Advanced topics in automata theory

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 lectures were written down by the first author and the exercises by the second author, but 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

$$a_1 a_2 a_3 \cdots$$

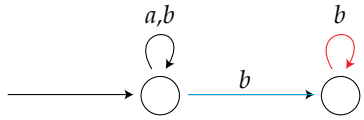
We write Σ^ω for the set of ω words over alphabet Σ. The topic of this chapter is McNaughton's Theorem, which shows that automata over ω-words can be determinised. A more in depth account of automata (and logic) for ω words can be found in [31].

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 an 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.

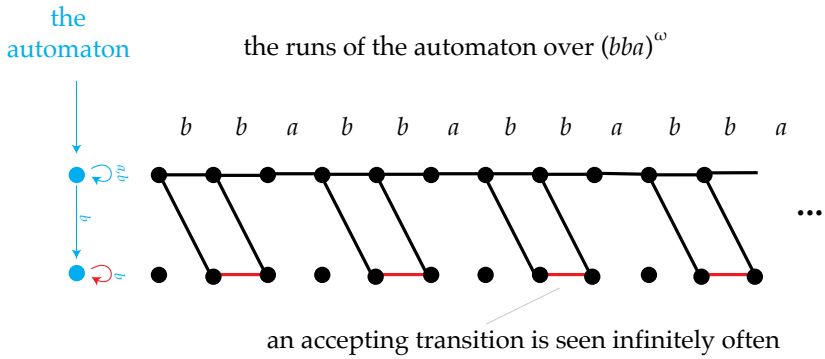
Example 1. Consider the set of words over alphabet $\{a, b\}$ where the letter a appears finitely often. This language is recognised by a nondeterministic Büchi

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:



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

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

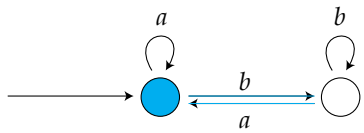
Proof. Take the automaton from Example 1. Suppose that there is a deterministic Büchi automaton that is equivalent, i.e. recognises the same language. Let us view the set of all possible inputs as an infinite tree, where the

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. ■

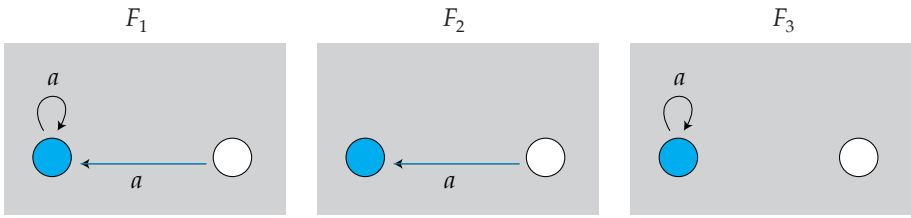
The above fact shows that if we want to determinize automata for ω-words, we need something more powerful than the Büchi condition. One solution is called the Muller condition, and is described below. Later we will see another, equivalent, solution, which is called the parity condition.

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 $\mathcal{F} \subseteq \mathsf{P}\Delta$ 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 pictured by showing the automaton with only transitions from the set

then the automaton will accept words which contain infinitely many a's and finitely many b's. Note that actually only the third set F_3 in the picture above is meaningful, because the other sets cannot be achieved as sets of transitions that appear infinitely often in a run. In particular, setting $\mathcal{F}$ to be $\{F_3\}$ would not change the recognised language.
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 $\mathsf{P}\Delta - \mathcal{F}$. 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 that from the above discussion that

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

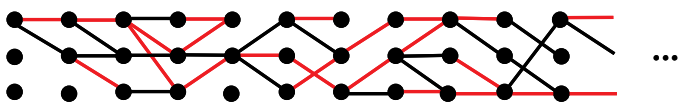
have the same expressive power, but deterministic Büchi automata are weaker. Theorem 1.2 was first proved in [22]. The proof here is similar to one by Muller and Schupp [23]. An alternative proof method is the Safra Construction, see e.g. [31].

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 finding a deterministic Muller automaton that recognising these languages. 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 parallel edges. 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*.

As we will show, 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 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

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

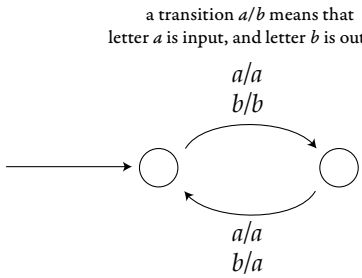
Define the *universal n state Büchi language* to be the set of words $w \in [n]^\omega$ which, when treated as a width n dag, contain a path that starts in the initial vertex and visits accepting edges infinitely often. The key to McNaughton's theorem is the following proposition.

Proposition 1.3. *For every $n \in \mathbb{N}$ there is a deterministic Muller automaton recognizing 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 from 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 with outputs, this term appears already in Shannon's original paper on information theory [29, Section 8]. Later in this book we will see other – and more powerful – types of transducers, with names like bimachine transducer or register transducer. If the input alphabet is Σ and the output alphabet is Γ, then a sequential transducer defines a function

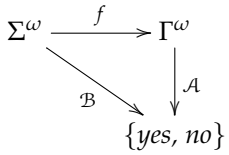
$$f : \Sigma^\omega \to \Gamma^\omega.$$

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.



□

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, f is a sequential transducer, there is a deterministic Muller automaton $\mathcal{B}$ which makes the following diagram commute:*

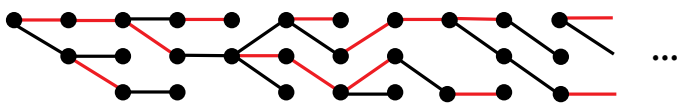


Proof. The states of automaton $\mathcal{B}$ are the product of the states for f and $\mathcal{A}$. (The assumption that the transducer is letter-to-letter is not necessary, but the construction becomes more complicated.) ■

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. By composing the transducer with the automaton from Proposition 1.3, we get a deterministic Muller automaton equivalent to $\mathcal{A}$.
It 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 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.

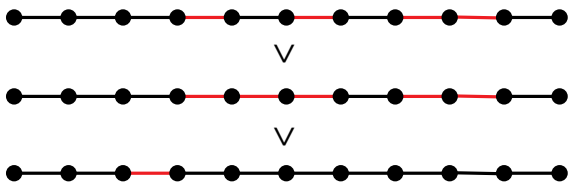
Lemma 1.6. *There exists a deterministic Muller automaton with input alphabet $[n]$ such that for every input that is a tree, the automaton accepts if and only if the input contains a path with infinitely many accepting edges.*

Combining the two lemmas, 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 Reduction to trees

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 idea is that the automaton keeps for each vertex a path which reaches the vertex and uses the first accepting edge as early as possible, and then the second accepting edge as early as possible, and so on.

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 consider order profiles lexicographically, with "accepting" is smaller than "non-accepting".



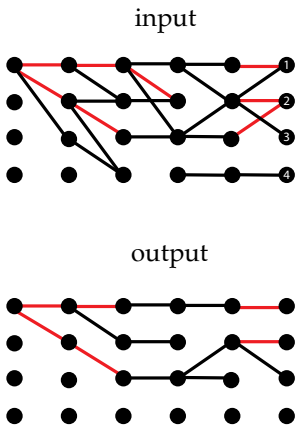
A finite path π in a width n dag is called *profile optimal* if it has lexicographically least profile among paths in w that have the same source and target.

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 $f(w)$ is profile optimal.

Proof. After reading the first n letters, the transducer keeps in its memory the lexicographic ordering on the profiles of optimal paths lead from the root to nodes at depth n. Here is a picture of the construction:



The state of the tranducer is this information:

The reachable vertices are

❶ ❷ ❸

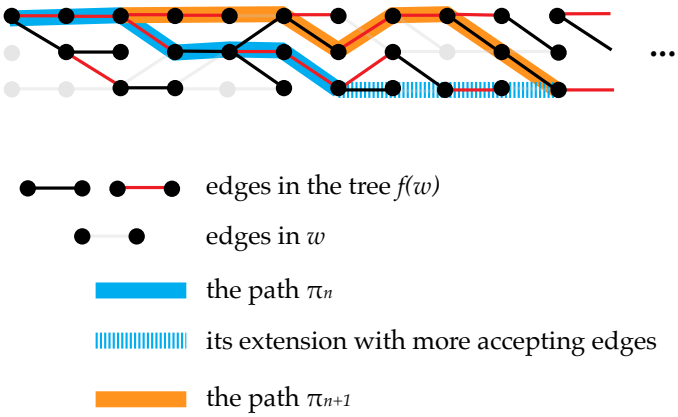
and the least profiles for reaching them are ordered as

❶ = ❷ < ❸

This information is enough to update the output. ∎

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.*

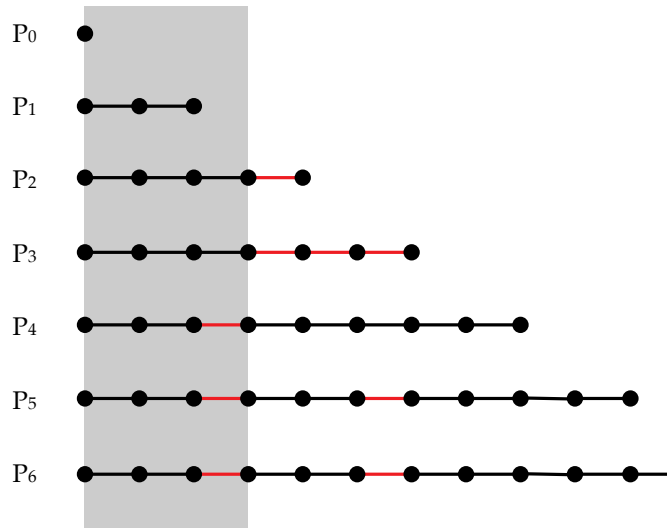
Proof. Assume that the input w to f contains a path with infinitely many accepting edges. Define a sequence $\pi_0, \pi_1, \ldots$ of finite paths in $f(w)$ as follows by induction. In the definition, we preserve the invariant that each path in the sequence $\pi_0, \pi_1, \ldots$ can be extended to an an accepting path in the input 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 path in the graph w, and therefore we can extend π_n to a finite path 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 has the same source and target as the new path that extends π_n with at at leat one accepting edge.



Define P_n to be the profile of the path π_n. 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.*

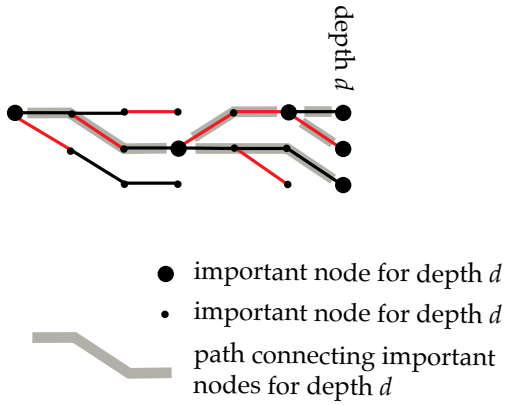
Proof. Toward a contradiction, suppose that P has the letter "accepting" finitely often, i.e. there is some i such that after i, only the letter "non-accepting" appears in P. Choose n so that $\pi_n, \pi_{n+1}, \ldots$ have profile consistent with P on the first i letters. By construction, the profile 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. ■

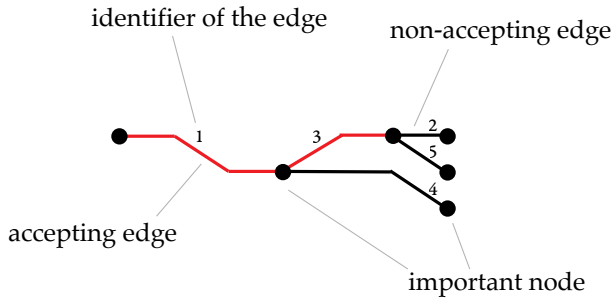
1.3 *Finding an accepting path in a tree graph*

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 a node to be *important* for depth d if it 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 solid lines representing accepting edges, and dotted lines representing non-accepting edges):



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 corresponds 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

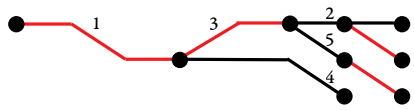
therefore it has less than $2n$ edges. The automaton also keeps track of a colouring of the edges, with each edge being marked as accepting or not, where "accepting" means that the corresponding path in the input tree contains at least one accepting edge. Finally, the automaton remembers for each edge an identifier from the set $\{1, \ldots, 2n-1\}$, with the identifier policy being described below. A typical memory state looks like this:



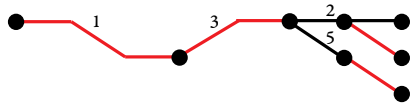
All identifiers are distinct, i.e. different edges get different identifiers. It might be the case (which is not true for the picture above), that the identifiers used at a given moment have gaps, e.g. identifier 4 is used but not 3.
The initial state of the automaton is a tree which has one node, which 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 do 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 the collapsed path is inherited from the first edge on the path. If the collapsed path had at least one accepting edge, then it turns into an accepting edge, otherwise it turns into a non-accepting edge. In the above picture, this means eliminating the unary nodes that are the targets of edges with identifiers 1 and 5:

Note how the edge with identifier 5 becomes accepting even though previously it was not accepting, this is because it got merged with an accepting edge.

