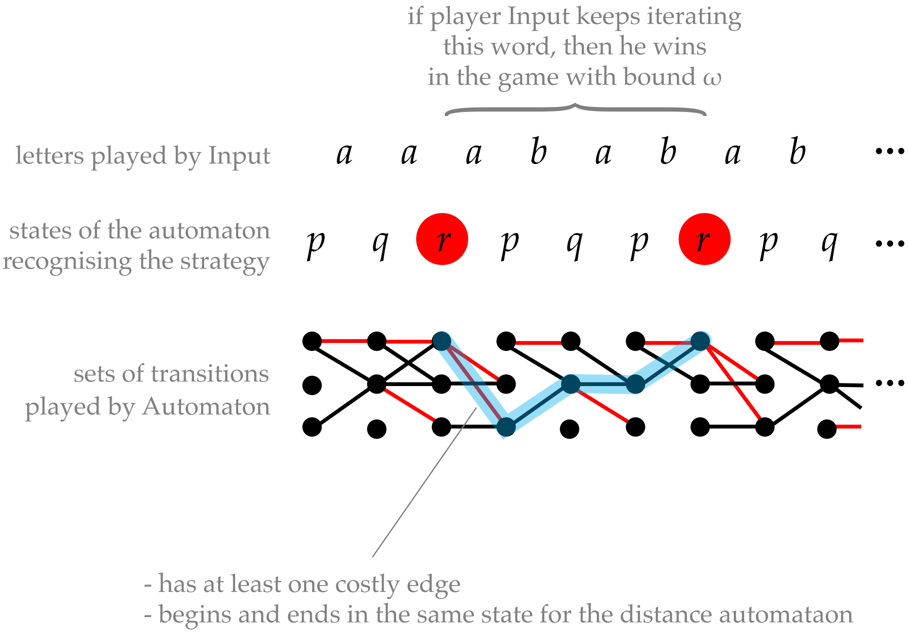
Implication from 3 to 2. Suppose that player Automaton wins the limitedness game with bound ω. We will prove that player Automaton can also win the limitedness game with a finite bound.

By the Büchi-Landweber theorem, if player Automaton can win the game with bound ω, then he can also win the game with a finite memory strategy. We will show that this finite memory strategy is actually winning for a finite bound.

Suppose that the input alphabet of the original distance automaton is Σ. A finite memory strategy of player Automaton in the limitedness game is a function

$$\sigma : \Sigma^* \to \text{sets of transitions}$$

which is recognised by a finite automaton, i.e. there is a deterministic finite automaton such that $\sigma(w)$ depends only on the state of the automaton after reading w. We claim that this same winning strategy produces runs where the cost is at most (number of states in the distance automaton) times (number of states in the automaton recognising the strategy), thus proving the implication from 3 to 2 in the lemma. To prove the claim, suppose that the strategy σ loses in the game with the above described finite bound. Using a pumping argument we find a loop that can be exploited by player Input to force player Automaton into a path that has infinitely many costly edges, contradicting the assumption that σ wins in the game with bound ω, as in the following picture:


if player Input keeps iterating
this word, then he wins
in the game with bound ω
letters played by Input
a a a b a b a b ...
states of the automaton
recognising the strategy
p q r p q p r p q ...
sets of transitions
played by Automaton
...
- has at least one costly edge
- begins and ends in the same state for the distance automataon

■

Problem 30. Show that limitedness remains decidable when distance automata are equipped with a reset operation. (The cost of a run is the biggest number of costly transitions between some two consecutive resets.)

Problem 31. Let $\mathcal{A}$ be a distance automaton with input alphabet Σ. The problem of *limitedness of $\mathcal{A}$ on regular language $L \subseteq \Sigma^*$* asks whether there exists $n \in \mathbb{N}$ such that for every word $w \in L$ the cost of w with respect to $\mathcal{A}$ is not bigger than n. Show that this problem is decidable.

Problem 32. We say that a regular language L has the *finite power property* if there exists $n \in \mathbb{N}$ such that $L^* = L^0 \cup L^1 \cup \ldots \cup L^n$. Show that one can decide if a regular language has the finite power property. is decidable.

Problem 33. We say that languages $K \subseteq \Sigma^*$ and $L \subseteq \Sigma^*$ are *separated by* language $S \subseteq \Sigma^*$ if $K \subseteq S$ and $L \cap S = \emptyset$. For $u, v \in \Sigma^*$ we say that $u = a_1 \cdots a_k$ is a *subsequence* of v, denoted $u \preceq v$, if $v \in \Sigma^* a_1 \Sigma^* \ldots \Sigma^* a_k \Sigma^*$. A language L is called *upward closed* if for every $u \in L$ and $u \preceq v$ also $v \in L$. Show that deciding whether two given regular languages K and L are separated by some upward closed language is decidable.

Problem 34. Let $\mathcal{F}$ be the class of finite unions of languages of the form $\Sigma^* w_1 \Sigma^* \ldots \Sigma^* w_k \Sigma^*$, where all w_i are words from Σ^*. Show that for given regular languages K and L it is decidable whether they are separated by a set from $\mathcal{F}$.
Remark: Note that $\mathcal{F}$ contains all upward closed languages defined in the Problem 33. To see this recall that Higman's Lemma implies that there is no infinite antichain in the $\preceq$ order. Therefore every upward closed language has finitely many minimal elements. Thus every upward closed language is a finite union of languages of the form $\Sigma^* a_1 \Sigma^* \ldots \Sigma^* a_k \Sigma^*$, where all $a_i \in \Sigma$.

Problem 35. Show that it is decidable if a regular language is of star height one, i.e. it can be defined by a regular expression that uses Kleene star, maybe multiple times, but does not nest it.

5
Monadic second-order logic

In this section we discuss the connection between monadic second-order logic (MSO) and automata, specifically tree automata. The presentation here is largely based on [56]. One of the crowning achievements of logic in computer science is Rabin's Theorem [45], which says that MSO on infinite trees is decidable, and has the same expressive power as automata. We prove Rabin's Theorem in this chapter.

Actually, we already have the tools to prove Rabin's Theorem[1], namely McNaughton's Theorem on determinisation of ω-automata from Chapter 1, and memoryless determinacy of parity games from Chapter 2. It remains only to deploy the appropriate definitions and put the tools to work.

5.1 Monadic second-order logic

Monadic second-order logic (MSO) is a logic with two types of quantifiers: quantifiers with lowercase variables $\exists x$ quantify over elements, and quantifiers with uppercase variables $\exists X$ quantify over sets of elements. The term "monadic" means that one cannot quantify over sets of pairs, or over sets

[1] Büchi says this in [14, page 2]: "Given the statement of this lemma [the complementation lemma for automata on infinite trees], and given McNaughton's handling of sup-conditions by order vectors, and given time, everybody can prove Rabin's theorem."

triples, etc. The syntax and semantics of the MSO are explained in the following example.

Example 6. Suppose that we view an directed graph as relational structure (i.e. a model as in logic), where the universe is the vertices and there is one binary relation $E(x,y)$ for the edges; this relation is not necessarily symmetric because the graph is directed. The formula

$$\forall x \forall y\ E(x,y)$$

says that the graph is a directed clique. The formula only quantifies over vertices, i.e. it uses only first-order quantification. Now consider a formula which uses also set quantification, which says that the input graph is not strongly connected:

$$\underbrace{\exists X}_{\text{exists a set}} \quad \underbrace{(\forall x \forall y\ x \in X \wedge E(x,y) \Rightarrow y \in X)}_{X \text{ is closed under outgoing edges}} \wedge \underbrace{(\exists x\ x \in X) \wedge (\exists x\ x \notin X)}_{X \text{ is neither empty nor full}}$$

The above formula illustrates all syntactic constructs in MSO: one can quantify over elements, over sets of elements, one can test membership of elements in sets, and one can use the relations available in the input model (in the case of directed graphs, only one binary relation).

Here is another example for graphs. The following MSO formula says that the input graph is three colourable (in the formula, the direction of the edges plays no role):

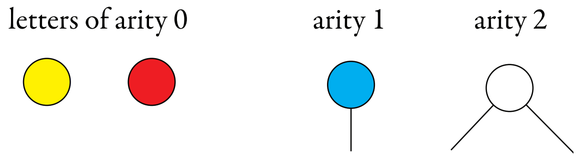
$$\exists X_1 \exists X_2 \exists X_3 \quad \underbrace{\forall x \bigvee_i x \in X_i}_{\text{every vertex is coloured}} \quad \wedge \quad \underbrace{\forall x \forall y\ E(x,y) \Rightarrow \bigwedge_i x \notin X_i \vee y \notin X_i}_{\text{no edge has both endpoints with the same colour}}$$

□

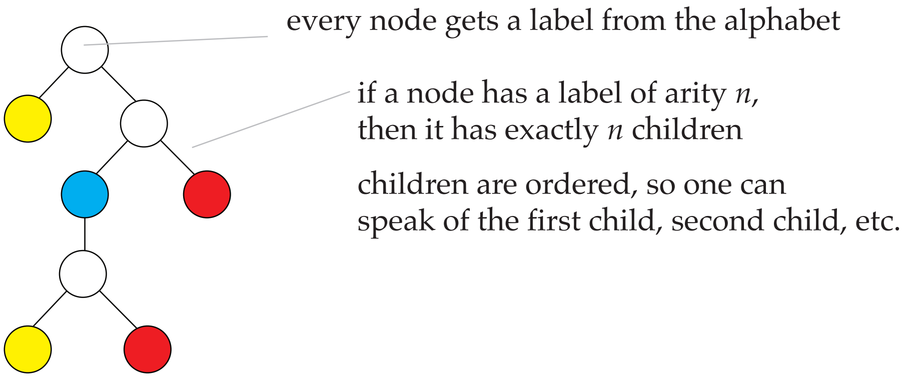
We say that a property of relational structures over some vocabulary (e.g. graphs as in the above example) is MSO *definable* if there is a formula of MSO which is true exactly in those structures which have the property. In this chapter, we use MSO to describe properties of trees (finite and infinite). In the next chapter, we talk about finite graphs.

5.2 *Finite trees*

Define a *ranked alphabet* to be a finite set Σ where every element $a \in \Sigma$ has an associated arity in $\{0, 1, \ldots\}$. Here is a picture of a ranked alphabet:



A tree over a ranked alphabet Σ is defined as in the following picture:


every node gets a label from the alphabet
if a node has a label of arity n,
then it has exactly n children
children are ordered, so one can
speak of the first child, second child, etc.

In this section, Section 5.2, we will be interested only in finite trees. Trees as defined above are sometimes called *ranked and ordered*. One can consider other variants, where the label does not determine the number of children (unranked) or where the siblings are not ordered (unordered). The goal of this section is to show that, over finite trees, automata have the same expressive power as MSO.

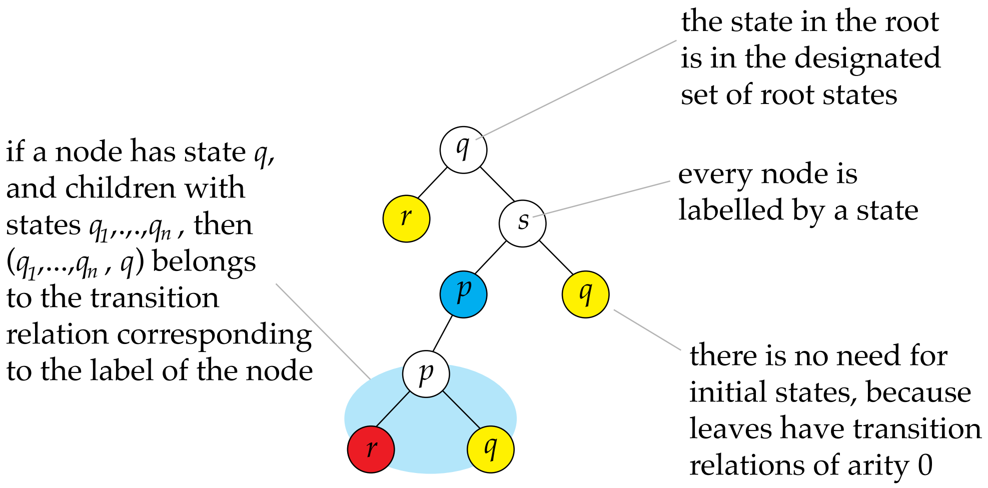
Tree automata. We begin by defining automata for finite trees.

Definition 5.1. *A* nondeterministic tree automaton *consists of:*

- *an input alphabet Σ, which is a ranked alphabet;*
- *a finite set of states Q with a distinguished subset of root states $R \subseteq Q$*

- *for every letter $a \in \Sigma$ of rank n, a transition relation $\delta_a \subseteq Q^n \times Q$.*

A tree automaton is called bottom-up deterministic *if every transition relation is a function $Q^n \to Q$. An automaton is called* top-down deterministic *if it has one root state and the transition relation is a partial function $Q^n \leftarrow Q$. A tree is accepted by the automaton if there exists an accepting run, as explained in the following picture:*

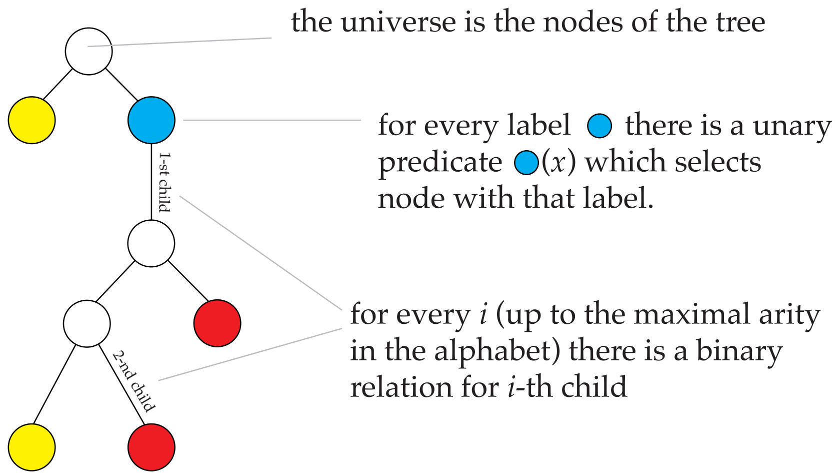

the state in the root is in the designated set of root states
if a node has state q, and children with states q_1,..,q_n , then (q_1,...,q_n , q) belongs to the transition relation corresponding to the label of the node
every node is labelled by a state
there is no need for initial states, because leaves have transition relations of arity 0
q
r
s
p
q
p
r
q

Lemma 5.2. *Languages recognised by nondeterministic tree automata are closed under union, intersection and complementation.*

Proof. For union, take the disjoint union of two nondeterministic tree automata. Intersection can be done using a cartesian product, or by using union and complementation. For complementation, we use determinisation: the same proof as for automata on words – the subset construction – shows that for every nondeterministic tree automata there is an equivalent one that is bottom-up deterministic (top-down deterministic automata are strictly weaker, see Exercise **??**.). Since bottom-up deterministic automata can be complemented by complementing the root states, we get the lemma. ∎

MSO **on finite trees.** We now define how MSO can be used to define a tree language, and show that tree languages defined this way are exactly those that are recognised by tree automata.

A tree (finite or infinite) over an alphabet Σ is viewed as a relational structure in the following way:



the universe is the nodes of the tree

for every label ● there is a unary predicate ●(x) which selects node with that label.

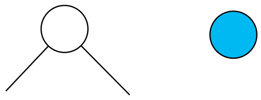
for every i (up to the maximal arity in the alphabet) there is a binary relation for i-th child

We say that an MSO formula is true in a tree if it is true in the relational structure described above. This only makes sense for formulas that have no free variables (sentences), and which use the vocabulary (relation names) described above, i.e. unary relations for labels and binary relations for child numbers. We say that a set of finite trees L over a ranked alphabet Σ is MSO definable if there is an MSO formula φ such that

$$\varphi \text{ is true in } t \quad \text{iff} \quad t \in L \qquad \text{for every finite tree } t \text{ over } \Sigma$$

The formula does not need to check if its input is a finite tree. However, the set of finite trees is MSO definable, as a subset of all relational structures over the appropriate set of relation names, and therefore the definition of MSO definable languages of finite trees would not be affected by requiring the formula to check that inputs are finite trees.

Example 7. Suppose that the ranked alphabet is



The set of trees with an odd number of nodes is MSO definable, namely the formula is "true". This is because all trees over the above ranked alphabet have an odd number of nodes. More effort is required for "odd number of leaves". Here the formula says that there exists a set X of nodes, which contains the root, and such that every node belongs to X if and only if it has an even number of children in X. □

The following theorem shows that for finite trees, tree automata have the same expressive power as monadic second-order logic. The connection of between automata and MSO was originally discovered simultaneously by three authors: Büchi [13], Elgot [26] and Trakthenbrot [57], in their quest to answer a question by Tarski: "is the MSO theory of the natural numbers with successor decidable"? We present below the version of the result for finite trees, which has essentially the same proof as for finite words (a word can be viewed as a tree over a ranked alphabet where all letters have arity zero or one), and was first observed in [55].

Theorem 5.3. *The following conditions are equivalent for every set of finite trees over a finite ranked alphabet:*

1. *definable in MSO;*

2. *recognised by a nondeterministic (equivalently, bottom-up deterministic) tree automaton.*

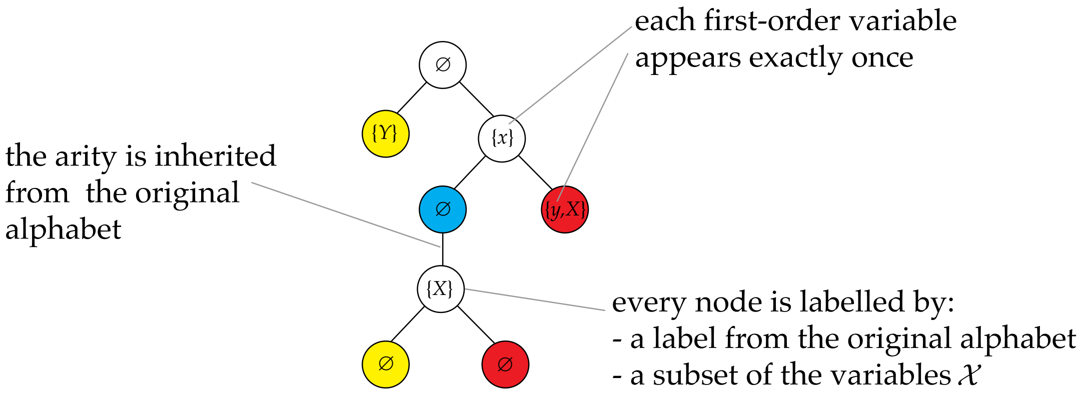
Proof.

$1 \Leftarrow 2$ Let $\mathcal{A}$ be a nondeterministic tree automaton. We show that MSO can formalise the statement "there exists an accepting run of $\mathcal{A}$". Without loss of generality, assume that the states of $\mathcal{A}$ are numbers $\{1, \ldots, n\}$. Here is the sentence that defines the language of $\mathcal{A}$:

$$\overset{\substack{\text{there exists a}\\ \text{labelling of}\\ \text{nodes with states}}}{\exists X_1 \cdots \exists X_n} \quad \wedge \left\{ \begin{array}{l} \text{every node has exactly one state} \\ \forall x \displaystyle\bigvee_{q\in\{1,\ldots,n\}} x \in X_q \wedge \bigwedge_{p\neq q} x \notin X_p \\[2ex] \text{the root has a root state} \\ \forall x \ \mathrm{root}(x) \Rightarrow \displaystyle\bigvee_{i\in R} x \in X_i \\[2ex] \text{for every node, a transition of the automaton is used} \\ \displaystyle\bigwedge_{a\in\Sigma} \forall x \ a(x) \Rightarrow \bigvee_{(q_1,\ldots,q_k,q)\in\delta_a} \left(x \in X_q \wedge \bigwedge_{i\in\{1,\ldots,k\}} \mathrm{child}_i(x) \in X_{q_i} \right) \end{array} \right.$$

Formally speaking, root(x) is a shortcut for a formula which says that x is not a child of any node, and child$_i(x) \in X_{q_i}$ is a shortcut for a formula which says that there exists a node that is the i-th child of x (because we have children as relations and not functions) and belongs to q_i.

1 ⇒ 2 By induction on formula size, we show that every MSO formula can be converted into an automaton. The main issue is that when we go to subformulas, free variables appear, and we need to say how an automaton deals with free variables. Consider a formula φ of MSO whose set of free variables is $\mathcal{X}$ (some of these variables are first-order, some are second-order). To encode a tree together with a valuation of free variables $\mathcal{X}$, we use a tree over an extended alphabet like this:


each first-order variable
appears exactly once
∅
{Y}
{x}
the arity is inherited
from the original
alphabet
∅
{y,X}
{X}
every node is labelled by:
- a label from the original alphabet
- a subset of the variables $\mathcal{X}$
∅
∅

A tree as above is said to satisfy φ if φ is true under the valuation which maps each first-order variable to the unique node that has it in the label, and maps each second-order label to the set of nodes that have it in their label. Define the *language* of φ to be the trees (over the extended alphabet with sets of variables) that satisfy φ. By induction on the size of an MSO formula, we show that its language, as defined above, is recognised by a tree automaton. For Boolean operations we use Lemma 5.2, for existential quantification we use nondeterminism.

■

[?]

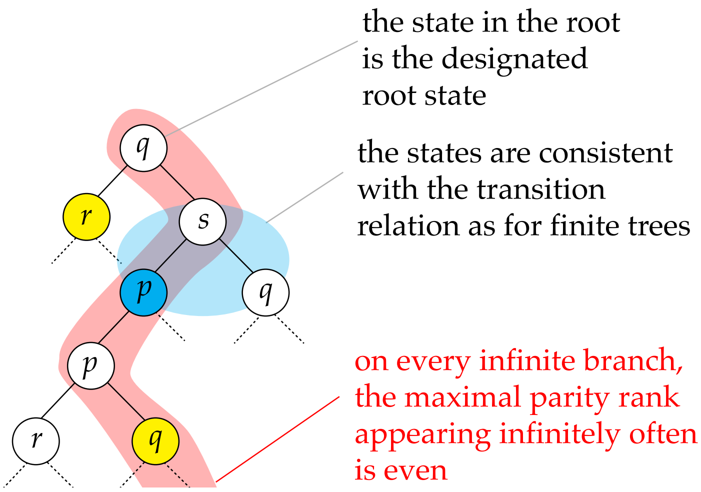
5.3 *Infinite trees*

We now move to infinite trees and Rabin's Theorem. For simplicity of notation, we use ranked alphabets where all letters have rank 2. For such alphabets, the set of nodes is always the same, and can be identified with $\{\text{left child}, \text{right child}\}^*$. For arbitrary alphabets, infinite trees can have various shapes, e.g. an infinite tree is allowed to have subtrees that are finite. To recognise properties of infinite trees, we use parity automata.

Definition 5.4. *The syntax of a nondeterministic parity tree automaton consists of*

- *an input alphabet Σ, which is a finite ranked set where all letters have rank 2;*
- *a finite set of states Q with a distinguished root state;*
- *a parity ranking function $Q \to \mathbb{N}$;*
- *for every letter $a \in \Sigma$, a set of transitions $\delta_a \subseteq Q^2 \times Q$.*

The automaton accepts an infinite tree over Σ if there exists an accepting run as explained in the following picture:



the state in the root is the designated root state

the states are consistent with the transition relation as for finite trees

on every infinite branch, the maximal parity rank appearing infinitely often is even

We now state Rabin's Theorem. Rabin's original proof did not use the parity acceptance condition, but what is now called the *Rabin condition*, see [56].

Theorem 5.5 (Rabin's Theorem)**.** *The following conditions are equivalent for every set of (necessarily) infinite trees over a finite ranked alphabet where all letters have arity 2:*

1. *definable in* MSO*;*
2. *recognised by a nondeterministic parity tree automaton.*

The proof has the same structure as in the case of finite trees. The only difference is that for infinite trees, closure under complementation, as stated in the following lemma, is far from obvious.

Lemma 5.6 (Complementation Lemma)**.** *Languages recognised by nondeterministic parity automata are closed under complement.*

The difficulty in the Complementation Lemma is that we use only nondeterministic automata; in fact no deterministic model for infinite trees is known that would be equivalent to MSO. Rabin's Theorem will follow immediately once the Complementation Lemma is proved, so the rest of this chapter is devoted to proving the Complementation Lemma.
A corollary of the statement of Rabin's Theorem as in Theorem 5.5, and of decidability of emptiness for nondeterministic parity tree automata (see Exercise **??**) is that the following logical structure has decidable MSO theory: the universe is the nodes of the complete binary tree, and there are two binary relations for left child and right child. This corollary is the original statement of Rabin's Theorem, see [45, Theorem 1.1.].

Alternating parity tree automata. To show complementation of nondeterministic tree automata, we pass through a more powerful model. The syntax of an *alternating parity tree automaton* is defined the same as in Definition 5.4 for nondeterministic automata, with the following differences: (1) to each state we assign an *owner*, which is either "player 0" or "player 1"; and (2) for each letter a, the transition relation has form

$$\delta_a \subseteq Q \times \{\epsilon, 0, 1\} \times Q.$$

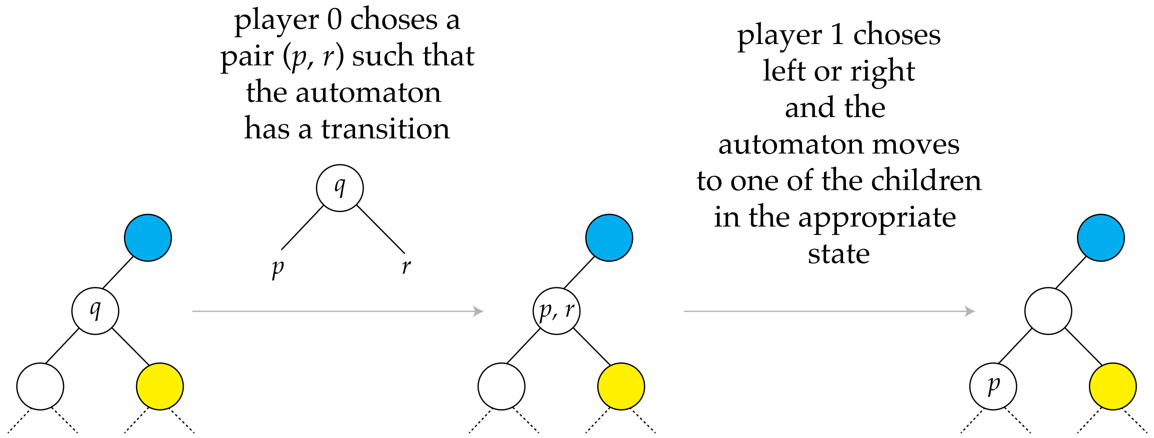
To define whether or not an automaton $\mathcal{A}$ accepts an input tree t over Σ, we consider a parity game $G_{\mathcal{A}}(t)$ defined as follows. The positions of the game are pairs (state of the automaton, node of the input tree). The initial position is (root state, root of the tree). Suppose that the current position is (q, v), and assume that state q is owned by player $i \in \{0, 1\}$. In such a position, player i chooses some pair (x, p) such that (q, x, p) belongs to the transition relation corresponding to the label of v. If there is no such pair, then player i loses immediately. Otherwise, the new position is set to $(p, v \cdot x)$, and the play continues. If the play continues forever, then the winner is declared using the parity condition, i.e. player 0 wins if and only if the maximal rank of a state appearing infinitely often is even. This completes the definition of the game $G_{\mathcal{A}}(t)$. A tree t is accepted if player 0 has a winning strategy in the game.

Theorem 5.7 (Dealternation Theorem)**.**

1. *For every nondeterministic parity tree automaton, one can compute an alternating one that recognises the same language.*
2. *Languages recognised by alternating parity tree automata are closed under complement.*
3. *For every alternating parity tree automaton, one can compute a nondeterministic one that recognises the same language.*

Before proving the above result, we show how it completes the proof of the Rabin's Theorem. Recall that the only missing ingredient was the Complementation Lemma. Using the Dealternation Theorem, we can easily complement nondeterministic parity tree automata: (1) make the automaton alternating, (2) complement it, (3) make it nondeterministic again.

Proof of Theorem 5.7. For item 1, let $\mathcal{A}$ be a nondeterministic parity tree automaton with states Q. The simulating alternating automaton has states $Q + Q^2$. The initial state is the root state of $\mathcal{A}$, and the transitions are explained in the following picture:


player 0 choses a pair (p, r) such that the automaton has a transition
q
p
r
q
p, r
player 1 choses left or right and the automaton moves to one of the children in the appropriate state
p

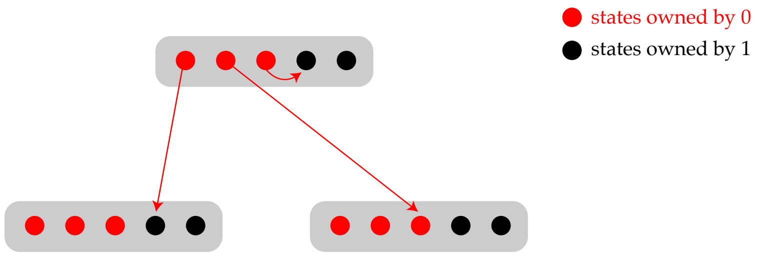
The parity condition for states from Q is inherited from the original nondeterministic automaton, and all states from Q^2 are assigned the least important rank.

For item 2, let $\mathcal{A}$ be an alternating parity tree automaton. Define $\overline{\mathcal{A}}$ to be the alternating parity tree automaton obtained from $\mathcal{A}$ by swapping the roles of players 0 and 1, and incrementing the ranking function so that even ranks become odd and vice versa, but the precedence order on ranks is maintained. To prove that $\overline{\mathcal{A}}$ is the complement of $\mathcal{A}$, we show below that the following conditions are equivalent for every input tree t:

1. $\mathcal{A}$ accepts t;
2. player 0 has a winning strategy in the game $G_{\mathcal{A}}(t)$;
3. player 1 has a winning strategy in the game $G_{\overline{\mathcal{A}}}(t)$.
4. player 1 does not have a winning strategy in the game $G_{\overline{\mathcal{A}}}(t)$.
5. $\overline{\mathcal{A}}$ rejects t.

The equivalences $1 \Leftrightarrow 2$ and $4 \Leftrightarrow 5$ are by definition of the language recognised by an alternating automaton. The equivalence $2 \Leftrightarrow 3$ is by construction of $\overline{\mathcal{A}}$. The equivalence $3 \Leftrightarrow 4$ is because $G_{\overline{\mathcal{A}}}(t)$ is a parity game, and it is therefore determined, i.e. one of the players has a winning strategy. The reason why this proof works is that: (a) the parity condition is self-dual, which allows one to define $\overline{\mathcal{A}}$; and (b) games with the parity condition are determined.

It remains to show the last item of the theorem, namely that alternating parity tree automata can be made nondeterministic. Suppose that $\mathcal{A}$ is an alternating parity tree automaton, with states Q and input alphabet Σ. By memoryless determinacy of parity games, it follows that a tree t is accepted if and only if player 0 has a memoryless winning strategy σ_0 in the game $G_{\mathcal{A}}(t)$. We will find a nondeterministic parity automaton on trees which checks this. Define Γ to be an alphabet which consists of functions from states controlled by player 0 to pairs in $Q \times \{\epsilon, 0, 1\}$. Here is a picture of a such a letter:



A memoryless strategy σ_0 for player 0 can be represented as a tree over this alphabet as follows: the label of node v is the function which maps state q to the pair (p, x) such that strategy σ_0 goes from (q, v) to $(p, v \cdot x)$.
We will show that the language

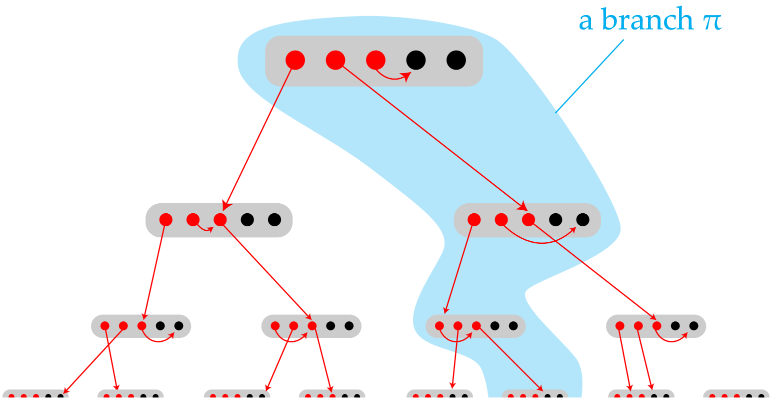
$$\{ \underbrace{(t, \sigma_0)}_{\substack{\text{tree over } \Sigma \times \Gamma \\ \text{representing } t \text{ and } \sigma_0}} : \sigma_0 \text{ is a memoryless strategy for player 0 in } \mathcal{G}_{\mathcal{A}}(t)\} \tag{5.1}$$

is recognised by a (even deterministic top-down) parity automaton on trees. This will complete the proof of the Dealternation Theorem, because a nondeterministic parity automaton can guess the part of the labelling that describes σ_0. The key observation is the following claim. (A branch is defined to be an inclusion-wise maximal set of nodes that are totally ordered by the descendant relation.)

Claim 5.8. *There is a* nondeterministic *parity automaton $\mathcal{B}$ over ω-words over the alphabet $\Sigma \times \Gamma \times \{0, 1\}$ such that the following conditions are equivalent for every tree t, branch π and memoryless strategy σ_0 for player* 0*:*

1. *There exists a strategy of player σ_1 such that if the players use strategies (σ_0, σ_1) in the game $\mathcal{G}_{\mathcal{A}}(t)$, then the resulting play stays on the branch π and* violates *the parity condition.*

2. *The automaton $\mathcal{B}$ accepts the ω-word $(t, \sigma_0)|\pi$ defined as follows: the i-th letter is of the label of the i-th node in π as well as the turn that π takes after that node. Here is a picture:*



Proof. The automaton $\mathcal{B}$ uses nondeterminism to choose the moves of the strategy σ_1. ■

Apply the above claim, yielding a nondeterministic parity automaton. By McNaughton's Theorem, see Chapter 1, there exists an equivalent deterministic parity automaton, call it $\mathcal{D}$. It is not difficult to see that a memoryless strategy σ_0 wins in the game $G_{\mathcal{A}}(t)$ if and only if every branch in the tree (t, σ_0) is rejected by the automaton $\mathcal{D}$. This can be checked by a (deterministic top-down) parity automaton on trees, which runs the automaton $\mathcal{D}$ on every branch (and has the acceptance condition complemented). ■

Problem 36. The translation from MSO to automata in Theorem 5.3 does an exponential blowup whenever it determinises the automaton, and therefore an upper bound on the running time is n-fold iteration of exponential, where n is the size of the formula. Here is a matching lower bound. Consider MSO on words, i.e. there is a successor relation and unary predicates for the labels. Show that for every n, there is a formula of MSO (in fact, first-order logic is enough) which has size polynomial in n and is true in a unique word which has length

$$\underbrace{2^{2^{2^{2^{\cdot^{\cdot^{\cdot^{2^{2^{2^{2}}}}}}}}}}}_{n \text{ times}}$$

Problem 37. Show that the set $\mathbb{N}^*$ equipped with the prefix relation has decidable MSO theory.

Problem 38. Show that emptiness is polynomial time and universality is ExpTime-complete for nondeterministic tree automata on finite trees.

Problem 39. Show that emptiness for nondeterministic parity tree automata reduces in polynomial time to solving parity games.

Problem 40. Determine whether the following tree languages are regular:

1. trees with an even number of nodes;

2. trees with an even number of a-labelled nodes;

3. trees over leaf alphabet $0, 1$ and internal alphabet $\vee, \wedge$ which evaluate to true when treated as boolean expressions;

4. balanced trees (every leaf is at the same depth).

Problem 41. Determine which of the following four variants of tree automata: deterministic / nondeterministic, top-down / bottom-up tree automata are equivalent.

Problem 42. Define the *yield* of a tree to be the word composed from labels of its leaves written in infix order. Show that for every $L \subseteq \Sigma^*$ the following are equivalent

1. L is context-free;

2. L is the set of yields of some regular tree language.

Problem 43. Show that deterministic top-down tree automata cannot recognize the language "some node has label a".

Problem 44. Show that the language of words of even length is definable in MSO.

Problem 45. Show that the following languages of infinite trees are regular (accepted by some nondeterministic automaton):

1. on every path, the sequence of labels belongs to a given ω-regular language L;

2. some node has label a;

3. in every subtree some node has label a.

Problem 46. In Existential Second Order Logic ($\exists$SO) one can write $\exists_{R_1,\ldots,R_n}\phi$, where R_i are any relations (possibly of arity greater than 1) and ϕ is a first order sentence (which of course may use R_i). Show that the language of words of composite (non-prime) length is expressible in $\exists$SO.

Problem 47. Consider the following game. There are two players *Insider* and *Outsider*. They choose in an alternating manner bits: 0 or 1 and create in that way an ω-word w. If w belongs to a given regular language $W \subseteq \{0,1\}^\omega$ then Insider wins a play, otherwise Outsider wins. Show that it is decidable to check which player has a winning strategy in that game. *Remark:* use MSO logic.

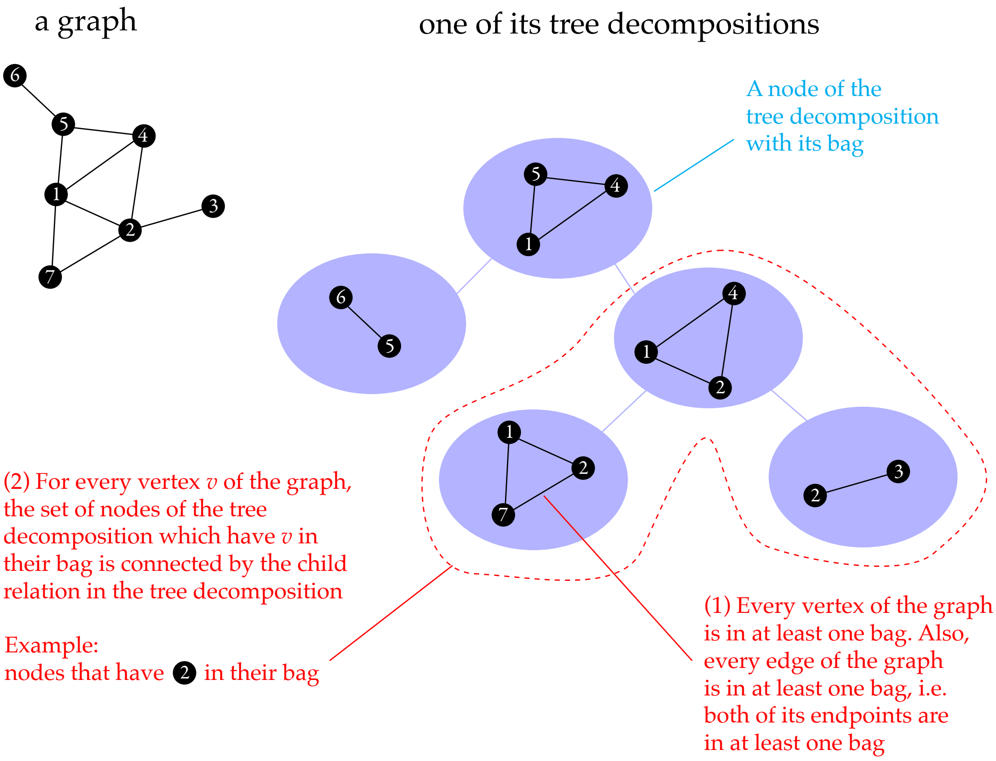
6

Treewidth

In this chapter, by graphs, we mean finite undirected graphs. We treat a graph as a logical structure, where the universe is the vertices and there is a binary edge relation, which is necessarily symmetric (for a different representation, see the exercises). We present Courcelle's Theorem, which says that every formula of MSO on graphs can be evaluated in linear time on graphs that have bounded treewidth. Treewidth is a graph parameter, i.e. every graph has a some treewidth, which is a natural number. The treewidth of a graph describes the smallest width of a tree decomposition that can produce the graph. The general idea is that small width tree decompositions can be obtained for graphs that are similar to trees. Treewidth is not the only way of quantifying similarity to a tree, alternatives include cliquewidth, see [23, Section 2.5] or treedepth [42, Chapter 6].

6.1 Treewidth and how to compute it

Consider a graph G. Define a *tree decomposition* of G to be a tree, where each node of the tree is labelled by a set of vertices in the graph, called the *bag of the node*, subject to conditions (1) and (2) depicted in the following picture:



A node of the tree decomposition with its bag

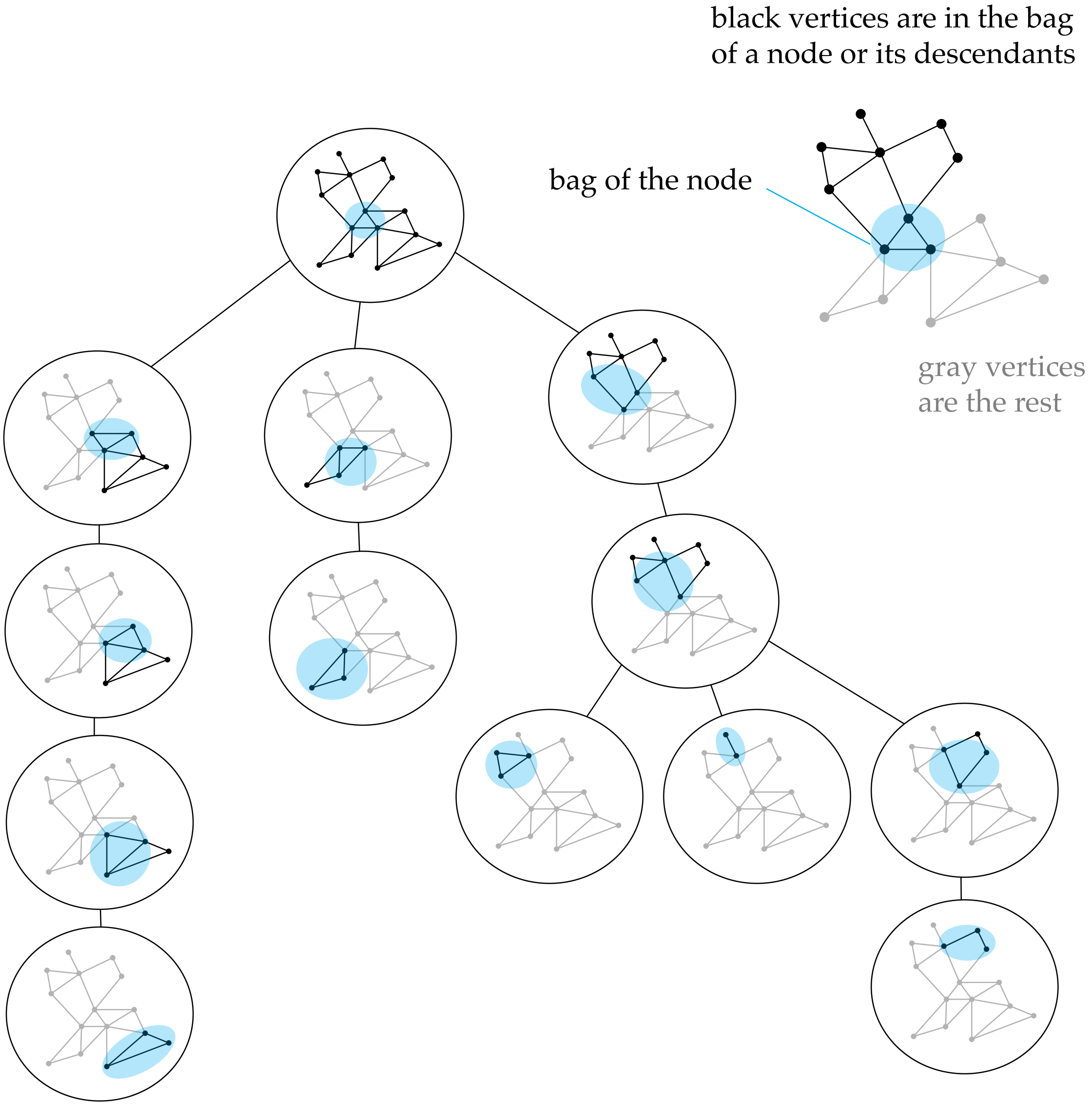
(2) For every vertex v of the graph, the set of nodes of the tree decomposition which have v in their bag is connected by the child relation in the tree decomposition

Example: nodes that have ❷ in their bag

(1) Every vertex of the graph is in at least one bag. Also, every edge of the graph is in at least one bag, i.e. both of its endpoints are in at least one bag

In the tree decomposition, we allow nodes to have unbounded arity, i.e. there is no requirement that each node has at most two children. The tree in the tree decomposition is unordered (i.e. there is no ordering on the siblings), but it is rooted, i.e. it makes sense to talk about descendants and children. Define the *width* of a tree decomposition to be the maximal size of a bag minus one. In the picture above, the width is 2, because the maximal bag size is 3. The reason for the minus one is so that trees have treewidth one. Another reason is that the width of a tree decomposition is the intersection between neighbouring bags (assuming the tree decomposition does not use the same bag twice, which can be assumed without loss of generality). The *treewidth* of a graph is the minimal width of a tree decomposition of it. Treewidth is a fundamental concept in graph theory, which plays a prominent role in the graph minor project of Robertson and Seymour.