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. In the above example, identifiers 3 and 4 are deleted, and both are then added again.

2. **Refresh.** In step 3, a whole path can be collapsed into its first edge. If there was an accepting edge in the newly added part, i.e. in any edge of the collapsed path not counting the first edge, then we say that the identifier of the first edge (and also of the collapsed path) is *refreshed*. This is the case for identifiers 1 and 5 in the above example.

3. **Nothing.** An identifier might be neither deleted nor refreshed. This is the case for identifier 2 in the above 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 state. 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 regular:

1. prefix of v belongs infinitely often to the fixed regular language of finite words $L \subseteq \Sigma^*$;
2. word v contains infinitely many infixes of the form $ab^p a$, where p is prime;
3. word v contains infinitely many infixes of the form $ab^p a$, where p is even.

Problem 2. Let UP be the set of *ultimately periodic words*, i.e. $UP = \{uv^\omega : u, v \in \Sigma^*\}$. Let L be a regular language of infinite words. Show that if $L \cap UP = \emptyset$ then $L = \emptyset$.

Problem 3. Let K and L be regular languages of infinite words. Show that if $L \cap UP = K \cap UP$ then $K = L$.

Problem 4. Are the following languages regular:

1. word v contains arbitrary long infixes in the fixed regular language of finite words L;
2. prefix of v belongs infinitely often to the fixed language of finite words $L \subseteq \Sigma^*$ (not necessarily regular).

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

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

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

Problem 8. Assume that we have changed the acceptance condition into such which investigates which sets of transitions are visited infinitely often. Does it affect the expressivity of automata? How it is for Büchi acceptance condition? And how for Muller acceptance condition?

Problem 9. Show that nonemptiness is decidable for automata with Muller acceptance condition.

Problem 10. Let us define a metric on infinite words: $d(u,v) = \frac{1}{2^{\mathrm{diff}(u,v)}}$, where $\mathrm{diff}(u,v)$ is the smallest index on which u and v differ. Language L jest open (in this metric) if for every $w \in L$ there exist some open ball centered in w which is included in L (so the standard definition). Prove that the following conditions are equivalent for a regular language L:

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

Problem 11. Consider a transducer, which defines a function $f : \Sigma^\omega \to \Gamma^\omega$ and metrics on Σ^ω and Γ^ω defined as $d(u,v) = \frac{1}{2^{\mathrm{diff}(u,v)}}$. Show that such an f is continuous.

Problem 12. We will look for a candidate for Myhill-Nerode relation for infinite words, i.e. an equivalence relation $\sim_L$ such that $\sim_L$ has finite index iff L is regular. Check whether this fact is true for the following relations

1. $\sim_L \subseteq \Sigma^* \times \Sigma^*$ such that $u \sim_L v$ iff for all $w \in \Sigma^\omega$ it holds $uw \in L \Leftrightarrow vw \in L$;
2. $\sim_L \subseteq \Sigma^\omega \times \Sigma^\omega$ such that $u \sim_L v$ iff for all $w \in \Sigma^*$ it holds $wu \in L \Leftrightarrow wv \in L$;
3. $\sim_L \subseteq \Sigma^* \times \Sigma^*$ such that $u \sim_L v$ iff for all $w \in \Sigma^\omega$ and $s,t \in \Sigma^*$ it holds $uw \in L \Leftrightarrow vw \in L$ and $s(ut)^\omega \in L \Leftrightarrow s(vt)^\omega \in L$.

2
Infinite duration games

In this chapter, we prove the Büchi-Landweber Theorem [10, Theorem 1], see also [31, 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 [12] and sometimes called "Church's Problem".

2.1 *Games*

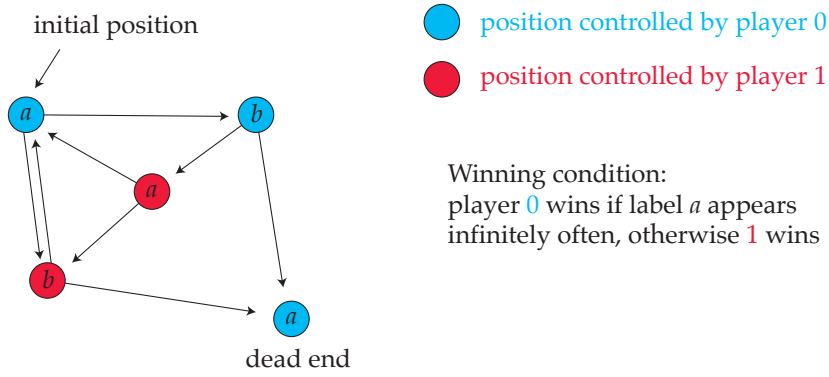
In this chapter, we consider games played by two players (called 0 and 1), which are zero-sum, perfect information, and most importantly, of potentially infinite duration.

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

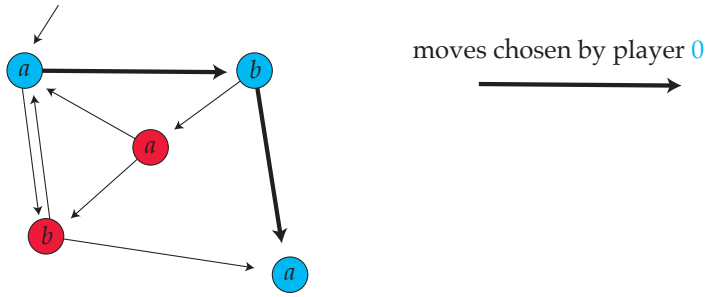
Intuitively speaking, the winning condition inputs a sequence of labels produced in an infinite play, and says which player wins. The definition is written in a way which highlights the symmetry between the two players; this symmetry will play an important role in the analysis. Here is a picture.



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 decided using the winning condition.
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 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 which are not determined because none 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 controlled by i, a function f_v from M to the neighbors 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 where there might be infinitely many positions, but we will only care about finite memory sets M (which leads us to consider finite state automata over infinite alphabets). An important special case is when the set M has only one element, in which case the strategy is called *memoryless*. In this case, 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.

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

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

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.*

The above theorem is proved in Section 2.2. The second step of the Büchi-Landweber theorem is the reduction to parity games. This essentially boils down to transforming deterministic Muller automata into called deterministic parity automata. In a parity automaton, there is a ranking function from states to numbers, and a run is considered accepting if the minimal 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 [17]. 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*

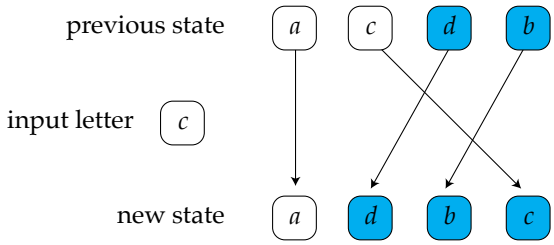
$$g : Q \to \mathcal{P}(\Sigma)$$

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 biggest 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 blue positions, and in case of the same number of blue 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 plays 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). In a position (v, q), the player controlling position v chooses an edge in the original game, and the state is updated deterministically according to the transition function of the automaton. We view the product game as a parity game, with the ranks inherited from the parity automaton. 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.

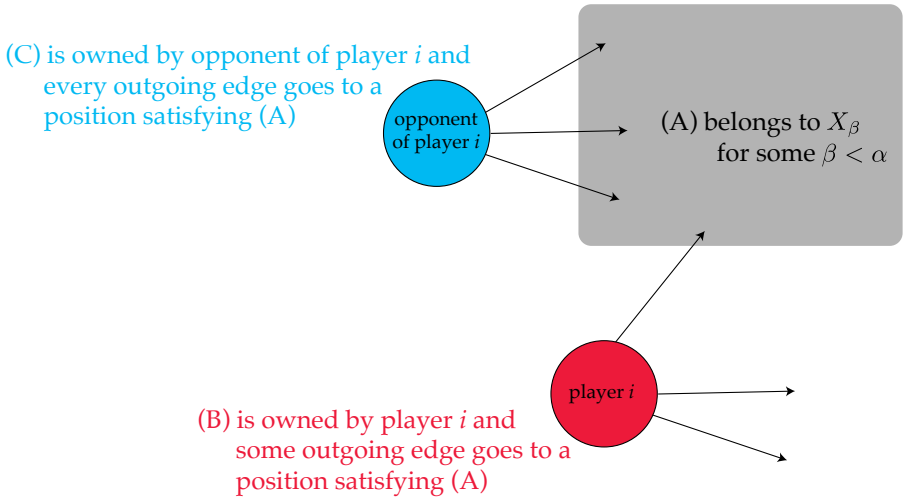
The implication from 1 to 2 crucially uses determinism of the automaton and would fail if a nondeterministic automaton were used (under an appropriate definition of a product game). Since the product game is a parity game, Theorem 2.5 says that for every position v, condition 2 must hold for one of the 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 [34] and [31]. 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

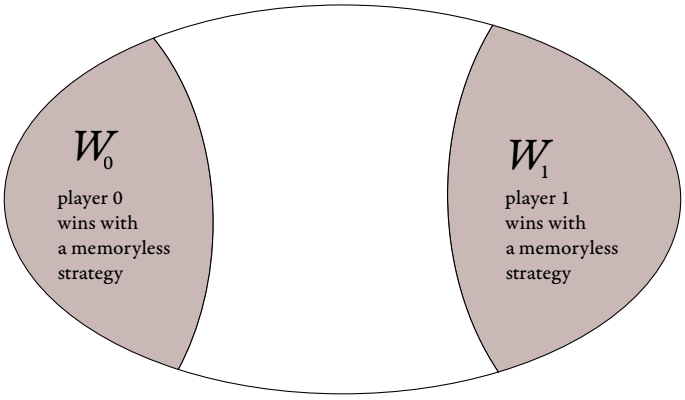
no ranks at all, and hence the theorem is vacuously true. For the induction step, we use the notion of attractors, which is defined below.

Attractors. Consider a set of positions X in a parity game (actually the winning condition is 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 the set X. The attractor is approximated using ordinal numbers. Define X_0 to be X. 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:



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 end up in X or 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 W is empty. Here is the picture:



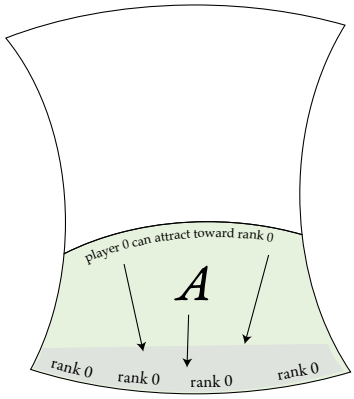
By definition, for every position in $w \in W_i$, player i has a memoryless winning strategy that wins when starting in position w. In principle, the memoryless strategy might depend on the choice of w, but the following lemma shows that this is not the case.

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,

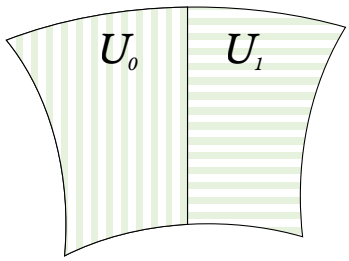
under a well-founded ordering, of some set that is nonempty (because it contains w). Define a consolidated strategy as follows: when in position w, play according to the strategy of the companion of w. The key observation is that for every play using this consolidated strategy, the sequence of companions is non-decreasing 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. Some positions that were not dead ends in the original game might become dead ends in the game restricted to U, because outgoing edges to position outside U are removed in the restriction process. Define A to be the 0-attractor, inside the game limited to U, of positions that are in U and have minimal rank 0. Here is the picture of the game restricted to U:

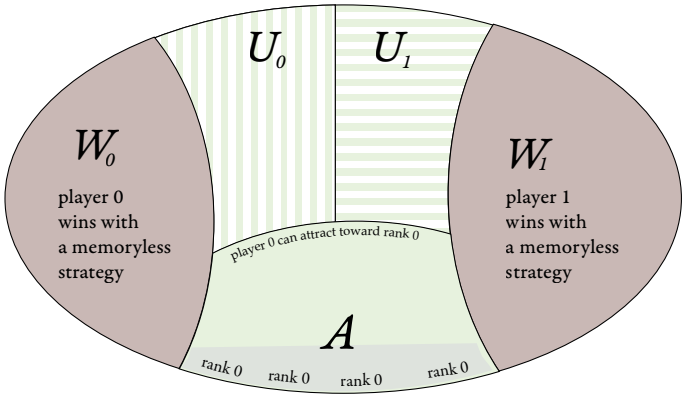


Consider a position in A that is controlled by player 1. In the original game, all outgoing edges from the position go to $A \cup W_0$; because if there would be an edge to W_1 then the position would also be in W_1. It follows that in the original game, if the play begins in a position from A and player 0 plays the attractor strategy on the set A, then the play is bound to either end up in a position in U

that has rank 0, or in the set W_0. Let us consider the game restricted to set $U - A$. 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 U_i has a memoryless winning strategy, assuming that the game is restricted to $U - A$:



Here is how the sets U_0, U_1 can be interpreted in terms of the bigger original game. For every $i \in \{0,1\}$, if in the original game, the play begins in a position from U_i and player i uses the memoryless winning strategy corresponding to U_i, then either the play stays forever in $U - A$ and player i wins, or it eventually leaves $U - A$.
Here is a picture of the original game with all sets:



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 never leaves $U - A$, then player 1 wins by assumption on the strategy. Suppose that the play does leave $U - A$. If it enters W_0 or A, this would have to be a choice of player 0, but 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 entire game, consider the following memoryless strategy for player 0:

- in U_0 use the winning memoryless strategy inherited from the game restricted to $U - A$;
- in W_0 use the winning memoryless strategy from Lemma 2.8;
- in A use the attractor strategy to reach rank 0 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. Next we observe that 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, by using the attractor, 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 13. We say that game is *finite* if its game tree is finite. Prove that every finite game is determined.