An alternative way of drawing tree decompositions is in the following picture:


black vertices are in the bag of a node or its descendants
bag of the node
gray vertices are the rest

Fact 6.1. *If a graph has treewidth k, then the number of edges in the graph is at most* $k \cdot (k+1)/2$ *times the number of vertices.*

Proof. A tree decomposition can always be modified so that the bag of a node contains at least one vertex that is not present in the bags of its descendants.

Therefore, the number of nodes in the tree decomposition is at most the number of vertices in the underlying graph. Each edge must be present in some node, and each node can have at most $k \cdot (k+1)/2$ edges, which proves the fact. The bound in the fact is optimal, as witnessed by a clique over $k+1$ vertices. ■

Computing a tree decomposition. We present an algorithm that computes tree decompositions of approximately optimal width (at most four times worse, see below for the exact statement) and which runs in quadratic time when the treewidth is fixed. The algorithm is from Robertson and Seymour, see also [24, Theorem 7.18].

Theorem 6.2. *There is a function $f : \mathbb{N} \to \mathbb{N}$ and an algorithm which runs in time $f(k) \cdot n^2$ that approximates tree decompositions in the following sense:*

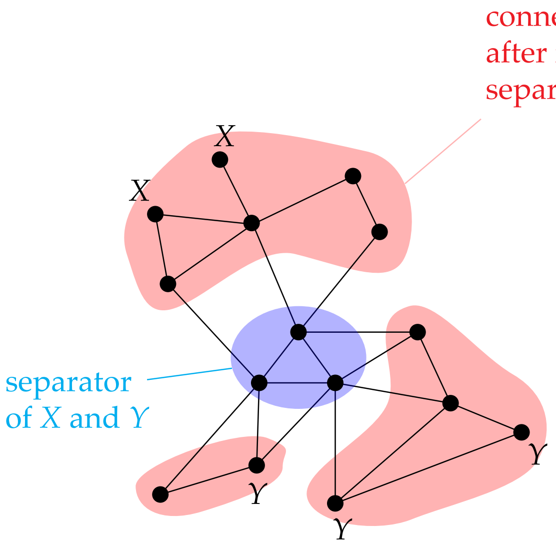
- **Input**. *k and a graph with n vertices;*
- **Output.** *A tree decomposition of the graph which has width $< 4k$, or a certificate that the graph has treewidth $\geq k$.*

The algorithm from the theorem is not optimal. The optimal algorithm, by Bodlaender [9], runs in linear time instead of quadratic time, and computes tree decompositions of optimal width (i.e. $< k$ instead of $< 4k$). The function $f(k)$ is exponential, and there is little hope for improvement, because the following problem is NP-complete [5]: given k and a graph, decide if the graph has treewidth at most k. The theorem gives a (prototypical) example of a an algorithm that is *fixed parameter tractable*, i.e. the input has two parameters k, n and the running time is of the form:

$$\underbrace{f(k)}_{\text{some computable function}} \cdot \underbrace{n^c}_{\text{a polynomial with degree independent of } k}$$

The algorithm uses the following lemma on computing separators. Recall that a separator of vertex sets X and Y in a graph G is a set of vertices S disjoint from

$X \cup Y$ such that $G - S$ does not contain any path connecting X with Y, as in the following picture:



Lemma 6.3. *Given a graph G and disjoint sets of vertices X, Y, one can compute a separator of minimal size in time*

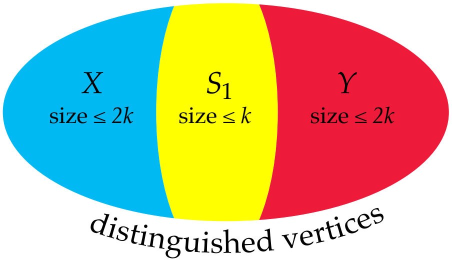
$$\mathcal{O}((\textit{number of edges} + \textit{number of vertices}) \cdot (\textit{size of the separator})).$$

We do not prove the above lemma, it can be shown using the Ford-Fulkerson algorithm for computing maximum flow, see the discussion in [24, p. 198]. When the treewidth is fixed, the number of edges is linear in the number of vertices, and the size of the separator is bounded by a function of k (see the proof of Lemma 6.4), and therefore the running time of the algorithm is linear. The main step in proving Theorem 6.2 is the following lemma.

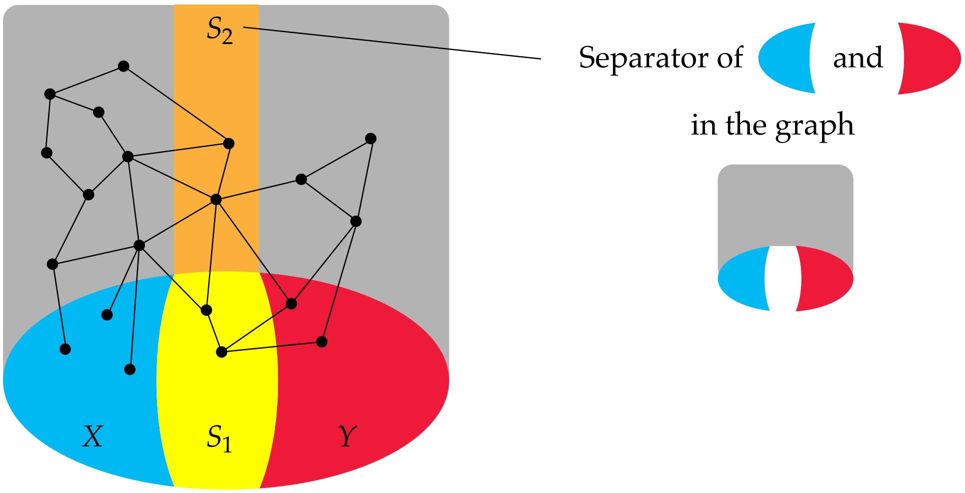
Lemma 6.4. *Let $k \in \mathbb{N}$. There is a linear time algorithm which does this:*

- **Input.** *k and a graph G with $\leq 3k$ distinguished vertices;*
- **Output.** *A certificate that the graph has treewidth $\geq k$, or a set S of $\leq k$ vertices so that $G - S$ has at least two connected components, and each connected component has $\leq 2k$ distinguished vertices.*

Proof. We begin with the algorithm, and then justify why it succeeds on graphs of treewidth $< k$. We enumerate all possible partitions of the distinguished vertices into three parts as follows

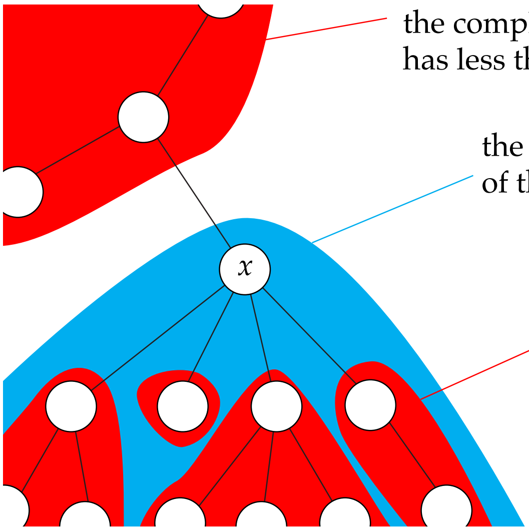


The idea is that S_1 is the intersection of the separator with the distinguished vertices. The number of such partitions is exponential in k, but is a constant if k is assumed to be fixed. For each such partition, compute a minimal size separator S_2 of X and Y in the graph $G - S_1$, as depicted in the following picture



Report success if the size of $S_1 \cup S_2$ is at most k, and return $S_1 \cup S_2$ as the separator. This completes the algorithm. The running time is linear, because the size of the separator is fixed, and the number of edges is linear in the number of vertices by Fact 6.1.

We now justify that if G has treewidth $< k$ then the algorithm succeeds. If the graph has treewidth $< k$, then there is a tree decomposition where all bags have size $\leq k$. Let t be this tree decomposition. Choose a node x of the tree decomposition so that half or more of the distinguished vertices of G appear in bags of x and its descendants, but this is no longer true for any of the children of x. Here is a picture:



the complement of the blue subtree
has less than half of the distinguished vertices

the blue subtree has at least half
of the distinguished vertices

each red subtree has less than half
of the distinguished vertices

Define S to be the bag of x. The size of S is $\leq k$. By choice of x we know that every connected component of $G - S$ has at most half of the distinguished vertices. In particular, there must be at least two connected components, because

$$\underbrace{3k}_{\substack{\text{distinguished}\\\text{vertices}}} > \underbrace{k}_{\substack{\text{distinguished}\\\text{vertices in } S}} + \underbrace{3k/2}_{\substack{\text{distinguished}\\\text{vertices in each}\\\text{connected component}}}$$

For each connected component of $G - S$, we count the number of distinguished nodes in that component; this is a number that is at most half of $3k$. The following claim, when applied to the numbers of distinguished vertices in the connected components of $G - S$, shows that the connected components can be grouped into two groups, so that each group has at most $2k$ distinguished vertices, thus proving the lemma.

Claim 6.5. *Let $n_1 \geq n_2 \geq \cdots \geq n_p$ be numbers in $\{1, \ldots, 2k\}$ with sum $\leq 3k$. Then*

$$\overbrace{n_1 + \cdots + n_i}^{\leq 2k} \quad \overbrace{n_{i+1} + \cdots + n_p}^{\leq 2k} \qquad \textit{for some } i$$

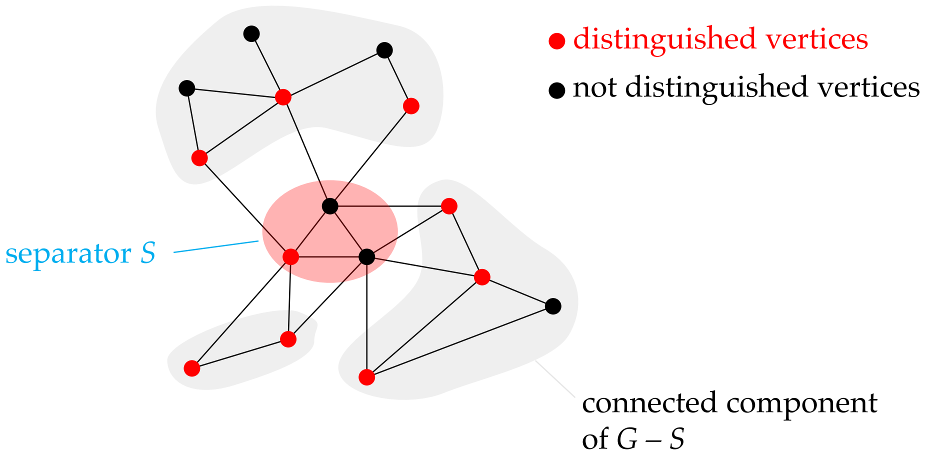
Proof. Take the first i such that the sum of the first i elements is $\geq k$. ■

■

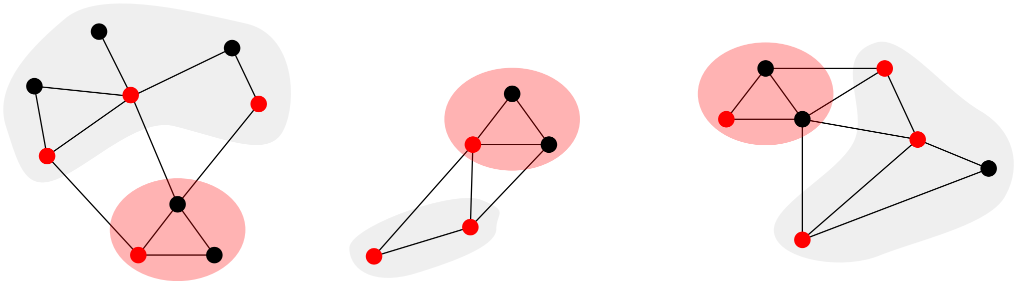
Proof of Theorem 6.2. We use a more detailed statement of the algorithm, as described below.

- **Input** k and a graph with $\leq 3k$ distinguished vertices;
- **Output.** A certificate that the graph has treewidth $\geq k$, or a tree decomposition of the graph which has width $< 4k$ and where the root bag consists exactly of the distinguished vertices.

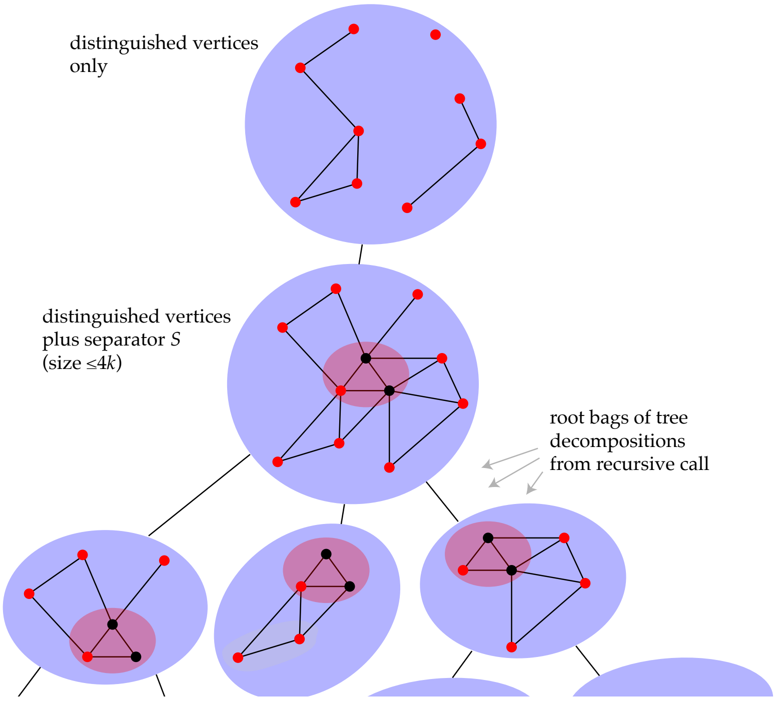
Suppose that G is the graph. If there are $< 3k$ distinguished vertices, we add some arbitrary vertices to make the set have size exactly $3k$. Apply Lemma 6.4, computing S, X and Y. If the input graph has treewidth $< k$ then the algorithm from the lemma must succeed. Find all connected components of the graph $G - S$, of which there are at least two. Each connected component has $\leq 2k$ distinguished vertices. Here is a picture:



For each connected component U of the graph $G - S$, define G_U to be the graph induced by $U \cup S$. This graph is smaller than G, because $G - S$ has at least two connected components. Here are are the graphs G_U for our picture above:



For each of the graphs G_U, recursively call the algorithm, with the distinguished vertices being S plus the original distinguished vertices from U. We are allowed to do the recursive call, since U has $\leq 2k$ distinguished vertices and S has at $\leq k$ vertices. Combine the tree decompositions yielded by the recursive calls into a single tree as follows:


distinguished vertices only
distinguished vertices plus separator S (size ≤$4k$)
root bags of tree decompositions from recursive call

It is not difficult to check that this is a tree decomposition of G. The size of bags is $\leq 4k$, and therefore the width of the decomposition is $< 4k$ (recall that the width was size of bags plus one). The algorithm does a linear computation, followed by recursive calls to smaller instances; and therefore its running time is quadratic. ■

6.2 *Courcelle's Theorem*

In this section we prove Courcelle's Theorem, which says that MSO can be evaluated efficiently on graphs of bounded treewidth. The key ingredient is the following lemma, which is proved the same way as Courcelle's original result that MSO definable graph properties are recognisable, see [22, Theorem 4.4].

Lemma 6.6. *For every $k \in \mathbb{N}$ and every formula of* MSO *φ on graphs, there is a linear time algorithm which does the following:*

- **Input.** *A graph together with a tree decomposition of width $\leq k$;*

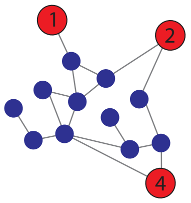
- **Question.** *Does the graph satisfy φ?*

The proof of the lemma is essentially this: we view the tree decomposition as a tree over a finite alphabet, convert the formula φ into a tree automaton, and then run the tree automaton over the tree in linear time. If we combine the lemma with an algorithm that computes tree decompositions, we do not need to get the tree decomposition on input. This yields the following formulation of Courcelle's Theorem (the algorithm for computing tree decompositions in these notes gives only a quadratic running time, for the linear time bound one needs the algorithm of Bodlaender from [9]):

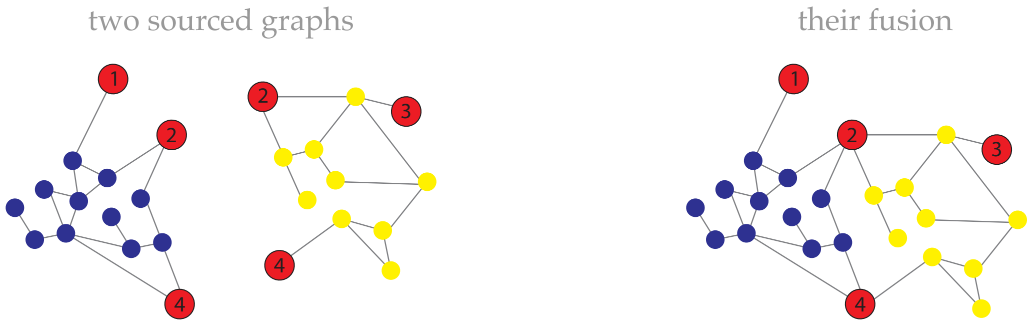
Theorem 6.7 (Courcelle's Theorem)**.** *For every $k \in \mathbb{N}$ and every formula of* MSO *φ on graphs, there is a linear time algorithm evaluates φ on graphs of treewidth $\leq k$.*

The rest of this chapter is devoted to proving Lemma 6.6. To this end, we present a more algebraic way of defining treewidth, so that tree decompositions can be viewed as trees over a finite ranked alphabet.

The algebra of tree decompositions. Define a *sourced graph* to be a graph with some but not necessarily all vertices being assigned natural numbers. The vertices with numbers are called the *sources* and the numbers are called the *source names*. Each source name can be used for at most one source. A width k sourced graph is one where the source names are from $\{0, \ldots, k\}$, note that $k+1$ source names are allowed; this corresponds to bags having size $k+1$ in a width k tree decomposition. A sourced graph with no sources is the same as a graph. Here is a picture of a width 4 sourced graph, which does not use source names 0 and 3:



The purpose of sourced graphs is to combine them using the following fusion operation. The fusion operation inputs two sourced graphs, and outputs their disjoint union with each pair of sources that have the same name being merged together into a single vertex, as in the following picture


two sourced graphs

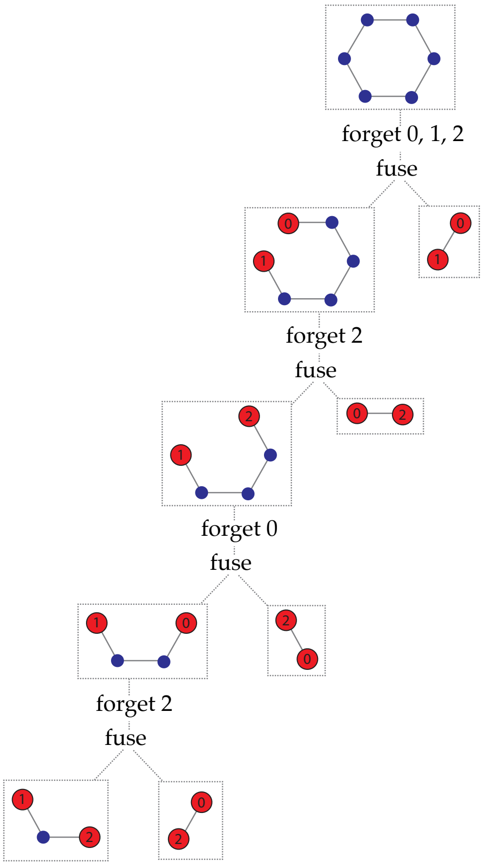
their fusion

Besides fusion, we also use an operation that forgets some source names, illustrated below:

a sourced graph

after forgetting 1, 4.

For $k \in \mathbb{N}$, define *the algebra of width k sourced graphs* to be the algebra where the universe is width k sourced graphs, and which is equipped with a binary fusion operation and a family of unary forget operations (one for every subset of source names). Here is a term in the algebra of width k sourced graphs that generates a cycle of length 6:



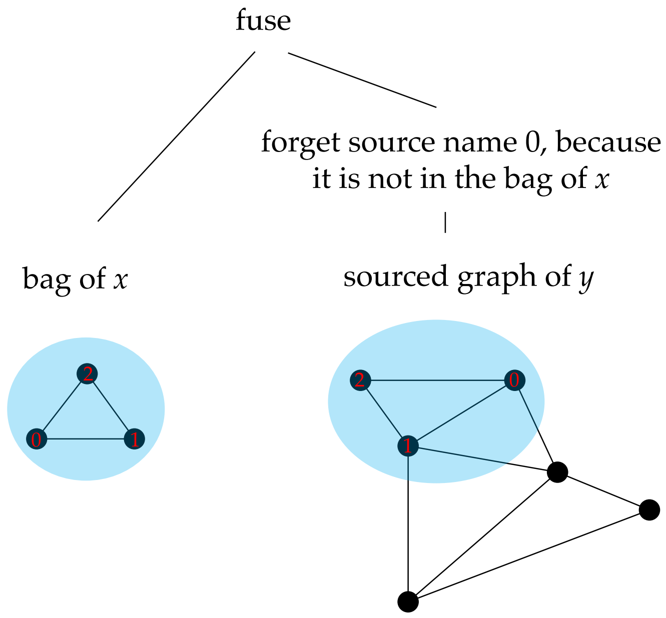
Fact 6.8. *A graph has treewidth k if and only if (when viewed as a sourced graph without any sources) it can be generated by a term in the algebra of width k sourced graphs, starting with constants that have at most $k + 1$ vertices.*

Proof. We only do the top-down implication. Consider a tree decomposition (in the standard, non-algebraic way) of width k. Using at top-down greedy algorithm, one can colour the vertices of the graph with colours $\{0, \ldots, k\}$ so that for each bag of the tree decomposition, all vertices in the bag have different colours. For a node x of the tree decomposition, define a sourced graph as follows:

- the graph is the subgraph induced by the union of bags of x and its descendants (this is sometimes known as the *cone* of node x);

- the sources are the bag of x, with source names taken from the colouring.

By induction on the number of descendants of x, we show that the sourced graph corresponding to x in the above sense can be generated by a term in the algebra of sourced graphs as in the statement of the fact. In the induction step, we do the following. For every child y of x, we combine the sourced graph generated by the subtree of y with the bag of x as follows:



Then we fuse all of the resulting graphs, with y ranging over children of x. ■

A term as in Fact 6.8 can be viewed as a tree over a ranked alphabet Σ_k where:

- leaves are width k sourced graph with at most $k+1$ vertices;
- unary nodes are forget operations for subsets $I \subseteq \{0, \ldots, k\}$;
- binary nodes all have the same label "fuse".

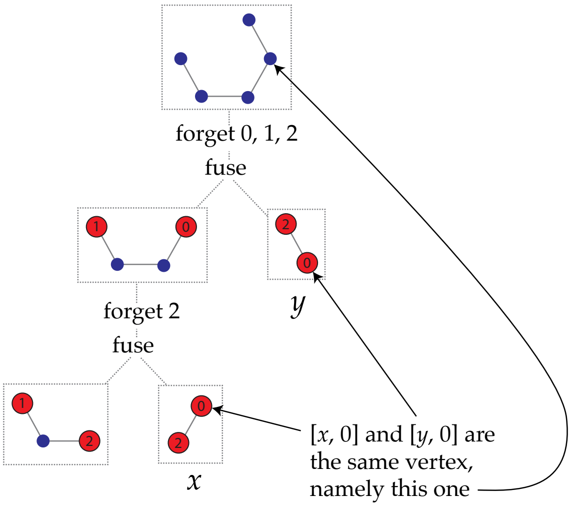
A width k tree decomposition can be converted into a corresponding tree over the above alphabet in linear time. Since the fusion operation as used in Σ_k has arity two, the conversion produces tree decompositions with binary branching (which can break properties, like no bag being used twice). By Theorem 5.3, for finite trees over alphabet Σ_k, MSO is equivalent to tree automata. Since tree automata can be evaluated in time linear in the size of the input tree (it is easier to use the bottom-up deterministic variant), it follows that MSO formulas on trees can also be evaluated in linear time. Therefore, Lemma 6.6 will follow once we prove the following lemma.

Lemma 6.9. *Let $k \in \mathbb{N}$ and let φ be an* MSO *formula over graphs. There is a* MSO *formula $\hat{\varphi}$ on trees over alphabet Σ_k such that*

t satisfies $\hat{\varphi}$ iff the graph of t satisfies φ

holds for every width k tree decomposition, viewed as a tree over Σ_k.

Proof. Consider a tree t as in the statement of the lemma. To a node x in the tree and a source name $i \in \{0,\ldots,k\}$, there corresponds a vertex $[x,i]$ of the graph generated by t in the natural way, as depicted in the following picture:



The encoding $(x,i) \mapsto [x,i]$ is partial, because it is undefined if the source name i is not present in the sourced graph that is generated by the subtree of x. It is not hard to see that for every source names $i,j \in \{0,\ldots,k\}$ the following binary relations on nodes x,y of t are definable in MSO:

- $[x,i], [y,i]$ are both defined and equal;
- the graph has an edge from $[x,i]$ to $[y,j]$.

Using the above relations, one can simulate an MSO formula φ over the graph generated by t using an MSO formula over t itself. When φ quantifies over a set of vertices U, then $\hat{\varphi}$ quantifies over $k+1$ sets of nodes, namely:

$$\{x : [x,0] \in U\}, \ldots, \{x : [x,k] \in U\}.$$

The professional terminology for the construction described above is "the graph generated by t can be produced from t using an MSO transduction", see [23, Section 1.7]. ■

Problem 48. Show that a graph has treewidth 1 iff it is a forest.

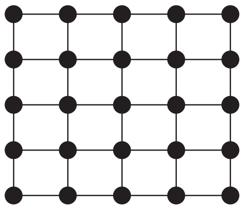
Problem 49. Compute the treewidth of the clique of n vertices.

Problem 50. Consider the following game on a graph G between k cops and one robber. The robber has a fast motorbike, cops have helicopters. In between moves everybody occupies one vertex. A round of the game is played as follows:

- some subset of the cops starts flying their helicopters and declares where they are going to land (different cops might land in different places) at the end of the round; the remaining cops stay on the ground,
- the robber moves along a path; he cannot pass through vertices that are occupied by cops who are on the ground,
- the cops in helicopters land on the declared vertices.

The cops win if they manage to land on the vertex with the robber. Show that if a graph has treewidth k then $k+1$ cops have a winning strategy in this game. Remark: if a graph has treewidth k then the robber has a winning strategy against k cops, but this is harder to show.

Problem 51. Let G_k be a grid $k \times k$ (with k^2 vertices). Show that G_k has treewidth which is either k or $k-1$. The actual answer is k, but showing this is a bit technical.



square grid of dimension 5

Problem 52. Determine the treewidth of the full bipartite graph with n vertices on the left and n vertices on the right.

Problem 53. Show that the vertex cover problem can be solved on a graph G in time $2^{\mathcal{O}(\mathrm{tw}(G))} \cdot n^{\mathcal{O}(1)}$.

Problem 54. A graph G is called a *minor* of graph H, denoted $G \trianglelefteq H$, if G can be obtained from H by a sequence of operations of one of the following three types: 1) deleting a vertex, 2) deleting an edge, 3) contracting an edge, i.e. unifying two endpoints of this edge. Show that $G \trianglelefteq H$ implies $\mathrm{tw}(G) \leq \mathrm{tw}(H)$.

Problem 55. Show that there exists a function f such that if a graph G is connected then it has a walk (a path which is allowed to visit vertices multiple times) that visits all vertices and visits every edge at most $f(\mathrm{tw}(G))$ times. Show that this is no longer true if we want to limit the number of visits to every vertex.

Problem 56. Show that the following problem is decidable: given an MSO formula φ and $k \in \{1, 2, \ldots\}$, decide if φ is true in some graph of treewidth at most k.

Problem 57. Consider two representations of graphs as logical structures:

- *Edge representation.* The universe is the vertices and there is a binary relation for neighbourhood.

- *Incidence representation*. The universe is the vertices and the edges, and there is a binary relation for incidence of a vertex with an edge.

With edge representation, MSO can quantify over sets of vertices, while with incidence representation, MSO can quantify over sets of vertices and edges. When proving Lemma 6.6, we used edge representation. Show that the lemma and also Problem 56 remain true with the incidence representation.

Problem 58. Show that the language of connected graphs is definable in MSO on graphs (assume edge representation, as described in the Problem 57).

Problem 59. Show that the language of all forests is definable in MSO on graphs (assume edge representation, as described in the Problem 57).

Problem 60. Show that the language of grids is definable in MSO on graphs and find an appropriate formula (assume edge representation, as described in the Problem 57).

Problem 61. Recall the edge and incidence representations from Problem 57. Show a property of graphs that is definable using incidence representation but not using edge representation.

Problem 62. Recall the edge and incidence representations from Problem 57. Find a class of graphs $\mathcal{C}$ such that the following problem is decidable for the edge representation but not for incidence representation: given a formula of MSO, decide if it is true in some graph from $\mathcal{C}$.

Problem 63. Show that "has an Euler cycle" is a graph property that is not definable in MSO, even if one uses the incidence representation from Problem 57.

Problem 64. Consider the extension of MSO, called counting MSO, where one can write a formula "the size of set X is divisible by n" for every n. Show that having an Euler cycle is definable in counting MSO.

Problem 65. Show that Lemma 6.6 remains true when we use counting MSO (see Problem 64) and incidence representation.

Problem 66. The grid theorem [46, 17] says that if a class of graphs has unbounded treewidth, then it has square grids of arbitrarily large dimensions as minors. Using the grid theorem, show that if a class $\mathcal{C}$ of finite graphs has unbounded treewidth, then the following problem is undecidable: given an MSO formula φ, decide if it is true in some graph from $\mathcal{C}$.

Problem 67. Show that for every $k \in \mathbb{N}$ there exists $t \in \mathbb{N}$ such that if a graph has treewidth $\geq t$ then it has k vertex disjoint cycles. *Hint:* use the grid theorem.

Problem 68. Show that for a planar graph one can check in time $2^{\mathcal{O}(\sqrt{k}\log(k))} \cdot n^{\mathcal{O}(1)}$ whether it contains a simple path with at least k vertices. *Hint:* use the grid theorem for planar graphs in the following form: if a planar graph has treewidth $\geq 5k$ then it has the $k \times k$ grid as a minor.

Problem 69. Recall ∃SO from Problem 46. Let us model a graph as relational structure using the edge representation discussed in Problem 57 (for the incidence representation, the same result would be true). Show that a property of graphs is definable in ∃SO if and only if it is in the class NP (this is Fagin's theorem).

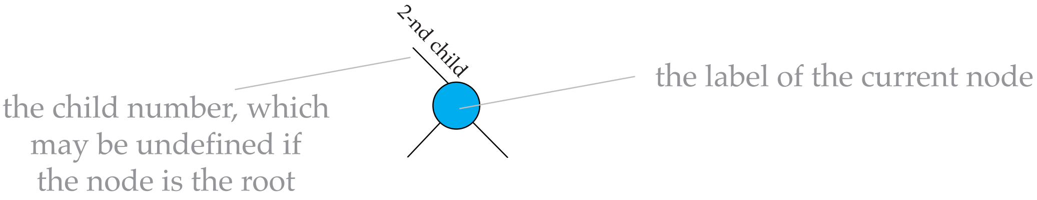
Problem 70. Show that the following problem is undecidable: the input is a formula of ∃SO that uses only equality (and the quantified relations); the question is if this formula is true in some finite structure (i.e. a finite universe equipped with equality only).

Problem 71. Show that there is a polynomial time algorithm deciding whether a given graph is planar. *Hint:* assume that there exists a polynomial algorithm deciding whether a given graph G is a minor of an input graph H.

Problem 72. Show that there exists a polynomial time algorithm deciding whether a given graph can be drawn on torus without crossing edges.

7
Tree-walking automata

In this chapter we discuss a less standard automaton model for finite trees – as opposed to tree automata discussed in Chapter 5 – namely tree-walking automata. We show that tree-walking automata cannot be not determinised. The point of a tree-walking automaton is to have a run that is sequential, i.e. it is a sequence – and not a tree – of configurations. The idea is that at any moment of its run, the automaton is in a single node of the input tree. Based on the local view, which is this information


2-nd child
the label of the current node
the child number, which may be undefined if the node is the root

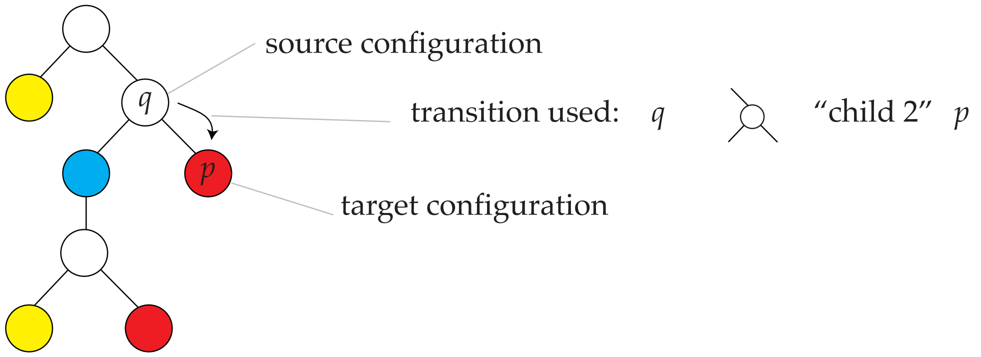
and the current state, the automaton chooses the new state and a neighbour (possibly the parent) to move to, or to accept or reject. The information on the child number is useful for operations like depth-first search, and the model becomes powerless if the child number is not included in the local view, see the exercises.

Definition 7.1 (Tree-walking automaton)**.** *The syntax of a* (nondeterministic) tree-walking automaton *consists of: a finite ranked set* Σ *called the* input alphabet, *a finite set of states Q with designated initial and accepting subsets, and a transition relation of the form*

$$\delta \subseteq \overset{\text{source state}}{Q} \times \underbrace{\text{local views over } \Sigma}_{\text{current local view}} \times \overset{\text{target state}}{Q} \times \overbrace{\{\text{parent, stay, child 1, ..., child } \underset{\text{maximal arity in } \Sigma}{n}\}}^{\text{where to move}}.$$

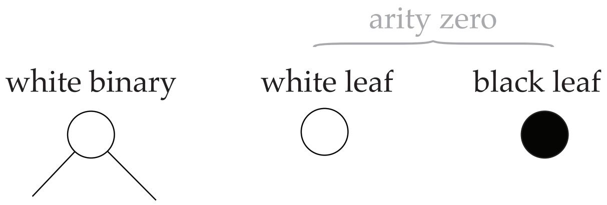
We assume that for every transition, the local view (i.e. the second coordinate) is consistent with the direction of the move (i.e. the last coordinate) in the sense that if the direction is "parent" then local view says that there is a parent, and if the direction is "child i" then local view says that there is an i-th child. The automaton is called deterministic *if it has one initial state and the transition relation is a function from the first two arguments to the last two arguments.*

Consider an input tree over the input alphabet. A *configuration* of the automaton is a pair (state, node of the input tree). A run of the automaton is a sequence of configurations, such that every two consecutive configurations are connected by the transition relation in the natural way, illustrated in the following picture:

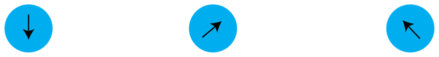


The automaton accepts a tree if there is a run which begins in an initial state at the root of the tree, and which ends in a configuration that uses an accepting state. For a deterministic tree-walking automaton, there are two ways of rejecting an input tree: (a) reaching a configuration that has no applicable transition; (b) entering an infinite loop. Using a construction from [53, Theorem 1], one can show that every deterministic tree-walking automaton can be modified so that rejection is done only via (a), see [41, Proposition 1].

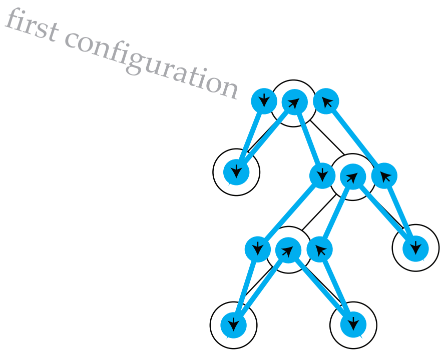
Example 8. Consider the following ranked alphabet



The tree language "at least one black leaf" can be recognised by a nondeterministic tree-walking automaton, which uses nondeterminism to find the black leaf. If we want to avoid nondeterminism, or we want to recognise the language "all leaves are white", then we can use an automaton that does a depth-first search through the tree. This automaton has three states:



which stand, respectively, for the first, second and third visit to a node. The transition relation is defined so that the run looks like in the following picture:



□

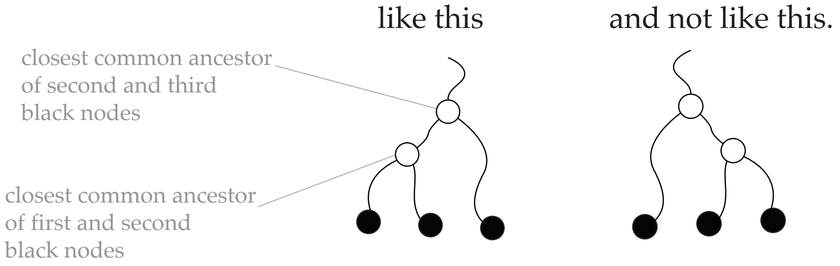
7.1 *Tree-walking automata cannot be determinised*

The goal of this chapter is to prove that tree-waking automata cannot be determinised.

Theorem 7.2. *[11, Theorem 5] There is a tree language that is recognised by a nondeterministic tree-walking automaton, but not by any deterministic tree-walking automaton.*

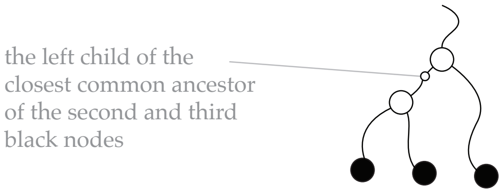
In the exercises, we show that nondeterministic tree-walking automata can be converted into tree automata as defined in Chapter 5. The converse does not hold, i.e. there is a language recognised by a tree automaton that is not recognised by any (nondeterministic) tree-walking automaton [12, Theorem 2]. The rest of this chapter is devoted to proving Theorem 7.2. We begin by defining the language which separates deterministic and nondeterministic tree-walking automata, and we show that the language can be recognised by a nondeterministic tree-walking automaton. In the next section we show the lower bound – the separating language cannot be recognised by a deterministic automaton.

The separating language. The input alphabet for the separating language is the same as in Example 8, i.e. there is a binary white letter and two leaf letters in the colours white and black. The language consists of trees with exactly three black leaves, such that the black leaves are distributed


like this
and not like this.
closest common ancestor of second and third black nodes
closest common ancestor of first and second black nodes

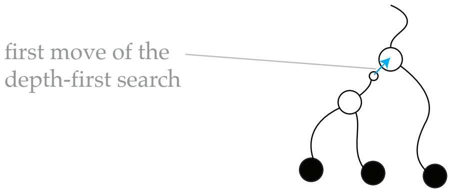
Lemma 7.3. *The separating language is recognised by a nondeterministic tree-walking automaton.*

Proof. The automaton first does a depth-first search, as explained in Example 8, to check if the tree contains exactly three black nodes. If not, it rejects. Otherwise, it uses a depth-first search to return to the first black node, and then it uses nondeterminism to go to some ancestor v of the first black node. The idea is that v will be chosen as in the following picture:

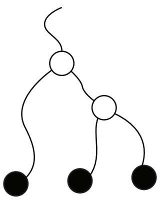

the left child of the closest common ancestor of the second and third black nodes

(*)

Next, the automaton continues a depth-first search, as if it has just visited v for the third time:



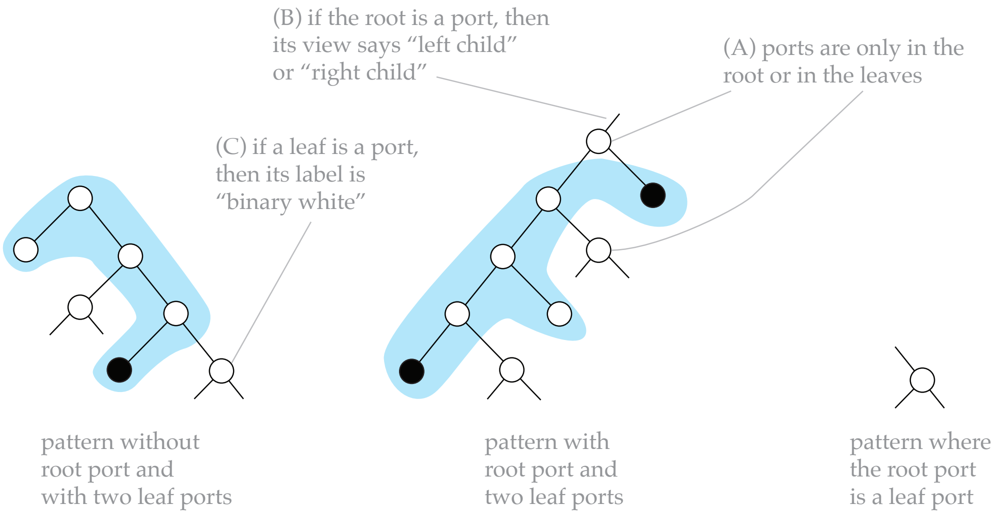
The automaton accepts if during the remainder of this depth-first search, it sees exactly one black node. If the input tree belongs to the language, then by choosing v as in (*), the automaton will accept. If the input tree is outside the language, i.e. the shape is like this:



then no matter how v is chosen, the automaton will see either zero or two black leaves in its depth-first search after visiting v. Therefore the automaton accepts all trees in the separating language, and no other trees. ■

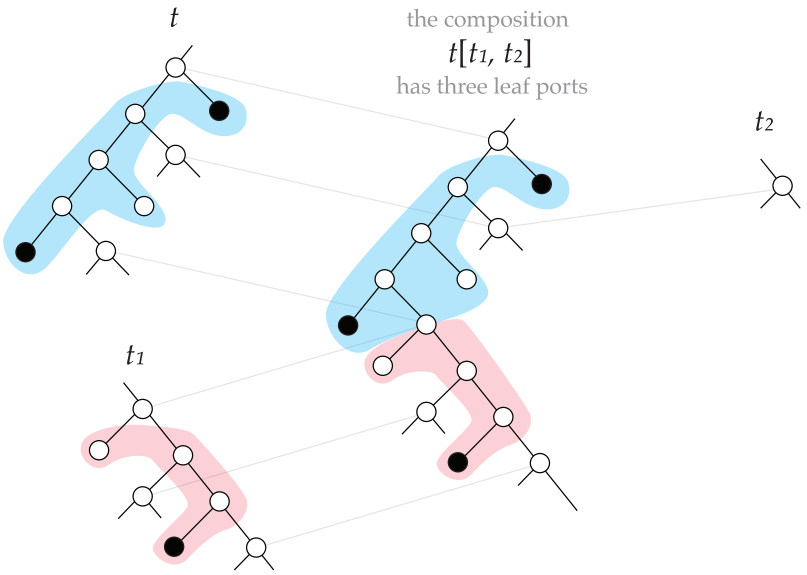
The lower bound. We now turn to the heart of Theorem 7.2, i.e. showing that a deterministic tree-walking automaton cannot recognise the separating language. We will show that for specially crafted inputs, a deterministic tree-walking automaton can only do a depth-first search, and this is not enough to recognise the separating language. Fix for the rest of this section a deterministic tree-walking automaton. We will prove that this automaton does not recognise the separating language. From now on, when talking about local views, we talk about local views for the alphabet in the separating example.

Patterns. A *pattern* is defined similarly to a tree (over our fixed input alphabet), with the exception that a subset of nodes – called *ports* – is labelled by local views instead of letters from the input alphabet, subject to constraints (A), (B), (C) in the picture below (the ports are the nodes that are not on a blue background):

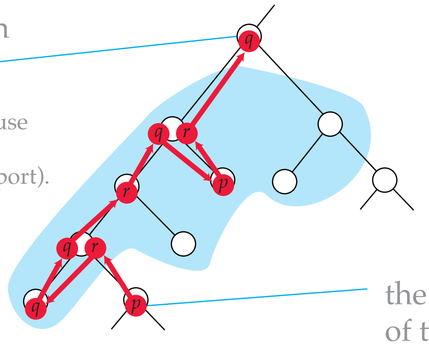

(B) if the root is a port, then its view says "left child" or "right child"
(A) ports are only in the root or in the leaves
(C) if a leaf is a port, then its label is "binary white"
pattern without root port and with two leaf ports
pattern with root port and two leaf ports
pattern where the root port is a leaf port

A pattern without any ports is the same as a tree. Define the *arity* of a pattern to be the number of leaf ports.

Patterns are composed as follows. Let t be an n-ary pattern and let $t_1, \ldots, t_n$ be patterns such that for every $i \in \{1, \ldots, n\}$, the root of pattern t_i is a port which has the same local view as the i-th leaf port in t. Then their composition is defined by fusing the corresponding ports – the i-th leaf port of t with the root port of t_i – as in the following picture:



Equivalence. To show that our fixed deterministic tree-walking automaton cannot recognise the separating language, we will find two patterns that are equivalent with respect to the automaton, but which are not equivalent with respect to the language. Define a *port-to-port* run inside a pattern to be a sequence of configurations as depicted in the following picture



the run is cut off as soon as it visits a port

(this might never happen, because the run might accept, reject or loop before visiting another port).

the automaton begins in one of the ports

the behaviour in the ports is well-defined, because a pattern includes the local view of each port

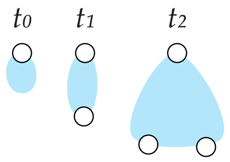
The *type* of a port-to-port run is defined to be the following information: (a) what is the state and port in the source configuration (b) what is the state and port in the target configuration (or accept / reject / loop if the run does not reach any port). In the type, we do not store the actual port node, only its number (i.e. is it the root port, leaf port 1, leaf port 2, etc.). In particular, the number of types of runs is finite once the arity of a pattern is fixed. Two patterns are called *equivalent* if: they have the same local view in the root (or the root is not a port in both patterns), they have the same number of leaf ports with the same respective local views, and they have the same types of port-to-port runs. Pattern equivalence is a congruence, i.e. composing equivalent patterns gives equivalent results.
We now move to the first main ingredient in the proof. If T is a set of patterns, then we write T^+ for the least set of patterns that contains T and is closed under pattern composition.

Lemma 7.4. *[Homogeneous Pattern Lemma] There exist patterns t_0, t_1, t_2 with the following properties:*

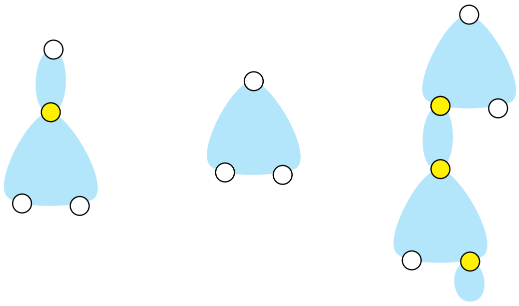
1. *For every $i \in \{0,1,2\}$, t_i does not have black nodes, has a root port and i leaf ports, and has the local view* *in all ports.*

2. *For every $i \in \{0,1,2\}$, all i-ary patterns in $\{t_0, t_1, t_2\}^+$ are equivalent.*

We draw the patterns from the lemma like this:



with the white circles standing for ports. The point of all ports having the same local view ⅄ is that the patterns can be freely composed. We use the name *homogeneous pattern* for patterns in $\{t_0, t_1, t_2\}^+$. In this terminology, item 2 of the lemma says that all homogeneous patterns of fixed arity $i \in \{0, 1, 2\}$ are equivalent. For example, all of the following homogeneous binary patterns are equivalent, because they have arity 2 (we adopt the convention that ports are drawn as white, and ports connecting patterns are drawn in colours like yellow or red, although they represent nodes with local view ⅄):



When proving the Homogeneous Pattern Lemma, we use the following result about finite semigroups. Recall that a semigroup is a set with an associative product operation.

Lemma 7.5 (Semigroup Lemma)**.** *Let S be a finite semigroup with elements a, b. There exist elements $x \in aS$ and $y \in bS$ which satisfy*

$$x = xx = xy.$$

Proof. The key result is the following well-known observation on finite semigroups: there is an *idempotent power*, i.e. a number $\omega \in \{1, 2, \ldots\}$ such that every element satisfies $s^\omega = s^\omega s^\omega$. More specifically, we take ω to be the factorial of the size of the semigroup. We now show that

$$s^\omega = s^\omega s^\omega \qquad \text{for every } s \in S.$$

Let then $s \in S$. By considering the first repetition in the sequence $s^1, s^2, \ldots$ we see that there exist numbers i, j, which are at most the size of the semigroup, and such that $s^i = s^{i+1}$. Because $i \leq \omega$ and j divides ω, we get

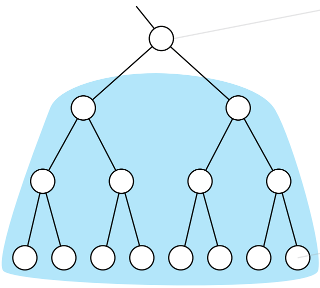
$$s^\omega = s^{\omega+j} = s^{\omega+2j} = \cdots = s^{\omega+\omega} = s^\omega s^\omega,$$

thus proving that ω is an idempotent power. To prove the lemma, define

$$y \overset{\text{def}}{=} b^\omega \qquad x \overset{\text{def}}{=} (ay)^\omega$$

In particular, both x and y are idempotents, because each is obtained by taking some element to the idempotent power. This establishes $x = xx$. Furthermore, since x ends with the idempotent y, we get $x = xy$. ■

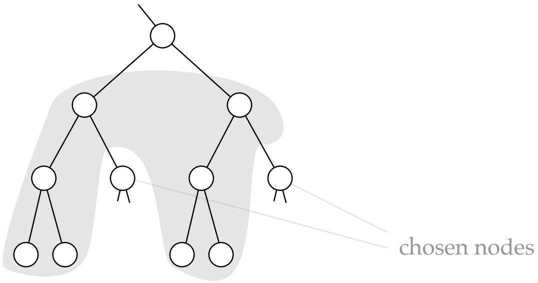
Proof of the Homogeneous Patterns Lemma. For $n \in \{1, 2, \ldots\}$, define s_n to be the pattern depicted in the following picture:



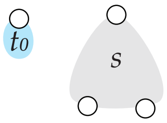
the root is the only port and it has local view

all leaves are white and have depth n

Because equivalence on patterns has finitely many equivalence classes, there must be some numbers $n < N$ such that s_n and s_N are equivalent. Define t_0, the first of the homogeneous patterns from the statement of the lemma, to be s_n. Choose distinct nodes in the pattern s_N which are right children and have subtrees of depth n. Define s to be the binary pattern obtained from s_N by putting a leaf port in each of these chosen nodes, here is a picture of s for $n = 2$ and $N = 4$:

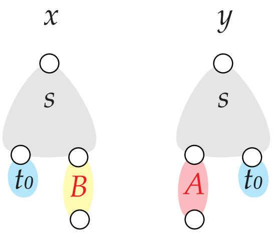


For the rest of the proof, we will draw the patterns t_0 and s like this:

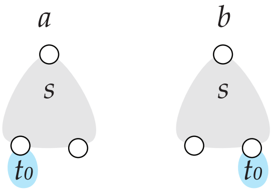


By definition, $s[t_0, t_0] = s_N$, which is equivalent to $s_n = t_0$. By induction, this generalises to the fact that all rank 0 patterns in $\{s, t_0\}^+$ are equivalent.
In the following claim, we use multiplicative notation for composition of unary patterns.

Claim 7.6. *There are unary patterns $A, B \in \{s, t_0\}^+$ such that $x = xx = xy$ holds for*

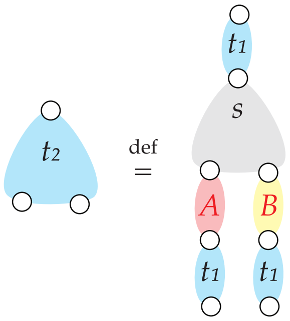


Proof. Apply the Semigroup Lemma to the semigroup of unary patterns (modulo pattern equivalence) generated by the following patterns:

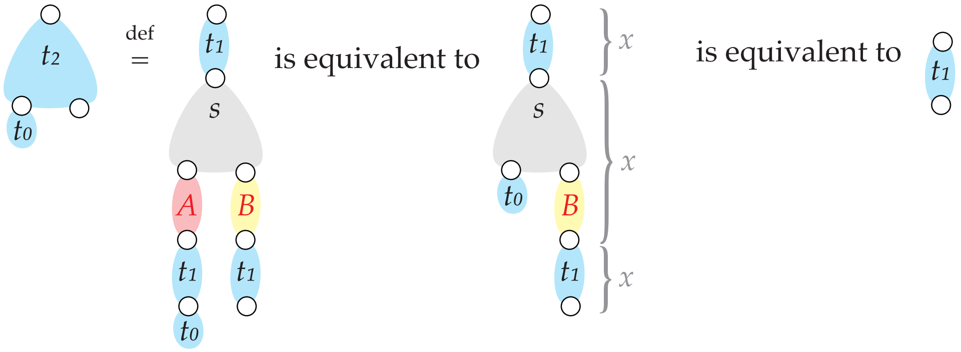


■

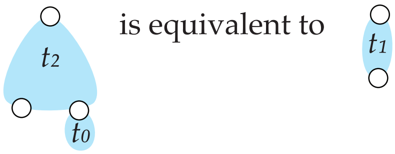
Let A, B and x, y be as in the above claim. Define t_1 to be x and define



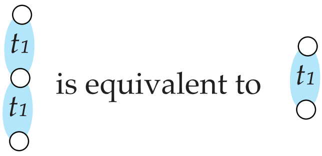
This completes the definition of the patterns t_0, t_1, t_2. From the claim, we get the following equivalence:



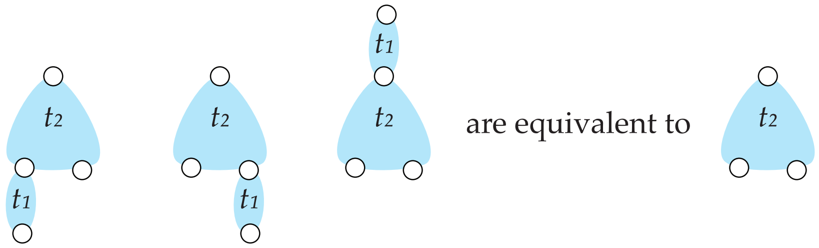
A symmetric proof also gives the following equivalence:



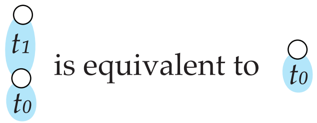
Directly from the claim, t_1 is idempotent in the following sense



Since t_2 has t_1 attached to each of its ports, idempotence of t_1 implies that:



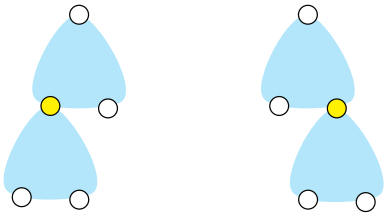
Because t_1 is from $\{s, t_0\}^+$, we get that



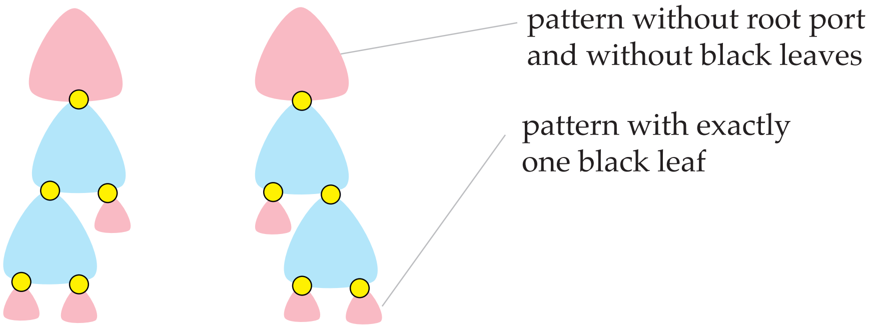
Using the above equivalences, and induction on the size of a pattern, one shows that every homogeneous patterns of arity $i \in \{0, 1, 2\}$ is equivalent to t_i. ■

The Homogeneous Pattern Lemma would also work for nondeterministic tree-walking automata, and even tree automata as in Chapter 5 ones under a suitable notion of equivalence. In contrast, the following lemma crucially depends on determinism (actually, a stronger result is true, namely every two homogeneous patterns of same arity are equivalent, see Exercise 79).

Lemma 7.7 (Rotation Lemma). *The following two patterns are equivalent:*



The lemma immediately implies the lower bound from Theorem 7.2, i.e. that a deterministic tree-walking automaton cannot recognise the separating language, thus finishing the proof of Theorem 7.2. Indeed, take the two patterns in the Rotation Lemma, and put them into the following environment:

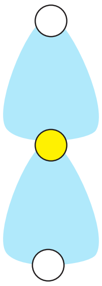


The tree on the left should be accepted and the tree on the right should be rejected, but the automaton will behave the same way on both trees by the Rotation Lemma. It remains to prove the Rotation Lemma.

7.2 *Proof of the rotation lemma*

In this section, we prove the Rotation Lemma. The proof uses a detailed analysis of what a deterministic tree-walking automaton can do in a homogeneous pattern. The bottom line is that the most interesting behaviour that it can do is a depth-first search.

Closure of a state. For a state q, consider the run of the automaton which begins in state q in the yellow node below:



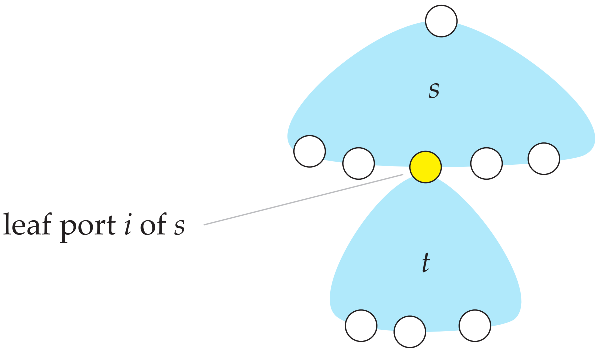
and which is cut off at the first visit to a port node (white in the picture). Define the *closure* of q, denoted by $\overline{q}$, to be the following information:

- *if the run reaches a port*: the state of the last visit in the yellow node;

- *if the run does not reach a port*: does it accept / reject / loop.

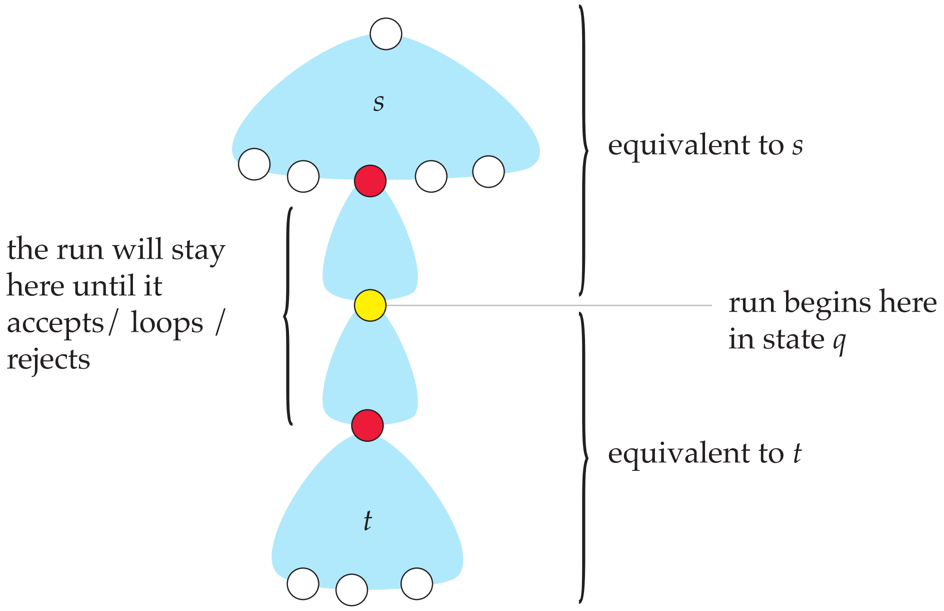
We might have $q = \overline{q}$ if the run goes directly to from the yellow node to some port, without every seeing the yellow node again. The definition of closure is based on the behaviour of the automaton on the interface between two copies of t_1. The following lemma shows that the same behaviour will be witnessed on the interface between any two homogeneous patterns of nonzero arity.

Lemma 7.8. *Let s, t be homogeneous patterns of nonzero arity, and let i be one of the leaf ports in s. If the automaton begins in state q in the yellow node here:*



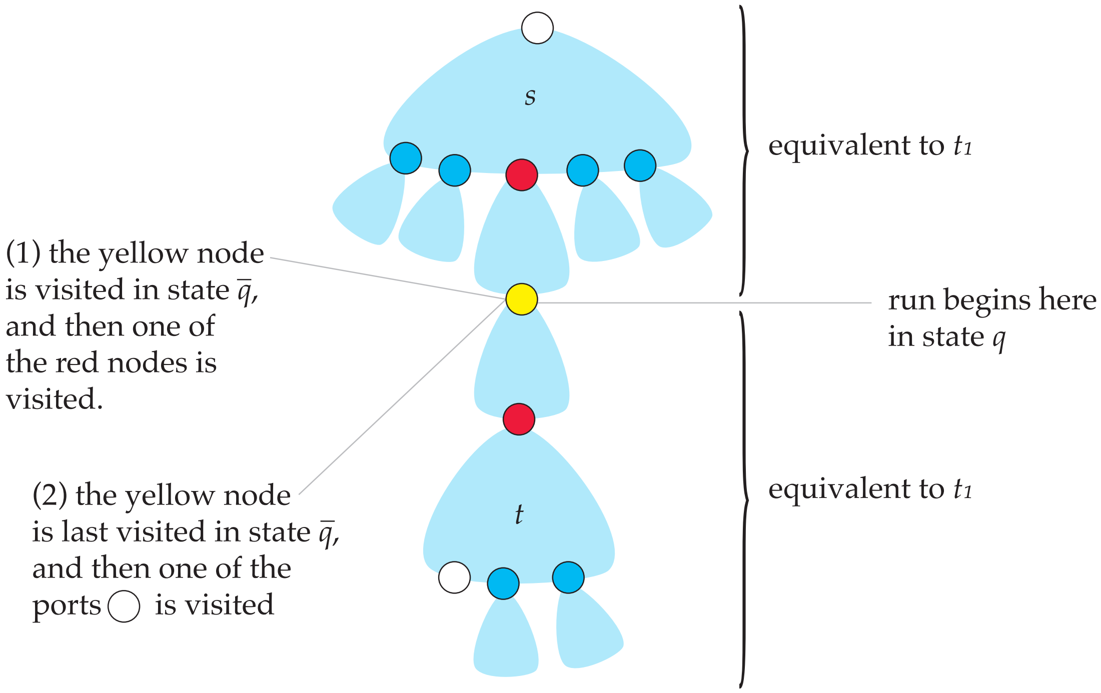
then: (a) if $\overline{q}$ is one of accept / reject / loop then the automaton will do the same without reaching any ports; and (b) if $\overline{q}$ is a state then the automaton will visit the yellow node for the last time in state q and then go to some port.

Proof. Case (a) is illustrated in the following picture



s
equivalent to s
the run will stay here until it accepts / loops / rejects
run begins here in state q
equivalent to t
t

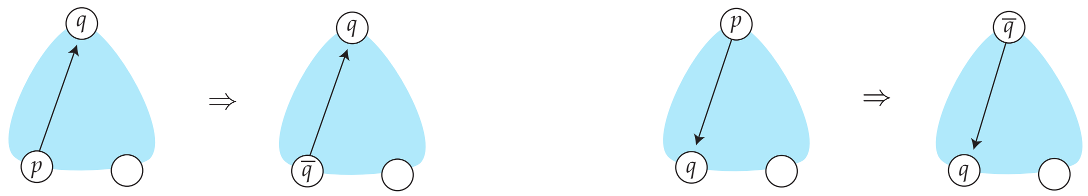
For case (b), suppose that $\overline{q}$ is a state. Consider the following run



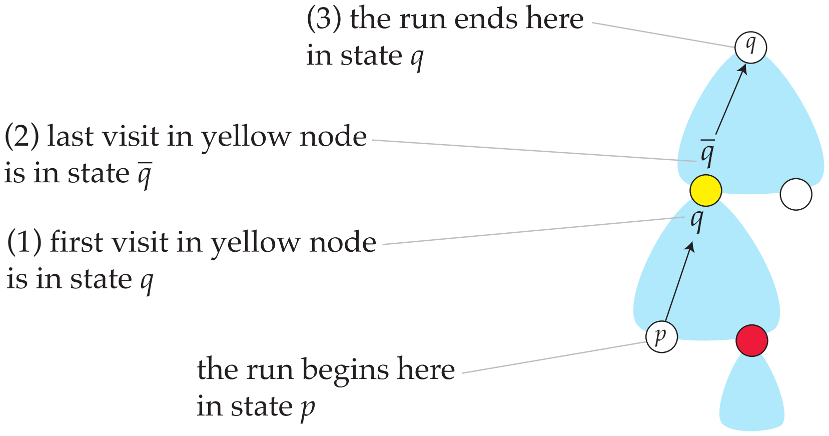
s
equivalent to t_1
(1) the yellow node is visited in state $\overline{q}$, and then one of the red nodes is visited.
run begins here in state q
(2) the yellow node is last visited in state $\overline{q}$, and then one of the ports ○ is visited
t
equivalent to t_1

Item (1) above is by the definition of $\overline{q}$ applied to the part of the pattern between the red nodes. Item (2) above is by the definition of $\overline{q}$ applied to the entire pattern. Consider now the port-to-port run which starts in the yellow node of the pattern from case (a). By item (1), the yellow node will be visited in state $\overline{q}$ before any of the ports are visited. By item (2), the last visit in the yellow node will be in port $\overline{q}$, and then one of the ports will be visited. ■

Lemma 7.9. *For every states p, q we have the following implications (and their symmetric versions with leaf port 2 used instead of leaf port 1):*



Proof. We only prove the first implication, the other ones are proved the same way. Consider the following port-to-port run:



(3) the run ends here in state q

(2) last visit in yellow node is in state $\overline{q}$

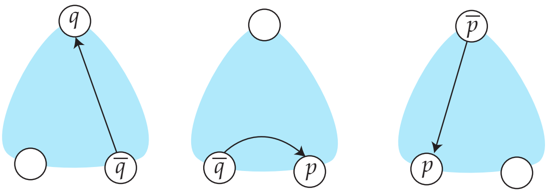
(1) first visit in yellow node is in state q

the run begins here in state p

Item (1) in the picture is the assumption of the implication. Item (2) is by definition of $\overline{q}$ and Lemma 7.8. Item (3) is again the assumption of the implication applied to the entire pattern. The run from (2) to (3) witnesses the conclusion of the implication. ■

Search behaviour. We now turn to the crucial definition in the proof of the Rotation Lemma.

Definition 7.10. *We say that a state q is a left-to-right search if there exists some state p such that the automaton admits the following port-to-port runs:*

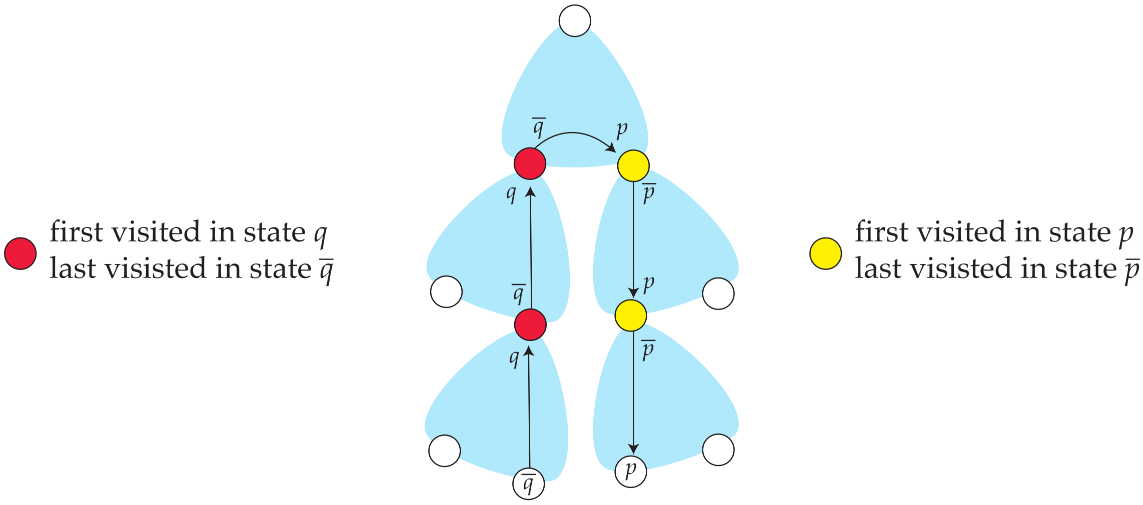


The following straightforward lemma shows that a left-to-right search will visit leaf ports in left-to-right order.

Lemma 7.11. *Assume that a state q is a left-to-right search. Then:*

(*) *if the automaton enters a homogeneous n-ary pattern in state $\overline{q}$ at leaf port $i < n$, then it will exit through the next leaf port $i + 1$.*

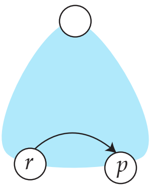
Proof. Here is the run that witnesses (*).


first visited in state q
last visited in state $\overline{q}$
$\overline{q}$
p
q
$\overline{p}$
$\overline{q}$
p
q
$\overline{p}$
$\overline{q}$
p
first visited in state p
last visited in state $\overline{p}$

In the picture above, we use Lemma 7.8 to prove that if a node is first visited in state q, then it is last visited in state $\overline{q}$, likewise for p. ∎

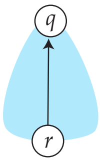
We now state the most technical part of the proof, which says that the conclusion (*) above is also true for any state r which goes from leaf port 1 to leaf port 2 in the pattern t_2.

Lemma 7.12. *Suppose r, p are states which admit the following port-to-port run:*



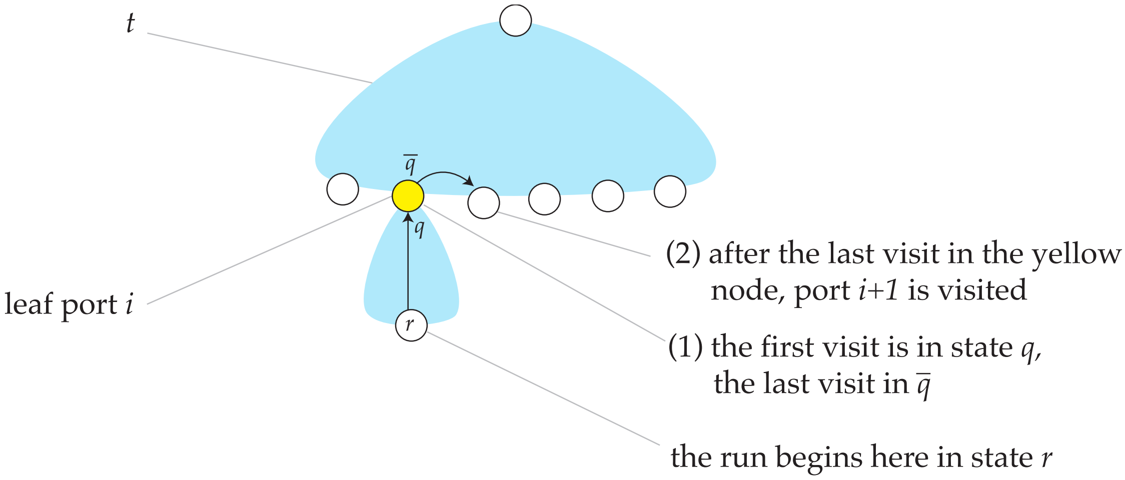
Then the conclusion (*) of Lemma 7.11 is true with r used instead of $\overline{q}$.

Proof. We claim that there is a state q which is a left-to-right search and satisfies



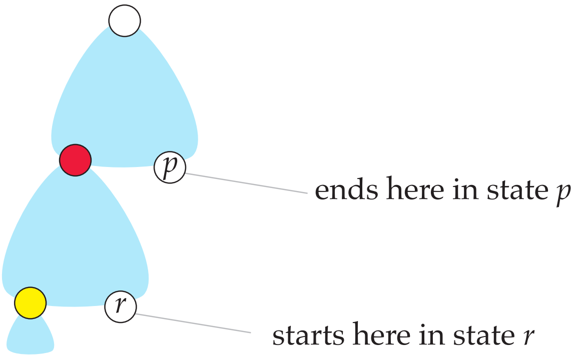
$$(\uparrow)$$

Before proving the claim, we use it to prove the lemma. Let t be an n-ary homogeneous pattern and let $i < n$ be one of the root ports. The following picture shows the run that witnesses (*) in the conclusion of the lemma.


t
$\overline{q}$
q
r
leaf port i
(2) after the last visit in the yellow node, port i+1 is visited
(1) the first visit is in state q, the last visit in $\overline{q}$
the run begins here in state r

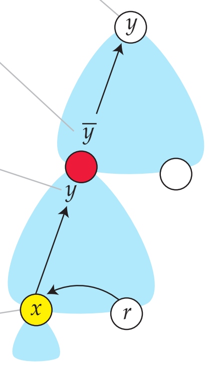
Item (1) is by (↑) and Lemma 7.8, and item (2) is from Lemma 7.11.

The rest of the proof is devoted to proving the claim, i.e. finding a state q which satisfies (↑) and is a left-to-right search. Consider the following port-to-port run, which exists by the assumption of the lemma:



(♣)

The red node must be visited by the above run, since it is on the way from leaf port 1 to leaf port 2. We first claim that in the run above, the yellow node cannot be visited. Otherwise, part of the run would look like this



(4) the root port is visited in state y

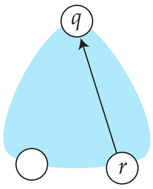
(3) the last visit in the red node is in state $\overline{y}$

(2) y is defined to be the state of the first visit to the red node after (1)

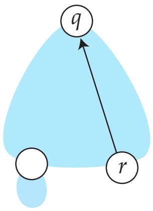
(1) x is defined to be the state of the last visit to the yellow node

Items (3) and (4) are by Lemma 7.9 applied to the part of the run between (1) and (2). Item (4) contradicts the assumption that the first port visited by the run in (♣) is leaf port 2.

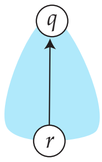
Therefore, the run from (♣) never visits the yellow node. Define q to be the state of the first visit in the red node. Because the red node is visited first, we have



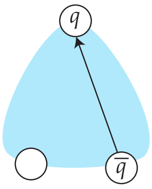
and therefore also



which is equivalent to

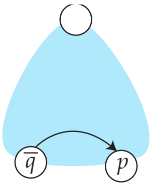


The conclusion above is (↑) from the claim at the beginning of the proof. It remains to prove that q is a left-to-right search. Applying Lemma 7.9 to the leftmost picture above, we get



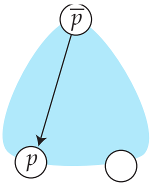
(*)

The last time the red node is visited in the run from (♣) is in state $\overline{q}$. After this visit, the run goes to leaf port 2 in state p, thus proving



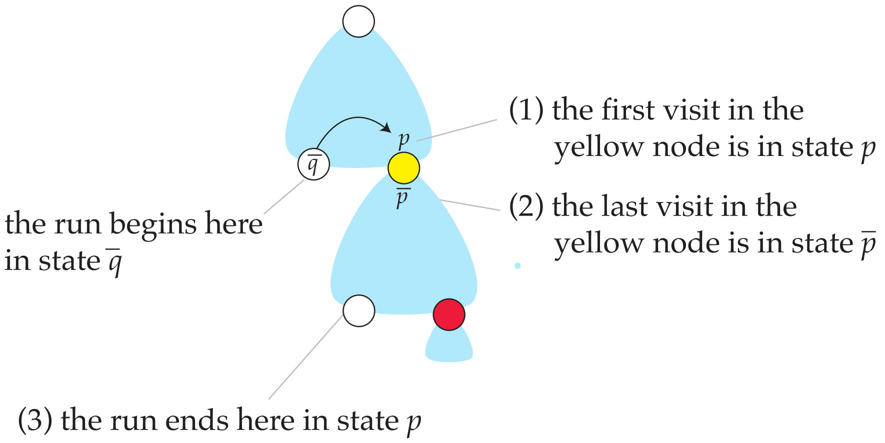
(**)

To complete the proof that q is a left-to-right search, it remains to show:



(***)

Consider the port-to-port run described in the following picture:

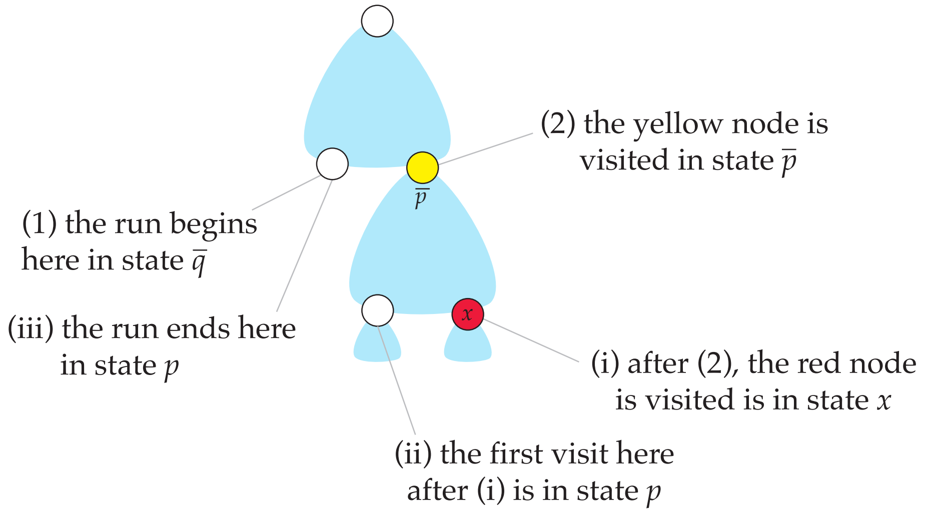

p
$\overline{q}$
$\overline{p}$
(1) the first visit in the yellow node is in state p
(2) the last visit in the yellow node is in state $\overline{p}$
the run begins here in state $\overline{q}$
(3) the run ends here in state p

(♠)

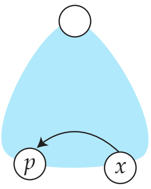
Claim 7.13. *The red node is not visited between configurations (2) and (3).*

The claim establishes (***), which was the last ingredient required to show that state q is a left-to-right search. It remains to show the claim (actually, a finer analysis would show that the red node is not visited at all during the run.)

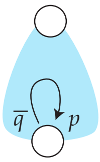
Proof. Toward a contradiction, suppose that the red node is visited between configurations (2) and (3). Let x be the state of the last visit to the red node. Then the automaton would have a run like this:


(2) the yellow node is visited in state $\overline{p}$
$\overline{p}$
(1) the run begins here in state $\overline{q}$
(iii) the run ends here in state p
x
(i) after (2), the red node is visited is in state x
(ii) the first visit here after (i) is in state p

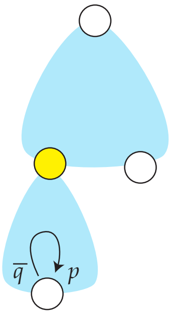
The run from (2) to (i) is taken from the assumption on x. The run from (i) to (ii) is because the run in (♣) goes from the red node to leaf port 2 in state p. In particular, the run from (i) to (ii) implies that



which in turn yields the run from (ii) to (iii). The run from (1) to (iii) shows



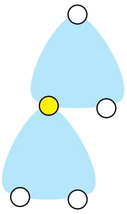
and therefore also

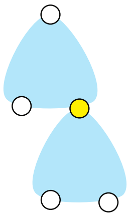


which contradicts (**). ■

■

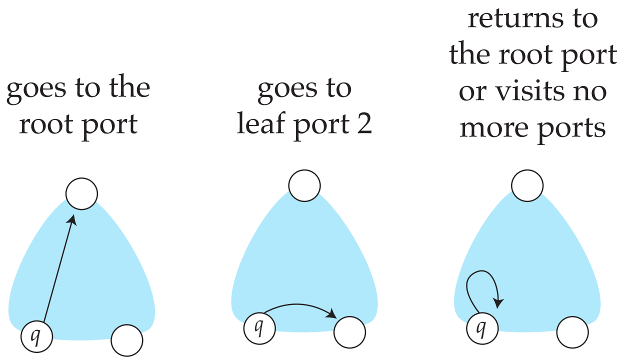
Proof of the Rotation Lemma. To prove the Rotation Lemma, let ρ, ρ' be port-to-port runs in the two patterns



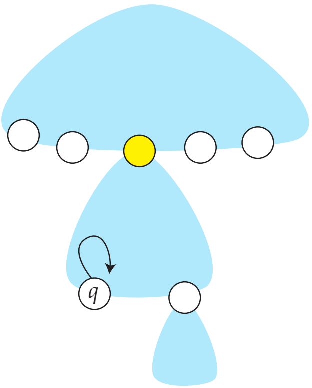


from the statement of the Rotation Lemma, respectively, which have equivalent source configurations. To prove equivalence, we need to show that target configurations are equivalent. Let q be the state in the source configuration of the two runs. We consider two cases depending on the source port.

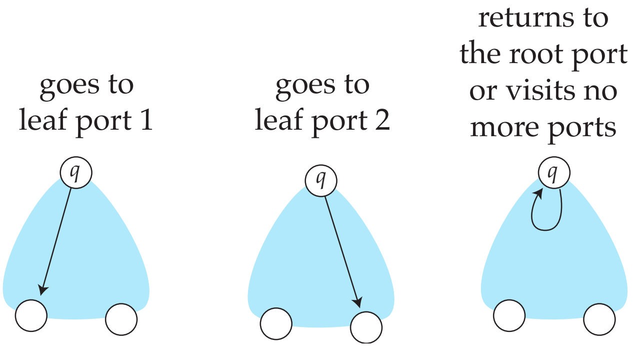
1. *The runs begin in a leaf port.* The cases of leaf port 1 and 3 are symmetric, see we ignore the case of leaf port 3. Suppose that the automaton starts in leaf port 1 or 2. Let us see what happens in the pattern t_2 if we start in leaf port 1 in state q. There are three cases to consider:

   In the first case, we use Lemma 7.9 to show that both ρ and ρ' end in the root port (and in the same state). In the second case, we use Lemma 7.11 to show that both ρ and ρ' end in leaf port $i+1$ (and in the same state). In the third case, we conclude that for every homogeneous pattern, if the automaton begins in state q in a leaf port, then it will return to the same port (in some fixed state) or never visit any other ports (and have the same behaviour among accept / reject / loop). Here is the picture:



2. *The runs begin in the root port.* Consider three cases for what happens if the automaton starts in the roof of t_2 in state q:



For the first case, we use Lemma 7.9 to prove that both ρ and ρ' will end in leaf port 1. A symmetric reasoning is applied in the second case, with the target being leaf port 3. The third case is dealt with the same way as in the previous item.

■

Problem 73. Consider trees over an alphabet Σ containing two letters $0, 1$ of arity zero, one letter $\neg$ of arity one and two letters $\vee, \wedge$ of arity two. Let T be the trees over the above alphabet which evaluate to value 1 under the standard semantics of boolean expressions. Show a tree-walking automaton that recognizes T.

Problem 74. Show that every language recognized by a two-way automaton on finite words is regular.

Problem 75. Show that every language recognized by a tree-walking automaton on finite trees is regular, i.e. recognised by a deterministic (branching) bottom-up tree automaton.

Problem 76. Show that every language recognized by a deterministic tree-walking automaton is also recognized by some deterministic tree-walking automaton that never loops.

Problem 77. Following [34]. Consider a model of tree-walking automata where the automaton sees only the label and whether or not the node is a root or leaf, but it does not see the child number. Show that this model, even in the nondeterministic variant, cannot recognise the language "every leaf has label a".

Problem 78. (Answer unknown) Prove or disprove: for every deterministic top-down branching tree automaton, there is a deterministic tree-walking automaton that recognises the same language.

Problem 79. Show the following generalisation of the Rotation Lemma: every two homogeneous patterns of same arity are equivalent.

Problem 80. Consider a variant of tree-walking automata that can use 1 pebble. The pebble operations are: place the pebble on the current node (assuming it is not already placed anywhere else), pick it up from the current node. The local view includes the information about whether or not the pebble is on the current node. Show that this extension of tree-walking automata can still be simulated by branching tree automata.

Problem 81. Consider an extension of the pebble automaton from the previous exercise, where 2 pebbles are allowed. Show that (a) this extension of tree-walking automata can still be simulated by branching tree automata if we keep a stack discipline (i.e. if pebble 2 is present in the tree, then any actions on pebble 1 are disallowed); (b) if stack discipline is lifted, then the model cannot be simulated by tree automata and in fact has undecidable emptiness.

8
Weighted automata over a field

This chapter is about automata which input words and output rational numbers. The original definition comes from Schützenberger [49]. We show that these automata can be minimised (even in polynomial time) and can be tested for equivalence (again, in polynomial time), but the following version of the emptiness problem is undecidable:

is the output 0 for at least one input?

Note that the dual problem,

is the output 0 for all inputs?