Problem 14. Show that a finite reachability game can be solved in polynomial time.

Problem 15. We will now show an example of a game, which is not determined. We will construct this example by a sequence of a few exercises. First consider a following riddle. There are infinitely many players (countable many), every one has a hut: either white or black. Everybody sees the color of everybody else hut, but not of his own. Everybody should say what is the color of his hut, such that only finitely many players will make a mistake. They can fix a strategy before, but they cannot tell anything to each other after they will see the huts. What is the winning strategy?

Problem 16. Define a function xor : $\{0,1\}^\omega \to \{0,1\}$, called an *infinite xor*, such that changing one bit or an argument changes the result.

Problem 17. Consider the following two player game. 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 have to pick on still existing piece and eat with piece together with everything which is towards the left and bottom from the picked piece (the picked piece is right upper corner of the currently eaten part). The one who has to eat rotten piece loses. Determine who has a winning strategy.

Problem 18. Define a non determined game.

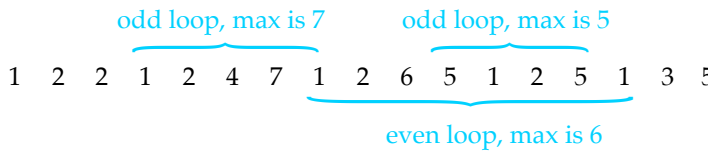
3
Parity games in quasipolynomial time

In this chapter, we show that parity games with n positions can be solved (i.e. one can compute which player wins) in quasipolynomial time (i.e. time of the form n to a power that is polylogarithmic in n). More precisely we present an algorithm that solves parity games in time at most $n^{\mathcal{O}(\log n)}$. Whether or not parity games can be solved in polynomial time is an important open problem. The presentation here is based on the original paper [11], with some new terminology (notably, the use of separation).

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

The strategy of the proof is to reduce parity games to reachability games in quasipolynomial time, and then solve reachability games in linear time.

A safety automaton for loop parity. Define a *safety automaton* to be the special case of a deterministic parity automaton where all states have even rank (e.g. all states have rank 0) and the transition relation is a partial function $Q \times \Sigma \to Q$. The only way that a safety automaton can reject an input is by reaching a situation where the transition function is undefined.
Consider an ω-word of natural numbers. A *loop* is an infix that begins and ends with the same letter. An loop is called *even* if the maximal number in it is even, an *odd* otherwise. Here is an example:

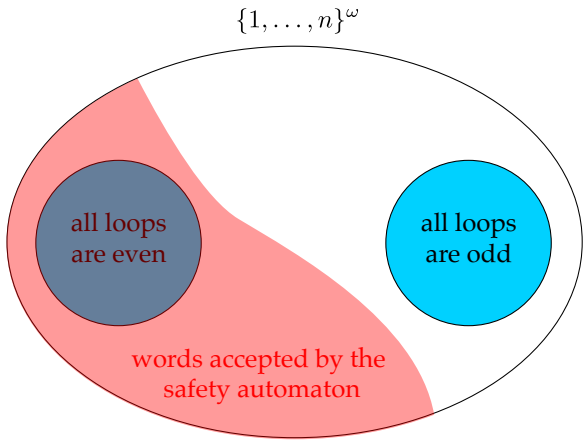


The following is the key combinatorial construction in the proof of Theorem 3.1.

Lemma 3.2. *Let $n \in \mathbb{N}$. There is a safety automaton with input alphabet $\{1, \ldots, n\}$ and at most $(n+1)^{\lceil \log n \rceil}$ states such that:*

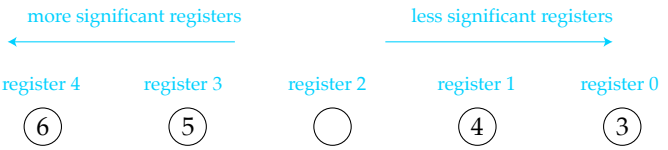
- *it accepts every input that has only even loops;*
- *it rejects every input that has only odd loops.*

Here is a picture:



Proof. We prove a stronger variant of the lemma; our automaton will reject all words which violate the parity condition, and not just those where all loops are odd. For solving parity games, we use the weaker and more symmetric statement of the lemma.
Let $k = \lceil \log n \rceil$. The automaton uses k registers with names $\{0, \ldots, k-1\}$. Each register stores a number from $\{1, \ldots, n\}$, or it is undefined. A state of the

automaton is a valuation of these registers, the number of states is therefore as in the statement of the lemma. Furthermore, we respect the invariant that when restricted to defined values, the numbers stored in the registers are non-decreasing, which makes the state space smaller. Here is a picture of a state of the automaton, where blanks indicate undefined numbers:



In the initial state, all registers are empty. The automaton does the following when reading an input letter $a \in \{1, \ldots, n\}$. Consider the following two registers (which might be undefined):

1. most significant nonempty register that stores value $< a$;
2. (only defined if a is odd:) least significant register that is empty or stores an even number.

If at least one of the above registers is defined, then value a is inserted into the more significant one that is defined, call it r, and all registers $< r$ are emptied. If neither of the registers is defined and a is even (which means that all nonempty registers have values $\geq a$, because the register from item 2 is necessarily undefined for even inputs), then the state is left unchanged. If neither of the registers is defined and a is odd (which means that all registers are nonempty and store odd values $\geq a$), then the automaton rejects. This completes the definition of the automaton.

Claim 3.3 (Invariant)**.** *Suppose that the automaton has read a finite word w so far, and has not rejected. Then the register valuation is nondecreasing on nonempty registers, i.e. register contents grow as registers become more significant. Furthermore, one can associate to each nonempty register r a word, such that:*

1. *the concatenation of these words, in decreasing order of nonempty registers, forms a suffix of the input read so far; and*

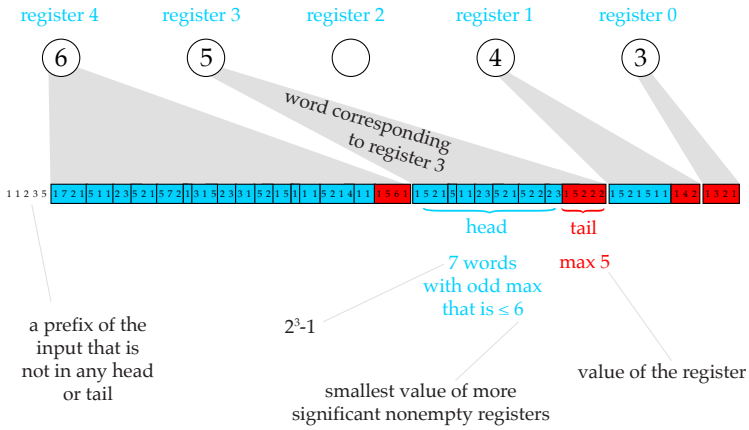
2. *if a register r is nonempty and stores a, then:*

   (a) *all words associated to nonempty registers $< r$ use letters $\leq a$;*

   (b) *the word associated to r is a concatenation of:*

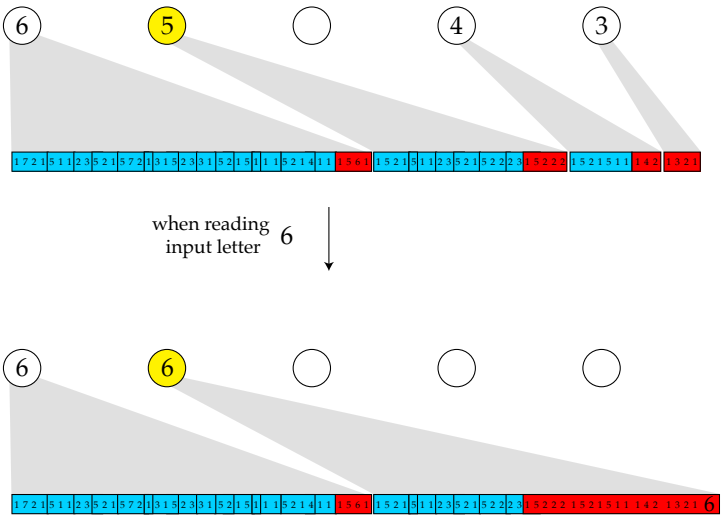
   - HEAD: $2^r - 1$ *words with odd maximum, followed by*
   - TAIL: *a word with maximum exactly a.*

Here is a picture of the invariant:



Proof. We prove that the invariant is preserved. Suppose that the invariant holds for some input word w, and the input letter is a. Recall the two registers in items 1 and 2 from the definition of the state update function. We prove the invariant depending on which one is used in the update function.

1. The first case is when the input letter a is inserted into the most significant register, call it r, which stores value $< a$. To preserve the invariant, we append all words (head and tail) corresponding to nonempty registers $< r$, plus the input letter, to the tail of register r. Here is the picture, with register r in yellow:



2. The second case is when an odd letter a is inserted into the least significant register, call it r, that is empty or contains an even value. To preserve the invariant, we do the following

   (a) If r was nonempty before the update (in which case it had an even value), then append the word corresponding to r to the head of the closest most significant register, call it s. If s does not exist, i.e. r was the most significant nonempty register, then the head and tail of r become part of the input prefix that is not covered by any head or tail.

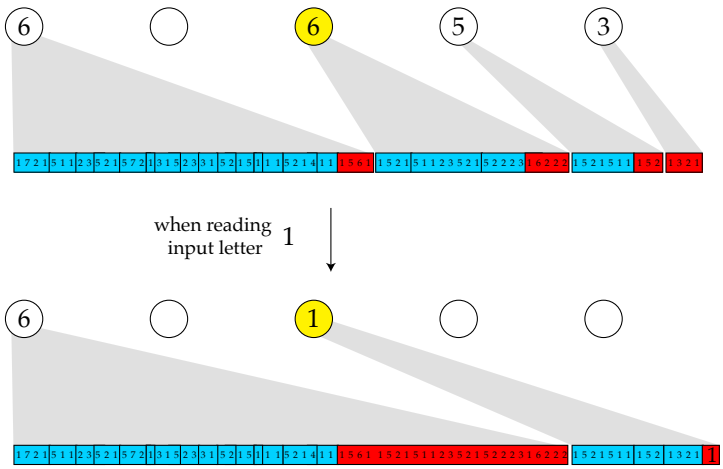
   (b) Put all words (head and tail) corresponding to registers $< r$ into the head of r. Before the update, the head of register $s < r$ could be decomposed as $2^s - 1$ words with odd maximum, and the tail was one more such word. Therefore, after the update the head of register r decomposes into

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

   words with odd maximum, preserving the invariant.

(c) Put the input letter into the tail of r. The newly inserted letter a is not bigger than any nonempty register $> r$, because otherwise we would have chosen the first item.

Here is the picture, with register r in yellow:



This completes the proof of the invariant. ∎

Claim 3.4. *If the automaton rejects, then the input contains an odd loop.*

Proof. From the invariant it follows that if the automaton rejects, then some suffix of the input can be decomposed into 2^k parts, each one with odd maximum. By the pigeonhole principle, at least two of these parts must have the same maximum, say achieved in positions i and j of the input word. The infix from i to j is a loop, and its maximal number is odd (it is either the number stored in position i, or one of the maximal numbers in the parts between i and j). ∎

Claim 3.5. *If the input violates the parity condition, then the automaton rejects.*

Proof. For $i \in \{1, \ldots, k\}$, define $\mathcal{A}_i$ to be variant of the automaton as described in the lemma, except that the number of registers is i instead of k. By induction on i, we prove the following generalisation of the claim, which yields the claim for the case of $i = k$:

(*) Suppose that $\mathcal{A}_i$ is initialised in some state q (not necessarily the initial state with all registers empty), and the input violates the parity condition. Then $\mathcal{A}_i$ will reject, i.e. it will read an odd letter in a state where all registers are nonempty and odd.