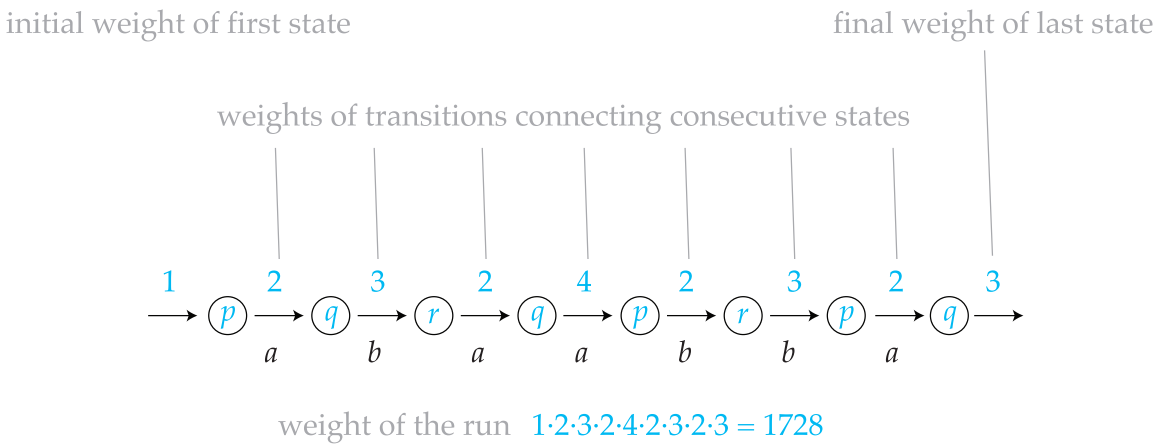
is a special case of the equivalence problem, and is therefore decidable in polynomial time. We use the field of rational numbers, but most results would work for other fields. The automata can be viewed in two ways: as a nondeterministic device with states from a finite set (we call these weighted automata) and as a deterministic device with states from a vector space (we call these vector space automata). Both views are useful, so we present both of them.

Weighted automata. In the nondeterministic view, the automaton has a state space which is a finite set, and many possible runs. Each run has an associated weight, and the weight of an input word is the sum of weights of all the runs.

Definition 8.1 (Weighted automaton)**.** *A* weighted automaton *consists of:*

1. *a finite set Σ, called the* input alphabet*;*
2. *a finite set P of* states*;*
3. *for each state, an* initial weight *and* final weight*, which are rational numbers;*
4. *a* transition function *from $P \times \Sigma \times P$ to rational numbers.*

Define the weight *of a run of the automaton to be the product of: the initial weight of the first state, the weights of all transitions used, and the final weight of the last state, as in the following picture:*


initial weight of first state
final weight of last state
weights of transitions connecting consecutive states
1
2
3
2
4
2
3
2
3
p
q
r
q
p
r
p
q
a
b
a
a
b
b
a
weight of the run 1·2·3·2·4·2·3·2·3 = 1728

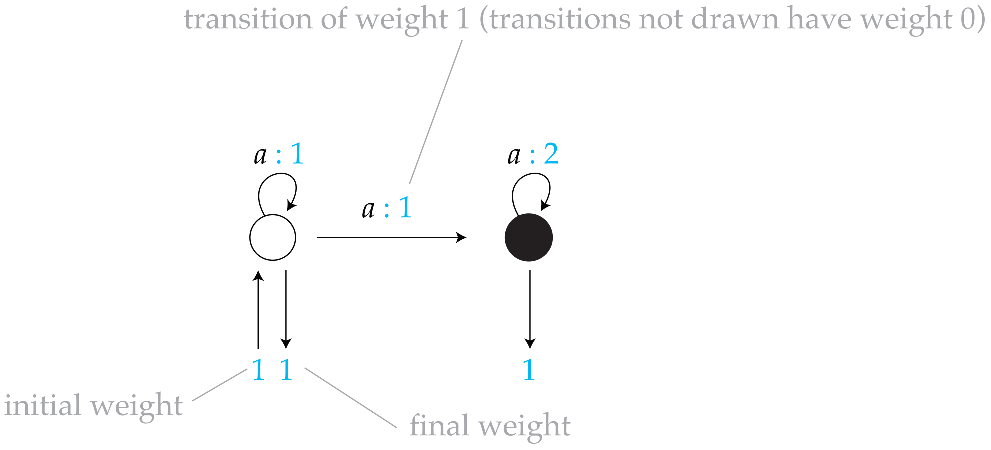
Define the weight *of a word to be the sum of the weights of all runs. The function* recognised *by the automaton is the function that maps a word to its weight.*

The above definition makes sense for an arbitrary semiring, i.e. a set equipped with product and sum operations, such that sum is commutative and there is an appropriate distributivity law. If we take the semiring

$$\Big(\underbrace{\{0,1,2,\ldots,\infty\}}_{\text{universe of the semiring}} \;,\; \underbrace{\min}_{\text{sum of the semiring}} \;,\; \underbrace{+}_{\text{product of the semiring}} \Big)$$

then we recover distance automata as discussed in Chapter 4. For this chapter, however, it will be important that we use the rational numbers, or more generally a field, so that we can use linear algebra.

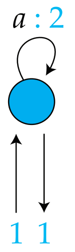
Example 9. Consider the following weighted automaton with three states.


transition of weight 1 (transitions not drawn have weight 0)
a : 1
a : 1
a : 2
1 1
1
initial weight
final weight

The weight of a run that stays in ○ is 1, and the weight of a run that goes from ○ to ● is 2^n, where n is the number of times the run loops around ●. Other runs have cost zero. If the input word has length n, then the weight of the word is

$$\underbrace{2^{n-1} + 2^{n-1} + \cdots + 2^1}_{\text{runs from } \bigcirc \text{ to } \bullet} + \underbrace{1}_{\text{loop in } \bigcirc} = 2^n$$

To recognise the same function $a^n \mapsto 2^n$ we could also use this automaton



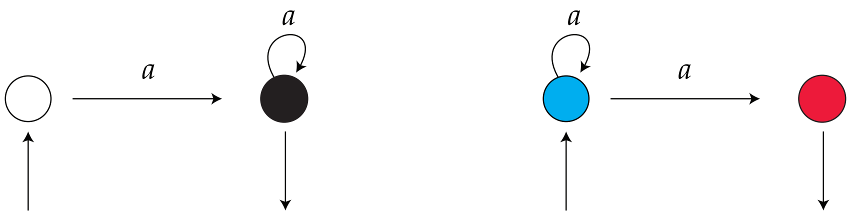
As we will see later in this chapter, weighted automata can be minimised. The second automaton is in fact the minimal automaton – how could it be smaller?

– and there exists an automaton homomorphism (see later in the chapter for the definition) from the first automaton to the second one, namely the function

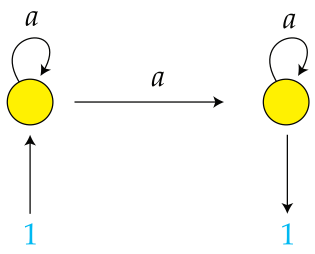
$$x \cdot \circ + y \cdot \bullet \qquad \mapsto \qquad (x+y) \cdot \bullet.$$

□

Example 10. [Running example] We describe two weighted automata which will be used as the running example in this chapter. We can view a nondeterministic automaton as a special case of a weighted automaton, by assuming that every arrow (including dangling arrows that indicate initial and final states) has weight 1. In this view, the semantics of weighted automata will map an input word to the number of accepting runs. Consider the following two nondeterministic automata over input alphabet $\{a\}$



which both recognise the language "nonempty words". Both automata are unambiguous, i.e. on each accepted word they have exactly one run. Therefore, if we treat the automata as weighted automata, then the recognised function will be the characteristic function of the set of nonempty words. Note that the semantics of weighted automata is finer, i.e. leads to more non-equivalent automata, than the standard nondeterministic semantics. For example, the automaton



also recognises – as a nondeterministic automaton – the set of nonempty words, but it has n runs on inputs of length n, and therefore it is not equivalent to the unambiguous automata when seen as a weighted automaton. We will return to the unambiguous automata later in the chapter, and show that they are isomorphic as weighted automata. □

Vector space automata. We now present a deterministic view on weighted automata. In this view, the automaton has a state space that is a vector space, and each letter deterministically updates the state using a linear function. This definition is almost the same as the original definition of Schützenberger [49, Definition 1], except that the original definition also allowed control states from a finite set. We do not use control states, because they do not contribute to expressive power of the model (although they make constructions easier), see the proof of Lemma 8.12.

Definition 8.2 (Vector space automaton). *A* vector space automaton[1] *consists of:*

1. *an input alphabet, which is a finite set Σ;*
2. *a set Q of states, which is a vector space of finite dimension over $\mathbb{Q}$;*
3. *an initial state $q_0 \in Q$;*
4. *for each letter $a \in \Sigma$, a linear map from Q to itself, denoted by $q \mapsto qa$;*
5. *a linear map from Q to the rational numbers, called the* output function*.*

The automaton begins in the initial state, and when reading a letter $a \in \Sigma$, it updates its state using the transition function from item 4. After reading all the letters of the input word, the output function is applied to the last state, yielding the output of the automaton.

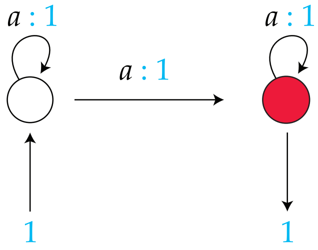
[1] This definition is designed so that it can be generalised to categories other than the category of vector spaces, see e.g. [21]

The state space of a vector space automaton is isomorphic to $\mathbb{Q}^n$ for some $n \in \mathbb{N}$, since these are the vector spaces of finite dimension. Therefore, when representing a vector space automaton for the use of algorithms, we simply indicate the dimension n, and use matrices to represent the transitions from item 4 and the output function from item 5.

Example 11. Without increasing the expressive power of the model, we could allow affine functions in the transitions of a vector space automaton. The construction is illustrated on the following example.
Consider the length function over a one letter alphabet $\{a\}$. The most natural approach to recognise this function would be to have the one dimensional vector space $\mathbb{Q}$ as the state space and use the affine function $q \mapsto q+1$ as the transition function. However, Definition 8.2 requires linear transition functions, so we use a workaround. The state space is $\mathbb{Q}^2$ and the initial state is $(1,0)$. When reading a letter a, the automaton applies the function

$$(x,y) \mapsto (x, x+y)$$

and the output function is $(x,y) \mapsto y$. As an alternative to the above vector space automaton, we can use the following weighted automaton



If we ignore the weights, the above picture shows a nondeterministic automaton, which has exactly n accepting runs on a word of the form a^n, and thus using the weighted semantics, we get the length function. (This is the weighted automaton at the end of Example 10.) □

Equivalence of the models. A closer inspection of the vector space automaton and the weighted automaton used in Example 11 shows that these are actually the same automaton, only drawn using different pictures. This sameness is formalised in the following lemma.

Lemma 8.3. *Weighted automata and vector space automata recognise the same functions.*

Proof. Actually, the proof shows something more, namely that the two definitions of automata are just different syntaxes for the same object. To transform between syntaxes, we use the transformation

$$\mathcal{A} \in \text{weighted automata} \qquad \mapsto \qquad \mathsf{vec}\mathcal{A} \in \text{vector space automata},$$

described below, which preserves the recognised function. The transformation is easily seen to be reversible, thus proving the lemma.
The vector space automaton $\mathsf{vec}\mathcal{A}$ is defined as follows.

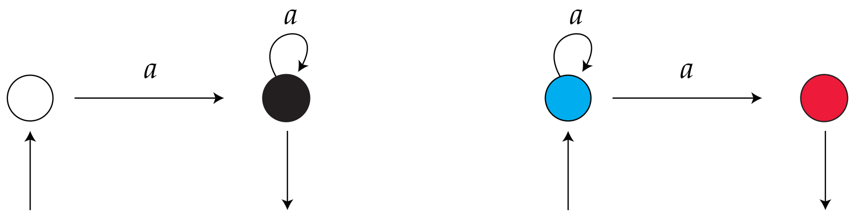
- The state space of $\mathsf{vec}\mathcal{A}$ is $\mathbb{Q}^P$, where P is the states of $\mathcal{A}$.
- The initial state of $\mathsf{vec}\mathcal{A}$ assigns to each state its initial weight.
- The output function of $\mathsf{vec}\mathcal{A}$ multiplies each coordinate by its final weight.
- For each input letter $a \in \Sigma$, the state update $q \mapsto qa$ of $\mathsf{vec}\mathcal{A}$ maps a vector q to a vector which stores the following number on coordinate $p \in P$:

  $$\sum_{r \in P} (\text{coordinate } r \text{ of } q) \cdot (\text{weight of transition } r \xrightarrow{a} p \text{ in } \mathcal{A}).$$

  An alternative view is that the linear map above is described by the matrix which is obtained by looking at the weights of transitions that read letter a in the automaton $\mathcal{A}$.

■

Example 12. [Running example] Recall the two weighted automata:



We now show the corresponding vector space automata. The state spaces of the automata are 2-dimensional vector spaces with bases $\{\bigcirc, \bullet\}$ and $\{\text{🔵}, \text{🔴}\}$, respectively. The initial vectors are ○ and 🔵, respectively, while the transition functions are

$$(x \cdot \bigcirc + y \cdot \bullet) \cdot a = (x+y) \cdot \bullet \qquad (x \cdot \text{🔵} + y \cdot \text{🔴}) \cdot a = x \cdot \text{🔵} + x \cdot \text{🔴}.$$

The output function in the first automaton is projection to coordinate ● and the output function in the second automaton is projection to coordinate 🔴. □

8.1 *Minimisation of weighted automata*

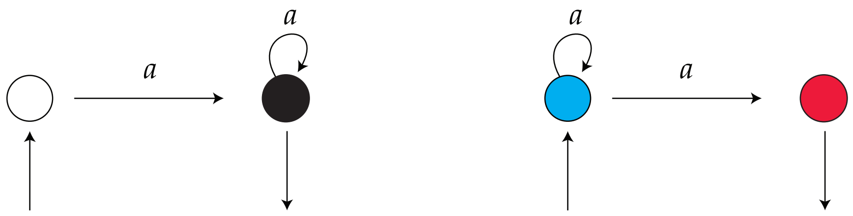
In this section, we prove a Myhill-Nerode style theorem on the existence of a minimal automaton, which is unique up to isomorphism (although the notion of isomorphism is a bit more involved than usual).

Homomorphisms of weighted automata. Let $\mathcal{A}$ and $\mathcal{B}$ be vector space automata over the same input alphabet Σ. A *homomorphism* from $\mathcal{A}$ to $\mathcal{B}$ is defined to be a linear map from the states of $\mathcal{A}$ to the states of $\mathcal{B}$ which is consistent with the structure of the automata, in the following sense:

$$\underbrace{\text{initial state}}_{\mathcal{B}} = \underbrace{h(\text{initial state})}_{\mathcal{A}} \qquad (h(q))\underset{\mathcal{B}}{\cdot}a = h(q\underset{\mathcal{A}}{\cdot}a) \qquad \underbrace{\text{output}}_{\mathcal{B}}(h(q)) = \underbrace{\text{output}(q)}_{\mathcal{A}}$$

If there is such a homomorphism, then the functions computed by the two automata are clearly the same. An *isomorphism* is a homomorphism which has an inverse that is also a homomorphism. If a homomorphism is surjective, as a function on state spaces, and the dimensions of the state spaces are the same, then it is an isomorphism. This is because on vector spaces of finite dimension, a surjective dimension preserving linear map has a linear inverse.

Example 13. [Running example] Recall these weighted automata



and their corresponding representations as vector space automata. We present a homomorphism, in fact an isomorphism, from the first automaton to the second automaton (as vector space automata). This is the function h defined by

$$x \cdot \circ + y \cdot \bullet \qquad \mapsto \qquad (x+y) \cdot \text{🔵} + y \cdot \text{🔴}.$$

Note that h is an isomorphism between the two state spaces. Clearly h maps the initial state ○ of the first automaton to the initial state 🔵 of the second automaton. The following diagram shows that h is consistent with the transition functions:

$$\begin{tikzcd}
x \cdot \circ + y \cdot \bullet \arrow[r, "q \mapsto qa \text{ in left automaton}"] \arrow[d, "h"'] & (x+y) \cdot \bullet \arrow[d, "h"] \\
(x+y) \cdot \text{🔵} + y \cdot \text{🔴} \arrow[r, "q \mapsto qa \text{ in right automaton}"'] & (x+y) \cdot \text{🔵} + (x+y) \cdot \text{🔴}
\end{tikzcd}$$

The following diagram shows that h is consistent with the output functions:

$$\begin{tikzcd}
x \cdot \circ + y \cdot \bullet \arrow[d, "h"'] \arrow[dr, "\text{output in left automaton}"] & \\
(x+y) \cdot \text{🔵} + y \cdot \text{🔴} \arrow[r, "\text{output in right automaton}"'] & y
\end{tikzcd}$$

We have thus shown that h is a homomorphism. Since it was an isomorphism of vector spaces, its inverse is also a homomorphism of automata (the diagrams are easily seen to invert), and therefore the two automata are isomorphic as vector space automata. □

We now state the minimisation theorem. Call a vector space automaton *reachable* if every state in its state space is a finite linear combination of reachable states, i.e. states that can be reached from the initial state by reading some input word.

Theorem 8.4. *Let $f : \Sigma^* \to \mathbb{Q}$ be a function recognised by a vector space automaton. There exists a vector space automaton, called the minimal automaton of f, which recognises f and such that every reachable vector space automaton recognising f admits a homomorphism into the minimal automaton.*

Proof. The proof is essentially the same as for the classical Myhill-Nerode theorem. Actually, the theorem remains true for vector space automata that can use infinite dimensional vector spaces as states.

The set of functions $\Sigma^* \to \mathbb{Q}$ can be viewed as an infinite dimensional vector space, with functions seen as vectors indexed by input words. There is a natural right action of words $w \in \Sigma^*$ on this vector space, defined by

$$q : \Sigma^* \to \mathbb{Q} \quad \mapsto \quad qw : \Sigma^* \to \mathbb{Q} \qquad \text{where } qw \text{ is defined by } v \mapsto q(wv).$$

For every word w, the map $q \mapsto qw$ is linear, because it simply rearranges the coordinates of q when seen as a vector.

Let f be a function as in the statement of the theorem. Define the minimal automaton of f as follows. The state space, which is a subspace of the infinite dimensional space $\Sigma^* \to \mathbb{Q}$, is all finite linear combinations of functions of the form fw for $w \in \Sigma^*$. The initial state is f. The transition function is defined using the right action $q \mapsto qa$ defined above. The output function takes a state q to its value on the empty word. The automaton clearly recognises the function f. We will justify below why the state space has finite dimension, and therefore the automaton is indeed a vector space automaton as per Definition 8.2.

We now show that every reachable vector space automaton recognising f admits a surjective homomorphism onto the minimal automaton defined above.

Let $\mathcal{A}$ be be a vector space automaton recognising f. For a state q of $\mathcal{A}$, define $[q] : \Sigma^* \to \mathbb{Q}$ to be the function recognised by the vector space automaton obtained from $\mathcal{A}$ by changing its initial state to q.

Claim 8.5. *The function* $[\_]$ *is a surjective homomorphism from* $\mathcal{A}$ *onto the minimal automaton.*

Proof. The function $[\_]$ is a linear map from states of $\mathcal{A}$ to the vector space $\Sigma^* \to \mathbb{Q}$, because the state update and output functions in $\mathcal{A}$ are linear functions. Note that the state space of the minimal automaton is not all of $\Sigma^* \to \mathbb{Q}$, but only a subspace, so we still need to show that $[\_]$ has its state space contained in that subspace. The function $[\_]$ is compatible with transitions, i.e.

$$\underbrace{[qa]}_{\text{transition in } \mathcal{A}} \qquad = \qquad \underbrace{[q]a}_{\text{transition in the minimal automaton}}.$$

Indeed, the left side describes the function: "what $\mathcal{A}$ will do if it starts in state qa and reads a word w", while the right side describes the function "what $\mathcal{A}$ will do if it starts in state q and reads a word aw". The initial state of $\mathcal{A}$, call it q_0, is mapped by $[\_]$ to the function f recognised by the automaton $\mathcal{A}$. It follows that

$$[q_0 w] = fw \qquad \text{for every } w \in \Sigma^*.$$

By the above and the assumption on $\mathcal{A}$ being reachable, it follows that the image of $[\_]$ consists of linear combinations of functions of the form fw, and therefore the image of $[\_]$ is contained in – in fact, equal to – the state space of the minimal automaton. Finally, the function $[\_]$ is compatible with the output functions of the automata, because the value of the output function of $\mathcal{A}$ on state q is the same as $[q](\epsilon)$. ■

A surjective linear map cannot increase the dimension of a vector space, and therefore the above claim also implies that the minimal automaton has a finite dimensional state space, assuming that f was recognised by a vector space automaton with a finite dimensional state space ■

Example 14. [Running example] The function recognised by the automata in the running example is the characteristic function of the set of nonempty words. This function is not recognised by any vector space automaton with a one dimensional state space (equivalently, by any weighted automaton with one state) because if the state space has one dimension, then the recognised function is of the form

$$a^n \mapsto \lambda_0 \cdot \lambda^n, \qquad \text{for some } \lambda_0, \lambda \in \mathbb{Q}$$

which is not the case for the characteristic function of nonempty words. Therefore, dimension ≥ 2 is necessary to recognise the function from the running example, and thus each of the two automata in the running example is a minimal automaton. □

So far, we have only proved that a minimal automaton exists. In the next section, we show that it can also be efficiently computed.

8.2 Algorithms for equivalence and minimisation

In this section we give polynomial time algorithms for equivalence and minimisation of vector space automata. We use the following lemma to implement operations of vector spaces.

Lemma 8.6. *Assume that rational numbers are represented in binary notation, linear subspaces of $\mathbb{Q}^d$ are represented using a basis, and linear maps are represented using matrices. The following operations on linear subspaces can be done in polynomial time: (a) test for inclusion, (b) compute the subspace spanned by a union of two subspaces, (c) compute the image under a linear map.*

Equivalence. We begin with a simple algorithm for finite vector space automata: computing linear combinations of reachable states. Computing the actual reachable states, and not their linear combinations, is a different story and leads to undecidability, as we will see in Section 8.3. To compute linear combinations of reachable states we use a simple saturation procedure. We begin with $Q_0 \subseteq Q$ being the vector space spanned by the singleton of the initial state, i.e. this is the one dimensional vector space whose basis is the initial state. Then, assuming that a vector space $Q_i \subseteq Q$ has already been defined, we define Q_{i+1} to be the vector space spanned by

$$Q_i \cup \bigcup_{a \in \Sigma} Q_i \cdot a.$$

A representation of Q_{i+1} can be computed in polynomial time from a representation of Q_i, using the toolkit from Lemma 8.6. We also use the following observation: the coefficients in the basis for $Q_i \cdot a$ can only grow, as compared with the coefficients for Q_i, by a constant amount depending on the linear map $q \mapsto qa$. This way we get a growing chain of linear subspaces

$$Q_1 \subseteq Q_2 \subseteq \cdots \subseteq Q.$$

Since the dimension cannot grow indefinitely, this sequence must stabilise after a number of iterations that is at most the dimension of Q, and this point is the set of reachable states.
Here is a corollary of the reachability algorithm described above.

Theorem 8.7. *The following problem is in polynomial time:*

- **Input.** *Two vector space automata $\mathcal{A}, \mathcal{B}$.*
- **Question.** *Do they compute the same function $\Sigma^* \to \mathbb{Q}$?*

Proof. Using a product construction, compute a vector space automaton which computes the function $\mathcal{A} - \mathcal{B}$. In the resulting product automaton, compute the linear combinations of reachable states. The automata $\mathcal{A}, \mathcal{B}$ are equivalent if and only if, the function $\mathcal{A} - \mathcal{B}$ is constant zero. The latter can be tested by computing the linear combinations of reachable states in the product automaton, taking the image under the output function, and testing if the result is equal to the zero-dimensional space $\{0\}$. ∎

Computing the minimal automaton. We show that the minimal automaton from Theorem 8.4 can be computed in polynomial time from any vector space automaton recognising the function f.

Theorem 8.8. *The following problem is in polynomial time:*

- **Input.** *A vector space automaton $\mathcal{A}$.*
- **Output.** *The minimal automaton of the function recognised by $\mathcal{A}$.*

Proof. Consider a vector space automaton $\mathcal{A}$ with state space Q. For $n \in \{0, 1, \ldots\}$, define states $q, p \in Q$ to be n-equivalent if for every input word w of length $\leq n$, the states qw and pw have the same values under the output function. This equivalence relation can be seen as a subset of

$$E_n \subseteq Q \times Q.$$

By linearity of the automaton, the subset is linear. We can also compute the equivalence relations as follows. The set E_0 is the inverse image of $\{0\}$ under the linear map

$$(p, q) \mapsto F(p) - F(q)$$

while the set E_{n+1} is the intersection

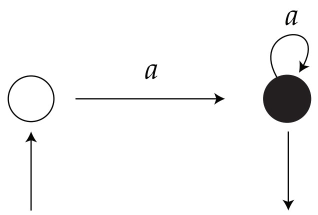
$$E_{n+1} = \bigcap_{a \in \Sigma} (f_a)^{-1}(E_n) \qquad \text{where } f_a \text{ is the linear map } (p, q) \mapsto (pa, qa).$$

We have a sequence of linear subspaces

$$Q \times Q \supseteq E_0 \supseteq E_1 \supseteq E_2 \supseteq \cdots$$

By the same arguments as in the equivalence algorithm, the sequence above must stabilise at some equivalence relation, call it E_*, which can be computed in polynomial time. This stable equivalence relation E_* is the Myhill-Nerode equivalence relation, which identifies states if they produce the same outputs on all inputs. In the terminology of the proof of Theorem 8.4, two states are equivalent under E_* if and only if they have the same image under the function $[\_]$. The quotient of Q under E_* is therefore the minimal automaton; and this quotient can be computed in polynomial time, see Exercise 87. ■

Example 15. [Running Example] To finish the running example, we run the minimisation algorithm on the vector space automaton that corresponds to



The equivalence E_0 identifies two states if they agree on the coordinate ●. The equivalence E_1 identifies two states

$$x \cdot \circ + y \cdot \bullet \qquad \text{and} \qquad x' \cdot \circ + y' \cdot \bullet$$

if they are equivalent with respect to E_0, i.e. $y = y'$, and furthermore applying a to both states gives equivalent results with respect to E_0, i.e.

$$(x + y) \cdot \bullet \qquad \text{and} \qquad (x' + y') \cdot \bullet$$

agree on coordinate ●, which means that they are equal, and therefore also $x = x'$. Summing up, E_1 is the identity equivalence relation, and therefore the automaton is already minimal. □

8.3 Undecidable emptiness

In Theorem 8.7, we showed that equivalence of vector space automata (and therefore also of weighted automata) is decidable in polynomial time. A corollary is that one can decide if a weighted automaton maps all inputs to zero. We now show that a dual problem, namely mapping some word to zero, is undecidable. For the undecidability proof, it will be more convenient to use the syntax of weighted automata and not that of vector space automata.

Theorem 8.9. *The following problem is undecidable:*

- **Input.** *A weighted automaton.*

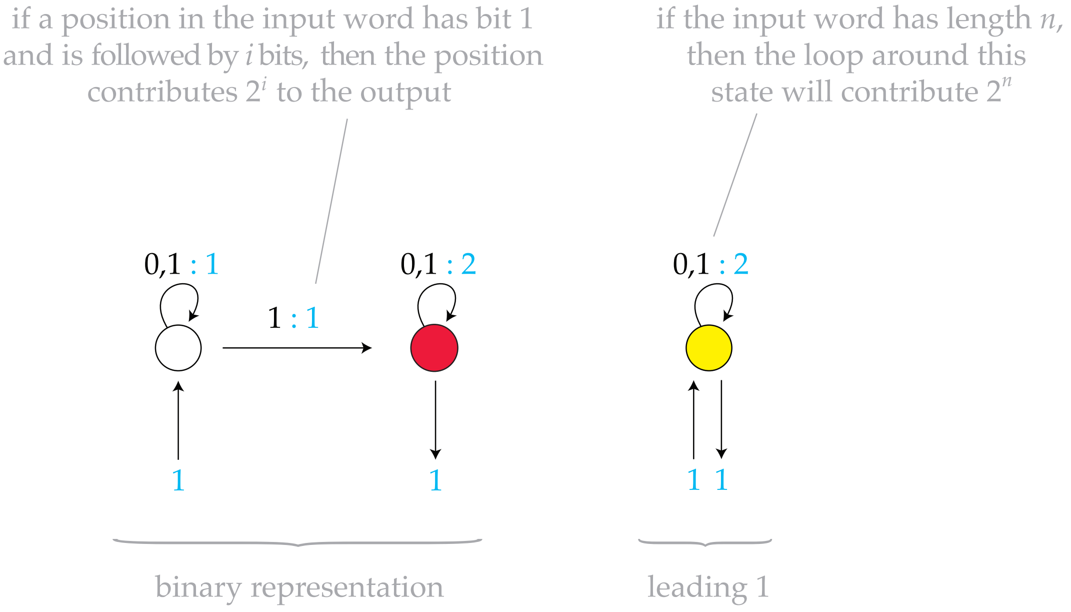
- **Question.** *Is some word mapped to* 0*?*

Changing 0 to any other number would not make the problem decidable, because if f is recognised by a weighted automaton, then so is $x \mapsto f(x) - c$ for every constant $c \in \mathbb{Q}$. There are two basic ingredients in the proof: hashing words as numbers, and composing weighted automata with NFA's with output. These ingredients are described below.

Hashing. A weighted automaton can map a string of digits to its interpretation as a fraction stored in binary (or ternary, etc) notation. This construction is described in the following lemma.

Lemma 8.10. *For every alphabet* Σ *there is a weighted automaton which computes an injective function from* Σ^* *to the strictly positive rational numbers.*

Proof. We only show the construction when Σ has two letters $\{0, 1\}$. The idea is that the weighted automaton maps a word w to the number represented in binary by the word $1w$. We use the leading 1 so that the representation of w takes into account leading zeroes. Here is the automaton.


if a position in the input word has bit 1
and is followed by i bits, then the position
contributes 2^i to the output
if the input word has length n,
then the loop around this
state will contribute 2^n
0,1 : 1
1 : 1
0,1 : 2
0,1 : 2
1
1
1 1
binary representation
leading 1

■

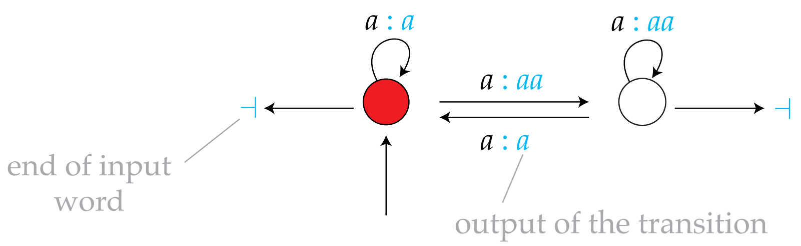
Composition with NFA's. To give a high-level description of the undecidability proof, it will be convenient to compose weighted automata with a certain kind of word-to-word functions.

Definition 8.11. *An* NFA with output *consists of:*

1. *An* NFA $\mathcal{A}$, *called the* underlying automaton*;*
2. *An* output alphabet Γ*;*
3. *For each transition of* $\mathcal{A}$*, an associated* output word *in* Γ^**;*
4. *For each final state of* $\mathcal{A}$*, an associated* end of input word *in* Γ^**.*

The output of a run, which is a word over the output alphabet, is defined by concatenating the output words for all transitions in the order that they are used, followed by the end of input word for the last state in the run. Given a word w over the input alphabet, the output of the automaton $\mathcal{A}(w)$ consists of the outputs of all of its accepting runs. We view $\mathcal{A}(w)$ as a multiset, so that if n different runs produce the same output word, then this output word is counted n times.

Example 16. Consider the following NFA with output where the input alphabet is $\{a\}$ and the output alphabet is $\{a, \dashv\}$.



On an input word a^n, the automaton has 2^n possible runs – and therefore also 2^n output words including repetitions – because after reading each letter, the automaton can be in either the red or white state. The function recognised by the automaton is

$$a^n \qquad \mapsto \qquad \sum_{X \subseteq \{1,\ldots,n\}} a^{n+|X|} \dashv$$

where sum denotes multiset addition. For example, in the multiset $\mathcal{A}(a^{10})$, the word a^{12} appears $\binom{10}{2}$ times. □

Weighted automata can be composed with NFA's with output.

Lemma 8.12. *If $\mathcal{A}$ is an* NFA *with output, which has input alphabet Σ and output alphabet Γ, and $\mathcal{B}$ is a weighted automaton with input alphabet Γ, then the function $\mathcal{B} \cdot \mathcal{A}$ defined by*

$$w \in \Sigma^* \qquad \mapsto \qquad \sum_{v \in \mathcal{A}(w)} \mathcal{B}(v)$$

is also recognised by a weighted automaton. In the sum above, outputs are counted with repetitions, i.e. an output word produced n times contributes n times to the sum.

Proof. For a run of the weighted automaton $\mathcal{B}$, define its *transition weight* to be the product of the weights of the transitions used in the run, without taking into account the initial weight of the first state or the final weight of the last state. A natural product construction does the job. Define a product automaton as follows. States of the product automaton are pairs (state of $\mathcal{B}$, state of $\mathcal{A}$). The initial weight of a pair (p, q) is defined to be 0 if q is not an initial state in $\mathcal{A}$, and otherwise it is defined to be the initial weight of p. The weight of a transition

$$(p, q) \xrightarrow{a} (p', q')$$

in the product automaton is defined to be the 0 if $\mathcal{A}$ does not admit a transition $q \xrightarrow{a} q'$, otherwise it is defined to be the sum

$$\sum_{\rho} \text{transition weight of } \rho$$

where ρ ranges over runs of the weighted automaton $\mathcal{B}$ that begin in p, read the output word labelling the transition $q \xrightarrow{a} q'$, and end in p'. The final weight of a pair (p, q) is defined to be 0 if q is not a final state in $\mathcal{A}$, and otherwise it is defined to be

$$\sum_{\rho} (\text{transition weight of } \rho) \cdot (\text{final weight of last state in } \rho)$$

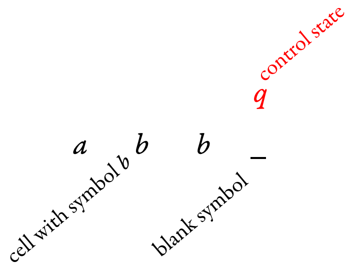
where ρ ranges over runs of the weighted automaton $\mathcal{B}$ which begin in state p and read the end of input word for state q in the automaton $\mathcal{A}$. ∎

The multiset semantics of NFA's with output were chosen so that the proof above works. In our undecidability proof below, we use the above lemma in the special case when $\mathcal{A}$ has at most one run over every input word, and so the multiset semantics do not play a role. Equipped with Lemmas 8.10 and 8.12, we prove the undecidability result from Theorem 8.9.

Proof of Theorem 8.9. We first introduce some notation and closure properties for weighted automata. If $\mathcal{A}, \mathcal{A}'$ are weighted automata, then we write $\mathcal{A} + \mathcal{A}'$ for the disjoint union of the automata; on the level of recognised functions this corresponds to addition of outputs. We write $-\mathcal{A}$ for the weighted automaton obtained from $\mathcal{A}$ by multiplying all initial weights by -1, on the level of recognised functions this corresponds to multiplying the output values by -1. We also write $\mathcal{A} - \mathcal{A}'$ instead of $\mathcal{A} + (-\mathcal{A}')$. Finally, if L is a regular language, then there is a weighted automaton, call it $\mathsf{char}(L)$ which recognises the characteristic function of L, i.e. maps words from L to 1 and other words to 0. The proof of the theorem is by reduction from the halting problem for Turing machines. For a Turing Machine M, we define a weighted automaton which outputs 0 on at least one input word if and only if M has at least one halting computation. Suppose that Σ is the work alphabet of the machine M, which includes the blank symbol. We encode a configuration of the machine as a word over the alphabet

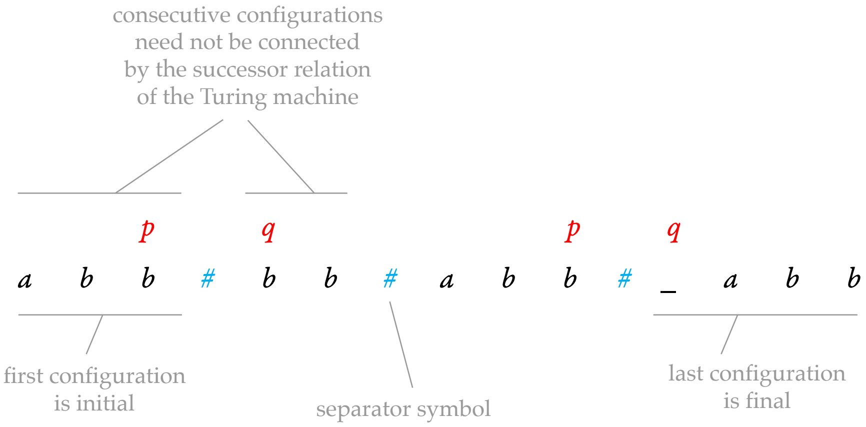
$$\Delta \overset{\text{def}}{=} \Sigma + \Sigma \times Q$$

in the natural way, here is a picture:



To ensure that each configuration has exactly one encoding, we assume that the first letter and the last letter are not just blank symbols, i.e. each one contains either the head, or a non-blank tape symbol, or both.

Define a *pre-computation* to be a word which is a sequence of encodings, in the sense above, of at least two configurations, separated by a fresh separator symbol #, such that the first configuration is initial and the last configuration is final. Here is a picture:


consecutive configurations need not be connected by the successor relation of the Turing machine
p q p q
a b b # b b # a b b # _ a b b
first configuration is initial
separator symbol
last configuration is final

A halting computation of the Turing machine is a pre-computation where consecutive configurations are connected by the successor relation on configurations of the Turing machine.

The set of pre-computations is a regular language. It is not hard to write NFA's with output $\mathcal{A}_1, \mathcal{A}_2$ such that if the input is not a pre-computation then the output for both $\mathcal{A}_1$ and $\mathcal{A}_2$ is the empty multiset, and if the input is a pre-computation, as witnessed by a (unique) decomposition

$$w_1\#w_2\#\cdots\#w_n \qquad w_1,\ldots,w_n \in \Delta^*$$

then $\mathcal{A}_1, \mathcal{A}_2$ have exactly one accepting run each, with respective outputs

$$w_2\#w_3\#\cdots\#w_n \qquad w_1'\#w_2'\#\cdots\#w_{n-1}'$$

where w_i' denotes the successor configuration of w_i. The Turing machine has a halting computation if and only if there is some pre-computation where $\mathcal{A}_1$ and $\mathcal{A}_2$ produce the same output. This is equivalent to the following weighted automaton producing 0 on at least one output:

$$\mathsf{char}(\text{words that are not pre-computations}) + \mathcal{H}\cdot\mathcal{A}_1 - \mathcal{H}\cdot\mathcal{A}_2,$$

where $\mathcal{H}$ is the hashing automaton from Lemma 8.10 and the product operation $\cdot$ is as in Lemma 8.12. ■

Problem 82. Construct weighted automata over unary alphabet, which for a word of length n output

1. n^2;
2. $n^2 + 2n$;
3. n^3;
4. n^k for constant $k \in \mathbb{N}$;
5. $p(n)$ for any polynomial $p \in \mathbb{Q}[x]$, i.e. a univariate polynomial with rational coefficients.

Problem 83. Show that for weighted automata with 2 states over a unary alphabet, it is decidable whether the automaton assigns value 0 to some word. *Remark:* for weighted automata over a unary alphabet with an arbitrary number of states, this is an important open problem, called the Skolem Problem in [43].

Problem 84. A probabilistic automaton is a vector space automaton where the initial state $q \in \mathbb{Q}^d$ is a probability distribution on $\{1,\ldots,d\}$, the linear updates are such that they preserve probability distributions, and the output function sums the coordinates corresponding to some accepting subset $F \subseteq \{1,\ldots,d\}$. Show that the following questions are undecidable for probabilistic automata:

1. is there some input word which produces output exactly $1/2$?
2. for fixed $p \in (0,1)$, is there some input word which produces output exactly p?
3. is there some input word which produces output at least $1/2$?

Problem 85. Show that the following question is decidable for probabilistic automata: is there some input word which produces output equal exactly 0?

Problem 86. Show that for every weighted automaton there is an isomorphic (using the notion of isomorphism inherited from vector space automata) one which has one initial and one final state.

Problem 87. Let $E \subseteq \mathbb{Q}^n \times \mathbb{Q}^n$ be a linear subspace which is an equivalence relation. Let $f_1,\ldots,f_k : \mathbb{Q}^n \to \mathbb{Q}^n$ be linear maps which respect the equivalence relation, i.e. if inputs are equivalent, then also outputs are also equivalent. Show that one can compute in polynomial time linear maps

$$h : \mathbb{Q}^n \to \mathbb{Q}^m \qquad f'_1,\ldots,f'_k : \mathbb{Q}^m \to \mathbb{Q}^m$$

so that E is the kernel of h, and the diagram

$$\begin{tikzcd}
\mathbb{Q}^n \arrow[r, "f_i"] \arrow[d, "h"'] & \mathbb{Q}^n \arrow[d, "h"] \\
\mathbb{Q}^m \arrow[r, "f'_i"'] & \mathbb{Q}^m
\end{tikzcd}$$

commutes for every $i \in \{1,\ldots,k\}$.

Problem 88. Consider a more symmetric model of NFA with output, as in Definition 8.11, where there is also a *start of input* word associated to each initial state, and the output of a run begins with the start of input word for its first state. Show that this model has the same expressive power as in Definition 8.11.

Problem 89. Call an NFA *unambiguous* if for every input there is at most one accepting run. Show that equivalence – i.e. are the same input words accepted – for unambiguous automata can be decided in polynomial time.

Problem 90. Construct an NFA with n states such that shortest rejected word rejected has length exponential wrt. n.

Problem 91. Show that if an NFA with n states is unambiguous and rejects at least one word, then it rejects some word of length at most $n - 1$.

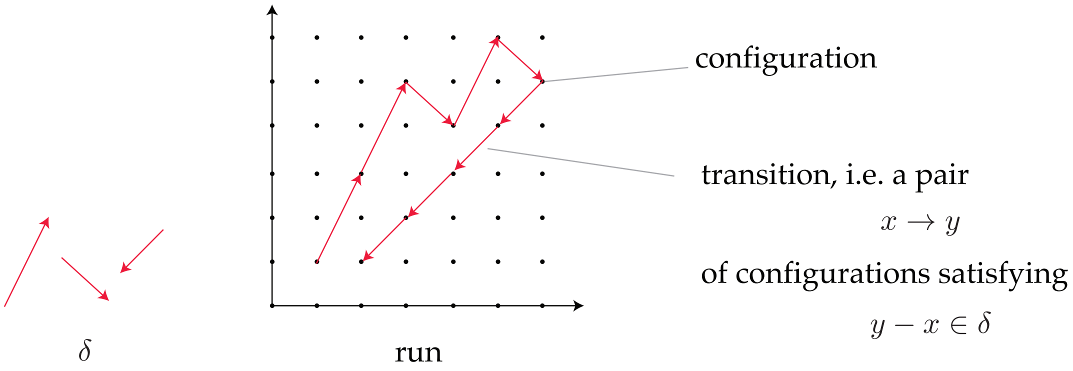
Problem 92. Show a polynomial time algorithm that decides if an NFA is unambiguous.

Problem 93. Give a more direct proof of Theorem 8.9 which uses the Post Correspondence Problem. Recall that the Post Correspondence Problem is the question: given two homomorphisms $f, g : \Sigma^* \to \Gamma^*$, decide if there is some nonempty word w such that $f(w) = g(w)$. This problem is undecidable.

9
Vector addition systems

This chapter is about vector addition systems. The definition of this device could hardly be simpler:

Definition 9.1 (Vector Addition System)**.** *The syntax of a* vector addition system *consists of a dimension $d \in \{1, 2, \ldots\}$ and a finite set $\delta \subseteq \mathbb{Z}^d$. A run of the system is a finite sequence of vectors in $\mathbb{N}^d$ (called* configurations*) such that every consecutive configurations in the run form a* transition *as explained in the following picture for dimension $d = 2$:*



The most famous problem for vector addition systems is *reachability*, i.e. given two configurations, decide if they can be connected by a run. Reachability is decidable, which was first shown by Mayr in [36], although the computational complexity of the problem remains unknown, see [47]. The reachability algorithm is complicated and beyond the scope of this book. It is crucial that configurations are vectors of natural numbers; for configurations that are integer vectors the reachability problem and related problems become much simpler, see Exercise 103.

In this chapter, we present a simple algorithm for a different problem, called coverability.