Suppose that we have already proved (*) for $i - 1$ (or $i = 1$ and there is nothing to prove) and we want to prove (*) for i. Consider a run of $\mathcal{A}_i$ on an input which violates the parity condition, i.e. the maximal number that appears infinitely often is some odd $a \in \{1, \ldots, n\}$. By the induction assumption, the most significant register must eventually become nonempty, because transitions that do not affect register i are transitions of the automaton $\mathcal{A}_{i-1}$. Wait until all letters $> a$ have already been read, and let b be the letter that is in register i at this moment. Again by induction assumption, at some point the automaton will reject, or it will replace b by some number that is $\leq a$. The next time a is seen, register i will be set to a (unless it was already a), and this a will never be dislodged from the register i for the rest of the run. Again by the induction assumption, the automaton will eventually reject. ∎

The contrapositive of Claim 3.4 is that if the input contains only even loops, then the automaton accepts. On the other hand, if all loops are odd, then the parity condition is violated, and by Claim 3.5 the automaton rejects. This finishes the proof of the lemma. ∎

Proof of Theorem 3.1. Let G be a parity game. Without loss of generality, we assume that each vertex has a different parity rank from $\{1, \ldots, n\}$. This can be done by using n which is at most twice the number of vertices. Apply Lemma 3.2 to n, yielding a safety automaton $\mathcal{A}$. Consider a product game $G \times \mathcal{A}$, as defined on page 28, 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. In other words, player 0 wins the game $G \times \mathcal{A}$ if a dead end of player 1 is reached, or if the play is infinite and accepted by $\mathcal{A}$.

Claim 3.6. *If player $i \in \{0,1\}$ wins G, then player i also wins $G \times \mathcal{A}$.*

Proof. By symmetry, take $i = 0$. (Here it is convenient that the statement of Lemma 3.2 is symmetric.) 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 arena of G by fixing the memoryless strategy σ_0; paths in this graph 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. In other words, every infinite path in G_0 is accepted by the automaton $\mathcal{A}$. Therefore, the same strategy σ_0 also wins in the game $G \times \mathcal{A}$. ∎

Because $\mathcal{A}$ is a safety automaton, the product game $G \times \mathcal{A}$ is a safety game, i.e. the only way that player 0 can lose is by reaching a dead end. The size of this game n times the number of states in $\mathcal{A}$, i.e. consistent with bound in the theorem. Therefore, it remains to prove the following claim.

Claim 3.7. *The winner in a safety game can be found in time linear in the number of edges.*

Proof. By computing the attractors (from the perspective of player 1) of the dead ends of player 0, see the proof of Theorem 2.5. ∎

∎

Problem 19. Consider the following variant of the automaton from Lemma 3.2. Only odd numbers are kept in the registers, and the update function is the same as in Lemma 3.2 when reading an odd number. When reading an even number a, the automaton erases all registers store values $< a$. Show that this automaton does not satisfy the properties required in Lemma 3.2.

Problem 20. 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 21. Show that there is no safety automaton with $< n$ states which satisfies the properties required in Lemma 3.2.

Problem 22. A probabilistic reachability automaton is defined like a finite automaton, except that each transition is assigned a probability (a number in $[0;1]$) such that for every state, the sum 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 $\{1,\dots,n\}^\omega$, 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.

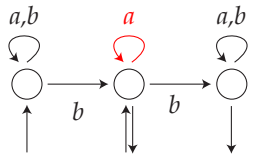
Problem 23.

Problem 24. Show that there is no safety automaton with $< n$ states which satisfies the properties required in Lemma 3.2.

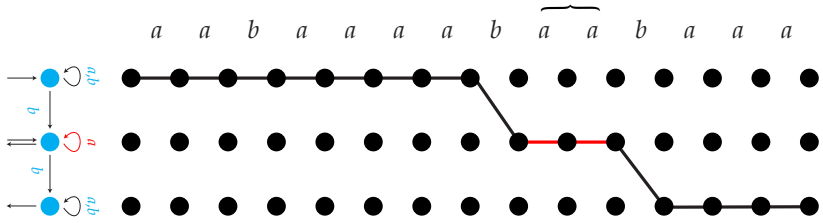
4
Distance automata

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 [18]. The theorem was part of Hashiguchi's solution to the star height problem [19].

Theorem 4.1 (Limitedness of distance automata)**.** *The following problem is decidable:*

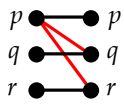
- **Input.** *A distance automaton.*
- **Question.** *Is the automaton bounded in the following sense: there is some $m \in \mathbb{N}$ such that every input word admits an accepting run of cost at most m.*

The problem in the above theorem was called limitedness in [18]. The algorithm we use, based on [5], uses the Büchi-Landweber theorem from Chapter 2.

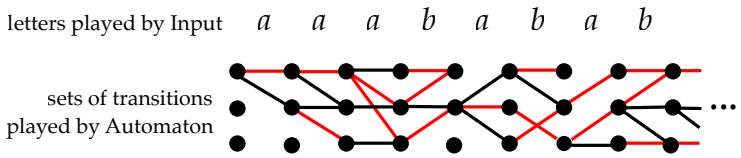
The limitedness game. Let us fix a distance automaton. For a number $m \in \mathbb{N} \cup \{\omega\}$, consider the following game, call it the *limitedness game with bound m*. The game is played in infinitely many rounds $1, 2, 3, \ldots$, by two players called Input and Automaton. In each round:

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



The picture will have length m. The winning condition for player Automaton is the following:

1. In every column, an accepting state must be reachable from an initial state in the first column; and

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

If either of the condition above is not met, then player Input loses. 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:*

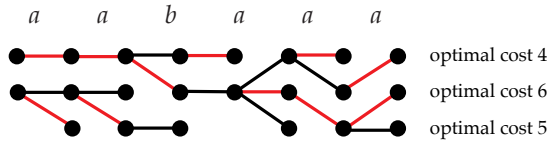
1. *the automaton is limited;*

2. *there is some m such that player Automaton wins the limitedness game with bound m;*

3. *player Automaton wins the limitedness game with bound ω*

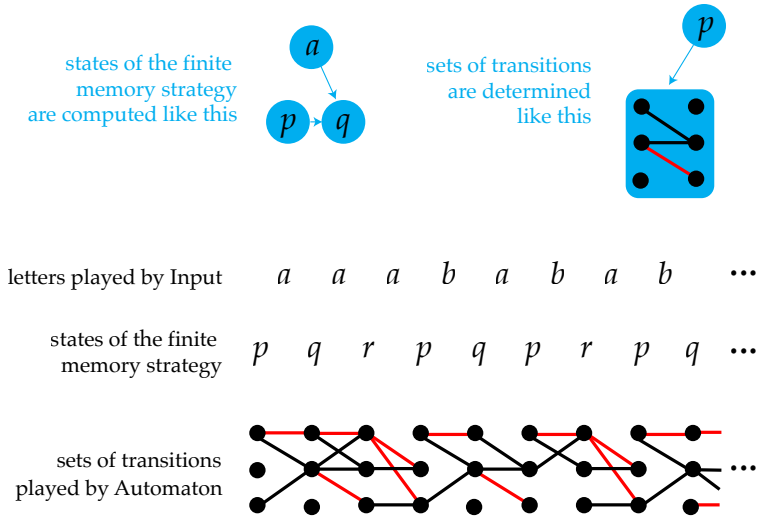
Proof. The implications from 2 to 1 and from 2 to 3 are immediate. For the other implications and the decidability part, the key is the observation that for every choice of $m \in \mathbb{N} \cup \{\omega\}$, the limitedness game is a special case of a game with a finite arena and an ω-regular condition. 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. Here is the strategy of player automaton. 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: if the input word is $a_1 \cdots a_i$, then in the i-th round, player Automaton responds with sets of transitions 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:



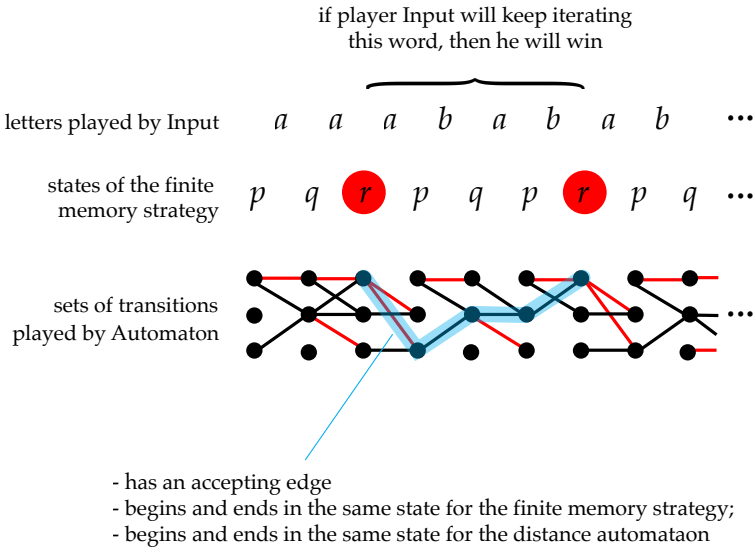
When player Input gives a new letter, player Automaton responds with a set of transitions in that connect the new configurations in a cost-minimising way.

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.
By unfolding the definition of a finite memory strategy in the limitedness game, there is a deterministic automaton, call it the strategy automaton, over the input alphabet of the original distance automaton, and a function f from the states of the strategy automaton to sets of transitions in the origial distance automaton, such that if in the first i rounds the letters produced by player Input were $a_1 \cdots a_i$, then the response of player Automaton in the i-th round is obtained by applying f to the state of the strategy automaton after reading $a_1 \cdots a_i$. Here is a picture of how the strategy works:



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 for the strategy), thus proving the implication from 3 to 2 in the

lemma. To prove the claim, suppose that the strategy produces a run exceeding the above bound. Using a pumping argument we can find a loop that can be exploited by player Input to force player Automaton into a path that has infinitely many costly edges, as in the following picture:



■

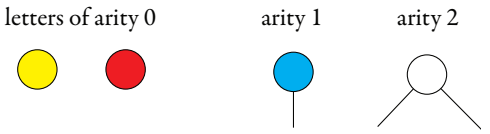
Problem 25. Show that limitedness remains decidable when distance automata are equipped with a reset operation. (The cost of a run is the biggest number of costly transitions between some two consecutive resets.)

Problem 26. 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.

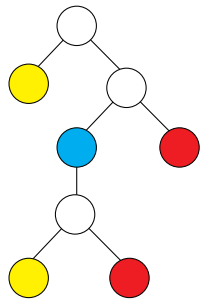
5
Tree-walking automata

In this chapter we discuss automata on finite trees. We introduce two models: branching tree automata and tree-walking automata. The main result is that tree-walking automata do not determinise.

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 to be a finite (rooted) tree where every node has a label from Σ. Furthermore, for each node the number of children is the same as the rank of the label, and we assume that these children are ordered, i.e. one can speak of the first child, second child, etc. Here is a picture of a finite tree over the alphabet from the example above:



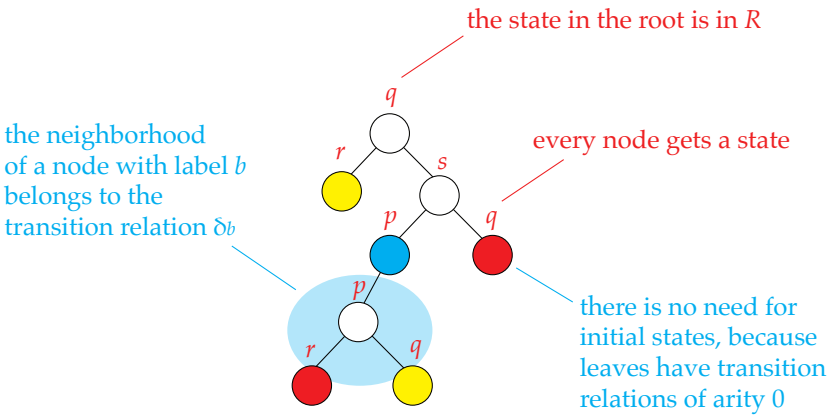
Trees are defined above are called *ranked and ordered*; one can consider other variants, where the label does not determine the number of children (unranked) or where the siblings are not ordered (unoredred). One can also consider infinite trees, this will be done in Chapter 6.
The goal of this chapter is to discuss several automata models for ranked and ordered trees (we will simply call them trees from now on).

Branching tree automata. We begin with branching tree automata. Because this is the "right" model of automata for trees, branching tree automata also known as tree automata (without any further qualifications).

Definition 5.1. *A* nondeterministic (branching) 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* $\delta_a \subseteq Q^n \times Q$*.*

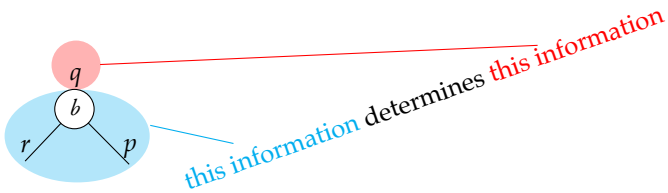
An automaton is called (bottom-up) deterministic *if every transition relation is a function* $Q^n \to Q$*. A tree is accepted by the automaton if there exists an accepting run, as explained in the following picture:*



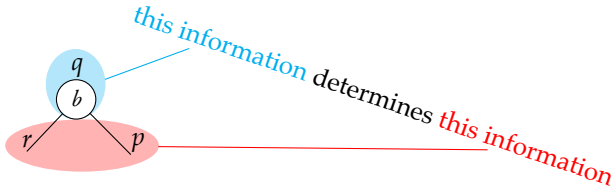
Lemma 5.2. *Languages recognised by nondeterministic tree automata are closed under union, intersection and complementation.*

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. Since bottom-up deterministic automata can be complemented by complementing the root states, we get the lemma. ∎

In the above lemma, we used bottom-up deterministic automata, and argued that they are equivalent to nondeterministic ones. This type of determinism can be illustrated as follows



Let us describe a different type of determinism, called *top-down* determinism. In such an automaton the transition relation for letters of rank n is a partial function of type $Q \to Q^n$, as in the following picture:



In a top-down deterministic automataon, we also require that there is only one root state. The only way that to-down deterministic automaton can reject a tree is by encountering an undefined result in the partial transition function.
Top-down deterministic automata are strictly less expressive than nondeterministic ones (and therefore also bottom-up deterministic ones), see Exercise 27.
This completes our discussion of branching tree automata. We now turn to tree-walking automata, where the combinatorics are more challenging.

5.1 *Tree-walking automata.*

Tree-walking automata are an alternative automaton model for trees, where the computation is sequential, i.e. it does not split into parallel threads. The idea is that at any given moment of its run, the automaton is in a single node of the input tree. Based on the current state and the local view, the automaton chooses the new state and a neighbour to move to (or to accept/ reject). The *local view* is defined to be the pair

$$(a, i) \in \underbrace{\Sigma \times \{\text{root}, 1, \ldots, n\}}_{\text{views}_\Sigma}$$

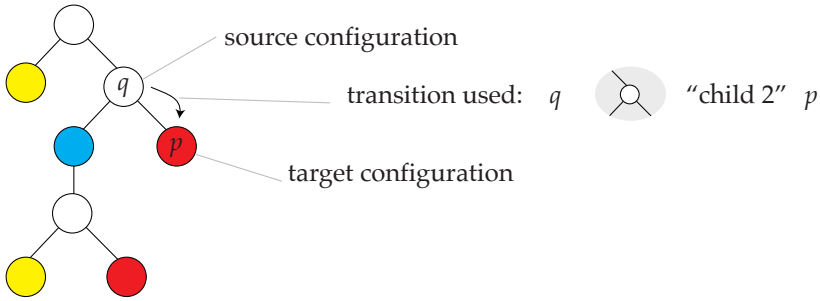
where a is the the label of the current node, and i says if the current node is the root, or otherwise what is its child number (here n stands for the maximal arity of a letter in the input alphabet).

Definition 5.3 (Tree-walking automaton)**.** *The syntax of a* (nondeterministic) tree-walking automaton *consists of: a finite ranked alphabet* Σ *called the* input alphabet*, a finite set of states* Q *with a designated initial state* q *and accepting states* $F \subseteq Q$*, and a transition relation*

$$\delta \subseteq \underbrace{Q \times \mathsf{views}\Sigma}_{\text{source state and view}} \times \underbrace{\{\mathit{parent}, \mathit{child}\ 1, \ldots, \mathit{child}\ n\}}_{\text{where to move, } n \text{ is maximal arity in } \Sigma} \times \underbrace{Q}_{\text{target state}}$$

We assume that for every transition $(q, \tau, d, p) \in \delta$*, the direction* d *is consistent with the view* τ *in the sense that if* d *is "parent" then* τ *is the view of a node with a parent, and if* τ *is "child* i*" then* τ *is the view of a node with at least* i *children. The automaton is called* deterministic *if the transition relation is a function from the first two arguments to the last two arguments.*

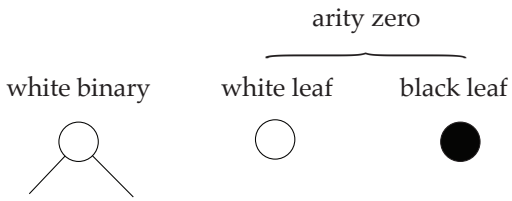
The semantics of a tree-walking automaton are defined like for two-way automata on words. Suppose that t is a tree over the input alphabet. A *configuration* of the automaton is a pair (state, node of the input tree). A run of the automaton is a sequence of configurations, such that every two consecutive 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 the initial state at the root of the tree, and which ends in a configuration that uses an accepting state. Note that there are two ways of rejecting an input tree: (a) reaching a

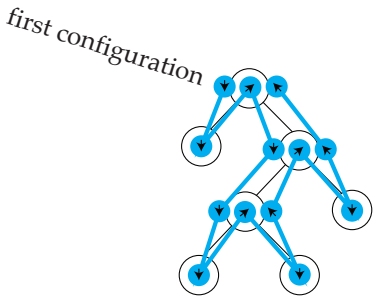
configuration that has no applicable transition; (b) entering an infinite loop. Using a construction from [30, Theorem 1], one can show that every deterministic tree-walking automaton can be modified so that rejection is done only via (a), see [24, Proposition 1].

Example 6. Consider the following ranked alphabet



The trees with at least one black leaf can be recognised by an nondeterministic tree-walking automaton, which simply guesses nondeterministically a path to a black leaf, and accepts if it sees one. If we want to avoid nondeterminism, or we want to check if all leaves are white, we can use an automaton that does a depht-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:

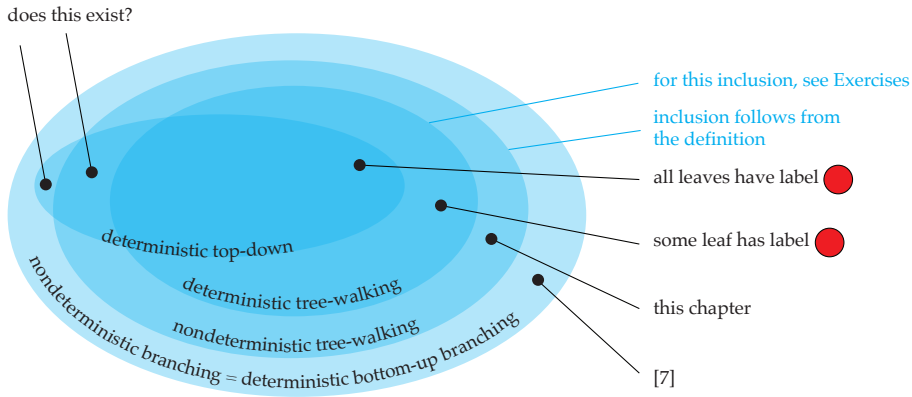


If we want the automaton to recognise the language "at least one black leaf", then it accepts whenever it sees a black leaf. If we want the automaton to recognise the language "all leaves are white", then it rejects whenever it sees a black leaf, and accepts when it reaches the root for the third time. □

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

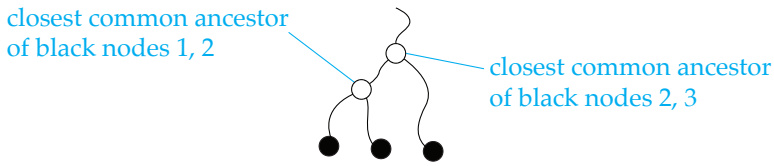
Theorem 5.4. *[6, Theorem 5] There is a tree language that is recognised by a nondeterministic tree-walking automaton, but not by any deterministic tree-walking automaton.*

In the exercises, we show that nondeterministic tree-walking automata can be converted into branching tree automata. The converse does not hold, i.e. there is a language recognised by a branching tree automaton that is not recognised by any (nondeterministic) tree-walking automaton [7, Theorem 2]. Summing up, here is the picture on the expressive power of various models of tree automata, including two regions about which we do not know if they are occupied.

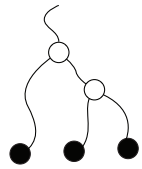


The rest of this chapter is devoted to proving Theorem 5.4. We begin by defining the language which separates deterministic and nondeterministic tree-walking automata, and we show that the language can be recognised by a

nondeterministic tree-walking automaton. In the next section we show the lower bound – the separating language cannot be recognised by a deterministic automaton.
The input alphabet for the separating language is the same as in Example 6, 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:

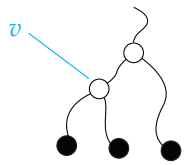


and not like this:

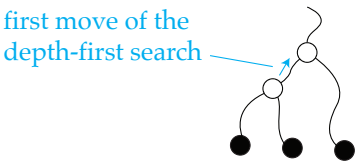


Lemma 5.5. *The separating language is recognised by a nondeterministic tree-walking automaton.*

Proof. The automaton first does a depth-first search, as explained in Example 6, to check if the tree contains exactly three black nodes. If not, it rejects.
Otherwise, it uses nondeterminism to go to some node v (the idea is that v will be the closest common ancestor of the first two black nodes):



Next, it behaves as if continuing a depth-first search after v has been visited for the third time, like this:

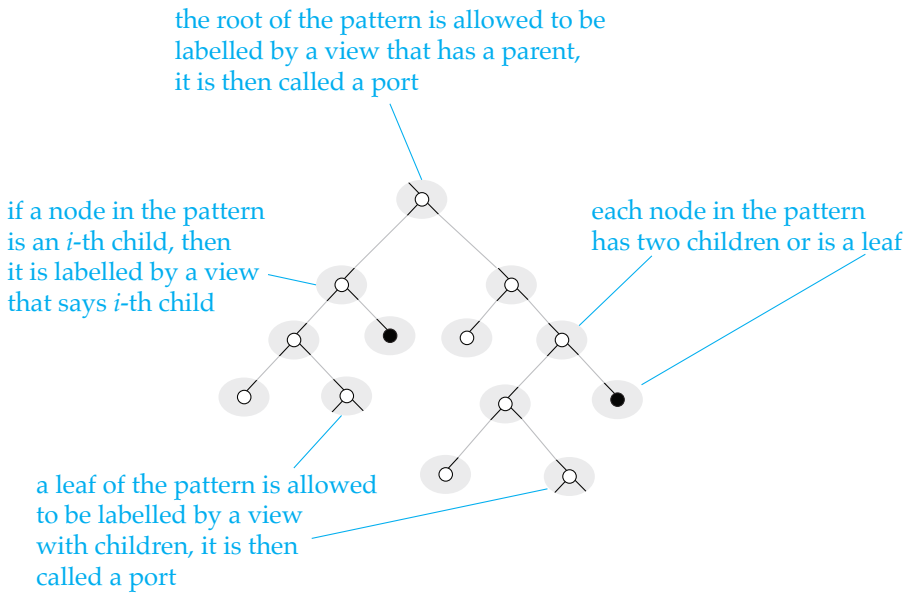


It accepts if during this depth-first search, it sees exactly one black node. If the input tree belongs to the language, then by choosing v to be the closest common ancestor of the first two black nodes, the automaton will accept. If the input tree is outside the language, then no matter how v is chosen, the automaton will see either zero or two black leaves in its depth-first search after visiting v. ■

5.2 *Deterministic automata do not recognise the separating language*

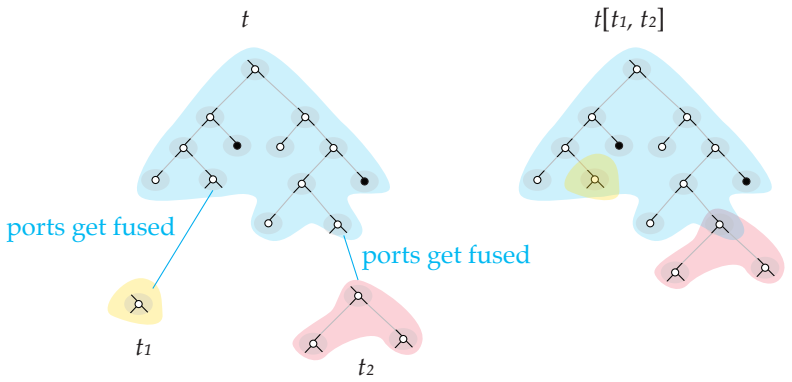
It remains to show that a deterministic tree-walking automaton cannot recognise the separating language. The idea is that for certain homogenous inputs, a deterministic tree-walking automaton can only do a depth-first search, and this is not enough to recognise the separating language.