Theorem 9.2. *The following problem, called coverability, is decidable:*

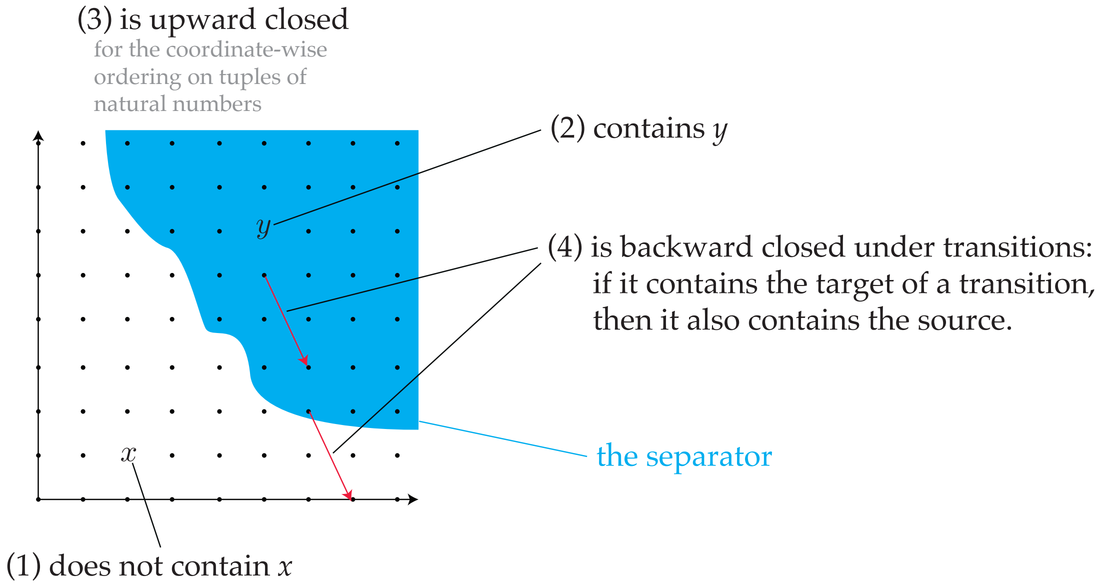
- **Input.** *A vector addition system with distinguished configurations x, y.*
- **Question.** *Is there a run from x to some configuration $\geq y$?*

In the above theorem, $\geq$ refers to the coordinate-wise ordering on vectors of natural numbers. Not only is the algorithm for the coverability problem conceptually simple, but it represents a technique that can be used to solve many other problems. The technique is known as *well-structured transition systems*, and *well quasi-orders* play a prominent role. See the exercises for more examples, and [48] for more on the topic.

Fix an input to the coverability problem, i.e. a vector addition system with distinguished configurations x and y. Let d be the dimension. Define a *semi-algorithm* for a decision problem to be an algorithm that terminates with success for "yes" instances, and which does not terminate for "no" instances. A decision problem is decidable if and only if both the problem and its complement have semi-algorithms. Clearly the coverability problem has a semi-algorithm – enumerate all runs that begin in x and terminate with success after finding a run that reaches a configuration $\geq y$. The following lemma completes the proof of Theorem 9.2, by giving a semi-algorithm for the complement of the coverability problem.

Lemma 9.3. *There is a semi-algorithm deciding non-coverability, i.e. an algorithm that inputs a vector addition system with configurations x, y and terminates with success if and only if there is no run from x to any configuration $\geq y$.*

Proof. Define a *separator* for configurations x and y to be a set of configurations that satisfies properties (1) - (4) depicted below


(3) is upward closed
for the coordinate-wise ordering on tuples of natural numbers
(2) contains y
y
(4) is backward closed under transitions: if it contains the target of a transition, then it also contains the source.
x
the separator
(1) does not contain x

We claim that the following conditions are equivalent:

1. there is no run from x to a configuration $\geq y$;
2. there is a separator for x and y.

For the top-down implication, one takes the separator to be the set of those configurations which can reach at least one configuration $\geq y$. This set is upward closed because the target set $\geq y$ is upward closed, and transitions can be moved up, i.e. if $a \to b$ is a transition, then also $a + c \to b + c$ is a transition, for every $c \in \mathbb{N}^d$. (The remaining conditions in the definition of a separator are easily seen to be satisfied.) For the bottom-up implication, we observe that the separator contains all configurations that can reach at least one configuration $\geq y$, and possibly other configurations as well.

To prove the lemma, it remains to show a semi-algorithm that checks if there exists a separator. We claim that a separator, actually any upward closed set, can be represented in a finite way using its minimal elements. First, every element in the separator is above some minimal element, because the order $\leq$ on $\mathbb{N}^d$ is well-founded. Second, every set has finitely many minimal elements, because minimal elements form an antichain (i.e. are pairwise incomparable with respect to $\leq$) and antichains are finite according to the following claim.

Claim 9.4 (Dickson's Lemma)**.** *Antichains in $\mathbb{N}^d$ are finite.*

Proof. We prove a slightly stronger statement: for every sequence

$$x_1, x_2, \ldots \in \mathbb{N}^d$$

there is an infinite (not necessarily strictly) increasing subsequence, i.e. consecutive elements in the subsequence are related by $\leq$. The stronger statement is proved by induction on d.

- *Induction base $d = 1$.* Take the first element x of the sequence such that all following elements are $\geq x$. Such an element must exist because $\leq$ on $\mathbb{N}$ is a well-founded total order. Put x into the subsequence, and then repeat the process for the tail of the sequence after x.

- *Induction step.* Using the induction assumption, extract a subsequence that is increasing on the first coordinate, and from that subsequence extract another one that is increasing on the remaining coordinates.

An alternative proof would use the infinite Ramsey theorem, see Problem 97. ■

By Dickson's Lemma, every upward closed set can be represented in a finite way as the upward closure of some finite set. The semi-algorithm from the statement of the lemma enumerates through all finite subsets $S \subseteq \mathbb{N}^d$, and for each one checks if its upward closure satisfies conditions (1)-(4) in the definition of a separator. The only interesting condition is (4), i.e. backward closure under transitions. Consider a potential counterexample for (4), i.e. a transition $a \to b$ such that the target is in the upward closure of S, but the source is not. If the counterexample $a \to b$ is chosen minimal coordinate-wise, then

(*) there is some $c \in S$ such that for every coordinate $i \in \{1, \ldots, d\}$, either the source has 0 on coordinate i, or the target agrees with c on coordinate i.

There are finitely many transitions which satisfy (*), namely at most $2^d|S|$, and we can go through all of them to check if (4) is satisfied. ■

Problem 94. Show that the following conditions are equivalent for every quasi-order (a binary relation that is transitive and reflexive, but not necessarily anti-symmetric):

1. every infinite sequence contains an infinite subsequence that is increasing (not necessarily strictly);

2. there are no infinite strictly decreasing sequences (i.e. the quasi-order is well-founded) and no infinite antichains (an antichain is a set of pairwise incomparable elements);

3. every upward closed set is the upward closure of a finite set.

A quasi-order that satisfies the above conditions is called a wqo.

Problem 95. Which of the following ordered sets are wqo's?

1. $\mathbb{N}^2$ with lexicographic order;

2. $\{a, b\}^*$ with lexicographic order;

3. $\mathbb{N}$ with divisibility order, i.e. x smaller than y if $x \mid y$;

4. Σ^* with prefix order;

5. Σ^* with infix order;

6. line segments with an order: $[a, b]$ smaller than $[c, d]$ if $(b < c) \vee (a = c \wedge b \leq d)$;

7. graphs with subgraph order (remove some edges and some vertices);

8. trees with subtree order (remove some nodes, but keep the descendant ordering).

Problem 96. Show that if $(X, \leq_X)$ and $(Y, \leq_Y)$ are both wqos then also $(X \times Y, \leq)$ is wqo, where $(x, y) \leq (x', y') \Leftrightarrow x \leq_X x' \wedge y \leq_Y y'$.

Problem 97. Prove the Infinite Ramsey Theorem: in every infinite clique, with edges coloured on finitely many colours there is an infinite monochromatic subgraph, i.e. subgraph such that all the edges in it are coloured by the same colour.

Problem 98. Let $(X, \preceq)$ be a wqo. Show that there is no infinite growing sequence of upward-closed subsets X, i.e. no sequence

$$U_1 \subsetneq U_2 \subsetneq \ldots,$$

s.t. for all $i \in \mathbb{N}$ set $U_i \subseteq X$ is upward-closed wrt. $\preceq$.

Problem 99. Show that given a d-dimensional VAS and $s \in \mathbb{N}^d$, one can compute the set of all configurations from which s is coverable. Hint: use Problem 98.

Problem 100. Show that given a vector addition system with a distinguished source configuration, one can decide if the set of configurations reachable from the source is finite.

Problem 101. Prove the following version of Higman's Lemma: if Σ is a finite alphabet, then Σ^* ordered by (not necessarily connected) subword is a wqo.

Problem 102. Define a *rewriting system* over an alphabet Σ to be finite set of pairs $w \to v$ where $w, v \in \Sigma^*$. Define $\to^*$ to be the least binary relation on Σ^* which contains $\to$, is transitive, and satisfies

$$w \to^* v \qquad \text{implies} \qquad aw \to^* av \text{ and } wa \to^* va \qquad \text{for every } a \in \Sigma.$$

There exist rewriting systems where $\to^*$ is an undecidable relation. Show that $\to^*$ is decidable if the rewriting system is *lossy* in the following sense: for every letter $a \in \Sigma$, the rewriting system contains $a \to \varepsilon$.

Problem 103. Define a *$\mathbb{Z}$-vector addtion system* in the same way as a vector addition system, except that configurations are vectors in $\mathbb{Z}^d$. Show that the reachability problem is decidable, i.e. one can decide if there is a run connecting two given configurations.

Problem 104. Define a *vector addition system with states* to be a finite set of states Q, a dimension d, and a finite set $\delta \subseteq Q \times \mathbb{Z}^d \times Q$. A configuration is an element of $Q \times \mathbb{N}^d$, and a transition is a pair

$$(q, x) \to (p, y) \qquad \text{such that } (q, y - x, p) \in \delta.$$

Show that the following problem is decidable: given states p, q decide if there is a run from the configuration $(p, \overline{0})$ to some configuration with state q.

Problem 105. Find a vector addition system, say of dimension d, where the reachability relation

$$\{(x, y) : \text{there is a run from from } x \text{ to } y\} \subseteq \mathbb{N}^{2d}$$

is not semilinear. Hint: use states and try to simulate exponentiation.

Problem 106. Find a family of vector addition systems with states, say of dimension d (the dimension does not need to be fixed for the family), where the reachability set

$$\{v : \text{there is a run from from the origin to } v\} \subseteq \mathbb{N}^d$$

is finite, but

1. of doubly exponential size,
2. of tower size

with respect to the number of transitions.

10 First-order theory of the reals

Consider the real numbers, equipped with binary functions for addition, subtraction, multiplication, and constants for zero and one, and a binary relation for the ordering

$$(\mathbb{R}, +, -, \times, 0, 1, <)$$

The goal of this section is to prove a Theorem of Alfred Tarski, which says that the first-order theory of this structure is decidable, i.e. there is an algorithm which inputs a sentence of first-order logic like

$$\forall x \exists y \; y \times y + x = 0$$

and says if the sentence is true in the real numbers. Remarkably this theorem represents a nontrivial algorithm dealing with first-order logic that was found in the late 1920s, before the notions of "first-order logic" and "algorithm" were well defined in the modern sense. Although proved earlier, the result was published only after the war [54].

There is some freedom in the choice of vocabulary. For example we could add division because it is definable in first order logic by

$$x/y = z \qquad \overset{\text{def}}{=} \qquad z \times y = x.$$

Conversely, we could remove subtraction, because it is definable in terms of addition, or we could remove the ordering, because it is defined by

$$x < y \qquad \overset{\text{def}}{=} \qquad \exists z\ x + z \times z = y.$$

Also, we could remove the constants for 0 and 1, because 0 is the unit for addition and 1 is the unit for multiplication. Summing up, as long as we care about first-order logic, the vocabulary could be reduced to have only + and ×. Nevertheless, we will care about quantifier-free formulas, and some of the above definitions are not quantifier-free.

Theorem 10.1. *For every formula of first-order logic over*

$$(\mathbb{R}, +, -, \times, 0, 1, <)$$

possibly with free variables, one can compute an equivalent (over the real numbers) formula that is quantifier-free. In particular, one can decide if a sentence, i.e. a formula without free variables, is true over the real numbers.

Before proving the theorem, we discuss its relation to decidability of first-order logic for other fields and rings, such as integers or rationals.

Example 17. The first-order theory of the integers with addition, subtraction and multiplication

$$(\mathbb{Z}, +, -, \times, 0, 1, <)$$

is undecidable. It follows that there is no first-order formula $\varphi(x)$ which defines the integers inside the real numbers. In contrast, every single integer can be defined.
A celebrated theorem of Julia Robinson says that there is a first-order formula that defines the integers inside the rational numbers, and therefore also the rational numbers cannot be defined inside the real numbers. □

Example 18. A complex number can be coded as two real numbers, its real and imaginary parts. Addition, subtraction and multiplication of complex numbers can be reduced to analogous operations on the real and imaginary parts. It follows that the first-order theory of the complex numbers

$$(\mathbb{C}, +, -, \times, 0, 1)$$

is decidable (note that order is not mentioned above, since it is not defined on the complex numbers). □

The rest of this chapter is devoted to proving Theorem 10.1. The second part of the theorem (deciding which sentences are true) is an immediate consequence of the first part (effective quantifier elimination). Indeed, if we want to know if a sentence is true in the real numbers, we first eliminate all quantifiers using the first part of the theorem, arriving at a Boolean combination of sentences that are similar to this example:

$$(1+1) \times (1+1+0) - 0 \times (1+0) > 0.$$

A straightforward evaluation leads to the desired true/false answer. It is therefore enough to prove the effective quantifier elimination. Here, it is enough to show that a single existential quantifier can be eliminated, since multiple quantifiers can then be eliminated one by one, starting with the innermost quantifiers (and using closure of quantifier-free formulas under Boolean operations).

Therefore, the essence of Theorem 10.1 is showing that for every formula

$$\exists x \qquad \underbrace{\varphi(x, y_1, \ldots, y_n)}_{\text{quantifier-free formula using } +, \times, -, 0, 1, <}$$

there exists, and can be computed, an equivalent one that is quantifier-free over the same vocabulary. We first observe that the formula φ can be without loss of generality assumed to be a Boolean combination of formulas of the form

$$p(x, y_1, \ldots, y_n) > 0$$

where each p is a polynomial with integer coefficients and variables $x, y_1, \ldots, y_n$. So the goal is to understand the behaviour of the polynomials p, once the arguments $y_1, \ldots, y_n$ have been fixed, as a function of the quantified parameter x. To understand this behaviour, we will use basic operations from calculus and algebra, like differentiation and dividing polynomials with remainders, and then observe that the effect of these operations can be formalised using quantifier-free formulas.

Instead of working directly with quantifier-free formulas, we introduce an intermediate computation model for the reals and show that (a) computation in this model can be simulated using quantifier-free formulas; and (b) quantifier elimination can be done using the computation model. The point of using the computation model is that when proving (b), we can appeal to algorithmic intuitions, like loops and conditionals, which are more cumbersome to formalise when working directly with quantifier-free formulas. The results (a) and (b) are presented in Sections 10.1 and 10.2 below.

10.1 Computation on the reals

Consider the following variant of a Turing machine, which we call a BSS *machine*, standing for Blum Shub and Smale, who introduced the model. The purpose of a BSS machine is to compute a partial function of type $\mathbb{R}^* \to \mathbb{R}^*$, i.e. a partial function that inputs and output tuples of real numbers. The machine has a single tape, infinite in both directions (i.e. indexed by integers), whose cells store real numbers plus a finite set R of registers that also store real numbers. Cells of the tape and registers can be undefined. At any given moment, the machine is in a state from a finite set of control states, and has a single head which points to one of the cells on the tape. In the initial configuration, the tape stores the input of the function (i.e. the i-th cell stores the i-th number of the input if the input has length $\geq i$, otherwise the cell is undefined), the registers are all undefined, the state is a designated initial sate and the head points to the cell 0 of the tape. Based on the answers to the following questions (the second question is actually several question, one for eachc of the finitely many registers):

- what the current state?

- what are the signs of the registers (negative, zero, positive, undefined)?

the transition function of the machine indicates deterministically a new state and an operation from the following set:

$$\text{accept} \qquad \text{move the head by } i \in \{-1, 1\} \qquad r := 1 \qquad r := s \ \texttt{op} \ t$$

where op ranges over the arithmetic operations (addition, subtraction, multiplication, dvision) and r, s, t range over registers or the contents of the cell under the head. For example, the effect of a transition can be that the contents of registers r, s are multiplied and the result is stored in the cell under the head. If a division by zero is performed, then the computation is not aborted, but the cell/register r storing the result becomes undefined. If the computation never performs the accept operation then the output of the computed function is undefined. Otherwise, the output of the computed function is defined to be the contents of the defined cells, read from left to right. If the output is shorter than the input, then division by zero is used to make cells on the tape undefined. The running time of a computation is defined to be the number of transitions that it uses.
As defined above, a BSS machine computes a partial function. We can also view BSS machines as computing languages, i.e. yes/no properties of tuples of reals, by assuming that the answer is "yes" if the function is defined and "no" otherwise. The following lemma shows that for languages of bounded dimension and running computation time, the BSS model can only compute quantifier-free properties.

Lemma 10.2. *If $S \subseteq \mathbb{R}^n$ is computed by a* BSS *machine which uses at most k computation steps in every accepting run, then S is definable by a quantifier-free formula using $+, -, \times, 0, 1, <$. Furthermore, this formula can be computed given the machine and k.*

Proof. By induction on the length of the computation, we can show that the control state, head position, and signs of the registers can be determined by asking quantifier-free queries to the input. In the induction step, we observe that, assuming that the history (sequence of transitions) of the computation is known, then the contents of each register/cell can be described by a term that uses the n input numbers, integer constants, and arithmetic operations (addition, subtraction, multiplication and addition). Furthermore, if t is such a term, then a straightforward induction on the size of t shows that the truth sign (negative, zero, positive) of t can be expressed using a quantifier-free formula as in the statement of the lemma. (Note that the term t is allowed to use division, but the quantifier-free formula is not.) For example,

$$\frac{x * y}{x - y + 1} > 0$$

is equivalent to the quantifier-free formula

$$\underbrace{(x \times y > 0) \land (x - y + 1 > 0)}_{\text{enumerator and denominator are positive}} \quad \lor \quad \underbrace{(x \times y > 0) \land (x - y + 1 > 0)}_{\text{enumerator and denominator are negative}}$$

■

10.2 *Quantifier elimination*

In Section 10.1 above we have shown that any bounded time procedure in the BSS model can be simulated using quantifier-free formulas. Therefore, to complete the proof of Theorem 10.1, it is enough to show that the truth values of formulas with one quantifier can be decided using the BSS model. In principle, the BSS model manipulates sequences of reals, but we will also use it to manipulate more structured entities, such as polynomials or finite sets of polynomials, implicitly using straightforward encodings as tuples of real numbers.

Lemma 10.3. *For every quantifier-free formula $\varphi(x, y_1, \ldots, y_n)$ there is a* BSS *machine which computes the following:*

- **Input.** *Real numbers $a_1, \ldots, a_n$.*
- **Output.** *Is $\exists x \varphi(x, a_1, \ldots, a_n)$ true in the real numbers?*

Furthermore, the running time of the algorithm is bounded by a number k that depends only on (and can be computed from) the formula φ, and does not depend on the numbers $a_1, \ldots, a_n$.

Together with Lemmas 10.2, the above lemma completes the proof of Theorem 10.1, and therefore the rest of this chapter is devoted to proving the above lemma. Let φ and $a_1, \ldots, a_n$ be as in the lemma. Without loss of generality we assume that φ is a Boolean combination of formulas of the form

$$p(x, y_1, \ldots, y_n) > 0 \tag{10.1}$$

where each p is a polynomial with integer coefficients and $n + 1$ variables. Given a tuple of real numbers $a_1, \ldots, a_n$ as in the input of the problem from Lemma 10.3, define P to be the finite set

$$\{p(x, a_1, \ldots, a_n) : p(x, y_1, \ldots, y_n) > 0 \text{ appears in } \varphi\}. \tag{10.2}$$

The set P contains polynomials with one variable x and real coefficients. The coefficients depend on the parameters $a_1, \ldots, a_n$, in a way that can be computed in the BSS model. Define the *sign* of a real number to be one of the three results (negative, zero, or positive) of comparing the number to 0. The truth value of the comparison in (10.1) depends only on the sign of the left side, and therefore in order to determine if a real number a satisfies $\varphi(a, a_1, \ldots, a_n)$, it is enough to look at the sign of $p(a)$ for all polynomials $p \in P$. This observation motivates the following definition. In the following definition, $\mathbb{R}[x]$ stands for polynomials with real coefficients and one variable x, and a nonzero polynomial is one with at least one nonzero coefficient.

Definition 10.4 (Sign table)**.** *Let $P \subseteq \mathbb{R}[x]$ be a finite set of nonzero polynomials, and let*

$$r_1 < r_2 < \cdots < r_n$$

be all the real numbers that are a root of at least one polynomial in P. (There are finitely many such numbers, because a nonzero polynomial has finitely many roots.) Define the sign table *of P to be the following information:*

- *the number n;*
- *for each $p \in P$ and $i \in \{0, \dots, n+1\}$, the sign of $p(r_i)$, where the sign in*

  $$r_0 \overset{def}{=} -\infty \qquad r_{n+1} \overset{def}{=} +\infty$$

  is defined by taking the limit in the natural way.
- *for each $p \in P$ and $i \in \{0, \dots, n\}$, what is the sign of p on the interval $(r_i; r_{i+1})$.*

From the discussion before the above definition, it follows that the truth of

$$\exists x \varphi(x, a_1, \dots, a_n)$$

can be (effectively) determined by looking at the sign table of the nonzero polynomials in P. Therefore, Lemma 10.2 and thus also the Tarski Theorem will follow from the following lemma.

Lemma 10.5. *The sign table of a finite set $P \subseteq \mathbb{R}[x]$ of nonzero polynomials can be computed by a BSS machine in time bounded by a function of the sum of degrees of P.*

Proof. The proof is essentially the observation that the usual method of plotting polynomials that is taught in high school can be formalised in the BSS model[1] We begin by observing that the BSS model can run the Euclidean algorithm.

Claim 10.6. *There is a BSS algorithm which inputs polynomials*

$$p, q \in \mathbb{R}[x] \qquad \textit{with} \qquad \textit{degree of } q \leq \textit{degree of } p$$

and outputs a polynomial $r \in \mathbb{R}[x]$ such that

$$\textit{degree of } r < \textit{degree of } q \qquad \textit{and} \qquad p = q \times s + r \quad \textit{for some } s \in \mathbb{R}[x].$$

Proof. Define a finite sequence of polynomials

$$p = p_0, p_1, \dots, p_n = r$$

[1] Tarski taught mathematics in a Warsaw high school for girls. A lesser man would complain about having to teach calculus, Tarski proved that it could be automated.

subject to the following invariant: (a) the degrees in the sequence strictly decrease; and (b) q divides $p - p_n$ for every n. We begin with $p_0 = p$. Suppose that p_i has been defined. If the degree of p_i is stritly smaller than the degree of q, then we output $r = p_i$; otherwise we define p_{i+1} to be the difference

$$p_i - \frac{\text{leading coefficient of } p_i}{\text{leading coefficient of } q} x^{(\text{degree of } p_i)\text{- }(\text{degree of } q)} \cdot q$$

which preserves the invariant. This procedure can clearly be implemented in the BSS model. ■

Apart from computing remainders using the Euclidean algorithm, we also use derivatives of polynomials (in the usual sense of calculus). The derivative of a polynomial $p \in \mathbb{R}[x]$ can clearly be computed in the BSS model. By repeatedly applying derivation and the Euclidean algorithm, we can extend any finite set of polynomials to a bigger one that is saturated in the following sense: it is closed under derivations and applying the Euclidean algorithm. Since computing a sign table can only get harder after adding polynomials, in order to show the lemma, it suffices to show that sign tables can be computed for sets P that are saturated.

Suppose then that P is a finite saturated set of polynomials. Take some $p \in P$ of maximal degree. The set $P - \{p\}$ is also saturated, because p has maximal degree, and both derivation and the Euclidean algorithm decrease degrees. Therefore, we can use induction to compute the sign table of $P - \{p\}$. Let

$$r_0 = -\infty \qquad r_1 < \cdots < r_n \qquad r_{n+1} = +\infty$$

be the numbers from the definition of a sign table, as applied to $P - \{p\}$. We do not know the exact values of these numbers, but we do know n and we also know how the signs of the polynomials from $P - \{p\}$ behave in the points r_i and the intervals that separate them. Our goal is to enrich this information to account for the polynomial p.

Claim 10.7. *For each $i \in \{0, 1, \ldots, n+1\}$ we can compute the sign of $p(r_i)$.*

Proof. For $i = 0$ and $i = n + 1$ we simply look at the sign of the leading coefficient of p, and the parity of its degree. This information determines the sign of p in $\pm\infty$. We are left with the case of $i \in \{1, \ldots, n\}$. By definition of r_i, there must be some polynomial $q \in P - \{p\}$ which has a root in r_i, and we can use the sign table to find that polynomial. Since P is closed under applying the Euclidean algorithm, there must be some $r \in P - \{p\}$ such that

$$p = q \times s + r \qquad \text{for some } s \in \mathbb{R}[x].$$

Since r_i is a root of q, the sign of $p(r_i)$ is the same as the sign of $r(r_i)$, and the latter sign is stored in the sign table. ■

We now proceed to investigate the behaviour of p in the intervals separating the roots r_i. The important observation is that p does not have any turning points in these intervals (a turning point is one where the polynomial changes behaviour between increasing/decreasing). The reason is that a turning point is also a root of the derivative, and all such roots are in the points $r_1, \ldots, r_n$ because the derivative of p is belongs to $P - \{p\}$. This observation yields the following claim:

Claim 10.8. *For every $i \in \{0, \ldots, n\}$, p has at most one root in the interval $[r_i; r_{i+1}]$.*

Proof. Otherwise there would be a turning point between r_i and r_{i+1}. ■

Using the above claim, we can describe the behaviour of p in an interval of the form $(r_i; r_{i+1})$. This is done using the following case analysis, which can readily be formalised in the BSS model.

- One of the signs of $p(r_i)$ and $p(r_{i+1})$ is zero, like this:

  picture

  In this case, thanks to Claim 10.8, the other sign is nonzero, p has no roots in the interval $(r_i; r_{i+1})$ and its sign on this interval is the same as in the nonzero endpoint.

- The signs of $p(r_i)$ and $p(r_{i+1})$ are the same, like this:

  picture

  If p would have a root in the interval $(r_i; r_{i+1})$, then it would also have a turning point in this interval, and this cannot happen. Therefore, p has no roots in this interval, and its sign in the interval is the same as in either one of its endpoints.

- The signs of $p(r_i)$ and $p(r_{i+1})$ are nonzero and different, like this:

  picture

  In this case p has exactly one root in the interval, which splits the interval into two parts; on the left part the sign of p is as in $p(r_i)$ and on right part the sign is as in $p(r_{i+1})$.

Doing the above case analysis for all $i \in \{0, \ldots, n\}$, we fill in the sign table for P. This completes the proof of Lemma 10.5, and therefore also of the Tarski Theorem. ■

Problem 107. Show that adding the function $\sin(x)$ to the real numbers yields an undecidable theory.

Problem 108. Show that the following structure has a decidable first-order theory: the universe consists of subsets of the Euclidean plane $\mathbb{R}^2$ that are points, lines or circles, and there is a binary predicate for inclusion.

Problem 109. Show that every $X \subseteq \mathbb{R}$ definable by a first-order formula with one free variable is a finite union of points and open intervals.

11
Polynomial grammars

In this chapter, we show that one can decide if a polynomial grammar – a type of grammar that generates numbers – has its language contained in $\{0\}$. The key tool is the Hilbert Basis Theorem. The application of the Hilbert Basis Theorem to problems in formal language theory dates back at least to the solution of the Ehrenfeucht Conjecture by Albert and Lawrence [2]. The presentation here is inspired by the more recent results from [51] and [6].

Definable reals. As our notion of "numbers", we use *definable reals*, which are those reals that can be defined in the first-order theory of the reals. More formally, a is definable real if and only if there is formula of first-order logic $\varphi(x)$ with one free variable that uses $+, \times, -, 0, 1, <$ and such that a is the unique real number which satisfies $\varphi(x)$.
For example, the formula

$$x^2 = 2 \land x > 0$$

is true only in $\sqrt{2}$, and hence $\sqrt{2}$ is a definable real. It is easy to see that the definable reals are a field – i.e. they are closed under addition, multiplication and inverses. We write $\mathbb{F}$ for the definable reals.
An *algebraic number* is defined to be a complex number that is a root of some nonzero polynomial in $\mathbb{Z}[x]$, i.e. a complex root of a nonzero univariate polynomial with integer coefficients. The following lemma shows that the

definable reals are exactly the algebraic numbers that are also real (i.e. have no imaginary part).

Lemma 11.1. *A real number is a definable real if and only if it is a root of some nonzero polynomial in $\mathbb{Z}[x]$.*

Proof. For the right-to-left implication, we observe that all roots of such a polynomial p are definable, the formulas being "first real root of p", "second real root of p", etc. Note how we use the order on the reals here.
For the left-to-right implication, we use Theorem 10.1, which says that if a is definable, then it is definable by a quantifier-free formula $\varphi(x)$. Such a formula is a finite Boolean combination of inequalities $p(x) > 0$ or $p(x) = 0$, where each $p \in \mathbb{Z}[x]$. If $\varphi(x)$ is true for a unique real a, then a has to be a root of one of the polynomials appearing in $\varphi(x)$, since otherwise there would be some other solution to $\varphi(x)$ in a sufficiently small neighbourhood of a. ■

Why do we use definable reals in this section? What about other fields or rings, e.g. the rings of reals, complex numbers, rationals or integers? None of these are going to work for us. The reals and complex numbers are uncountable, so they cannot be represented. Integers can be easily represented, but they raise another problem: they are not algebraically closed, and what is more, it is undecidable if a polynomial (with integer coefficients, and possibly more than one variable) has a root which uses only integers – this is the famous undecidability of Hilbert's 10th problem, see [44] for a brief history. In our approach, we need to find roots of polynomials, so the undecidability of Hilbert's 10th problem rules out the integers as a candidate for our ring. Similar problems arise with the rational numbers – it is unknown if the rational version of Hilbert's 10th problem is decidable, see [44, p. 348].

Polynomial grammars. When talking about polynomials, we mean polynomials where the coefficients are definable reals. In the definitions below, it will be convenient to use polynomial functions from vectors to vectors. Define a *polynomial function* to be a function of the form

$$p : \mathbb{F}^n \to \mathbb{F}^k \qquad \text{for } n, k \in \{0, 1, 2, \ldots\}$$

which is given by k polynomials representing the coordinates of the output vector, each one with n variables representing the coordinates of the input vector.

A polynomial grammar is a variant of a context free grammar. It generates definable reals (as opposed to words) and uses polynomial functions in the rules (as opposed to concatenation). Also, nonterminals are allowed to generate tuples of definable reals, although we require the starting nonterminal to generate only individual definable reals (this restriction is not important).

Definition 11.2 (Polynomial grammar). *A* polynomial grammar *consists of*

- *a set $\mathcal{X}$ of nonterminals, each one with an assigned* dimension *in* $\{1, 2, \ldots\}$*;*
- *a designated starting nonterminal of dimension 1;*
- *a finite set of productions of the form*

  $$X \leftarrow p(X_1, \ldots, X_k)$$

  where $k \in \{0, 1, \ldots\}$*,* $X_1, \ldots, X_k, X$ *are nonterminals, and*

  $$p : \overset{\text{dimension of } X}{\mathbb{F}^n} \to \overset{\text{sum of dimensions of } X_1, \ldots, X_k}{\mathbb{F}^m}$$

  is a polynomial function.

If a nonterminal has dimension n, then it generates a set of n-tuples of definable reals, which is defined as follows by induction. (The language generated by the grammar is defined to be the subset generated by its starting nonterminal.) Suppose that

$$X \leftarrow p(X_1, \ldots, X_k)$$

is a production and we already know that vectors $v_1, \ldots, v_k$ are generated by the terminals $X_1, \ldots, X_k$ respectively. Then the vector $p(v_1, \ldots, v_k)$ is generated by nonterminal X. The induction base is the special case of $k = 0$, where the polynomial p is a constant.

In all of our examples, the rules in the grammars will only use positive integers, and therefore the grammars will only generate tuples of positive integers. The fact that the grammars are allowed to use definable reals is an artefact of the method that we use to solve the grammars, and does not come from any desire to model definable reals.

Example 19. This grammar (there is only the starting nonterminal, which has dimension one) generates all odd natural numbers

$$X \leftarrow 1 \qquad X \leftarrow X + 2.$$

If we replace $X + 2$ by $X \times 2$, then we get the powers of two. The following grammar generates numbers of the form 2^{2^n}:

$$X \leftarrow 2 \qquad X \leftarrow X^2.$$

We can also generate factorials. Apart from the starting nonterminal X, we have a nonterminal Y of dimension two which generates pairs of the form $(n, n!)$. The crucial rule is this:

$$Y \leftarrow p(Y) \qquad \text{where } p \text{ is defined by } (a, b) \mapsto (a + 1, (a + 1) \cdot b).$$

The remaining rules are $Y \leftarrow (1, 1)$ and $X \leftarrow \pi_1(Y)$ where π_1 is defined by $(a, b) \mapsto a$. □

As usual, one can adopt an alternative fix-point view on grammars. We present this view since it will be used in the proof of Theorem 11.3. Define a *solution* to a grammar to be a function η which associates to each nonterminal X a set of vectors of definable reals of same dimension as X, and which satisfies all of the productions in the sense that

$$\underbrace{X \leftarrow p(X_1, \ldots, X_k)}_{\text{is a production}} \qquad \text{implies} \qquad \eta(X) \supseteq p(\eta(X_1) \times \cdots \times (\eta(X_k))$$

It is not difficult to see that the function which maps a nonterminal to the set of vectors generated by it is a solution, and it is the least solution with respect to coordinate-wise inclusion.
This chapter shows the following theorem.

Theorem 11.3. *One can decide if the language of a polynomial grammar is contained in* $\{0\}$*.*

The nonzeroness problem mentioned in the above theorem is clearly semi-decidable: one can enumerate all derivations of the grammar, and stop when a derivation is found that generates a nonzero number. The interesting part of the theorem is that the problem is also co-semi-decidable, i.e. one can find a finite witness proving that the generated language is contained in $\{0\}$. For this, we use the Hilbert Basis Theorem.

Hilbert's Basis Theorem. Let X be a set of variables. Define an *ideal* to be a set $I \subseteq \mathbb{F}[X]$ with the following closure properties:

$$\underbrace{p, q \in I \quad \Rightarrow \quad p + q \in I}_{\text{addition inside } I} \qquad \underbrace{p \in I, q \in \mathbb{F}[X] \quad \Rightarrow \quad pq \in I}_{\text{multiplication by arbitrary polynomials}}.$$

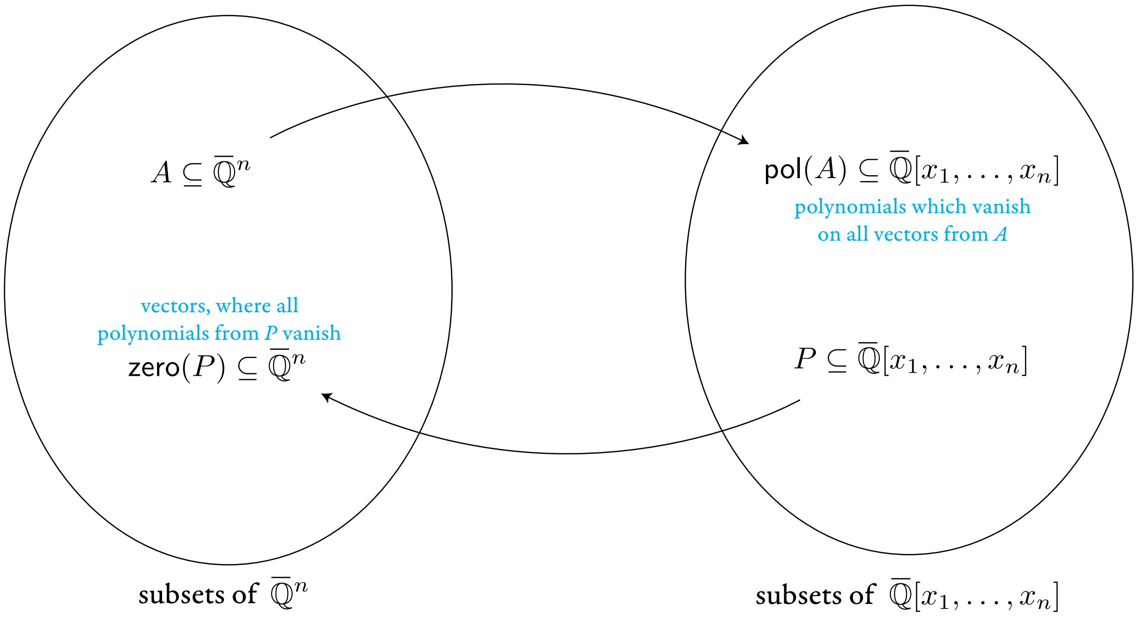
If $P \subseteq \mathbb{F}[X]$ is a set of polynomials, then the ideal generated by P, denoted by $\langle P \rangle$, is the set of polynomials of the form

$$p_1 q_1 + \cdots + p_n q_n \qquad \text{where } p_i \in P, q_i \in \mathbb{F}[X].$$

This is the inclusion-wise least ideal that contains P. We now state the Hilbert Basis Theorem.

Theorem 11.4. *If X is a finite set of variables, then every ideal in $\mathbb{F}[X]$ is finitely generated, i.e. of the form $\langle P \rangle$ for some finite set of polynomials.*

Algebraic closure. We say that a polynomial function $p : \mathbb{F}^n \to \mathbb{F}$ *vanishes* on a set $X \subseteq \mathbb{F}^n$ if p is the constant zero over this set. For a fixed dimension n, consider the following two operations pol and zero which go from sets of vectors to sets of polynomials, and back again:


$A \subseteq \overline{\mathbb{Q}}^n$
$\mathsf{pol}(A) \subseteq \overline{\mathbb{Q}}[x_1, \dots, x_n]$
polynomials which vanish
on all vectors from A
vectors, where all
polynomials from P vanish
$\mathsf{zero}(P) \subseteq \overline{\mathbb{Q}}^n$
$P \subseteq \overline{\mathbb{Q}}[x_1, \dots, x_n]$
subsets of $\overline{\mathbb{Q}}^n$
subsets of $\overline{\mathbb{Q}}[x_1, \dots, x_n]$

The picture above is a special case of what is known as a *Galois connection*. Note that the operation pol produces only ideals. For a set of vectors $A \subseteq \mathbb{F}^n$, define its *closure* as follows:

$$\overline{A} \quad \stackrel{\text{def}}{=} \quad \mathsf{zero}(\mathsf{pol}(A)).$$

The closure operation defined above is easily seen to be a closure operator, in the sense that it can only add elements to the set, it is monotone with respect to inclusion, and applying the closure a second time adds nothing. A *closed set* is defined to be any set obtained as the closure, equivalently a closed set is the vanishing set for some ideal of polynomials. By the Hilbert Basis Theorem, ideals of polynomials can be represented by giving a finite basis. Therefore, closed sets can be represented by giving a finite basis for the corresponding ideal of polynomials.

Example 20. Suppose that the dimension is $n = 1$. A univariate polynomial in $\mathbb{F}[x]$ has infinitely many zeros if and only if it is constant zero. Therefore $\mathsf{pol}(A)$

contains only the constant zero polynomial whenever A is infinite. This means that the algebraic closure of any infinite set $A \subseteq \mathbb{F}$ is the whole space $\mathbb{F}$. On the other hand, when A is finite, then there is a polynomial which vanishes exactly on the points from A, and therefore $\overline{A} = A$. For higher dimensions, an example of a closed set is the unit circle, because in this case the ideal $\mathsf{pol}(A)$ is generated by the polynomial $x^2 + y^2 = 1$. □

The following lemma is the key to deciding if a grammar generates only zero.

Lemma 11.5. *Let $\mathcal{G}$ be a polynomial grammar with nonterminals $\mathcal{X}$, and let η be a solution (not necessarily the least solution). Then $\overline{\eta}$, defined by $X \in \mathcal{X} \mapsto \overline{\eta(X)}$ is also a solution.*

Proof. We first prove two inclusions, (11.1) and (11.2), which show how closure interacts with polynomial images and Cartesian products. The first inclusion is about polynomial images:

$$p(\overline{A}) \subseteq \overline{p(A)} \qquad \text{for every } A \subseteq \mathbb{F}^n \text{ and polynomial } p : \mathbb{F}^n \to \mathbb{F}^m. \tag{11.1}$$

To prove the above inclusion, we need to show that if a polynomial vanishes on $p(A)$, then it also vanishes on $p(\overline{A})$. Suppose that a polynomial q vanishes on $p(A)$. This means that the polynomial $q \circ p$ vanishes on A, which means that $q \circ p$ also vanishes on $\overline{A}$, by definition of closure. Therefore q, vanishes on $p(\overline{A})$. The second inclusion is about Cartesian products:

$$\overline{A} \times \overline{B} \subseteq \overline{A \times B} \qquad \text{for every } A \subseteq \mathbb{F}^n \text{ and } B \subseteq \mathbb{F}^m. \tag{11.2}$$

We need to show that if a polynomial vanishes on $A \times B$, then it also vanishes on $\overline{A} \times \overline{B}$. Suppose that q vanishes on $A \times B$. Take some $b \in B$. The polynomial $q(\_, b)$ vanishes on A, and therefore it vanishes on $\overline{A}$ by definition of closure. Therefore, q vanishes on $\overline{A} \times B$. Applying the same reasoning again, we get that q vanishes on $\overline{A} \times \overline{B}$, proving the inclusion (11.2).

We are now ready to prove the lemma. Let η be some solution to the grammar. Take some rule $X \leftarrow p(X_1, \ldots, X_n)$ in the grammar $\mathcal{G}$. The following shows that $\overline{\eta}$ is compatible with the rule, and therefore $\overline{\eta}$ is a solution to the grammar by arbitrary choice of the rule.

$$\begin{aligned}
p(\overline{\eta}(X_1),\ldots,\overline{\eta}(X_n)) &= \text{by definition of } \overline{\eta}\\
p(\overline{\eta(X_1)},\ldots,\overline{\eta(X_n)}) &\subseteq \text{repeated application of (11.2)}\\
p(\overline{\eta(X_1)\times\cdots\times\eta(X_n))} &\subseteq \text{by (11.1)}\\
\overline{p(\eta(X_1)\times\cdots\times\eta(X_n))} &\subseteq \text{because } \eta \text{ is solution and closure is monotone}\\
\overline{\eta(X)} &= \text{by definition of } \overline{\eta}\\
\overline{\eta}(X) &
\end{aligned}$$

This completes the proof of the lemma. Note that the proof is not very specific to polynomials and definable reals, and it would work for more abstract notions of algebra, as will be defined in Section 11.1. ∎

We now complete the proof of Theorem 11.3. We use two semi-algorithms, as in Chapter 9. By enumerating derivations, there is an algorithm that terminates if and only if the grammar generates some nonzero vector. We now give an algorithm that terminates if and only if the grammar generates a language contained in $\{0\}$. The algorithm simply enumerates through all *closed solutions* to the grammar, i.e. solutions which map each nonterminal to a closed set. By Lemma 11.5, the generated language is contained in $\{0\}$ if and only if there is a an assignment η which maps nonterminals to closed sets such that:

1. η is a solution to the grammar; and

2. η maps the starting nonterminal to a subset of $\{0\}$.