Patterns. Define a *pattern* to be a tree where every node is labelled by a view in a consistent way as defined by the following picture:



A pattern without any ports (i.e. the root is not a port, and none of the leaves are 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 has the same view as the i-th leaf port in t. Then $t[t_1, \ldots, t_n]$ is defined by fusing the leaf ports of t with the corresponding root ports of $t_1, \ldots, t_n$ as in the following picture:



Equivalence. Fix for the rest of this section a deterministic tree-walking automaton. We will prove that this automaton does not recognise the separating language. For this 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 t to be a sequence of configurations (state, node of the pattern) that is consistent with the transition function of the automaton, has the source configuration in a port, and which is cut off at the first visit to a port after the source configuration. A port-to-port run either has its target configuration in a port, or it might not visit any more ports and accept/reject/loop.

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) does the run accept/reject/loop without visiting ports after its source configuration; (c) if the run does visit a port after the source port, then what is the first one and in what state. In items (a) and (c), 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 view in the root, they have the same number of leaf ports with the same respective views, and they have the same types of port-to-port runs. It is not difficult to see that equivalence is a congruence, i.e. if

$\sim$ denotes equivalence then:

$$t \sim t', t_1 \sim t'_1, \ldots, t_n \sim t'_n \qquad \text{implies} \qquad t[t_1, \ldots, t_n] \sim t'[t'_1, \ldots, t'_n].$$

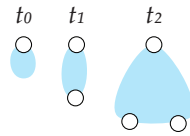
We now move to the first main ingredient in the proof. If T is a set of patterns, we write T^+ the least set of patterns that contains T and is closed under composition.

Lemma 5.6. *[Homogeneous Patterns Lemma] There exist patterns t_0, t_1, t_2 with the following properties:*

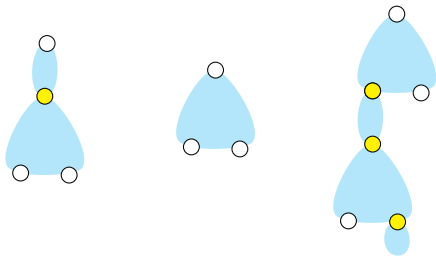
1. *For every $i \in \{0,1,2\}$, t_i is a pattern of arity i without black nodes where the root and all leaf ports have view*

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:



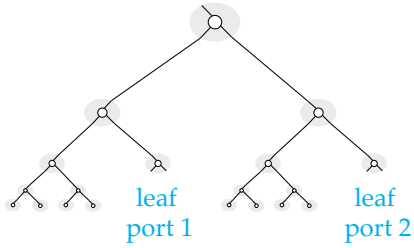
We use the name *homogeneous pattern* for patterns in $\{t_0, t_1, t_2\}^+$. In this terminology, item 2 of the lemma says that all homogeneous patterns of fixed arity $i \in \{0,1,2\}$ are equivalent. For example, all of the following homogeneous binary patterns are equivalent, because they have arity 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 a white binary label):



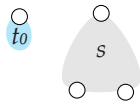
Proof of the Homogeneous Patterns Lemma. For $n \in \{1, 2, \dots\}$, define s_n to be the pattern which has only white nodes, root view

and where all leaves are at depth n (in particular, there are no ports). 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, or equivalently, s_N.

Choose distinct nodes v, w in the pattern s_N such that the subtrees of both v and w are the pattern s_n. Define s to be the binary pattern obtained from s_N by putting a leaf port in the nodes v and w. Here is the picture for $n = 3$ and $N = 5$:



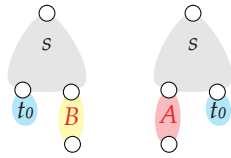
For the rest of the proof, we will draw the patterns t_0 and s like this:



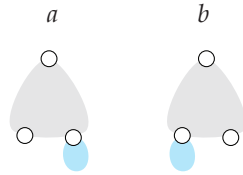
A straightforward induction shows that all rank 0 patterns in $\{s, t_0\}^+$ are equivalent. Also, every pattern from $\{s, t_0\}^+$ has view

in the root and all leaf ports.
In the following claim, we use multiplicative notation for composition of unary patterns.

Claim 5.7. *There exist rank 1 patterns $A, B \in \{s, t_0\}^+$ such that xy is equivalent to x whenever each of x, y is one of the following patterns:*



Proof. Define a and b to be the following unary patterns:



The set $\{a, b\}^+$ modulo pattern equivalence forms a finite semigroup, call it S. The statement of the claim follows from the following observation, which is true for any finite semigroup, and not just those that arise from tree-walking automata.

(*) Let S be a finite semigroup with elements a, b. There exist $A, B \in S$ such that

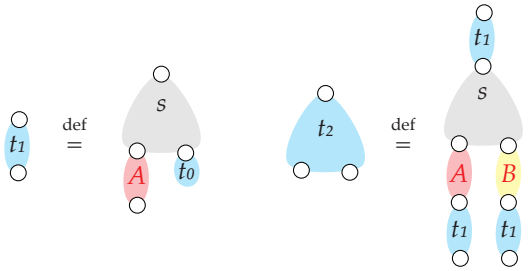
$$aAaA = aA = aAbB \qquad bBbB = bB = bBbA$$

To prove the claim, it remains to show (*) above. A short proof is using Green's relations: take some C which belongs to the least $\mathcal{J}$-class of the semigroup. Then define

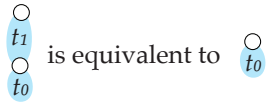
$$A = A(aC)^{\omega-1} \qquad B = B(bC)^{\omega-1}$$

where ω is the idempotent power of the semigroup, i.e. a number $\omega \in \{1, 2, \ldots\}$ such that s^ω is an idempotent for every element s of the semigroup. The equalities (*) then follow from a standard analysis of Green's relations. ∎

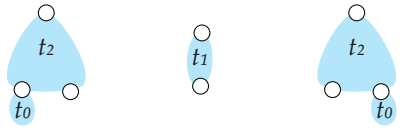
Let A, B be as in the above claim. Define t_1 and t_2 as follows:



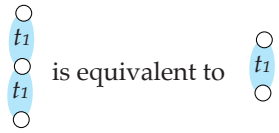
This completes the definition of the patterns t_0, t_1, t_2. Item 1 in the conclusion of the claim follows from the definition of the patterns. It remains to show item 2, i.e. that all patterns of same arity ≤ 2 are equivalent. Because t_1 is from $\{s, t_0\}^+$, we get that



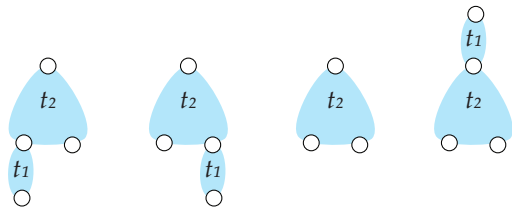
By Claim 5.7, the following three patterns are equivalent:



Also by Claim 5.7, t_1 is idempotent in the following sense



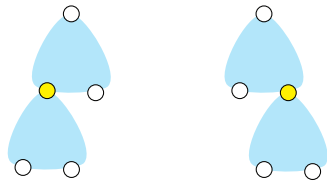
Since t_2 has t_1 attached to each of its ports, from idempotence of t_1 it follows that all of the following patterns are equivalent:



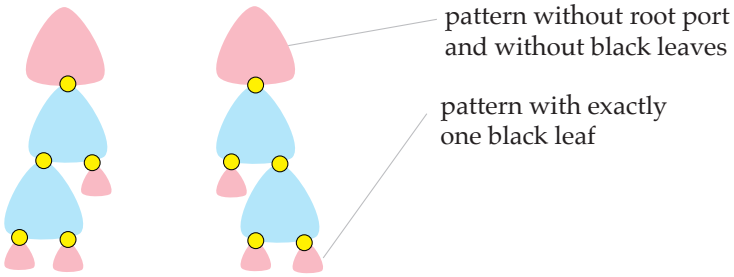
Using the above rules, and induction on the size of a pattern, one shows that every homogeneous patterns of arity $i \in \{0, 1, 2\}$ is equivalent to t_i. ∎

The Homogeneous Patterns Lemma would also work for nondeterministic tree-walking automata, and even branching ones, under a suitable notion of equivalence. The following lemma crucially depends on determinism (actually, a stronger result is true, see Exercise 31).

Lemma 5.8 (Rotation Lemma)**.** *The following two patterns are equivalent:*



The lemma immediately implies the lower bound from Theorem 5.4, i.e. that a deterministic tree-walking automaton cannot recognise the separating language, thus finishing the proof of Theorem 5.4. 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.

5.3 *Proof of the rotation lemma*

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

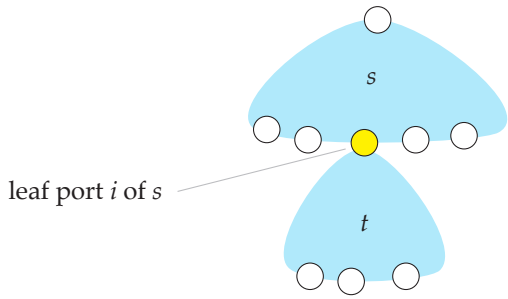
Closure of a state. For a state q, consider the run of the automaton which begins in state q in the yellow node below:

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 state of the last visit in the yellow node;

we might have $q = \overline{q}$ if the run does not visit the yellow node again, e.g. it goes directly to a port or rejects after starting in q in the yellow node.

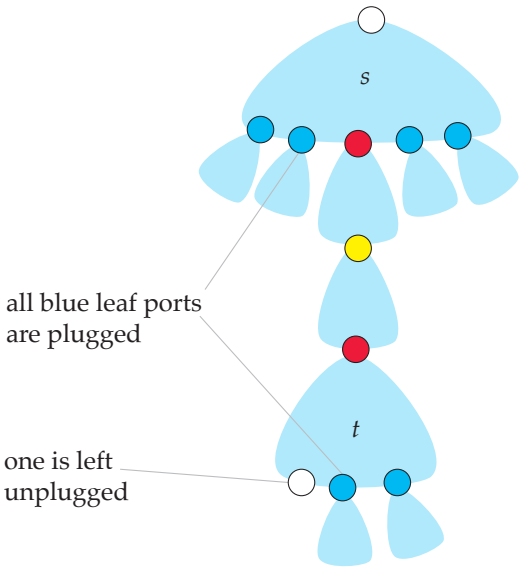
The definition of closure is based on the behaviour of the automaton on the interface between two copies of t_1. The following lemma shows that the same behaviour will be witnessed on the interface between any two homogeneous patterns of nonzero arity.

Lemma 5.9. *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:*



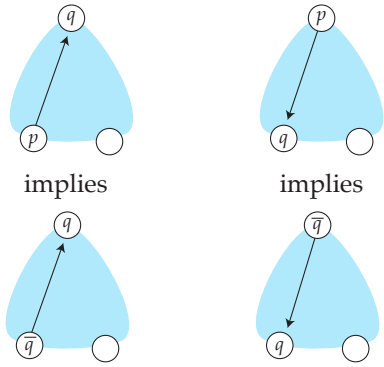
then the last visit in the yellow node (before any ports are reached) will be in state $\overline{q}$.

Proof. Consider the following pattern:

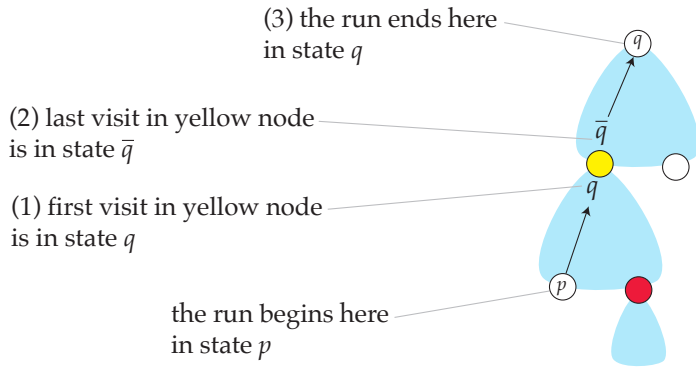


In the above pattern, consider the run of the automaton which begins in state q in the yellow node, and which is cut off whenever it reaches a port (a white node) for the first time. By definition of $\overline{q}$, because the parts above and bellow the yellow node are equivalent to t_1, the last visit to the yellow node is in state $\overline{q}$. For the same reason, the loop in the yellow node that goes from state q to state $\overline{q}$ never visits the red nodes; and once a red node is visited, then the yellow node is not visited again. After passing through a red node, the run continues – perhaps returning to the red node – until it reaches one of the ports (i.e. a white node) or one of the plugged ports (i.e. a blue node). ■

Lemma 5.10. *For every states p, q we have the following implications (and their symmetric versions with leaf port 2 used instead of leaf port 1):*



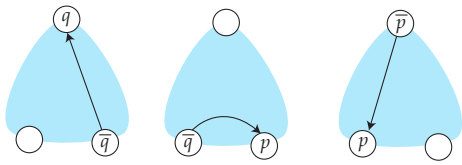
Proof. We only prove the first implication, the other ones are proved the same way. Consider the following port-to-port run:



Item (1) in the picture is by the assumption of the implication. Item (2) is by definition of $\overline{q}$ and Lemma 5.9. Item (3) is again by assumption of the implication, and because the above pattern is equivalent to t_2. 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 5.11. *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:*

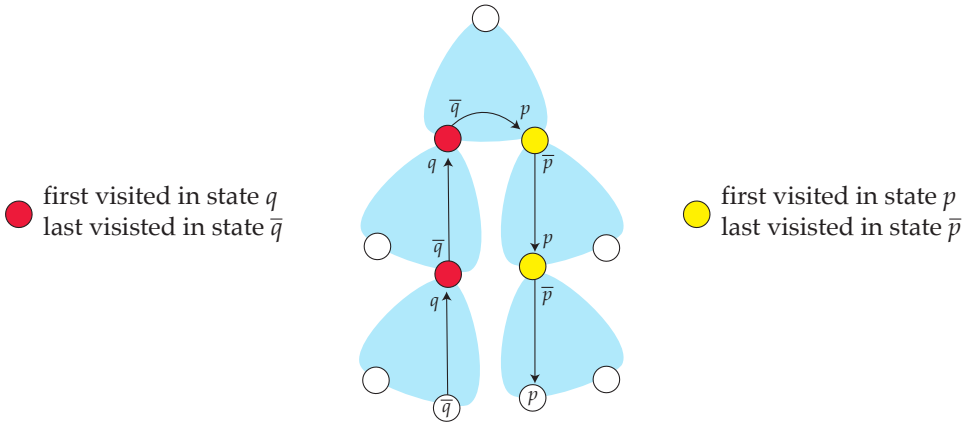


The following straightforward lemma shows that a left-to-right search will visit leaf ports in left-to-right order.