We assume that a closed set A is represented by a finite basis of the ideal $\mathsf{pol}(A)$. By Hilbert's Basis Theorem, we can enumerate candidates for η, by using finite sets of polynomials to represent closed sets. It remains show that, given η, one can check if conditions 1 and 2 above are satisfied. To do this, we use decidability of the first-order theory of the reals from Theorem 10.1, although there are more efficient algorithms that have been developed in the area computational algebraic geometry, see [7, Theorem 11].

Lemma 11.6. *Given a set of variables X and two finite sets $P, Q \subseteq \mathbb{F}[X]$ of polynomials, one can decide if* $\mathsf{zero}(P) \subseteq \mathsf{zero}(Q)$.

Proof. The question can be formalised in first-order logic; but there is a slightly subtle point, which needs to be explained. Let us write

$$\mathsf{zero}_{\mathbb{R}}(P) \supseteq \mathsf{zero}_{\mathbb{F}}(P)$$

for the set of tuples of reals (respectively, definable reals) where all polynomials from P vanish. Even if the coefficients of P are definable reals, the two sets above are typically not the same (although they would be the same if there would be only one variable in X). The question in the statement of the lemma is

$$\mathsf{zero}_{\mathbb{F}}(P) \subseteq \mathsf{zero}_{\mathbb{F}}(Q) \tag{11.3}$$

while a call to Theorem 10.1 allows us to decide the answer to the question

$$\mathsf{zero}_{\mathbb{R}}(P) \subseteq \mathsf{zero}_{\mathbb{R}}(Q). \tag{11.4}$$

However, the answers to the questions (11.3) and (11.4) are the same. The reason is that the symmetric difference

$$\mathsf{zero}_{\mathbb{R}}(Q) \oplus \mathsf{zero}_{\mathbb{R}}(Q).$$

is a definable set of reals, i.e. it can be defined in first-order logic. Every definable set of reals contains at least one definable real; which is most easily proved assuming that the definition is quantifier-free. ■

From the above lemma, it follows that inclusion on closed sets is decidable, and hence condition 2 is decidable. Condition 1 boils down to testing a finite number of inclusions of the form

$$p^{-1}(A) \supseteq A_1 \times \cdots \times A_n$$

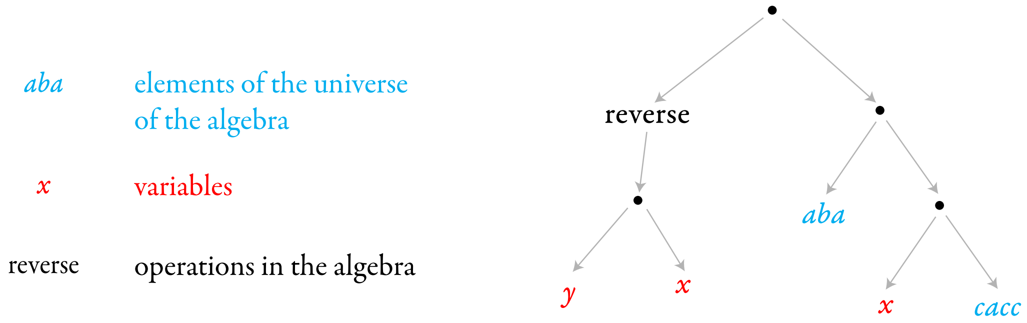
for closed sets $A, A_1, \ldots, A_n$ and some polynomial p. In Exercise 121 we show that closed sets are (effectively, using our representation) closed under products and inverse images of polynomials, which completes the algorithm.

11.1 *Application to equivalence of register automata*

In this section, we use Theorem 11.3 to decide equivalence for automata which use registers to manipulate values in certain kinds of algebras. In this section, we use the notion of algebra from universal algebra, i.e. an *algebra* is defined to be a set equipped with some operations. Examples include

$$(\mathbb{F}, +, \times) \qquad (\{a, b\}^*, \cdot).$$

We adopt the convention that boldface letters like **A** and **B** range over algebras, and the universe (the underlying set) of an algebra **A** is denoted by A. Define a polynomial with variables X in an algebra **A** to be a term built out of operations from the algebra, variables from X and elements of the universe. Here is a picture of a polynomial with variables $\{x, y\}$ in an algebra where the universe is $\{a, b, c\}^*$ and the operations are concatenation (binary) and reverse (unary):


aba
elements of the universe of the algebra
x
variables
reverse
operations in the algebra
reverse
y
x
aba
x
cacc

We write $\mathbf{A}[X]$ for the set of polynomials over variables X in the algebra **A**. Such a polynomial represents a function of type $A^X \to A$ in the natural way. We extend this notion to function of type $A^n \to A^m$ by using m-tuples of polynomials with n variables.

Example 21. If the algebra is $(\mathbb{F}, +, \times)$, then the polynomials are the polynomials in the usual sense, e.g. a binary polynomial is

$$x^2 + 3x^3y^4 + 7.$$

If we choose the algebra to be $\mathbb{F}$ with the operations being addition and the family of scalar multiplications $\{x \mapsto ax\}_{a \in \mathbb{F}}$, then the polynomials are exactly the affine functions. □

Definition 11.7 (Register automaton). *Let* $\mathbf{A}$ *be an algebra. The syntax of a* register automaton over $\mathbf{A}$ *consists of:*

- *a finite input alphabet* Σ*;*
- *a finite set* R *of registers;*
- *a finite set* Q *of states;*
- *an initial configuration in* $Q \times A^R$*;*
- *a transition function* $\delta : Q \times \Sigma \to Q \times (\mathbf{A}[R])^R$*;*
- *an output dimension* n *and an* output function $F : Q \to (\mathbf{A}[R])^n$.

The semantics of the automaton is a function of type $\Sigma^* \to A^n$ *defined as follows. The automaton begins in the initial configuration. After reading each letter, the configuration is updated according to*

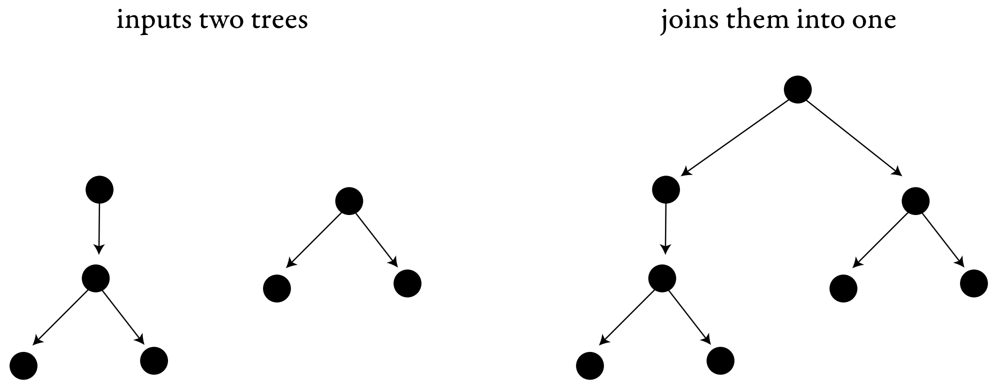
$$(q, v) \overset{a}{\mapsto} (p, f(v)) \qquad \textit{where } \delta(q, a) = (p, f).$$

If the configuration after reading the entire input is (q, v)*, then the output of the automaton is obtained by applying* $F(q)$ *to* v*.*

Example 22. A language $L \subseteq \Sigma^*$ is regular if and only if its characteristic function $\Sigma^* \to \{0, 1\}$ is recognised by a register automaton with no registers over the algebra with universe $\{0, 1\}$ and no operations. □

Example 23. Let Σ be a finite alphabet. Here is a register automaton over the algebra $(\Sigma^*, \cdot)$ which implements the reverse function $\Sigma^* \to \Sigma^*$. The automaton has one register, call it x and one state. When it reads a letter $a \in \Sigma$, it executes the register update given by the polynomial $a \cdot x$. The output function is the identity. □

Example 24. Consider an algebra **A** where the universe is the set of trees (viewed as directed graphs, with edges directed away from the root), and which has one binary operation depicted as follows:



It is not difficult to write a register automaton over the algebra **A**, with input alphabet $\{a\}$, which maps a word a^n to a balanced binary tree of depth n. □

The following result is a direct corollary of Theorem 11.3. The result was first shown in [6, Theorem 4], were the proof was also based on the Hilbert Basis Theorem.

Theorem 11.8. *The following problem is decidable:*

- **Input.** *Two functions $\Sigma^* \to \mathbb{F}^n$ given by register automata over $(\mathbb{F}, +, \times)$;*
- **Question.** *Are the functions equal?*

The above theorem generalises Theorem 8.7, because weighted automata can be viewed as a special case of register automata over $(\mathbb{F}, +, \times)$ where only linear maps are allowed as the register updates, instead of arbitrary polynomials, as allowed in Theorem 11.8. The generalisation of Theorem 8.7 is only in terms of decidability, since the running time of the algorithm for Theorem 11.8 is not estimated in any way, not to mention polynomial time. In fact, a lower bound of Ackermann time is given in [6, Theorem 1].

Proof. Suppose that the input functions are f, g. By doing a natural product construction, we can compute a register automaton that recognises the difference function $f - g$. Therefore, the problem boils down to deciding if a function h, e.g. the difference, is constant equal to zero. For this we use a grammar. Suppose that h is recognised by an register automaton with states Q and n registers. Define a grammar where the nonterminals are

$$\underbrace{Q}_{\text{dimension } n} + \underbrace{1}_{\text{dimension 1}} .$$

The starting nonterminal is 1. By copying the transitions of the automaton, we can write the rules of the grammar so that nonterminal q generates exactly those tuples $\bar{a} \in \mathbb{F}^n$ such that configuration $(q, \bar{a})$ can be reached over some input. By using the output function of the automaton, we can ensure that the starting nonterminal produces exactly the outputs of the automaton. Therefore, the language defined by the grammar is contained in $\{0\}$ if and only if the automaton can only produce 0, and the former is decidable by Theorem 11.3. ∎

Note that the above theorem, with the same proof, would also work for tree register automata, i.e. a generalisation of register automata for inputs that are trees.

Other algebras. In Theorem 11.8, we showed that equivalence is decidable for register automata over the algebra $(\mathbb{F}, +, \times)$. From this we can infer decidability for some other algebras, by coding them into rational numbers according to the following definition.

Definition 11.9. *Let* **A** *and* **B** *be algebras. We say that* **A** *can be simulated by polynomials of* **B** *(no relation to polynomial time computation) if there is some dimension n and an injective function*

$$\alpha : A \to B^n$$

with the following property. For every operation $f : A^m \to A$ in the algebra **A**, *there is a polynomial $g : B^{m\times n} \to B^n$ of* **B** *which makes the following diagram commute*

$$\begin{tikzcd}
A^m \arrow[r, "(\alpha,\dots,\alpha)"] \arrow[d, "f"'] & B^{m\times n} \arrow[d, "g"] \\
A \arrow[r, "\alpha"'] & B^n
\end{tikzcd}$$

It is easy to see that if **A** can be simulated by polynomials of **B**, then decidability of equivalence of register automata over **B** implies decidability equivalence of register automata over **A**.

Corollary 11.10. *For every finite alphabet Σ, the equivalence problem is decidable for register automata over the algebra $(\Sigma^*, \cdot)$.*

Proof. By Theorem 11.8, it suffices to show that $(\Sigma^*, \cdot)$ can be simulated by polynomials of $(\mathbb{F}, +, \times)$. The proof is the same as in Lemma 8.10. Assume without loss of generality that Σ is $\{0, \dots, n-1\}$. We use the coding:

$$a_i a_{i-1} \cdots a_0 \in \Sigma^* \qquad \overset{\alpha}{\mapsto} \qquad (n^i, \quad \underbrace{a_i n^i + \cdots a_0 n^0}_{\text{input as a number in base } n} \quad) \in \mathbb{F}^2$$

The unique operation of the algebra $(\Sigma^*, \cdot)$, namely string concatenation, is encoded by the polynomial

$$((a,b),(a',b')) \quad \mapsto \quad (a \times a', b \times a' + b')$$

By the remarks after Theorem 11.8, the decidability result would extend to tree automata over the algebra $(\Sigma^*, \cdot)$. This yields the result that equivalence is decidable for tree-to-string transducers, as considered in [51].

■

Problem 110. Which of the following structures are rings:

1. $(\mathbb{N}, +, \cdot, 0)$;

2. $(\mathbb{Z}, +, \cdot, 0)$;

3. $(\mathbb{R}, +, \cdot, 0)$;
4. $(\{0\}, +, +, 0)$;
5. $(\mathbb{Z}[x_1, \ldots, x_n], +, \cdot, 0)$;
6. $(\mathbb{Q}[x_1, \ldots, x_n], +, \cdot, 0)$;
7. $(\mathbb{Z}, \max, +, 0)$;
8. $(\mathbb{N}, \max, +, 0)$;
9. $(S_n, \cdot, \cdot, id)$;
10. $(\mathbb{Z}_n, +, \cdot, 0)$ for $n \in \mathbb{N}$.

Problem 111. Let $(R, +, \cdot, 0)$ be a ring. A subset $I \subseteq R$ is called an *ideal* if the following two conditions hold: 1) for all $i, j \in I$ it holds $i + j \in I$, 2) for all $i \in I, r \in R$ it holds $i \cdot r, r \cdot i \in I$. Find all ideals in the following rings:

1. $(\mathbb{Z}, +, \cdot, 0)$;
2. $(\mathbb{Q}, +, \cdot, 0)$.

Generalize the second case to any field.

Problem 112. A ring congruence is an equivalence relation $\equiv$ such that $x_1 \equiv y_1$, $x_2 \equiv y_2$ implies $x_1 * x_2 \equiv y_1 * y_2$ for $* \in \{+, \cdot\}$. For an ideal $I \subseteq R$ and $r_1, r_2 \in R$ we say that $r_1 \equiv_I r_2$ if $r_1 - r_2 \in I$. In case of the ring of integers all the $\equiv_I$ are actually $\equiv_n$, so in particular ring congruences. Show that for every ideal I, the relation $\equiv_I$ is a ring congruence.

Problem 113. Show that every ideal in $\mathbb{Q}[x]$ is generated by one element.

Problem 114. Is every ideal in the following rings generated by one element:

1. $\mathbb{Z}[x]$?
2. $\mathbb{Q}[x, y]$?

Problem 115. Is there a constant $c \in \mathbb{N}$ such that every ideal in $\mathbb{Z}[x]$ is generated by at most c elements?

Problem 116. Show that for every ring R the following conditions are equivalent:

1. every ideal in R is finitely generated;

2. every growing sequence of ideals $I_1 \subsetneq I_2 \subsetneq \ldots$ is finite.

Problem 117. Prove the Hilbert's Basis Theorem in the following formulation: if R is a ring where every ideal in R is finitely generated, then also every ideal in $R[x]$ is finitely generated.

Problem 118. Show that for every set $A \subseteq \mathbb{F}$ the set $\mathsf{pol(A)}$ is an ideal.

Problem 119. Consider the closure operation from $\mathbb{P}(\mathbb{Q})$ to $\mathbb{P}(\mathbb{Q})$ defined for $A \subseteq \mathbb{Q}$ as $\overline{A} = \mathsf{zero(pol(A))}$. Show that the following conditions are true for every $A, B \subseteq \mathbb{Q}$:

- $A \subseteq \overline{A}$;

- if $A \subseteq B$ then $\overline{A} \subseteq \overline{B}$;

- $\overline{A} = \overline{\overline{A}}$.

Problem 120. Consider the field of rational numbers $\mathbb{Q}$. Show that for every finite set of variables X and every ideal $I \subseteq \mathbb{Q}[X]$, there is an ideal $J \subseteq \mathbb{Q}[x]$ generated by a single polynomial such that $\mathsf{zero}(I) = \mathsf{zero}(J)$.

Problem 121. Assume that a closed set $A \subseteq \mathbb{F}^n$ is represented by a finite basis for the ideal $\mathsf{pol}(A)$. Show that closed sets are effectively closed under products and inverse images of polynomials, i.e. if A, B are closed and p is a polynomial, then the sets $A \times B$ and $p^{-1}(A)$ are closed, and their representations can be computed.

Problem 122. Show that the following problem is decidable: given a polynomial grammar and a finite set $X \subseteq \mathbb{Q}$, decide if the language generated by the grammar is equal to X.

12
Parsing in matrix multiplication time

The classical dynamic CYK algorithm for parsing context-free grammars runs in cubic time (in terms of the input word). In this chapter we present a parsing algorithm of Valiant [59], which parses context-free languages in approximately the same time as (Boolean) matrix multiplication. For readers who do not like matrices, multiplying two $n \times n$ Boolean matrices is the same as computing the composition $R \circ S$ of two binary relations $R, S \subseteq \{1, \ldots, n\}^2$. The naive algorithm for this problem runs in time n^3, but smarter algorithms run faster, e.g. the Strassen algorithm runs in time approximately $\mathcal{O}(n^{2.8704})$, and the record holder as of 2017 is $\mathcal{O}(n^{2.3727})$, see [60]

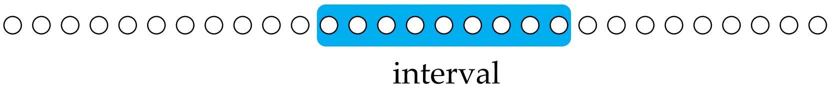
Theorem 12.1. *Assume that multiplication of $n \times n$ Boolean matrices can be computed in time $\mathcal{O}(n^\omega)$ for some real number ω. Then membership in a context-free language can be decided in time at most*

$$\mathrm{poly}(\mathcal{G}) \cdot n^\omega \cdot \log^2(n) \qquad \textit{where } \mathcal{G} \textit{ is the grammar and } n \textit{ is the length of the input.}$$

For the rest of this chapter, fix ω and a context-free grammar $\mathcal{G}$. We assume that the grammar is in Chomsky Normal Form, i.e. every rule is of the form $X \leftarrow YZ$ or $X \leftarrow a$, where X, Y, Z are nonterminals and a is a terminal. A grammar can be converted into Chomsky Normal Form in polynomial time, so this assumption can be made without loss of generality. Define a length *n parse* *matrix* to be a family

$$M = \{M_X\}_{X \text{ is a nonterminal}}$$

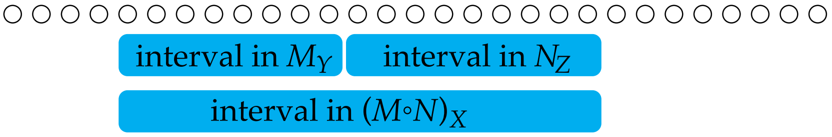
such that each M_X is a family of intervals in $\{1, \ldots, n\}$. Here an interval is a set of numbers connected by the successor relation, like this:



The intuition is that, given an input word, we put into M_X all those intervals that correspond to infixes of the input word which can be generated by nonterminal X. For parse matrices M, N of same length, define their product $M \circ N$ by

$$(M \circ N)_X \overset{\text{def}}{=} \bigcup_{X \to YZ} M_Y \circ M_Z$$

where the union ranges over rules of the grammar and $M_Y \circ M_Z$ is defined to be the family of intervals that can be decomposed as a disjoint union of an interval from M_Y followed by an interval from M_Z, as in the following picture:



A family of intervals contained in $\{1, \ldots, n\}$ can be seen as the binary relation over $\{0, \ldots, n\}$, where an interval $\{i, \ldots, j\}$ is represented as a pair $(i-1, j)$. Under this representation, product of sets of intervals is the same as composing binary relations, which in turn is the same as multiplying Boolean matrices. Hence, we get the following observation.

Lemma 12.2. *For length n parse matrices, product can be computed in time $\mathcal{O}(n^\omega)$.*

We say that a parse matrix M is *closed* if it satisfies $M \circ M \subseteq M$, and we say that it is closed on an interval $I \subseteq \{1, \ldots, n\}$ if it is closed when restricted to intervals contained in I. For a parse matrix M, define its *closure* M^* to be the least (with respect to inclusion) parse matrix that contains M and is closed.

Proposition 12.3. *There is an algorithm which runs in time, call it $T(n)$, at most*

$$\text{poly}(\mathcal{G}) \cdot \log(n) \cdot n^{\omega}$$

and which computes the closure of a length $2n$ parse matrix, assuming that it is closed on the intervals $\{1, \ldots, n\}$ and $\{n+1, \ldots, 2n\}$.

Before proving the proposition, we show how it implies the Theorem 12.1.

Proof of Theorem 12.1. Suppose that we want to know if the grammar $\mathcal{G}$ generates a word w of length n. Define M to be the length n parse matrix where M_X contains intervals $\{i\}$ such that nonterminal X generates the i-th letter of w, using a rule of the form $X \to a$. This parse matrix can be computed in time linear in n. The word w is generated by the grammar if and only if the closure M^* contains the interval $\{1, \ldots, n\}$ on the component corresponding to the starting nonterminal. It suffices therefore to compute the closure M^*. To make the computation easier, suppose that the length of the word is a power of two, i.e. $n = 2^k$. We do a divide an conquer approach: we compute the closures of the parse matrix for the first and second halves of w (using a recursive procedure), and then combine these using the algorithm from Proposition 12.3. The running time of this algorithm is at most

$$T(n) + 2T(\frac{n}{2}) + \cdots + 2^k T(\frac{n}{2^k}). \tag{12.1}$$

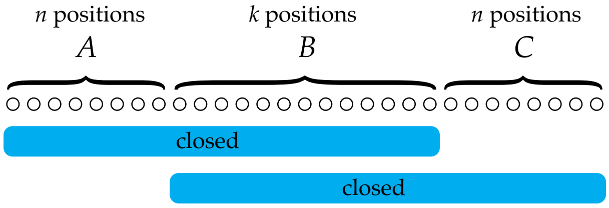
Because $T(n)$ is at least linear, it follows that

$$2^i T(\frac{n}{2^i}) \leq T(n),$$

which shows that the running time (12.1) is at most $\log n$ times slower than $T(n)$, thus proving the theorem, given the bounds on $T(n)$ from Proposition 12.3. ∎

It remains to prove the proposition. We use the following lemma.

Lemma 12.4. *Suppose that M is a length $k+2n$ parse matrix that is closed on the intervals $A \cup B$ and $B \cup C$ as depicted below:*



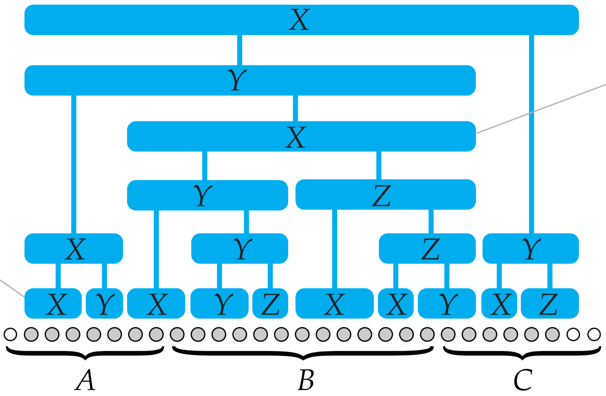
Then the closure M^* can be computed in time $\mathrm{poly}(\mathcal{G}) \cdot n^\omega + T(n)$.

Proof. Define N to be $M \cup M \circ M$ restricted to intervals that contain B or are disjoint with B. The main observation in the lemma is

$$M^* = M \cup N^*, \tag{12.2}$$

where the sum above is component-wise (recall that a parse matrix is a family of sets of intervals). Before proving the above equality, we note that the right side of the above equality can be computed in time as in the statement of the lemma, thus proving the lemma. By Lemma 12.2, N can be computed in time $\mathrm{poly}(\mathcal{G}) \cdot n^\omega$. Because the matrix M was closed over intervals A and C, it follows that N is also closed over these intervals. Since all entries of N contain B or are disjoint with B, it is essentially a matrix of length $2n$ whose first and second halves are closed. It follows that N^* can be computed in time $T(n)$.

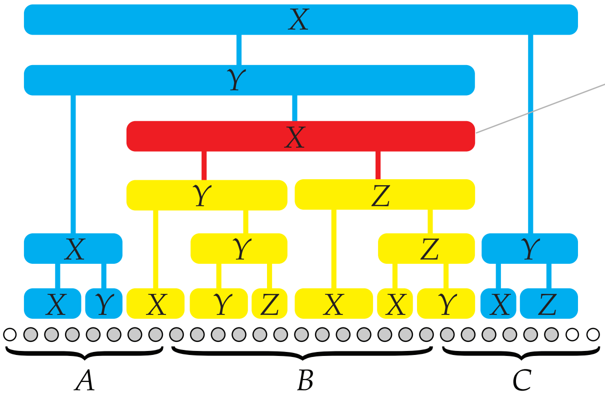
It remains to prove the equality (12.2). The inclusion $\supseteq$ is immediate, it remains to justify the inclusion $\subseteq$. We need to show that if M^* contains interval I on nonterminal X, then this is true for $M \cup N^*$. If I is contained in $A \cup B$ or $B \cup C$, then this implication holds by the closure assumptions on M. The remaining case is when I contains B. The reason for M^* containing I on nonterminal X is a parse tree as described in the following picture:



a non-leaf with label X represents an interval on component X of M^*

a leaf with label X represents an interval on component X of M

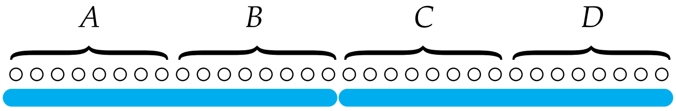
In the parse tree, use red consider the smallest interval which contains B, and use yellow for the descendants of the red interval:



smallest interval which contains B

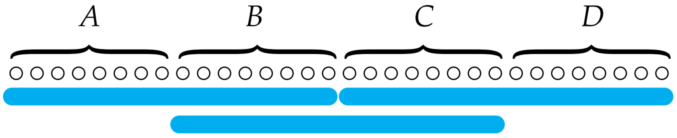
By minimality, each yellow interval is contained in either $A \cup B$ or $B \cup C$, and therefore belongs to M by the closure assumptions on M. Therefore, the red itself belongs to $M \circ M$. The red interval contains B, and the blue intervals are disjoint with B, therefore the red and blue intervals are in N. It follows that the red and blue intervals form a parse tree corresponding to the matrix N^*. ■

Proof of Proposition 12.3. Here is the algorithm. Suppose that M is a length $2n$ parse matrix which is closed on its first and second halves, as in the statement of the proposition. Let us write A, B, C, D for the intervals describing the four quarters of $2n$, as in the following picture:

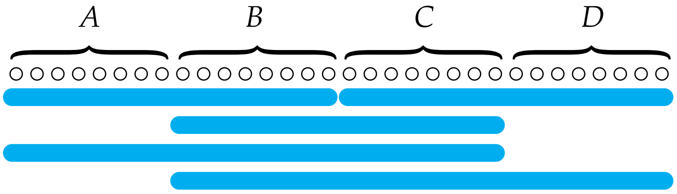


As in Lemma 12.4, the blue rectangles indicate the intervals which are closed.

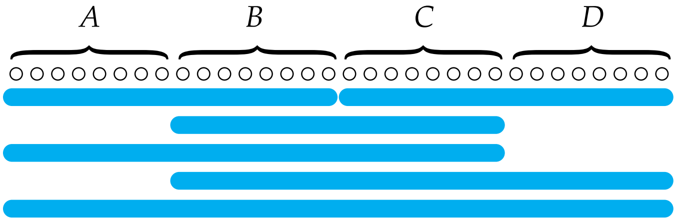
1. By induction, compute the closure of the interval $B \cup C$:



2. Using Lemma 12.4 twice, compute the closures of $A \cup B \cup C$ and $B \cup C \cup D$:



3. Using Lemma 12.4, compute the closure of $A \cup B \cup C \cup D$:



The cost of the above procedure is:

$$T(n) = \underbrace{T(n/2)}_{\text{step 1}} + \underbrace{2 \cdot (T(n/2) + c \cdot n^{\omega})}_{\text{step 2}} + \underbrace{T(n/2) + c \cdot n^{\omega}}_{\text{step 3}}$$

for some c polynomial in the grammar. Summing up,

$$T(n) = 4T(n/2) + 3c \cdot n^{\omega}.$$

Reasoning as in the end of the proof of Theorem 12.1, we get

$$T(n) = 3c \cdot n^{\omega} + 4 \cdot 3c \cdot (\frac{n}{2})^{\omega} + \cdots + 4^k \cdot 3c \cdot (\frac{n}{2^k})^{\omega}.$$

Because n^{ω} is at least quadratic (an algorithm for matrix multiplication must at least read two $n \times n$ matrices), it follows that

$$2^i \cdot (\frac{n}{2^i})^{\omega} \leq n^{\omega},$$

which gives the bound in the proposition. ■

Problem 123. Show that the operation $M \circ N$ is not associative.

Problem 124. Design an algorithm, which for an undirected graph G with n vertices answers whether there exists a subgraph of G, which is

1. a triangle, in time $\mathcal{O}(n^{\omega})$;

2. a cycle with 4 vertices, in time $\mathcal{O}(n^{\omega})$;

3. a cycle with k vertices, in time $\mathcal{O}(n^{\omega})$;

4. a clique with 4 vertices, in time $\mathcal{O}(n^{1+\omega})$;

5. a clique with 5 vertices, in time $\mathcal{O}(n^{2+\omega})$;

6. a clique with 6 vertices, in time $\mathcal{O}(n^{2\omega})$;

7. a clique with $3k$ vertices, in time $\mathcal{O}(n^{k\omega})$.

Problem 125. Design an algorithm, which for an undirected graph $G = (V, E)$ with $3n$ vertices answers whether there exists a subset $S \subseteq V$ with $|E(S, V - S)| \geq k$ in time $\mathcal{O}(2^{n\omega} \cdot \mathrm{poly}(n))$ (by $E(A, B)$ we denote the set of all the edges with one endpoint in A and another endpoint in B).

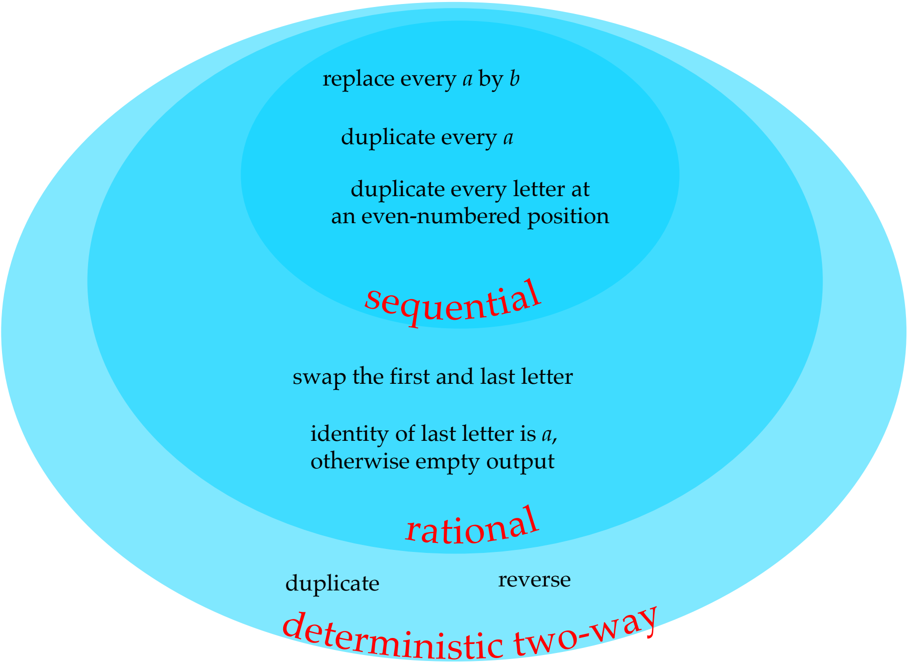
Problem 126. Let $U, V \subseteq \mathbb{N}^d$ be sets of d-dimensional vectors, each one with n vectors. Show that for $n \geq d$ one can check whether there are $u \in U$ and $v \in V$ such that $u \perp v$ in time $o(n^2 d)$.

Problem 127. Design an algorithm, which multiplies two matrices of size $n \times n$ in time $\mathcal{O}(n^{\log_2 7})$.

13

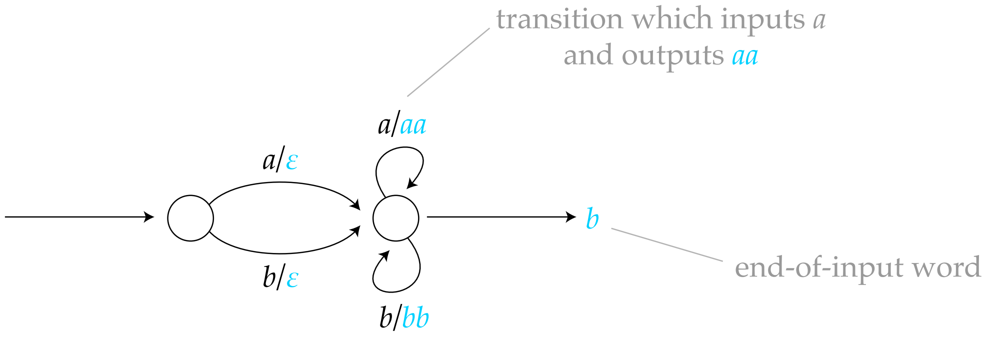
Two-way transducers

In this chapter, we talk about transducers, i.e. automata that input words and output words. We cover three families of transducers as shown below:


replace every a by b
duplicate every a
duplicate every letter at an even-numbered position
sequential
swap the first and last letter
identity of last letter is a, otherwise empty output
rational
duplicate
reverse
deterministic two-way

13.1 Sequential functions

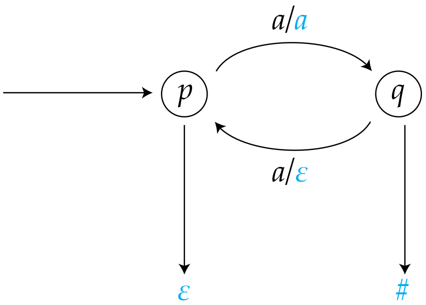
Recall the definition of a *nondeterministic finite automaton with output* from Definition 8.11. This is an NFA where every transition is labelled by a (possibly empty) *output word* over a designated output alphabet, and every final state is labelled by a (possibly empty) *end-of-input word*, also over the output alphabet. Here is an example:



The output of a run is obtained by concatenating the output words of all transitions used, followed by the end-of-input word of the last state used. The semantics of the automaton is defined to be the function which maps an input word to the multiset of words over the output alphabet that are produced by accepting runs (if the same output is produced by n different accepting runs, then it appears n times in the output multiset).

The automaton in the picture above has the following outputs: if the input word is empty, then the output multiset is empty; if the input word is nonempty, then the automaton produces exactly one output (i.e. a multiset with one word) which is obtained from the input by deleting the first letter, doubling the other letters, and appending b to the end.

Define a DFA *with output* to be the special case of an NFA with output where: (a) the transition relation is a deterministic, i.e. for every state there is a unique outgoing transition for each input letter; and (b) all states are final. Under these assumptions, the automaton produces exactly one output for every input, and therefore its semantics can be viewed as a function from words over the input alphabet to words over the output alphabet. Any function obtained this way is called a *(left-to-right) sequential function*[1] . Here is an example:



The transducer above erases a's at even-numbered positions, and appends # or nothing to the output, depending on the parity of the input length. Other examples of left-to-right sequential functions include: "erase all appearances of letter a" or "erase all appearances of letter a at even-numbered input positions".

Define a *right-to-left sequential function* symmetrically: the syntax is the same, except that in the semantics, the input letters are read from right to left, and the end-of-input word is produced after reading the leftmost position. The function "identity if the input ends with a, otherwise empty output" is a right-to-left sequential function but not a left-to-right sequential function.

[1] The name sequential is used for at least four transducer models in the literature, starting with the original transducer models described by Shannon [52, Section 8] and later developed by Moore [39] and Mealy [38]. Both the Moore and Mealy models – which are two non-equivalent models of letter-to-letter transducers – were called sequential by their authors. In those days, sequential seems to have been a synonym for "recognised by an automaton". Then, Ginsburg introduced a model, called submachines, that could produce words (and not just letters) in transitions [30]. Soon Ginsburg's model started to be called sequential, see e.g. [25, p. 298]. Then, Schützenberger extended submachines with end-of-input words [50]. Now it is Schützenberger's model – originally called subsequential – that is being called sequential, e.g. [28], and this is the convention that we adopt here.

13.2 Rational functions

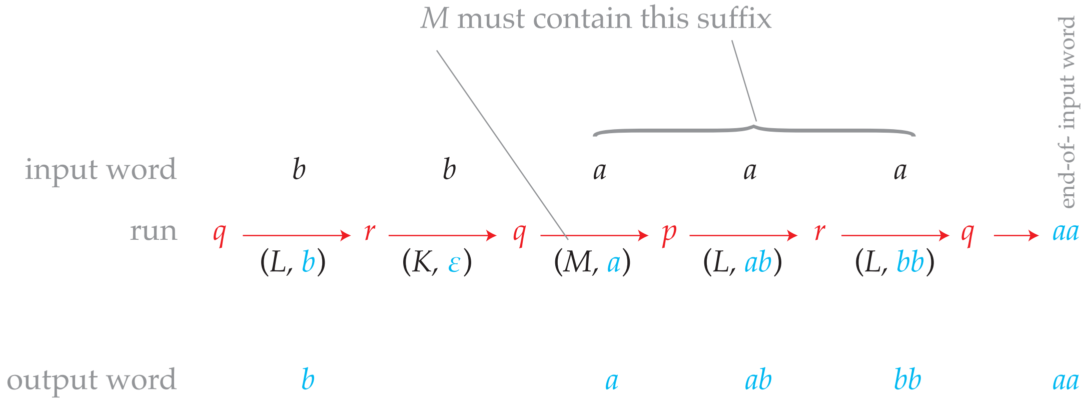
We now move to a richer class of functions from words to words, called the *rational functions*[2]. This class admits several equivalent definitions; we give five. Another advantage is that the class is symmetric, i.e. there is no need to define "right-to-left rational functions". We begin with two definitions that use NFA's with output.

Functional and unambiguous NFA's with output. We say that an NFA with output is *functional* if for every input word, the output multiset contains exactly one word, but possibly with multiplicities. In other words, there might be several accepting runs, but all accepting runs produce the same output word, and there is always at least one accepting run. We say that an NFA with output is *unambiguous* if for every input word, the output multiset contains exactly one word, used exactly one time. In other words, for every input there is exactly one accepting run. Functional, and therefore also unambiguous, NFA's with output can be viewed as recognising functions from words to words, by mapping an input word to the unique output word in the output multiset. Functional NFA's with output are essentially the same as the original definition of rational functions given by Eilenberg in [25, Chapter IX].
We will later show that – when viewed as recognisers of functions from words to words (without multiplicities of outputs) – functional and unambiguous automata have the same expressive power, i.e. nothing is gained by using functional but possibly ambiguous NFA's with output.

Lookahead DFA with output. A *lookahead* NFA *with output* is a model that extends an NFA with output as follows: instead of pairs (input letter, word over the output alphabet), the transitions are pairs (regular language over the input alphabet, word over the output alphabet). A transition labelled by a pair (L, w)

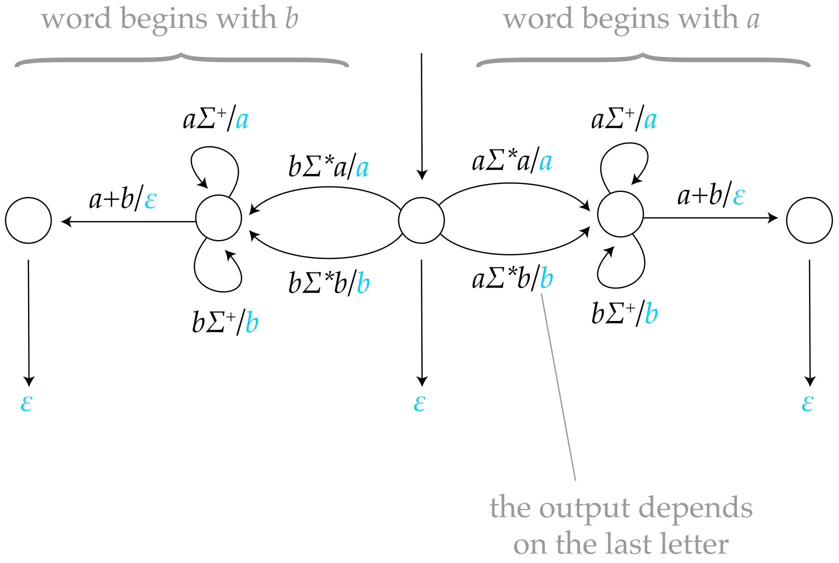
[2] The name rational comes from Eilenberg. Eilenberg introduced rational subsets of any monoid [25, Chapter VII], which covers the special case of rational relations [25, Chapter IX] defined as rational subsets of monoids of the form $\Sigma^* \times \Gamma^*$, which in turn covers the special case of *rational functions* which are functional rational relations.

can be applied if the unread part of the input belongs to L; the effect of using such transition is that w gets added to the output and one input letter is consumed. Here is a picture of a run:



A *lookahead* DFA *with output* is the special case where (a) for every state, the regular languages labelling outgoing transitions form a partition of all nonempty words; and (b) every state is final.

Example 25. The following lookahead DFA with output swaps the first and last letters:

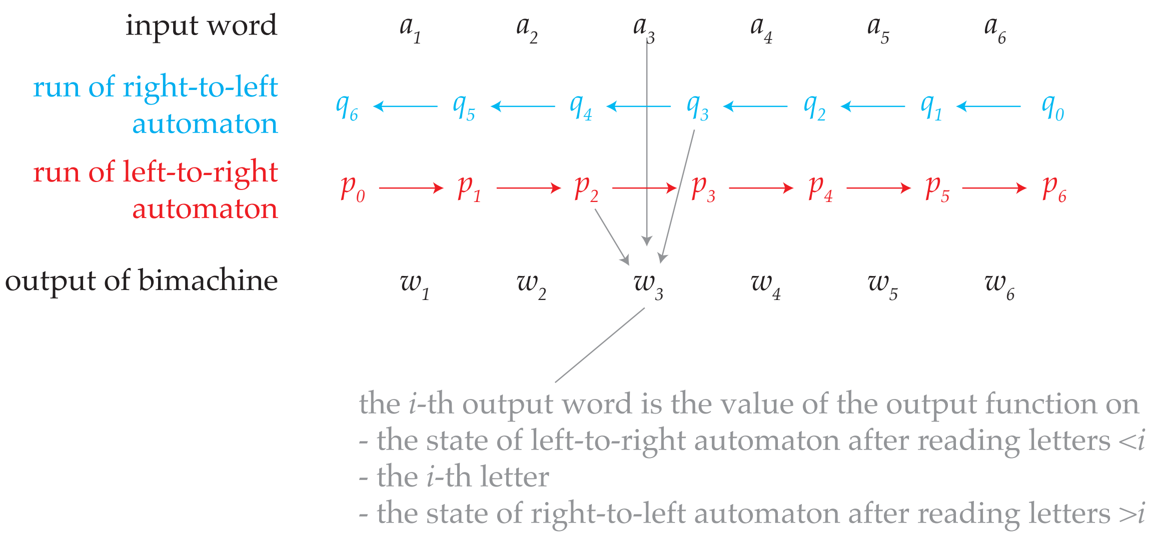


□

Eilenberg bimachine. We now present Eilenberg bimachines, which are essentially another syntax for lookahead DFA with output. An *Eilenberg bimachine* [25, Chapter XI.7] consists of two finite automata $\mathcal{A}, \mathcal{B}$ over the input alphabet – with $\mathcal{A}$ left-to-right deterministic and $\mathcal{B}$ right-to-left deterministic – as well as an output function of type

$$\text{states of } \mathcal{A} \times \text{input alphabet} \times \text{states of } \mathcal{B} \quad \to \quad (\text{output alphabet})^*.$$

In the automata $\mathcal{A}, \mathcal{B}$ the final states are irrelevant and can be omitted from the syntax. The semantics of the bimachine is defined as follows. Given a nonempty input word, define for each position in the input word an output word as described in the following picture:


input word
a_1 a_2 a_3 a_4 a_5 a_6
run of right-to-left automaton
q_6 q_5 q_4 q_3 q_2 q_1 q_0
run of left-to-right automaton
p_0 p_1 p_2 p_3 p_4 p_5 p_6
output of bimachine
w_1 w_2 w_3 w_4 w_5 w_6
the i-th output word is the value of the output function on
- the state of left-to-right automaton after reading letters $<i$
- the i-th letter
- the state of right-to-left automaton after reading letters $>i$

The output of the bimachine is defined to be the concatenation of the output words, in the order inherited from the input positions. To deal with empty inputs, an Eilenberg bimachine is equipped with an designated output word that is used for the empty input.