Lemma 5.12. *Assume that a state q is a left-to-right search. Then:*

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

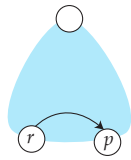
Proof. Here is the run that witnesses (*).



In the picture above, we use Lemma 5.9 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.

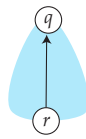
Lemma 5.13. *Suppose that a state r satisfies:*



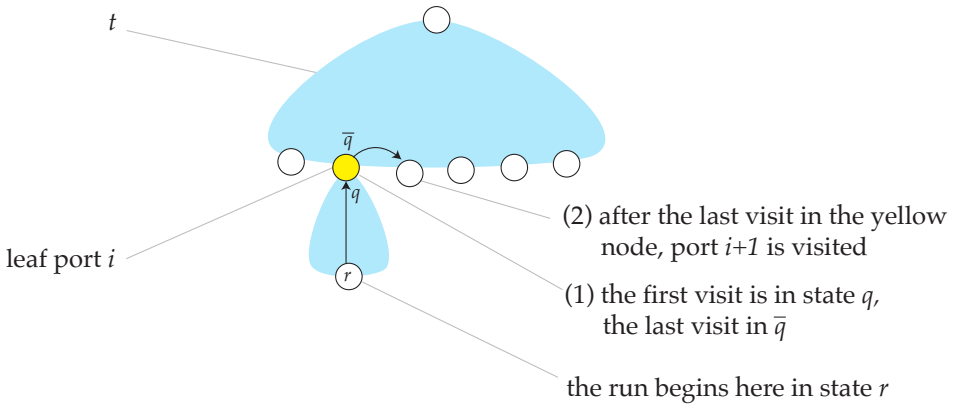
for some state p. Then the conclusion (*) of Lemma 5.12 is true with r used instead of q.

Proof. The key to the proof is the following claim.

Claim 5.14. *There is a state q which is a left-to-right search and satisfies*

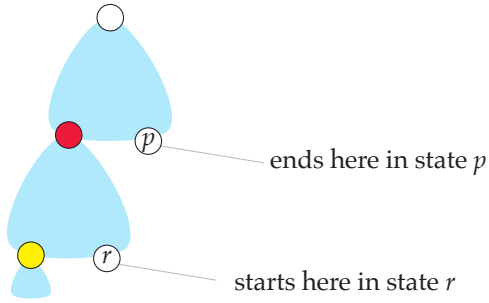


Before proving the claim, we used it to prove the lemma. Let t be an n-ary homogeneous pattern and let $i < n$ be one of the root ports. The following picture shows the run that witnesses (*) in the conclusion of the lemma.



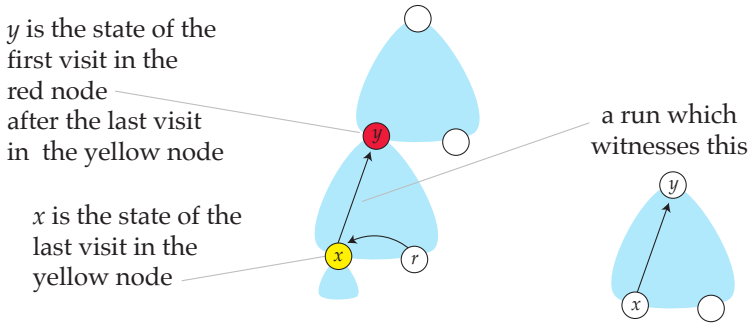
Item (1) is by the claim and Lemma 5.9, and item (2) is from Lemma 5.12.

Proof of the claim. Let p be as in the assumption of the lemma. Consider the following port-to-port run ρ:



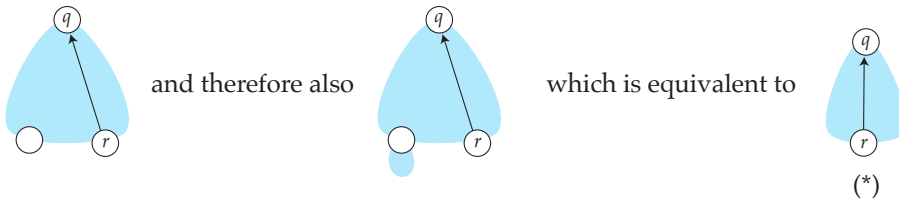
By definition of state p, such a run must exist, because the pattern above is homogeneous of arity 2, and the assumption of the lemma says that if the run begins in leaf port 1 in state r, then it will end in leaf port 2. In particular, the red node must be visited by ρ, since it is on the way from leaf port 1 to leaf port 2. Consider two cases, depending on which of the nodes – red or yellow – is first visited by ρ.

- *The yellow node is visited first.* We will show that this case cannot happen. Consider the states x, y defined in the following picture:

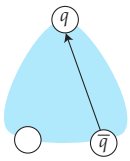


  The state of the last visit in the red node is $\overline{y}$. By Lemma 5.10, after visiting the red node in state y, the automaton returns in state $\overline{y}$, and then it proceeds to the root of the pattern, contradicting the assumption that ρ ends in leaf port 1.

- *The red node is visited first*. Define q to be the state of the first visit in the red node. Because the red node is visited first, we have

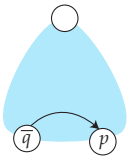


  Applying Lemma 5.10 to the leftmost picture above, we get



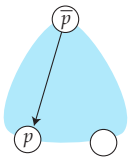
(**)

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

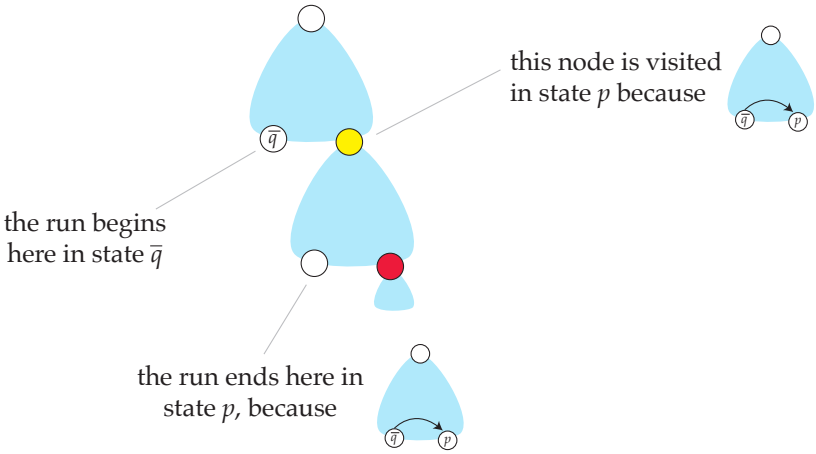


(***)

In the second case above, which is the only case, we proved (*), (**) and (***), which almost finishes the proof of the statement of the claim. The only missing ingredient is:



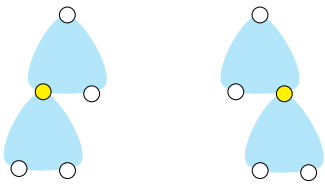
Let ρ be the port-to-port run described in the following picture:



The yellow node is visited the first time in state p, it is visited the last time in state $\overline{p}$. Therefore, the run after the last visit in the yellow node witnesses the missing ingredient. This concludes the proof of the claim, and therefore also of the lemma. ■

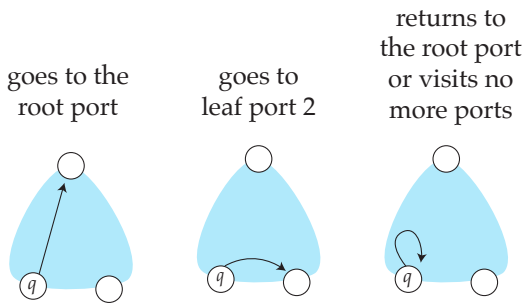
■

Proof of the Rotation Lemma. To prove that rotation preserves equivalence, let ρ, ρ' port-to-port runs in the two patterns

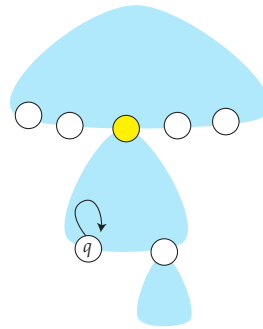


from the statement of the rotation lemma, respectively, which have equivalent source configurations. To prove equivalence, we need to show that target configurations are equivalent. Let q be the state in the source configuration of the two runs. We consider two cases depending on the source port.

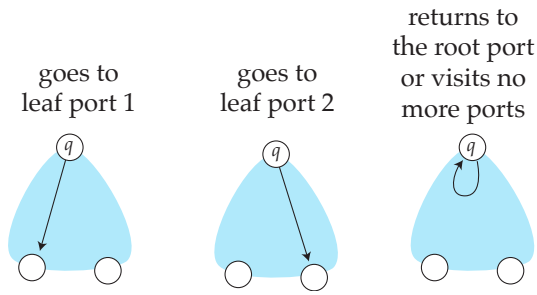
1. *The runs begin in a leaf port.* The cases of leaf port 1 and 3 are symmetric, see we ignore the case of leaf port 3. Suppose that the automaton starts in leaf port 1 or 2. Let us see what happens in the pattern t_2 if we start in leaf port 1 in state q. There are three cases to consider:



In the first case, we use Lemma 5.10 to show that both ρ and ρ' end in the root port (and in the same state). In the second case, we use Lemma 5.12 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 acceptance behaviour). 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 5.10 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 27. Show that deterministic top-down branching tree automata do not recognise the language "at least one leaf has label a", assuming that the alphabet has a binary letter and at least two leaf letters.

Problem 28. Following [20]. Consider a model of tree-walking automata where the automaton sees only the label and not the view (i.e. it does not know if it is in the root, or what is the child number). Show that this model, even in the nondeterministic variant, cannot recognise the language "some leaf has label a".

Problem 29. (Answer unknown) Prove or disprove: for every nondeterministic tree-walking automaton, there is a deterministic bottom-up branching tree automaton that recognises the same language.

Problem 30. Show that for every deterministic top-down tree automaton, there is a deterministic tree-walking automaton that recognises the same language.

Problem 31. Show the following generalisation of the Rotation Lemma: every two homogeneous patterns of same arity are equivalent.

6

Monadic second-order logic

In this section we discuss the connection between automata and monadic second-order logic (MSO). The presentation here is largely based on [31]. The connection was originally discovered simultaneously by three authors [8, 16, 32], in their quest to answer a question by Tarski: "is the MSO theory of the natural numbers with successor decidable"? The combination of logic and automata is an important theme of logic in computer science, Vardi calls this combination "a match made in heaven" [33]. A crowning achievement is Rabin's Theorem [25], 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.

[1] Büchi says this in [9, 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."

6.1 *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 triples, etc.

Example 7. Suppose that we view an undirected 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 symmetric. The following formula

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

says that the graph is a 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 connected:

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

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 graphs, only one binary relation).
Here is another example for graphs. The following MSO formula says that the input graph is three colourable:

$$\exists X_1 \exists X_2 \exists X_3 \quad \underbrace{\forall x \bigvee_i x \in X_i}_{\text{every vertex is coloured}} \quad \wedge \quad \underbrace{\forall x \forall y \; E(x, y) \Rightarrow \bigvee_{i \neq j} x \in X_i \wedge x \in X_j}_{\text{every edge has endpoints with different colours}}$$

□

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 graphs.