Equivalence of the models. The following theorem shows that all the models described above are equivalent. We use the name *rational function* for a word-to-word function that is defined by any one of the equivalent models in the theorem.

Theorem 13.1. *The following models are equivalent, in terms of the functions from words to words that they define:*

1. *functional* NFA *with output;*
2. *lookahead* DFA *with output;*
3. *unambiguous* NFA *with output.*
4. *Eilenberg bimachines.*
5. *compositions of right-to-left sequential functions with left-to-right sequential functions.*

Proof sketch.

$1 \subseteq 2$ Consider a functional NFA with output $\mathcal{A}$. We define an equivalent lookahead DFA as follows. The lookahead DFA computes some run of the functional NFA that can be extended to an accepting run. Each transition is chosen using the lookahead, to determine if it can be extended to an accepting run. If more than one transition can be chosen, some arbitrary tie-breaking mechanism is used.

$2 \subseteq 3$ Consider some lookahead DFA with output $\mathcal{A}$. We define an equivalent Eilenberg bimachine as follows. Let $\mathcal{B}$ be a right-to-left DFA (without output) that simultaneously recognises all the languages which are used in the transitions of $\mathcal{A}$, i.e. the lookahead languages. The simulating NFA with output guesses the runs of these two automata (the run for $\mathcal{B}$ is right-to-left, and the run for $\mathcal{A}$ is left-to-right, and depends on the run of $\mathcal{B}$). This guess is unambiguous, because the automata $\mathcal{A}$ and $\mathcal{B}$ are unambiguous.

$3 \subseteq 4$ Consider an unambiguous NFA with output $\mathcal{A}$. We define an equivalent Eilenberg bimachine as follows. The left-to-right automaton is a left-to-right powerset construction applied to states of $\mathcal{A}$, i.e. its states are sets of states in $\mathcal{A}$ and the transition function is defined by

$$P \cdot a = \{q : \text{the automaton } \mathcal{A} \text{ has a transition } p \overset{a/w}{\to} q \text{ for some } p \in P\}.$$

The right-to-left automaton is defined symmetrically, i.e. its transition function is defined by

$$a \cdot P = \{p : \text{the automaton } \mathcal{A} \text{ has a transition } p \overset{a/w}{\to} q \text{ for some } q \in P\}$$

The output function maps a triple (P, a, Q) to the unique output word w such that the automaton has a transition

$$p \overset{a/w}{\to} q \qquad p \in P, q \in Q.$$

This function is well defined by the assumption that $\mathcal{A}$ is unambiguous.

$4 \subseteq 1$ Consider an Eilenberg bimachine $\mathcal{A}$. We define an equivalent functional – in fact, unambiguous – NFA with output as follows. The states of the simulating automaton are pairs (state of the left-to-right automaton in $\mathcal{A}$, state of the right-to-left automaton $\mathcal{A}$). The transition relation is defined by

$$(q, ap) \overset{a/w}{\to} (qa, p)$$

w is the output word in the bimachine that is associated to the triple (q, a, p). This automaton is unambiguous by the determinism assumptions in the definition of a bimachine.

$2 \subseteq 5$ A right-to-left sequential function can label the input word with states of right-to-left automata recognising the lookahead, and a left-to-right sequential function can then simulate the DFA with lookahead.

$5 \subseteq 3$ Functional NFA with output are closed under compositions and generalise both left-to-right and right-to-left sequential functions.

■

13.3 *Deterministic two-way transducers*

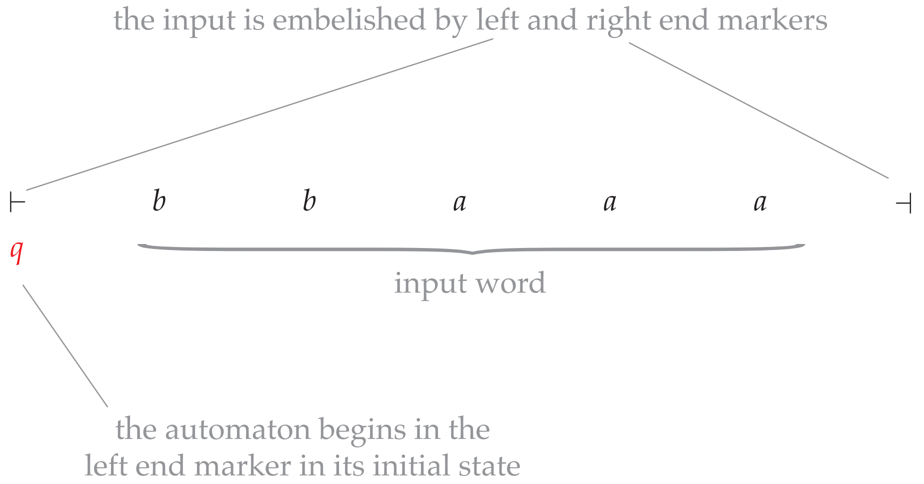
We now turn to the most powerful class of transducers discussed in this chapter, namely *deterministic two-way transducers*. In the next chapter, we will present an equivalent one-way model, which uses registers to store parts of the output.

Definition 13.2. *A* deterministic two-way transducer *consists of:*

- *finite* input and output alphabets Σ *and* Γ*;*
- *a finite set of* states Q *with a distinguished* initial *state;*
- *a transition function*

$$\delta : Q \times (\Sigma \cup \{\vdash, \dashv\}) \to \{accept\} \cup (Q \times \{left, stay, right\} \times \Gamma^*)$$

The semantics of the transducer are defined similarly to Turing machines. Actually, the model is equivalent to a Turing machine where there is one read-only input tape and one append-only output tape. The automaton begins in the following configuration:


the input is embelished by left and right end markers
⊢
b
b
a
a
a
⊣
q
input word
the automaton begins in the
left end marker in its initial state

(For two-way automata, the head is over a letter, as opposed to one-way automata, where the head is between letters.) At any given moment, the automaton applies its transition function to its current state and the symbol under the head. The result of the transition might be "accept", in which case the automaton ends its run, or a triple (state, direction, output word), in which case the new state is assumed, the head is moved in the direction, and the output word is appended to the output. The output letters are used in chronological order, i.e. those which are output at the beginning of the run are at the beginning of the output, regardless of the position of the head when executing the transition. The run of the automaton might fail, either by moving out of the word (i.e. moving left on the left marker or moving right on the right marker), or by entering an infinite computation that never sees a final state; such failing runs do not produce any output, and therefore the semantics of the automaton is a partial function from Σ^* to Γ^*.

Typical things that can be done using a two-way transducer are duplication or reversing the input. The main result of this chapter is that deterministic two-way automata are closed under composition.

Theorem 13.3 ([1, 19]). *Functions recognised by deterministic two-way transducers are closed under composition.*

For sequential and rational functions, closure under composition is done using a straightforward product construction. For two-way automata, the construction is much more challenging, since the automata begin composed might choose to move in different directions.

The rest of this chapter is devoted to proving Theorem 13.3. We do it in two steps. First, we show in Lemma 13.4 a weaker version – namely that deterministic two-way automata are closed under pre-composition with rational functions. Then we bootstrap the weaker version to get composition with deterministic two-way automata.

Rational preprocessing. We begin by proving that deterministic two-way transducers can be pre-composed with rational functions. A different perspective on this result is that deterministic two-way transducers would not become more expressive if equipped with "regular lookaround", i.e. transitions that depend not only on the letter under the head, but also on some regular properties of the words to the left and right of the head.

Lemma 13.4. *Deterministic two-way transducers are closed under pre-composition with rational functions. In symbols,*

$$\underbrace{\mathsf{2Det}}_{\substack{\textit{functions recognised by}\\ \textit{deterministic two-way automata}}} \quad = \quad \mathsf{2Det} \quad \circ \quad \underbrace{\mathsf{Rat}}_{\textit{rational functions}}$$

Proof. The left-to-right inclusion is immediate, because the identity is a rational function. For the converse inclusion, recall the following characterisation

$$\mathsf{Rat} = \underbrace{\mathsf{Seq}^{\rightarrow}}_{\substack{\text{left-to-right}\\ \text{sequential functions}}} \circ \underbrace{\mathsf{Seq}^{\leftarrow}}_{\substack{\text{right-to-left}\\ \text{sequential functions}}}$$

from Theorem 13.1. By the above, to prove the theorem it is enough to show

$$\mathsf{2Det} \supseteq \mathsf{2Det} \circ \mathsf{Seq}^{\rightarrow} \qquad \mathsf{2Det} \supseteq \mathsf{2Det} \circ \mathsf{Seq}^{\leftarrow}.$$

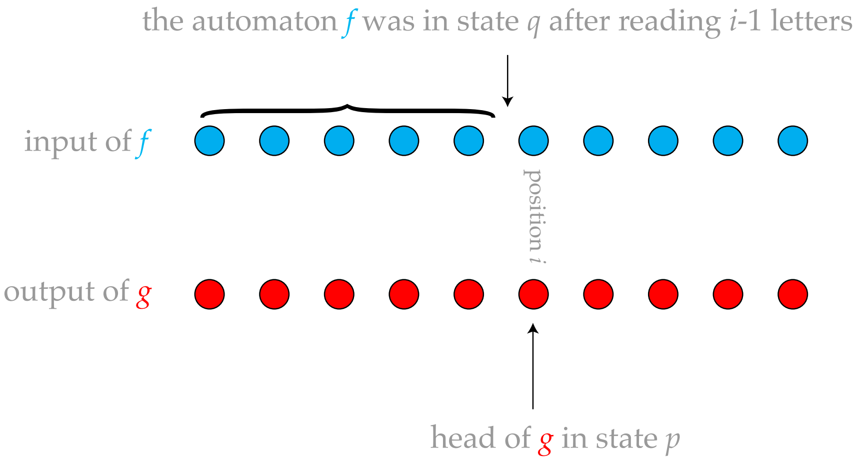
By symmetry of two-way automata, it is enough to prove the first inclusion. Summing up, it suffices to show that if f is left-to-right sequential and g is recognised by a deterministic two-way transducer, as in the following diagram,

$$\begin{tikzcd}
\Sigma^* \arrow[r, "\text{left-to-right sequential } f"] \arrow[dr, "f \circ g"'] & \Gamma^* \arrow[d, "\text{two-way } g"] \\
 & \Delta^*
\end{tikzcd}$$

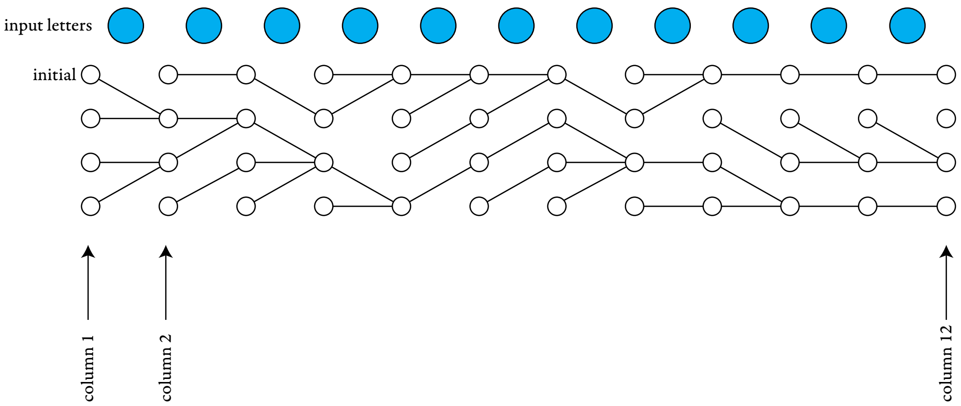
then the composition $f \circ g$ is also recognised by a deterministic two-way automaton. The difficulty is the machines for f and g have different types of movement.

The idea for the proof comes from Hopcroft and Ullman [58, Lemma 3]. To simplify notation, we assume that f is letter-to-letter, i.e. each transition of the underlying DFA with output produces exactly one output letter, and there are no end-of-input words. The proof for the general case – without the letter-to-letter assumption – can be easily inferred from the special case.

Suppose that the two-way automaton recognising g is in state p over the i-th position of its input (which is the output of f), like in the following picture:

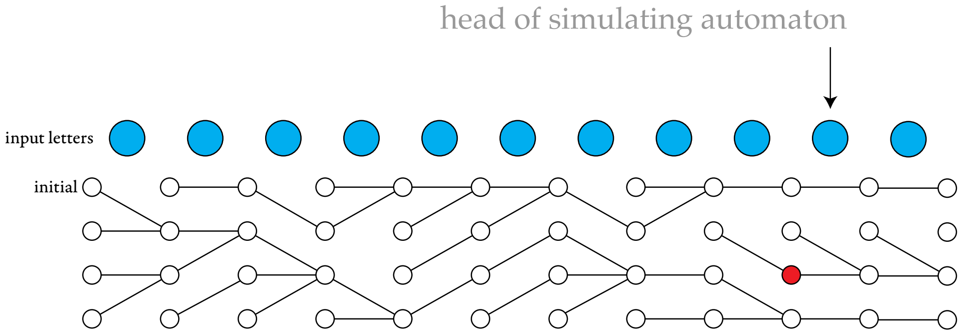

the automaton f was in state q after reading i-1 letters
input of f
position i
output of g
head of g in state p

Then the simulating two-way automaton for the composition $g \circ f$ has its head over the i-th position of the input word (which is the input of f), and knows the states p and q described in the picture above. The question is how to maintain this information, especially when the simulated two-way automaton g wants to move its head to the left. The key insight is to consider the graph which describes the states of f and how they are updated by the transition function. This graph looks likes this:



The vertices of the graph are configurations of f, i.e. pairs (state of f, column between positions in the word), and the edges correspond to transitions of the automaton. Each edge is labelled by an output letter. We number the columns beginning with 1. Because f is deterministic, the graph is a forest.

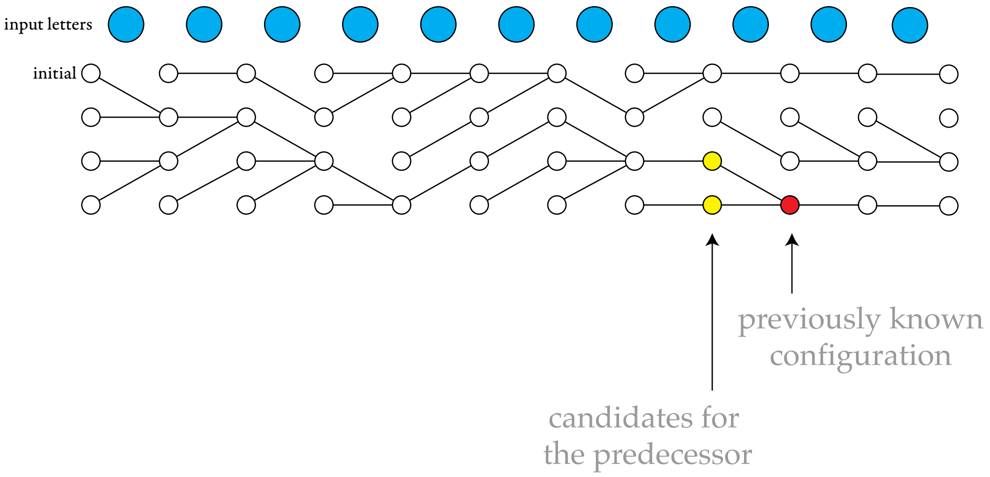
Define q_i to be the state of f in the i-th column, i.e. after reading the first $i-1$ letters of the input word. The simulating two-way automaton uses the state q_i to get the i-th letter in the output $f(w)$. Suppose that the head of the simulating two-way automaton is over some position i in the input word, and the state q_i of the oracle is known, as indicated by a red circle in the following picture:



We show below how to maintain the state of f when simulating one transition of the two-way automaton g. If the transition of the two-way automaton g does not move the head, or moves it to the right, there is no problem, since the transition function of f can be simply applied to the known state q_i.

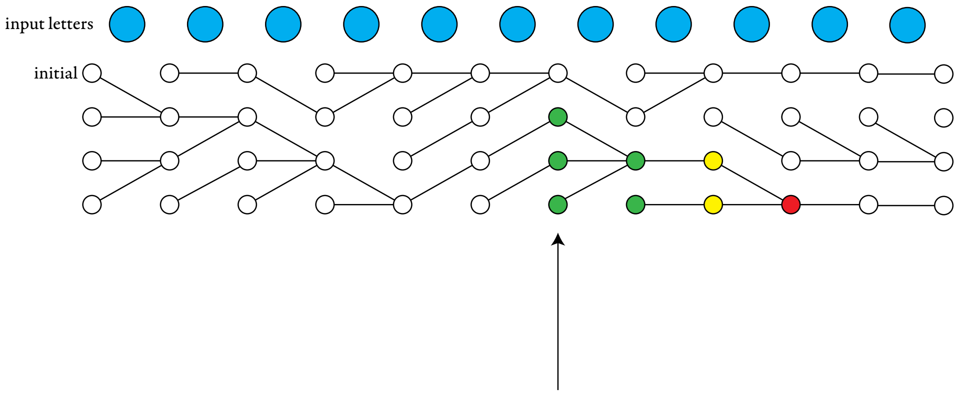
The issue is when the simulated two-way automaton f wants to move the head to the left, and we need to compute the state q_{i-1}.

Here is the solution. In terms of the forest in the pictures above, we want to determine the unique child of the red node which has the initial configuration in its subtree. To find this unique child, we do the following. We start by moving the head one step to the left, which identifies all possible candidates for the predecessor configurations. Here is the picture, with the candidates being coloured yellow:


input letters
initial
previously known configuration
candidates for the predecessor

If there is only one yellow configuration, i.e. only one candidate for the predecessor, then we are done. The more interesting case is when there is more than one yellow configuration. In this case, we keep moving to the left, and use green to colour all descendants of the yellow configuration (and therefore of the red configuration as well). For each green configuration we remember which of the yellow configurations is its ancestor. Two cases may happen.

1. We might reach a column where all green configurations are descendants of the same yellow configuration, as in this picture:



all are descendants of the same yellow configuration

In this case, the unique yellow configuration is the one that we want to compute. The question is how to return to this unique configuration? The solution is this: suppose that we stopped in column i, i.e. all green configurations in column i are descendants of the same yellow configuration, but this is not true for column $i+1$. We store in our memory the state of the unique yellow configuration that is the ancestor of all green configurations in column i. Then we start moving to the right, storing in each column that states reachable from the green configurations in column $i+1$. We stop when this set becomes a singleton – this happens exactly when we reach the column with the red node. Then we can move one step to the left and use our stored yellow state to determine the predecessor configuration of the red one.

2. The remaining case is when we reach the first column at the beginning of the input. Here we do the same trick to return to the red configuration, and we can keep in our state which branch of the subtree corresponds to the computation of the past oracle.

■

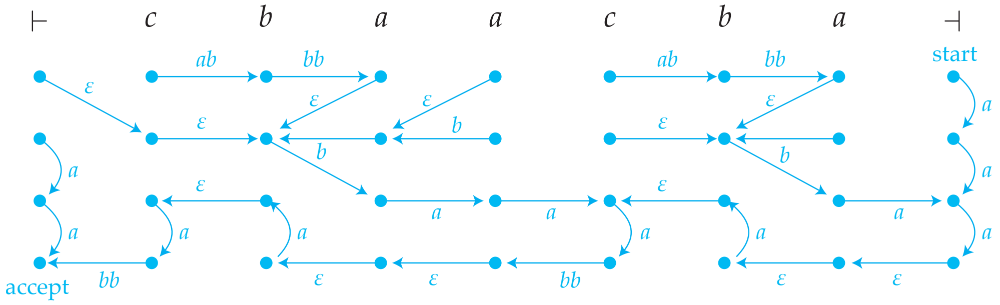
Closure under composition. Using Lemma 13.4 on pre-compositions with rational functions, we complete the proof of Theorem 13.3 on composition closure of deterministic two-way transducers. For our proof, it is more convenient to use a definition – clearly equivalent in terms of expressive power – of two-way transducers where the initial configuration is (initial state, end of input marker $\dashv$).

Fix two deterministic two-way transducers

$$\Sigma^* \xrightarrow{f} \Gamma^* \xrightarrow{g} \Delta^* .$$

We use the following colour coding. The first alphabet Σ is written in black. Blue is used for the states and output alphabet of f. Red is for the states and output alphabet of g. Our goal is to give a deterministic two-way transducer which recognises the composition $g \circ f$.

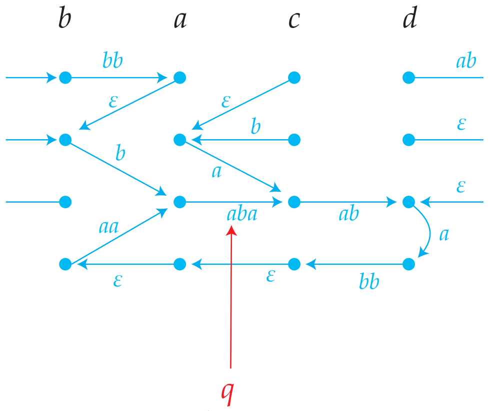
We begin with a naive construction that will not work. Take some input word $w \in \Sigma^*$, and consider the configuration graph of f on this input word, which looks like this:



Vertices of the graph – the blue dots – are pairs (state, position in w extended with end markers), and the edges correspond to transitions. The transitions are labelled by output words from the intermediate alphabet Γ. We can represent the configuration graph as a labelling of the input word, with arrows stored in the positions where they originate, and the descriptions of the end markers stored in the adjacent input positions.

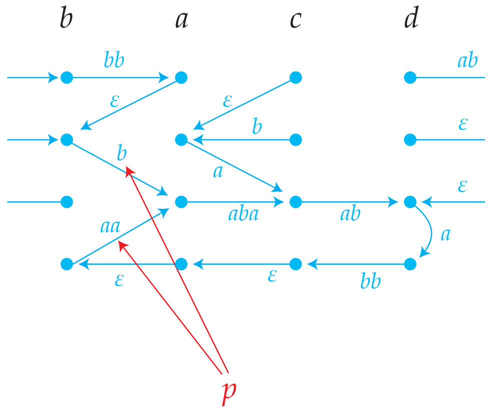
The natural construction for the composition $g \circ f$ would be to have an automaton which stores a state of g and a pointer to one of the letters from Γ

that are in the label of an edge in the configuration graph, as in the following picture:



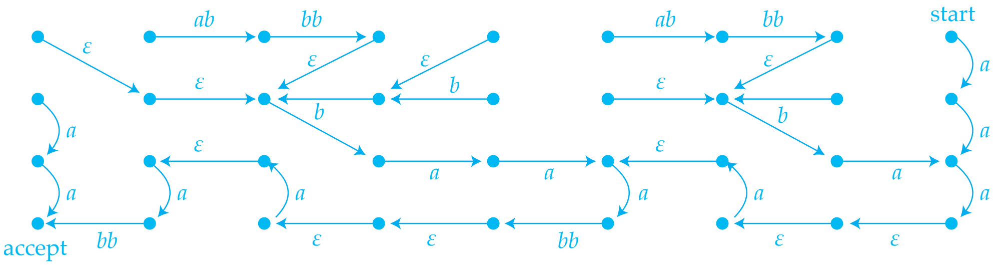
state of the automaton g

The problem with this construction is that a vertex in the configuration graph might have several incoming edges. For example, suppose that in the situation from the above picture, the automaton g decides to move its head to the left and change the state to p. Then the automaton for the composition $g \circ f$ would not know which of the following two choices should be made:

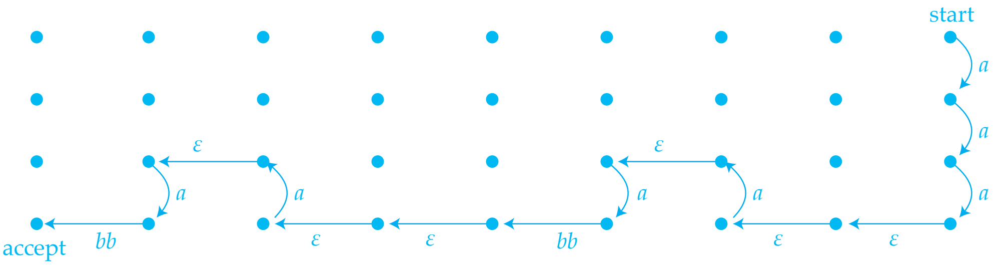


The solution – and also the reason why we use rational preprocessing from Lemma 13.4 – is to restrict the configuration graph of f to edges that are reachable from the initial configuration.

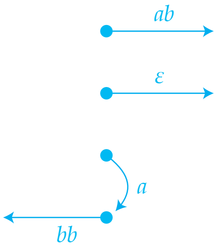
Lemma 13.5. *The following function is rational. The input is a configuration graph of f, like this:*



The output is the same graph, but only with those edges that are reachable from the initial configuration, like this:

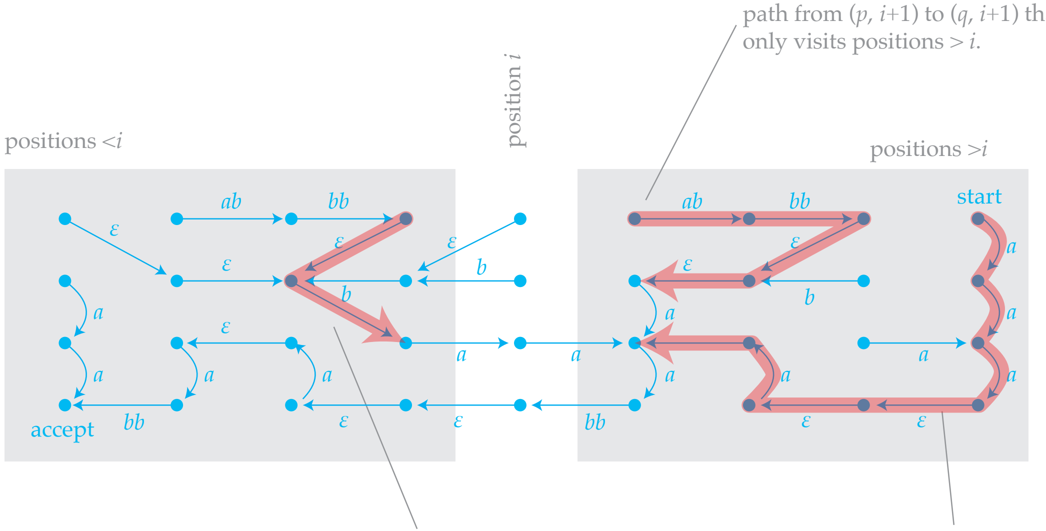


Proof. We assume that a configuration graph is represented as word where each letter represents the outgoing transitions from one column (i.e. position in the input word with end markers). Here is a picture of a letter



For this claim, it is convenient to use an Eilenberg bimachine as the representation of rational transducers. Given a position i in a configuration graph, the bimachine generates the following information:

(b) for which states p, q is there a path from (p, i+1) to (q, i+1) that only visits positions > i.

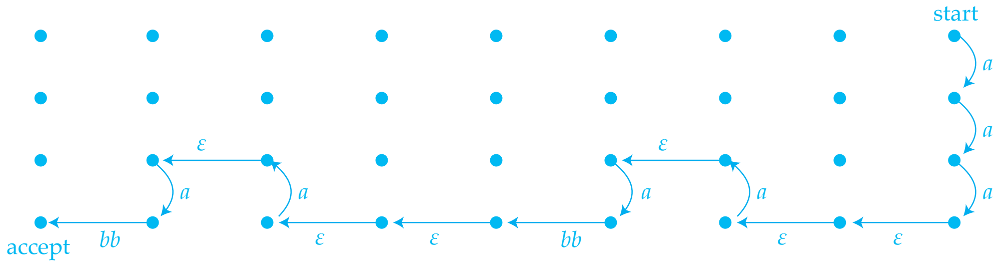


(a) for which states p, q is there a path from (p, i-1) to (q, i-1) that only visits positions < i.

(c) for which states q is there a path from the initial configuration to (q, i+1) that only visits positions > i.

The information can be generated by deterministic automata, as required by the definition of an Eilenberg bimachine, using the standard conversion of two-way automata (without output) to one-way automata. Based on this information and the label of the position i, one can determine which states in position i are reachable from the initial configuration. In its output, the bimachine only leaves edges that originate from reachable states. ∎

By Lemma 13.4, deterministic two-way transducers are closed under rational preprocessing, and by Lemma 13.5 a rational function can restrict a configuration graph to reachable configurations. Therefore, in order to find a deterministic two-way transducer for the composition $g \circ f$, it suffices to give a deterministic two-way transducer which inputs configuration graph of f restricted to reachable configurations, like this:



and outputs the value of g on the labelling of the unique path from the starting configuration to the accepting configuration. Since the blue nodes have indegree at most one, this can be done using the naive construction described before Lemma 13.5.

Problem 128. Show that deterministic two-way automata (seen as acceptors of words) can be complemented with polynomial blowup.

Problem 129. Consider a sequential transducer, which defines a function $f : \Sigma^\omega \to \Gamma^\omega$. Show that this function is continuous with respect to the distance defined in Problem 9.

Problem 130. Show that the reverse function is not left-to-right sequential.

Problem 131. Which of the following functions over a unary alphabet are sequential?

1. $a^n \mapsto a^{n^2}$;

2. $a^n \mapsto a^{\lfloor \sqrt{n} \rfloor}$.

Problem 132. Show that the duplication function $w \mapsto ww$ is not rational.

Problem 133. Show that left-to-right sequential functions are closed under compositon, i.e.

$$\mathsf{Seq}^{\to} = \mathsf{Seq}^{\to} \circ \mathsf{Seq}^{\to}.$$

Problem 134. Show that rational functions are closed under compositon, i.e.

$$\mathsf{Rat} = \mathsf{Rat} \circ \mathsf{Rat}.$$

Problem 135. Show that if f is recognised by a deterministic two-way transducer and and g is rational (with suitable input and output alphabets), then $g \circ f$ is recognised by a deterministic two-way transducer.

Problem 136. Consider nondeterministic two-way automata with output. Show that for every nondeterministic two-way automaton with output $\mathcal{A}$ there is a deterministic two-way automaton with output $\mathcal{B}$ that uniformises it in the following sense: for every input word, $\mathcal{B}$ produces one of the outputs of $\mathcal{A}$. (If there is no output of $\mathcal{A}$, then also there is no output of $\mathcal{B}$.)

Problem 137. Show that the following problem is in polynomial time: given two letter-to-letter (i.e. each transition produces exactly one letter) left-to-right sequential functions with the same input alphabet, decide if for every input they produce the same output.

Problem 138. Show that the following problem is undecidable: given two left-to-right sequential functions with the same input alphabet, decide if for some input, they produce the same output.

14
Streaming string transducers

In this chapter we present a one-way automaton model that has the same expressive power as two-way transducers.
We begin by defining register transducers, which are automata that use registers to store parts of their output. We have already seen register transducers in Chapter 11 – in a more general setting, for arbitrary algebras – and we have even proved in Corollary 11.10 that their equivalence is decidable for the specific algebra of words with concatenation that we use in this chapter. To make this chapter self-contained, we give a stand-alone definition below. Register transducers, as defined below, will turn out to be strictly more powerful than two-way transducers, but a model with the same expressive power as two-way transducers will be recovered by placing a certain copyless restriction on the register updates.

Definition 14.1. *A* register transducer *consists of:*

- *finite* input and output alphabets Σ *and* Γ*;*
- *a finite set of* states Q*;*
- *a finite set of* registers R*;*
- *an* initial configuration *in* $Q \times (R \to \Gamma^*)$*;*

- *a* transition function

$$\delta : Q \times \Sigma \to Q \times \underbrace{(R \to (R + \Gamma)^*)}_{\textit{register update}}$$

- *an* output function

$$out : Q \to (R + \Gamma)^*$$

The automaton is run as follows. Define a *register valuation* to be any function from registers to words over the output alphabet Γ, and define a *register update* to be any function from registers to words over the alphabet $R + \Gamma$. There is an action of updates on valuations

$$(v \in \text{register valuations}, \tau \in \text{register update}) \quad \mapsto \quad v \cdot \tau \in \text{register valuations}$$

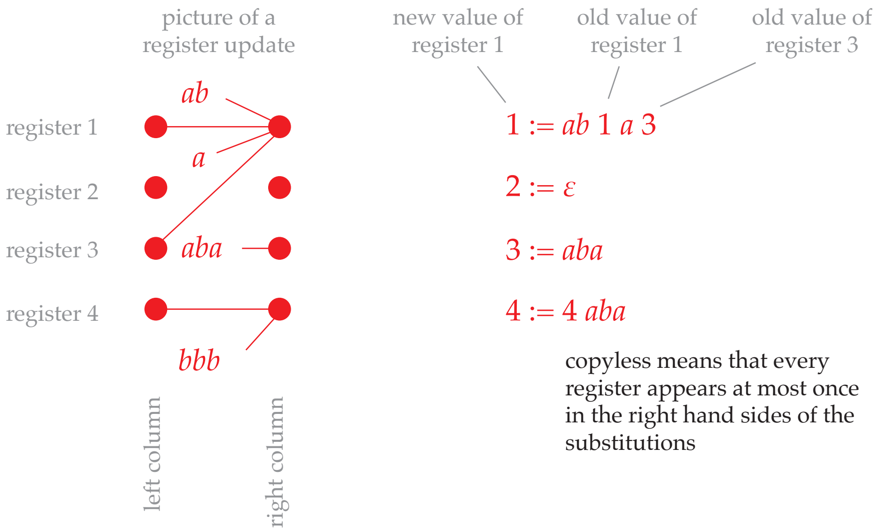
where $v \cdot \tau$ is obtained from τ by replacing each register name with its contents under τ. A *configuration* of the automaton is defined to be a pair (state, register valuation). The automaton begins in the initial configuration. When reading an input letter a, the automaton uses its transition function to determine its new state and the register update. More formally, the configuration is updated as follows:

$$(q, v) \cdot a \stackrel{\text{def}}{=} (p, v\tau) \qquad \text{where } \delta(q, a) = (p, \tau).$$

After the entire word has been processed, with the last configuration being (q, v), the automaton outputs $out(q)$, with register names replaced by their contents in v.

Example 26. Here is an automaton where the input and output alphabets are $\{a\}$, and the recognised function is $a^n \mapsto a^{5+3\cdot 2^n}$. The automaton has one register and one state. The initial configuration stores the word a in the unique register. When reading an input letter, the unique register r is updated by $r := rr$. The output function maps the unique state to $aaaaarrr$.
The function recognised by this register transducer is not recognised by any two-way transducer. There reason is that the function has exponential growth, while a two-way transducer has necessarily at most linear blowup, because a position in the input word can be visited at most once for each state. □

Copyless restriction. As argued in Example 26, register transducers can have exponential growth, and therefore are not in general equivalent to deterministic two-way transducers. To recover equivalence with two-way transducers, we use the *copyless restriction* (also known as the *single use restriction*) described in the following picture:



copyless means that every register appears at most once in the right hand sides of the substitutions

In other words, a register update is copyless if every vertex in the left column has outdegree at most one. The intuition is that the register contents are physical objects and can only be moved around and not duplicated.

Definition 14.2. *A* streaming string transducer[1] *is a register transducer where the transition function produces only copyless register updates.*

The output function need not be copyless. Requiring it to be copyless would not weaken the model, though, because the output function is applied only once. For example, if the output function uses each register at most k times, then by taking k disjoint copies of the registers we can make the output function copyless.

[1] The model and name of streaming string transducers comes from [3], although similar and essentially equivalent models have been known before in the literature on attributed grammars, e.g. attributed tree transducers from [29].

The goal of this chapter is to prove that streaming string transducers are equivalent to deterministic two-way transducers.

Theorem 14.3. *Streaming string transducers recognise the same word-to-word functions as deterministic two-way transducers*

The above theorem was proved by Alur and Cerny in [3]. A similar result (using a model of streaming string transducers with lookahead) can also be recovered from earlier work of Bloem, Engelfriet and Hogeboom: (a) MSO transductions are equivalent to deterministic two-way transducers [27]; and (b) MSO transductions are equivalent (even over trees) to a certain kind of attribute transducers [8].
We begin by describing the proof strategy. Our goal is to prove the equality

$$\underbrace{\mathsf{SST}}_{\substack{\text{functions recognised by}\\ \text{streaming string transducers}}} = \underbrace{\mathsf{2Det}}_{\substack{\text{functions recognised by}\\ \text{deterministic two-way transducers}}}. \tag{14.1}$$

As in the proof of Theorem 13.4, we write Rat for the class of rational functions. In Section 14.1, we prove the following inclusions

$$\mathsf{SST} \overset{\text{Lemma 14.4}}{\subseteq} \mathsf{2Det} \circ \mathsf{Rat} \qquad \text{and} \qquad \mathsf{SST} \circ \mathsf{Rat} \overset{\text{Lemma 14.5}}{\supseteq} \mathsf{2Det}.$$

In other words, every streaming string transducer can be recognised by a deterministic two-way automaton with preprocessing by a rational function, and likewise in the opposite direction. Rational functions are easily seen to be closed under composition, using a straightforward product construction, see Exercise 134. Combining the inclusions from Lemmas 14.4 and 14.5, and using closure of rational functions under composition, we get

$$\mathsf{SST} \circ \mathsf{Rat} = \mathsf{2Det} \circ \mathsf{Rat}. \tag{14.2}$$

To finish the proof of Theorem 14.3, it suffices to show that both streaming string transducers and deterministic two-way transducers are closed under preprocessing with rational functions. For deterministic two-way transducers, this was shown in Theorem 13.4 from Chapter 13. For streaming string transducers, this will be done in Lemma 14.7, which is the most challenging construction in this chapter. Combining these results, we get

$$\mathsf{SST} \overset{\text{Lemma 14.7}}{=} \mathsf{SST} \circ \mathsf{Rat} \overset{(14.2)}{=} \mathsf{2Det} \circ \mathsf{Rat} \overset{\text{Theorem 13.4}}{=} \mathsf{2Det}$$

which completes the proof of Theorem 14.3. It remains to prove Lemmas 14.4, 14.5 and 14.7.

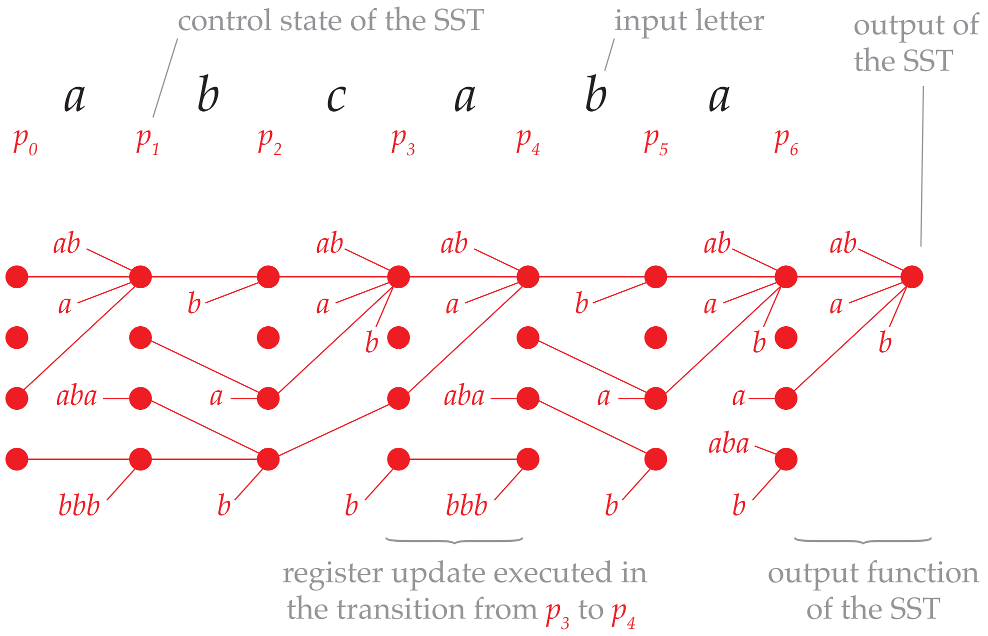
14.1 *Equivalence after rational preprocessing*

In this section, we prove that streaming string transducers and deterministic two-way transducers are equivalent if we allow rational preprocessing

Lemma 14.4. *Every streaming string transducer can be decomposed as a rational function followed by a deterministic two-way transducer. In other words*

$$\mathsf{SST} \subseteq \mathsf{2Det} \circ \mathsf{Rat}.$$

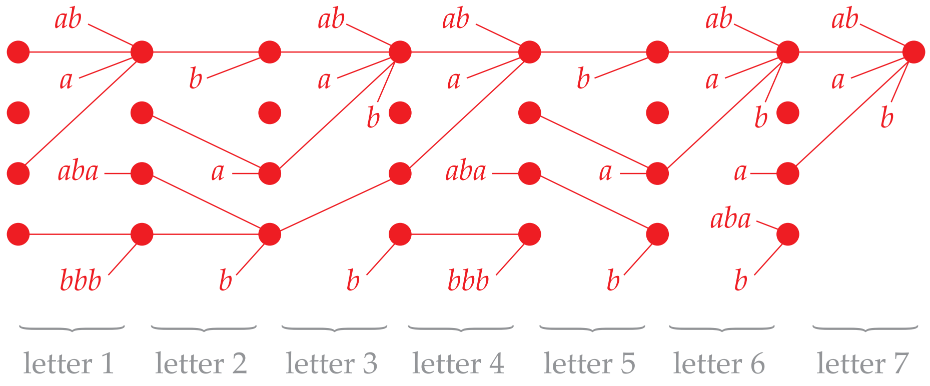
Proof. Fix a streaming string transducer. A run of the transducer looks like this:


control state of the SST
input letter
output of the SST
a b c a b a
p_0 p_1 p_2 p_3 p_4 p_5 p_6
ab
a
aba
bbb
b
b
a
b
ab
a
b
b
ab
a
aba
bbb
b
a
b
ab
a
b
a
aba
b
ab
a
b
register update executed in the transition from p_3 to p_4
output function of the SST

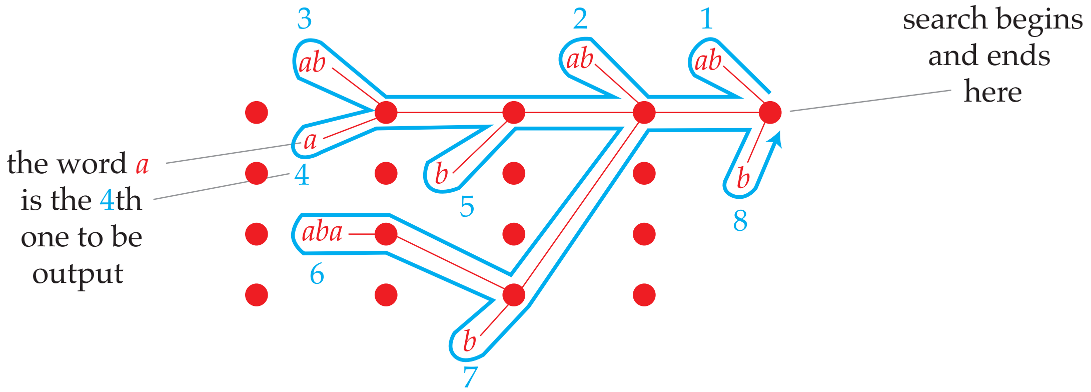
It is not hard to see that there is a rational – in fact left-to-right sequential – transducer which transforms an input word

$$a \qquad b \qquad c \qquad a \qquad b \qquad a$$

to a word describing the corresponding sequence of register updates:



By using the above rational transducer as a preprocessor, to prove the lemma it is enough to find a deterministic two-way transducer which inputs a tree that describes the register updates, and outputs the final value. To do this, we use a depth-first search through the tree as explained in the following picture



the word *a* is the 4th one to be output

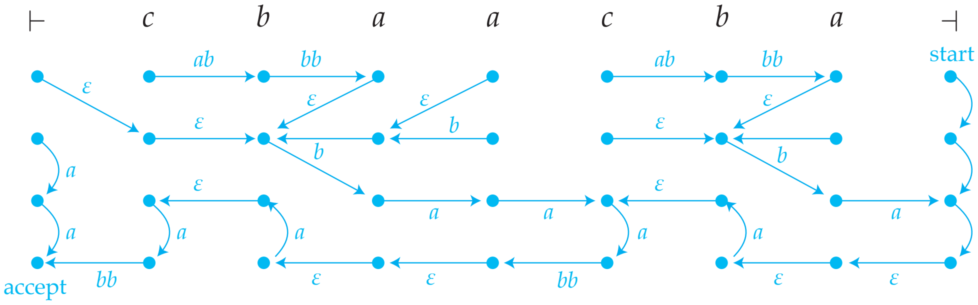
search begins and ends here

It is easy to implement a depth-first search using a deterministic two-way automaton. One simply has to remember the current register and the direction from which it came. ∎

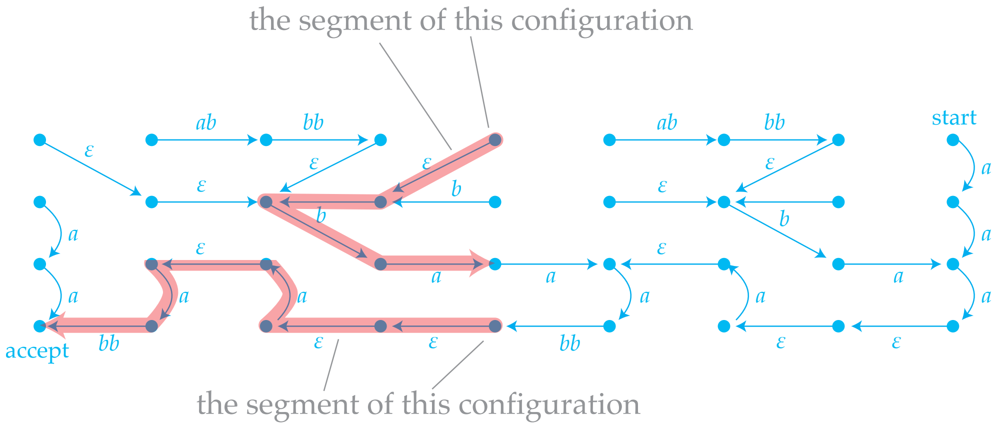
Lemma 14.5. *Every deterministic two-way transducer can be decomposed as a rational function followed by a streaming string transducer. In other words*

$$\mathsf{2Det} \subseteq \mathsf{SST} \circ \mathsf{Rat}.$$

Proof. As in the proof of Theorem 13.3, it is more convenient to use a definition of two-way transducers where the initial configuration is (initial state, end of input marker ⊣). Consider the configuration graph of the two-way automaton over a given input word, as in the following picture:

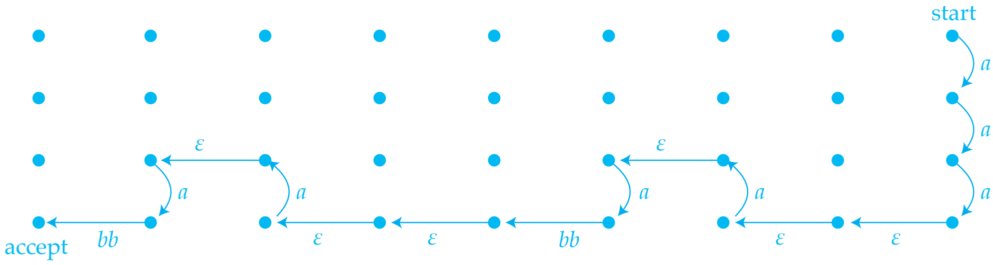


We begin with a naive idea, which will not work because of the copyless restriction. For a vertex in the configuration graph, define its *segment* to be the (unique, by determinism) path that begins in the configuration, and is cut off at the first visit to the same column as the source configuration, as in the following picture:


the segment of this configuration
ab
bb
ε
ε
ε
ε
b
b
ab
bb
start
ε
ε
b
a
a
a
ε
a
a
a
a
a
ε
a
a
a
accept
bb
ε
ε
bb
ε
ε
the segment of this configuration

The segment might accept/reject/loop without returning to the column of the source configuration. The naive idea would be to store for each state q the output word that is found by reading the labels on the segment of the configuration that has state q in last read position. The problem with this construction is that it violates the copyless restriction, because configurations can have more than one incoming edge, and therefore the labels of one segment can be shared by several longer segments.

Like in the proof of Theorem 13.3, the solution is to restrict the configuration graph to edges that are reachable from the initial configuration. As shown in Lemma 13.5, a rational function can be used to restrict the configuration graph to reachable configurations, so that the result looks like this:



When only reachable edges are used, the indegree is at most one, because otherwise the automaton would loop, which cannot happen by the assumption that it defines a total function. Using the naive idea, one can write a streaming string transducer which inputs a configuration graph with only reachable edges

– represented as a word over a finite alphabet in any natural way – and outputs the label of the segment corresponding to the initial configuration. ■

14.2 *Lookahead removal*

In this section we show that functions recognised by register transducers and streaming string transducers are closed under pre-composition with rational functions.

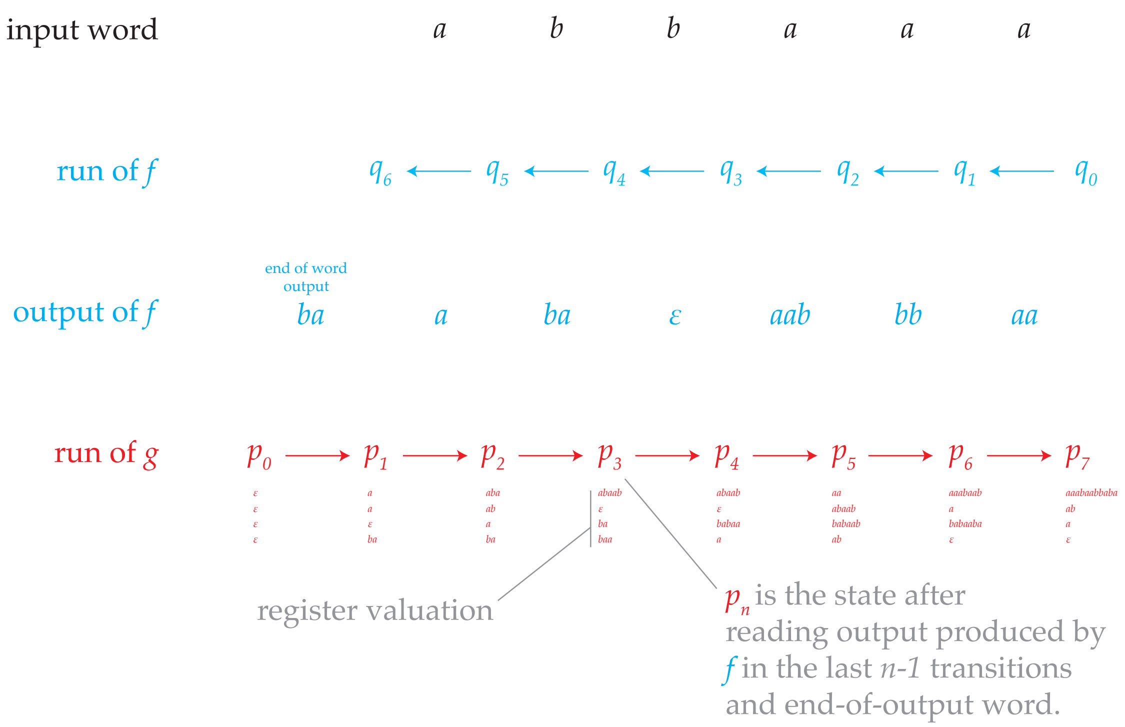
A different perspective on this result is that register transducers and streaming string transducers would not become more expressive if equipped with an oracle that gives regular information about the input word to the left and right of the head. Since the information about the word to the left of the head can be stored in the state, the interesting part of the oracle is the one that talks about the word to the right of the head. In other words, in this section we show that lookahead can be eliminated from the transducers without affecting expressive power.

Lemma 14.6. *Functions recognised by register transducers are closed under pre-composition with rational functions.*

Proof. Consider functions

$$\begin{tikzcd}\Sigma^* \arrow[r, "f"] & \Gamma^* \arrow[r, "g"] & \Delta^*\end{tikzcd}$$

such that f is rational and g is recognised by a register automaton. We use the following colour coding. The first alphabet Σ is written in black. Blue is used for the states and output alphabet of f. Red is for the states and output alphabet of g. A run of the composition $g \circ f$ looks like this:



The register transducer for the composition $f \circ g$ stores a function

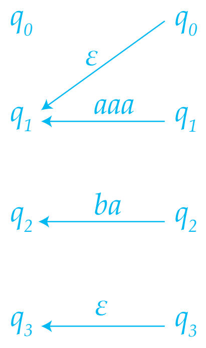
$$\text{states of lookahead } f \qquad \to \qquad \text{configurations of } g$$

which maps a state q of f to the configuration that would be used by g assuming that q is the state of the lookahead f after reading the unread part of the input (in a right-to-left pass). Such a function can be represented by using

$$(\text{number of states in lookahead } f) \times (\text{number of registers in } g)$$

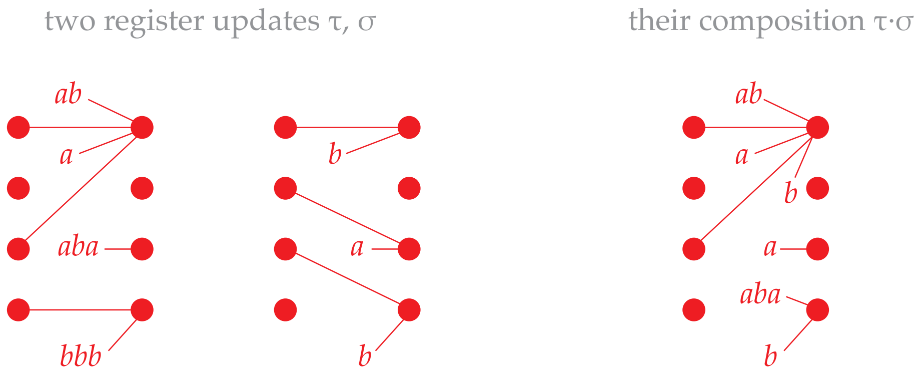
registers; and the representation can be updated in the transition function. After reading the entire word, the transducer for the composition looks at the value of the function under the initial state of f, and then applies the output function of g. ■

The construction in the above lemma cannot be used for streaming string transducers because it violates the copyless restriction. The violation comes from merging states in the right-to-left sequential function f. For example, suppose that the state transformation of f over some input letter $a \in \Sigma$ looks like this:



Then the register transducer described in the proof of Lemma 14.6 would duplicate the information stored for state q_1, using it for both q_0 and q_1. To eliminate lookahead for streaming string transducers, we use a data structure, called a transformation forest, which stores register updates organised in a forest structure so that composition can be done without copying. We describe this data structure below.

Composing register updates. We begin with defining a composition operation on register updates. Here is the picture:

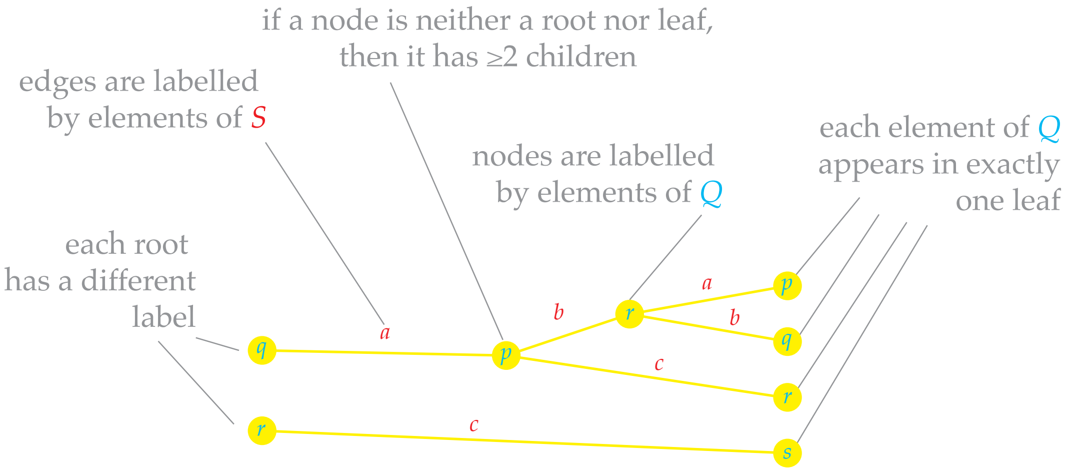


The composition operation is defined so that if τ, σ are two register updates and v is a register valuation, then

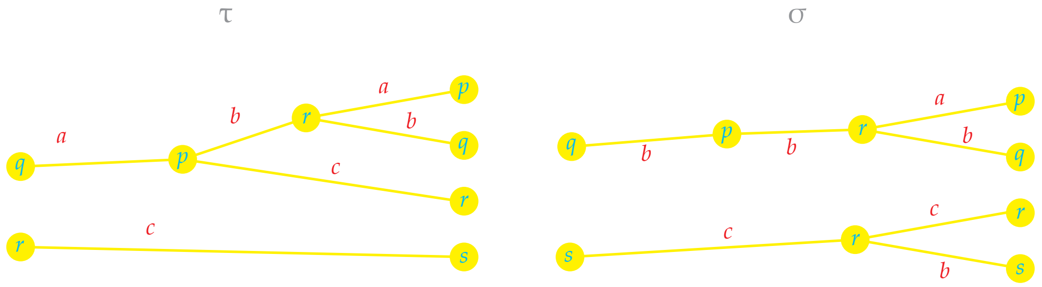
$$v \cdot (\tau \cdot \sigma) = (v \cdot \tau) \cdot \sigma.$$

Using the above composition, we can view the set of register updates – for a fixed set of register names and output alphabet – as a monoid.

Transformation forests. Suppose that M is a monoid and Q is a finite set. (Our intended application is that S is the monoid of register updates for some streaming string transducer, but the abstract definition requires less notation.) Define a *transformation forest* (over M and Q) to be any labelled forest of the following form:

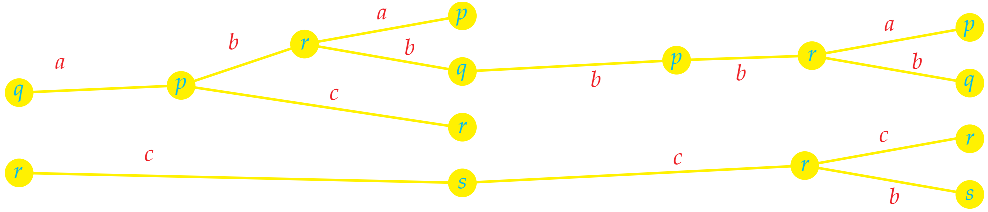

if a node is neither a root nor leaf, then it has ≥2 children
edges are labelled by elements of S
nodes are labelled by elements of Q
each element of Q appears in exactly one leaf
each root has a different label
q
a
p
b
r
a
p
b
q
c
r
r
c
s

We now describe how transformation forests can be composed. Suppose that we have two transformation forests τ and σ, as illustrated below:

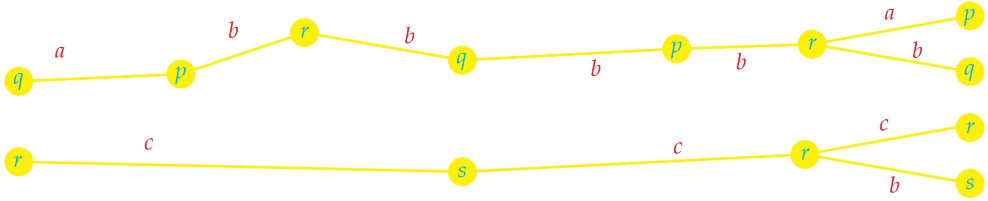


Their composition $\tau\sigma$ is obtained by doing the following steps.

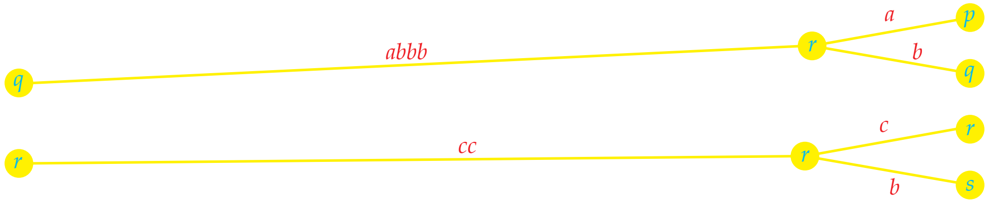
1. To each root of σ we can associate a unique leaf of τ with the same label, because roots of σ have different labels and all labels appear in leaves of τ. Merge each root of σ with the associated leaf of τ:



2. Eliminate nodes that do not reach any node leaf of σ:



3. Contract into a single edge every path that uses only nodes with unary branching (except the source and target):



   The label of a contracted path is the product, in the semigroup S, of the labels of edges on the path before the contraction.

It is not hard to see that this operation is associative, i.e.

$$\tau(\sigma\rho) = (\tau\sigma)\rho.$$

Also, there is a neutral element, namely the transformation forest where each leaf is a root (and there are no edges). Therefore, the set of transformation forests is a monoid, which we denote by $M^{[Q]}$. The reader might recognise transformation forests from Lemma 1.6 from Chapter 1. In that lemma, the monoid M had two elements "accepting" and "non-accepting". In this chapter, M will be the infinite monoid of copyless register updates.

Lookahead elimination for streaming string transducers. Equipped with the data structure of transformation forests, we are ready to prove the copyless variant of Lemma 14.6.

Lemma 14.7. *Functions recognised by streaming string transducers are closed under pre-composition with rational functions. In other words*

$$\mathsf{SST} = \mathsf{SST} \circ \mathsf{Rat}.$$

Proof. The left-to-right inclusion is immediate, since the identity is a rational function. For the converse inclusion, recall the following equality

$$\mathsf{Rat} \quad = \quad \underbrace{\mathsf{Seq}^{\rightarrow}}_{\substack{\text{left-to-right} \\ \text{sequential functions}}} \quad \circ \quad \underbrace{\mathsf{Seq}^{\leftarrow}}_{\substack{\text{right-to-left} \\ \text{sequential functions}}}$$

from Theorem 13.1. Since both streaming string transducers and left-to-right sequential functions are instances of left-to-right automata, a straightforward product construction can be used to yield the inclusion

$$\mathsf{SST} \supseteq \mathsf{SST} \circ \mathsf{Seq}^{\rightarrow}$$

Therefore, in order to prove the lemma it suffices to show

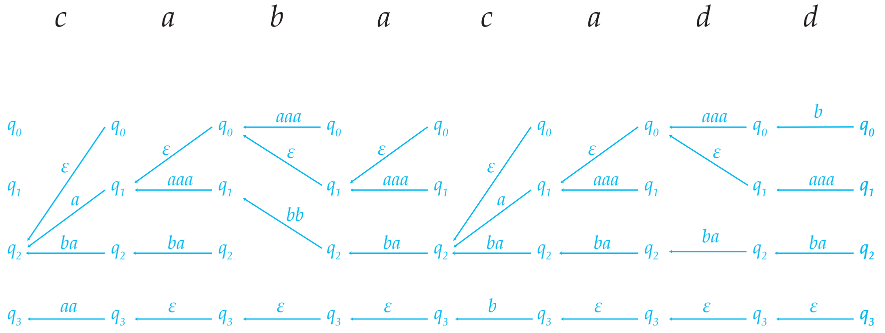
$$\mathsf{SST} = \mathsf{SST} \circ \mathsf{Seq}^{\leftarrow}.$$

Here we cannot use a simple product construction, because we compose automata that move in different directions. The rest of the proof is devoted to proving the above inclusion. We use the same notation and colour convention as in the proof of Lemma 14.6. Let

$$\begin{tikzcd} \Sigma^* \arrow[r, "f"] & \Gamma^* \arrow[r, "g"] & \Delta^* \end{tikzcd}$$

be functions such that f is right-to-left sequential and g is a streaming string transducer. Our goal is to design a streaming string transducer that recognises the composition $g \circ f$. To make notation lighter, we assume that f has empty end-of-input words. This assumption can be lifted without greater conceptual difficulty.

Overview of the construction. The idea is that instead of storing register valuations, the streaming string transducer for $g \circ f$ will store register updates, organised in a transformation forest. To illustrate this idea, consider the configuration graph of the right-to-left sequential function f over an input word $w \in \Sigma^*$, as shown in the following picture:

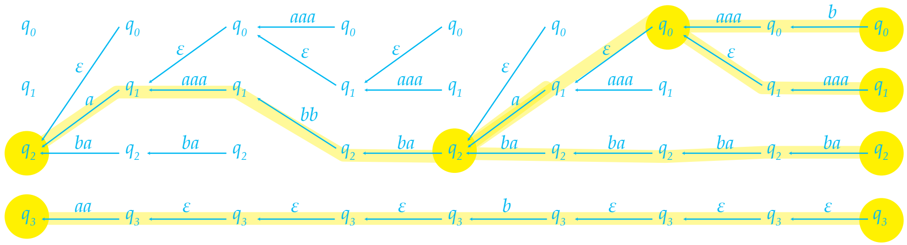


Nodes of the configuration graph are labelled by states of f and edges are labelled by output words of f. Because the f is right-to-left deterministic, the configuration graph is a forest, with the roots in the first column. The output of f is obtained by reading from left to right the labels on the path that goes from the unique leaf with the initial state of f to the unique root that is its ancestor. (We use the assumption that the end-of-input words are empty; otherwise we would need to add one more column at the left end of the picture.)

The automaton recognising the composition $g \circ f$ will store in its configuration a transformation forest

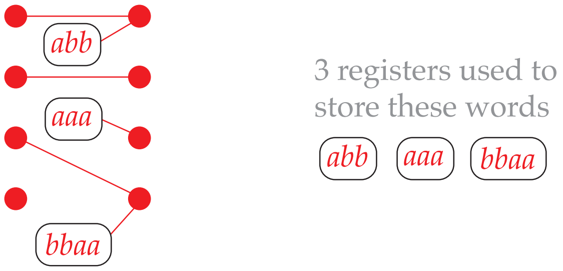
$$t \in \underbrace{(\text{register updates of } g)}_{\substack{\text{monoid of copyless register}\\ \text{updates for registers and}\\ \text{output alphabet of } g}}{}^{[\text{states of } f]}.$$

The nodes of this transformation forest will correspond to the leaves of the configuration graph, their closest common ancestors, and the roots that are reachable from leaves, as represented by the big yellow circles below:



For a path connecting two adjacent yellow nodes, the transformation forest t will store the register update done by g on that path. To describe the automaton in more detail, we begin by discussing how copyless register updates, and therefore also transformation forests over the monoid of copyless register updates, can be stored in the configuration of a streaming string transducer.

Storing register updates. Recall the graphical representation of register updates that was used when defining the copyless restriction. A copyless register update can be stored by a streaming string transducer like this:



In general, to store a copyless register update we need a bounded number of bits to store the tree structure of the update plus

$$2 \cdot (\text{number of registers in } g)$$

registers to store the output words used in the update. To store a transformation forest

$$t \in (\text{register updates of } g)^{[\text{states of } f]}.$$

we use a bounded number of bits to store the structure of the forest and its labelling by states of f, plus

$$\underbrace{2 \cdot (\text{number of registers in } g)}_{\substack{\text{registers to store}\\ \text{a register update}}} \cdot \underbrace{2 \cdot (\text{states in } f)}_{\substack{\text{number of edges in}\\ \text{a transformation forest}}}$$

registers to store the register updates. The following claim says that transformation forests can be updated in a copyless way.

Claim 14.8. *Fix a transformation forest*

$$s \in (\textit{register updates of } g)^{[\textit{states of } f]}.$$

Then the function

$$t \in (\textit{register updates of } g)^{[\textit{states of } f]} \quad \mapsto \quad ts \in (\textit{register updates of } g)^{[\textit{states of } f]}$$

can be done using a copyless register update.

Proof. Almost by definition, copyless register updates can be composed using a copyless register update. The same is true when composing transformation forests ts, because each label from t and each label from s is used at most once in the composition. In fact, copyless register updates can be seen as a special case of transformation forests, see Exercise 143. ■

The automaton. Before describing the automaton, let us introduce some notation that will be used in its definition and correctness proof. Let q be a state of f and let p be a state of g. Define f_q to be the right-to-left sequential function obtained from f by changing the initial state to q and define $[p, w, q]$ to be the run of g – viewed as a sequence of transitions – which begins in state p and reads the word $f_q(w)$. We have the following equality, which is obtained by unravelling the definitions:

$$[p, wa, q] = [p, w, aq] \cdot [p(f_q(a)), a, q] \qquad \text{for every } w \in \Sigma^* \text{ and } a \in \Sigma. \tag{14.3}$$

In the above, we write $\_q$ and $p\_$ for the state transformations of the automata underlying f and g.

Equipped with the above notation, we are ready to define the streaming string transducer recognising the composition $g \circ f$. After reading an input word $w \in \Sigma^*$, the transducer will store a transformation forest

$$t_w \in (\text{register updates of } g)^{[\text{states of } f]}$$

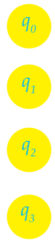
whose intuitive meaning was described at the beginning of the proof. The transformation forest t_w is stored as described before Claim 14.8, and it satisfies the following invariant:

(*) Let q be a state of f and let π be the unique root-to-leaf path in t_w that ends in a leaf with label q. Then the composition of register updates labelling π is the same as the register update done by the run [initial state of g, w, q].

To update its configuration, the transducer will also store in its finite state space the function δ_w defined by

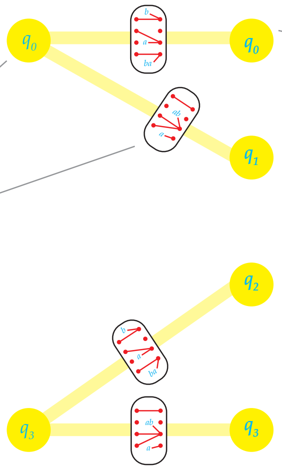
$$q \in \text{states of } f \quad \mapsto \quad \text{target state of the run [initial state of } g, w, q]$$

Using (14.3), it is not hard to see how δ_{wa} can be computed from δ_w and an input letter a. It remains to show how to update the transformation forest t_w. Initially, t_ε is a forest with no edges and one leaf per state of f, like this



every leaf is also
a root, and there
are no edges

and therefore the invariant (*) is satisfied because π is the empty path which yields an identity register update. When reading a letter a, the transformation forest is updated as follows. The new transformation forest t_{wa} is defined to be the composition – in the monoid of transformation forests – of t_w with the following transformation forest:



the parent of a leaf with label q
is a root with label aq

a leaf for
each state of f

the edge to a leaf with label q
is labelled by the register update
done by the run $[\delta_w(aq), a, q]$

Using the equality (14.3), it is not hard to check that t_{wa} satisfies the invariant. Furthermore, the update can be done while preserving the copyless discipline, by Claim 14.8.

It remains to define the output function so that the automaton recognises the composition $g \circ f$. By the invariant, once the automaton has finished processing an input w, by looking at the transformation forest t_w we can recover the register update τ that is done by the run of g on $f(w)$, i.e. the run

$$[\text{initial state of } g,\ w,\ \text{initial state of } f].$$

To get the output of $g \circ f$ on w, it remains to apply τ to the empty register valuation, and finally apply the output function of g to the resulting register valuation. All of this can be done using the register representation of the transformation forest t_w. ■

Problem 139. Show that the class SST of functions recognised by streaming string transducers has the following closure properties:

1. if f, g are in SST, then so is $w \mapsto f(w)g(w)$.
2. if f is in SST, then so is $w \mapsto$ reverse of $f(w)$.

Problem 140. Show that the class of regular languages is closed under inverse images of streaming string transducers, but not under forward images.

Problem 141. Show that a language $L \subseteq \Sigma^*$ is regular if and only if there is a streaming string transducer with input alphabet Σ and output alphabet $\{0, 1\}$ which recognises the characteristic function of L.

Problem 142. Define a *nondeterministic* streaming string transducer by (a) allowing several applicable transitions in each state; (b) distinguishing accepting states, so that only runs that end in an accepting state count. A *functional* streaming string transducer is a nondeterministic one where every accepting run produces the same output. Show that functional streaming string transducers recognise the same functions as deterministic ones.

Problem 143. Consider the least class of monoids that contains Γ^*, and is closed under:

- reversing the monoid operation, i.e. $m \cdot n$ becomes $n \cdot m$;
- submonoids;
- homomorphic images;
- if M is the class, then so is $M^{[Q]}$ for every finite set Q.

Show that this class contains, for any finite set R of registers, the monoid of copyless register updates with alphabet Γ and registers R.

Problem 144. A streaming string transducer is called *monotone* if its registers can be totally ordered as $r_1, \ldots, r_n$ so that every register update τ preserves the order in the following sense: after concatenating the words $\tau(r_1), \ldots, \tau(r_n)$ and keeping only the register names, we get a subsequence of $r_1, \ldots, r_n$. Show that every streaming string transducer can be decomposed as $g \circ f$ where f is a rational function and g is a monotone streaming string transducer.

Problem 145. Show that the following problem is PSPACE-hard (it is also in PSPACE, but this is more challenging to prove): given a streaming string transducer, decide if it produces the empty word for every input.

15
Learning automata

This chapter is about learning regular languages of finite words. All automata here are deterministic finite automata. The setup is that there are two parties: Learner and Teacher. Teacher knows a regular language. Learner wants to learn this language, and pursues this goal by asking two types of queries to the Teacher:

- *Membership.* In a membership query, Learner gives a word, and the Teacher says whether or not Teacher's language contains that word.
- *Equivalence.* In an equivalence query, Learner gives regular language, represented by an automaton, and Teacher replies whether or not the Teacher's and Learner's languages are equal. If yes, the protocol is finished. If no, Teacher gives a counterexample, i.e. a word where the Teacher's and Learner's languages disagree.

Membership queries on their own can never be enough to identify the language, since there are infinitely many regular languages that match any finite set of membership queries. Given enough time, equivalence queries alone are sufficient: Learner can enumerate all regular languages, and ask equivalence queries until the correct language is reached, without ever using membership queries. The lecture is about a more practical solution, which was found by Dana Angluin [4]. Angluin's algorithm is a protocol where Learner learns Teacher's language in a number of queries that is polynomial in:

- the minimal automaton of Teacher's language;
- the size of Teacher's counterexamples.

If Teacher provides counterexamples of minimal size, then the second parameter above is superfluous, i.e. the number of queries will be polynomial in the minimal automaton of Teacher's language. As mentioned above, we only talk about deterministic automata, and therefore the minimal automaton refers to the minimal deterministic automaton.

State words and test words. Suppose that Teacher's language is $L \subseteq \Sigma^*$. We assume that the alphabet is known to both parties, but the language is only known to Teacher. At each step of the algorithm, Learner will store an approximation of the minimal automaton of L, described by two sets of words:

- a set $Q \subseteq \Sigma^*$ of state words, closed under prefixes;
- a set $T \subseteq \Sigma^*$ of test words, closed under suffixes.

The idea is that the state words are all distinct with respect to Myhill-Nerode equivalence for Teacher's language, and the test words prove this. This idea is formalised in the following definitions.

Correctness and completeness. If T is a set of test words, we say that words $v, w \in \Sigma^*$ are T-equivalent if

$$wu \in L \quad \text{iff} \quad vu \in L \qquad \text{for every } u \in T$$

This is an equivalence relation, which is coarser or equal to the Myhill-Nerode equivalence relation of Teacher's language. In terms of T-equivalence we define the following properties of sets $Q, T \subseteq \Sigma^*$ that will be used in the algorithm:

- *Correctness.* All words in Q are pairwise T-non-equivalent;

- *Completeness.* For every $q \in Q$ and $a \in \Sigma$, there is some $p \in Q$ that is T-equivalent to qa.

If (Q, T) is correct and complete, then we can define an automaton as follows. The states are Q, the initial state being the empty word. When the automaton is in state $q \in Q$ and reads a letter a, it goes to the state p described in the completeness property; this state is unique by the correctness property. The accepting states are those states that are in Teacher's language.

Lemma 15.1. *If (Q, T) is correct but not complete, then using a polynomial number of membership queries, Learner can find some $P \supseteq Q$ such that (P, T) is correct and complete.*

Proof. If $q \in Q$ and $a \in \Sigma$ are such that no word in Q is T-equivalent to qa, then qa can be added to Q. The membership queries are used to test what is T-equivalent to qa. ■

The algorithm. Here is the algorithm.

1. $Q = T = \{\epsilon\}$

2. Invariant: (Q, T) is correct, not necessarily complete.

3. Apply Lemma 15.1, and enlarge Q, making (Q, T) correct and complete.

4. Compute the automaton for (Q, T) and ask an equivalence query for it.

5. If the answer is yes, then the algorithm terminates with success.

6. If the answer is no, then add the counterexample and its suffixes to T.

7. Goto 2.

Note that if (Q, T) is correct, then all words in Q correspond to different states in the minimal automaton (for Teacher's language). Furthermore, if the size of Q reaches the size of the minimal automaton, then Q represents all states of the

minimal automaton, and the transition function in the automaton for (Q, T) is the same as the transition function in the minimal automaton. Therefore, if Q reaches the size of the minimal automaton, the equivalence query in step 4 has a positive result.
To prove that the algorithm terminates, we show below that after step 6, (Q, T) is no longer complete. This will mean that step 3 will necessarily enlarge Q, and therefore the number of times we do "Goto 2" will be bounded by the size of the minimal automaton.

Lemma 15.2. *After step 6, (Q, T) is no longer complete.*

Proof. Let (Q, T) be the pair in step 4, and let $a_1 \cdots a_n$ be the counterexample, which witnesses that the automaton for (Q, T) does not recognise Teacher's language. Define T' to be T plus all suffixes of the counterexample, and suppose toward a contradiction that (Q, T') is complete. If (Q, T') is complete, then the automata for (Q, T) and (Q, T') are the same. Define q_i to be the state of either of these automata after reading $a_1 \cdots a_i$. By construction, the state q_i is a word which is T'-equivalent to $q_{i-1}a_i$, and since $a_{i+1} \cdots a_n \in T'$, it follows that

$$q_{i-1}a_i \cdots a_n \in L \qquad \text{iff} \qquad q_i a_{i+1} \cdots a_n \in L.$$

Since q_0 is the empty word, the above and induction imply that

$$a_1 \cdots a_n \in L \qquad \text{iff} \qquad q_n \in L$$

which means that the automaton gives the correct answer to the counterexample, a contradiction. ■

Problem 146. Show that one can design an algorithm for learning DFA without membership queries and counterexamples, which finds a correct DFA in exponential time. Show that one cannot do better.

Problem 147. Show that there is no algorithm, which asks only membership queries and guesses a correct DFA at the first time it asks an equivalence query. Show that the same holds for a fixed number of mistaken equivalence queries allowed.

Problem 148. Show that there is no algorithm running in polynomial time, which learns a correct DFA in the following setting: both membership and equivalence queries are allowed, but in the case when answer for an equivalence query is ″NO″ Teacher delivers no counterexample.

Bibliography

[1] Alfred V Aho and Jeffrey D Ullman. A Characterization of Two-Way Deterministic Classes of Languages. *J. Comput. Syst. Sci.*, 4(6):523–538, 1970.

[2] M H Albert and J Lawrence. A proof of Ehrenfeucht's Conjecture. *Theoretical Computer Science*, 41:121–123, 1985.

[3] Rajeev Alur and Pavol Cerný. Expressiveness of streaming string transducers. *FSTTCS*, 2010.

[4] Dana Angluin. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[5] Stefan Arnborg, Derek G Corneil, and Andrzej Proskurowski. Complexity of Finding Embeddings in a k-Tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, April 1987.

[6] Michael Benedikt, Timothy Duff, Aditya Sharad, and James Worrell. Polynomial automata - Zeroness and applications. *LICS*, pages 1–12, 2017.

[7] Anna M Bigatti, Philippe Gimenez, and Eduardo Sáenz-de Cabezón, editors. *Computations and Combinatorics in Commutative Algebra*, volume 2176 of *Lecture Notes in Mathematics*. Springer International Publishing, Cham, 2017.

[8] Roderick Bloem and Joost Engelfriet. A Comparison of Tree Transductions Defined by Monadic Second Order Logic and by Attribute Grammars. *J. Comput. Syst. Sci.*, 61(1):1–50, 2000.

[9] Hans L Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. *STOC*, pages 226–234, 1993.

[10] Mikolaj Bojanczyk. Star Height via Games. *LICS*, pages 214–219, 2015.

[11] Mikolaj Bojanczyk and Thomas Colcombet. Tree-walking automata cannot be determinized. *Theor. Comput. Sci.*, 350(2-3):164–173, 2006.

[12] Mikolaj Bojanczyk and Thomas Colcombet. Tree-Walking Automata Do Not Recognize All Regular Languages. *SIAM J. Comput.*, 38(2):658–701, 2008.

[13] J Richard Buchi. Weak Second-Order Arithmetic and Finite Automata. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 6(1-6):66–92, 1960.

[14] J Richard Buchi. State-Strategies for Games in F G. *J. Symb. Log.*, 48(04):1171–1198, 1983.

[15] J Richard Buchi and Lawrence H Landweber. Solving Sequential Conditions by Finite-State Strategies. *Transactions of the American Mathematical Society*, 138:295, April 1969.

[16] Cristian S Calude, Sanjay Jain, Bakhadyr Khoussainov, Wei Li, and Frank Stephan. Deciding parity games in quasipolynomial time. *STOC*, pages 252–263, 2017.

[17] Chandra Chekuri and Julia Chuzhoy. Polynomial Bounds for the Grid-Minor Theorem. *J. ACM*, 2016.

[18] Alonzo Church. Logic, Arithmetic, and Automata. pages 21–35, 1962.

[19] Michal Chytil and Vojtech Jákl. Serial Composition of 2-Way Finite-State Transducers and Simple Programs on Strings. *ICALP*, 52(Chapter 11):135–147, 1977.

[20] Thomas Colcombet and Damian Niwinski. On the positional determinacy of edge-labeled games. *Theor. Comput. Sci.*, 352(1-3):190–196, 2006.

[21] Thomas Colcombet and Daniela Petrisan. Automata and minimization. *SIGLOG News*, 2017.

[22] Bruno Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Information and Computation*, 85(1):12–75, March 1990.

[23] Bruno Courcelle and Joost Engelfriet. *Graph Structure and Monadic Second-Order Logic*. 2012.

[24] Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Daniel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, July 2015.

[25] Samuel Eilenberg. *Automata, Languages, and Machines, Volume A*. Pure and Applied Mathematics, a series of monographs and textbooks: vol. 59-A. Academic Press, 1974.

[26] Calvin C Elgot. Decision problems of finite automata design and related arithmetics. *Transactions of the American Mathematical Society*, 98(1):21–21, January 1961.

[27] Joost Engelfriet and Hendrik Jan Hoogeboom. MSO definable string transductions and two-way finite-state transducers. *ACM Transactions on Computational Logic*, 2(2):216–254, April 2001.

[28] Filiot, Emmanuel and Reynier, Pierre-Alain. Transducers, logic and algebra for functions of finite words. *SIGLOG News*, 2016.

[29] Zoltán Fülöp. On attributed tree transducers. *Acta Cybern.*, 1981.

[30] Seymour Ginsburg. Some remarks on abstract machines. *Transactions of the American Mathematical Society*, 96(3):400–400, March 1960.

[31] Yuri Gurevich and Leo Harrington. *Trees, automata, and games*. ACM, May 1982.

[32] K Hashiguchi. Limitedness theorem on finite automata with distance functions. *Journal of Computer and System Sciences*, 24(2):233–244, April 1982.

[33] Kosaburo Hashiguchi. Algorithms for Determining Relative Star Height and Star Height. *Inf. Comput.*, 78(2):124–169, 1988.

[34] Tsutomu Kamimura and Giora Slutzki. Parallel and two-way automata on directed ordered acyclic graphs. *Information and Control*, 49(1):10–51, 1981.

[35] Daniel Kirsten. Distance desert automata and the star height problem. *RAIRO - Theoretical Informatics and Applications*, 39(3):455–509, July 2005.

[36] Ernst W Mayr. An Algorithm for the General Petri Net Reachability Problem. *SIAM J. Comput.*, 13(3):441–460, 1984.

[37] Robert McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9(5):521–530, 1966.

[38] George H Mealy. A method for synthesizing sequential circuits. *Bell Syst. Tech. J.*, 34(5):1045–1079, 1955.

[39] Edward F Moore. Gedanken-Experiments on Sequential Machines. In *Automata Studies. (AM-34)*. Princeton University Press, Princeton.

[40] David E Muller and Paul E Schupp. Alternating automata on infinite trees. *Theoretical Computer Science*, 54(2-3):267–276, 1987.

[41] Anca Muscholl, Mathias Samuelides, and Luc Segoufin. Complementing deterministic tree-walking automata. *Inf. Process. Lett.*, 99(1):33–39, 2006.

[42] Jaroslav Nešetřil and Patrice Ossona de Mendez. *Sparsity*, volume 28 of *Algorithms and Combinatorics*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[43] Joël Ouaknine and James Worrell 0001. On linear recurrence sequences and loop termination. *SIGLOG News*, 2015.

[44] Bjorn Poonen. Undecidability in Number Theory. *Notices of the AMS*, 55(3):344–350, 2008.

[45] Michael O Rabin. Decidability of Second-Order Theories and Automata on Infinite Trees. *Transactions of the American Mathematical Society*, 141:1, July 1969.

[46] Neil Robertson and P D Seymour. Graph minors. V. Excluding a planar graph. *Journal of Combinatorial Theory, Series B*, 41(1):92–114, August 1986.

[47] Sylvain Schmitz. The complexity of reachability in vector addition systems. *SIGLOG News*, 2016.

[48] Sylvain Schmitz and Philippe Schnoebelen. The Power of Well-Structured Systems. In *CONCUR 2013 – Concurrency Theory*, pages 5–24. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[49] M P Schützenberger. On the definition of a family of automata. *Information and Control*, 4(2-3):245–270, September 1961.

[50] M P Schützenberger. Sur une variante des fonctions sequentielles. *Theoretical Computer Science*, 4(1):47–57, February 1977.

[51] Helmut Seidl, Sebastian Maneth, and Gregor Kemper. Equivalence of Deterministic Top-Down Tree-to-String Transducers is Decidable. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 943–962. IEEE, 2015.

[52] Claude E Shannon. A mathematical theory of communication, Part I, Part II. *Bell Syst. Tech. J.*, 27:623–656, 1948.

[53] Michael Sipser. Halting Space-Bounded Computations. *Theor. Comput. Sci.*, 10(3):335–338, 1980.

[54] Alfred Tarski. A Decision Method for Elementary Algebra and Geometry, 1951.

[55] J W Thatcher and J B Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1):57–81, March 1968.

[56] Wolfgang Thomas. Languages, Automata, and Logic. In *Handbook of Formal Languages*, pages 389–455. Springer, Berlin, Heidelberg, Berlin, Heidelberg, 1997.

[57] Boris A Trakthenbrot. Finite automata and monadic second order logic. *Siberian Mathematical Journal*, 3:103–131, 1962.

[58] J D Ullman and J E Hopcroft. An Approach to a Unified Theory of Automata. *Bell System Technical Journal*, 46(8):1793–1829, October 1967.

[59] Leslie G Valiant. General Context-Free Recognition in Less than Cubic Time. *J. Comput. Syst. Sci.*, 10(2):308–315, 1975.

[60] Virginia Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In *the 44th symposium*, pages 887–898, New York, New York, USA, 2012. ACM Press.

[61] Wieslaw Zielonka. Infinite Games on Finitely Coloured Graphs with Applications to Automata on Infinite Trees. *Theor. Comput. Sci.*, 200(1-2):135–183, 1998.