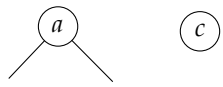
6.2 *Finite trees*

Recall finite trees over a ranked alphabet as defined in Section 5. A finite tree t over an alphabet Σ is viewed as a relational structure in the following way. The universe is the nodes. For every $i \in \{0, 1, \ldots\}$ there is a binary predicate $child_i(x, y)$ which says that y is the i-th child of x, and for every $a \in \Sigma$ there is a predicate $a(x)$ which says that x has label a. Using MSO, one can define a descendant predicate: a node y is a descendant of x if and only if y belongs to every set X that contains x and is closed under children. 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.
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 8. Suppose that the ranked alphabet is



The set of trees with an even number of nodes is MSO definable, namely the formula is "false". This is because all trees over the above ranked alphabet have

an odd number of nodes. More effort is required for "even number of leaves". Here the formula says that there exists a set X of nodes, which contains all leaves but not the root, and such that for every non-leaf node, it belongs to X if and only if it has an even number of children in X. □

Theorem 6.1. *Let Σ be a finite ranked alphabet. The following conditions are equivalent for every set L of finite trees over Σ:*

1. *L is definable in* MSO*;*
2. *L is recognised by a nondeterministic (equivalently, bottom-up deterministic) tree automaton.*

Proof.

1 ⇐ 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 $\mathcal{A}$:

there exists a labelling of nodes with states

$$\exists X_1 \cdots \exists X_n \quad \wedge \begin{cases} \text{every node has exactly one state} \\ \quad \forall x \bigvee_{q \in \{1,\ldots,n\}} x \in X_q \wedge \bigwedge_{p \neq q} x \notin X_p \\ \text{the root has a root state} \\ \quad \forall x \ \mathrm{root}(x) \Rightarrow \bigvee_{i \in R} x \in X_i \\ \text{for every node, a transition of the automaton is used} \\ \quad \bigwedge_{a \in \Sigma} \forall x \ a(x) \Rightarrow \bigvee_{(q_1,\ldots,q_k,q) \in \delta_a} \left(x \in X_q \wedge \bigwedge_{i \in \{1,\ldots,k\}} \mathrm{child}_i(x) \in X_{q_i} \right) \end{cases}$$

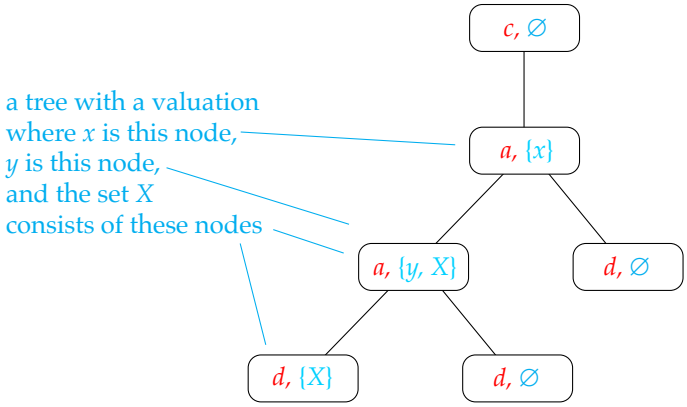
Formally speaking, $\mathrm{root}(x)$ is a shortcut for a formula which says that x is not a child of any node, and $\mathrm{child}_i(x) \in X_{q_i}$ is a shortcut for a formula which says that there exists a node that is the i-th child of x (because we have children as relations and not functions) and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

$1 \Rightarrow 2$ By induction on formula size, we show that every MSO formula can be converted into an automaton. The main issue is that 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). We define the *language* of such a formula to be the set of trees over an extended alphabet $\Sigma \times \mathcal{P}(\mathcal{X})$ defined as follows. (In the extended alphabet, the ranks are inherited from Σ.) A tree t over this alphabet belongs to the language if

1. For every first-order variable $x \in \mathcal{X}$, there is exactly one node in the tree whose label contains x on the second coordinate;

2. The formula φ is satisfied in the tree under the valuation which maps a first-order variable x to the unique node with x in its label, and which maps a second-order variable X to the set of nodes which have X in their label.

Here is a picture:



By induction on the size of an MSO formula, we show that its language, as defined above, is recognised by a tree automaton. The proof uses the closure properties from Lemma 5.2.

■

6.3 *Infinite trees*

We now move to infinite trees and Rabin's Theorem. For simplicity of notation, we consider only labelings of the complete binary tree by a finite alphabet. Let Σ be a finite alphabet (without any ranks, although intuitively one can think of the letters as having rank 2). An *infinite tree over* Σ is defined to be a function

$$t : \underbrace{\{0,1\}^*}_{\text{nodes of the complete binary tree}} \to \Sigma.$$

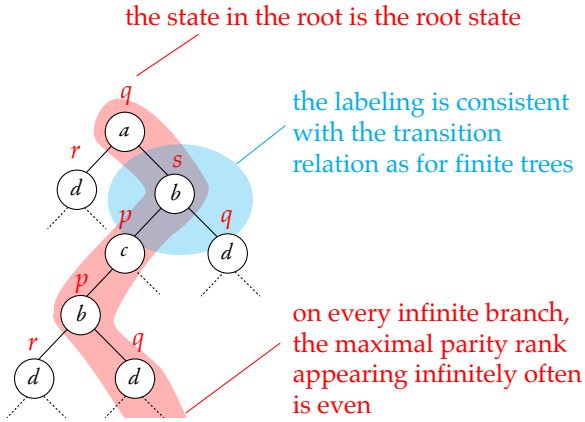
To recognise properties of infinite trees, we use parity automata.

Definition 6.2. *The syntax of a nondeterministic parity tree automaton consists of*

- *a finite input alphabet* Σ*;*

- *a finite set of states Q with a distinguished root state;*
- *a parity ranking function $Q \to \mathbb{N}$;*
- *for every letter a, a set of transitions $\delta_a \subseteq Q^2 \times Q$.*

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



We now state Rabin's Theorem. Rabin's original proof did not use the parity acceptance condition, but something that is now called the *Rabin condition*, see [31].

Theorem 6.3 (Rabin's Theorem)**.** *Let Σ be a finite alphabet. The following conditions are equivalent for every set L of infinite trees over Σ:*

1. *L is definable in* MSO*;*
2. *L is 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 is far from obvious. The difficulty is that we use only nondeterministic automata; in fact no deterministic model for infinite trees is known that would be equivalent to

MSO. Therefore, the rest of this chapter is devoted to proving closure under complementation for languages recognised by nondeterministic parity tree automata.
A corollary of the above statement of Rabin's theorem is that the structure

$$(\{0,1\}^*,\ \underbrace{v \mapsto v0}_{\text{left successor}}\ ,\ \underbrace{v \mapsto v1}_{\text{right successor}}\ ),$$

i.e. the relational structure describing the unlabelled (equivalently, labelled by a one letter alphabet) complete binary tree, has decidable MSO theory. This corollary is the original statement of Rabin's Theorem.

Alternating parity tree automata. To show complementation of nondeterministic tree automata, we will pass through a more powerful model. The syntax of an *alternating parity tree automaton* is defined the same as in Definition 6.2 for nondeterministic automata, with the following differences: (1) the set of states is partitioned into two subsets Q_0 and Q_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 belongs to Q_i with $i \in \{0,1\}$. 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 6.4.

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 Theorem 6.3. Recall that the only missing ingredient in Theorem 6.3 was complementation of nondeterministic parity tree automata. This is achieved by using Theorem 6.4 as follows: make the automaton alternating, complement it, make it nondeterministic again.

Proof of Theorem 6.4. For item 1, let $\mathcal{A}$ be a nondeterministic parity tree automaton with states Q and transitions Δ. The simulating alternating automaton has states $Q + \Delta$. In a state from Q, player 0 chooses a transition applicable to the current configuration (and the head of the simulating automaton stays in the same node of the input tree). In a state from Δ, player 1 chooses either the left or right child, and updates the state accordingly. The parity condition for Q is inherited from the original nondeterministic automaton, and all states from Δ 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 by swapping the roles of players 0 and 1, and incrementing the ranking function so that even ranks become odd and vice versa. 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 $G_{\overline{\mathcal{A}}}(t)$.

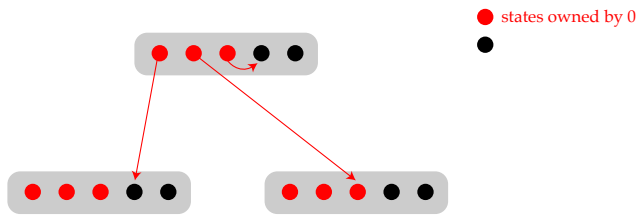
4. player 1 does not have a winning strategy in the game $G_{\overline{\mathcal{A}}}(t)$.

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.
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. A memoryless strategy σ_i for player 0 can be represented as a tree $[\sigma_i]$ over alphabet

$$\Gamma_i \quad \stackrel{\text{def}}{=} \quad Q_i \to (Q \times \{\epsilon, 0, 1\})$$

where the label of node v is the function which maps state q to the pair (p, x) such that strategy σ_i goes from (q, v) to $(p, v \cdot x)$. Here is a picture of a letter from Γ_0:



We will show that the language

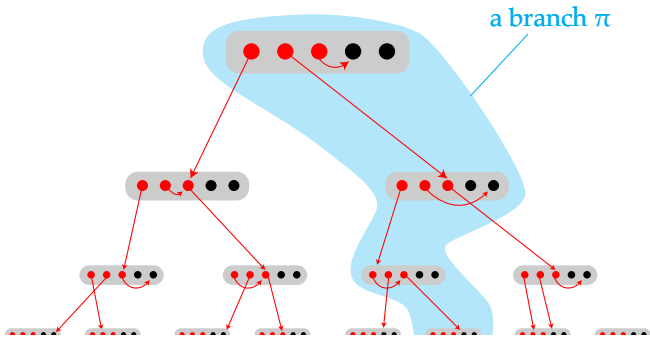
$$\{(t, [\sigma_0]) : \sigma_0 \text{ is a memoryless strategy for player 0 in } \mathcal{G}_{\mathcal{A}}(t)\} \tag{6.1}$$

is recognised by a (even deterministic top-down) parity automaton on trees. (In the above language, we treat a pair of trees as a single tree over a product alphabet.) This will complete the proof of the Dealternation Theorem, because a nondeterministic parity automaton can guess the labelling $[\sigma_0]$. The key observation is the following claim.

Claim 6.5. *There is a* nondeterministic *parity automaton $\mathcal{B}$ over ω-words, such that the following conditions are equivalent for every tree t, branch $\pi \in 2^\omega$ 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 over alphabet $\Sigma \times \Gamma_0 \times \{0,1\}$ that is obtained from (t, σ_0) by reading the branch $\pi \in 2^\omega$ (the i-th letter of the ω-word consists 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 chose 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 $(t, [\sigma_0])$ 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 32. Show that the set $\mathbb{N}^*$ equipped with the prefix relation has decidable MSO theory.

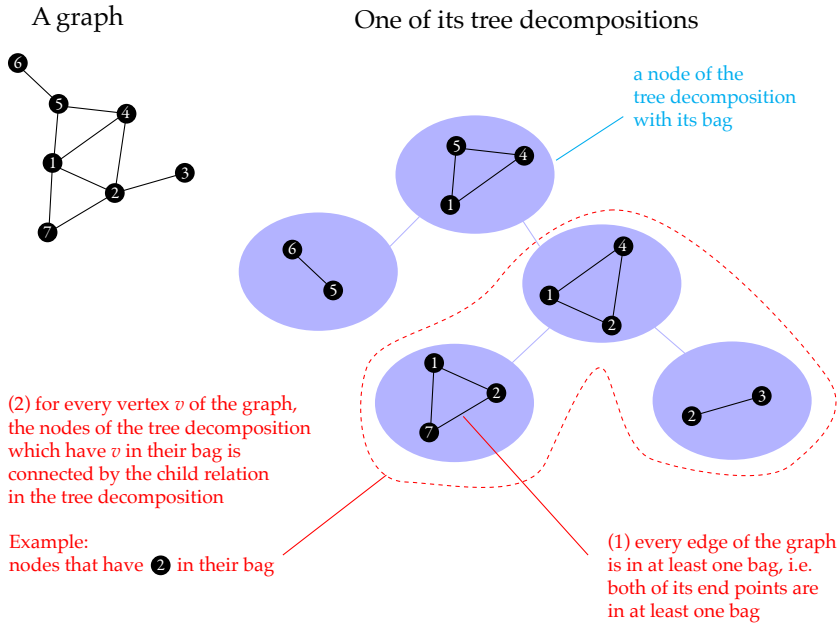
7
Treewidth

In this chapter, we present Courcelle's Theorem, which says that every formula of MSO can be evaluated in linear time on graphs that have bounded treewidth. (For MSO formulas defining properties of graphs, see Example 7.) Tree width 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, although there exists other inequivalent ways of quantifying similarity to a tree, e.g. clique width, see [14, Section 2.5].

7.1 Treewidth and how to compute it

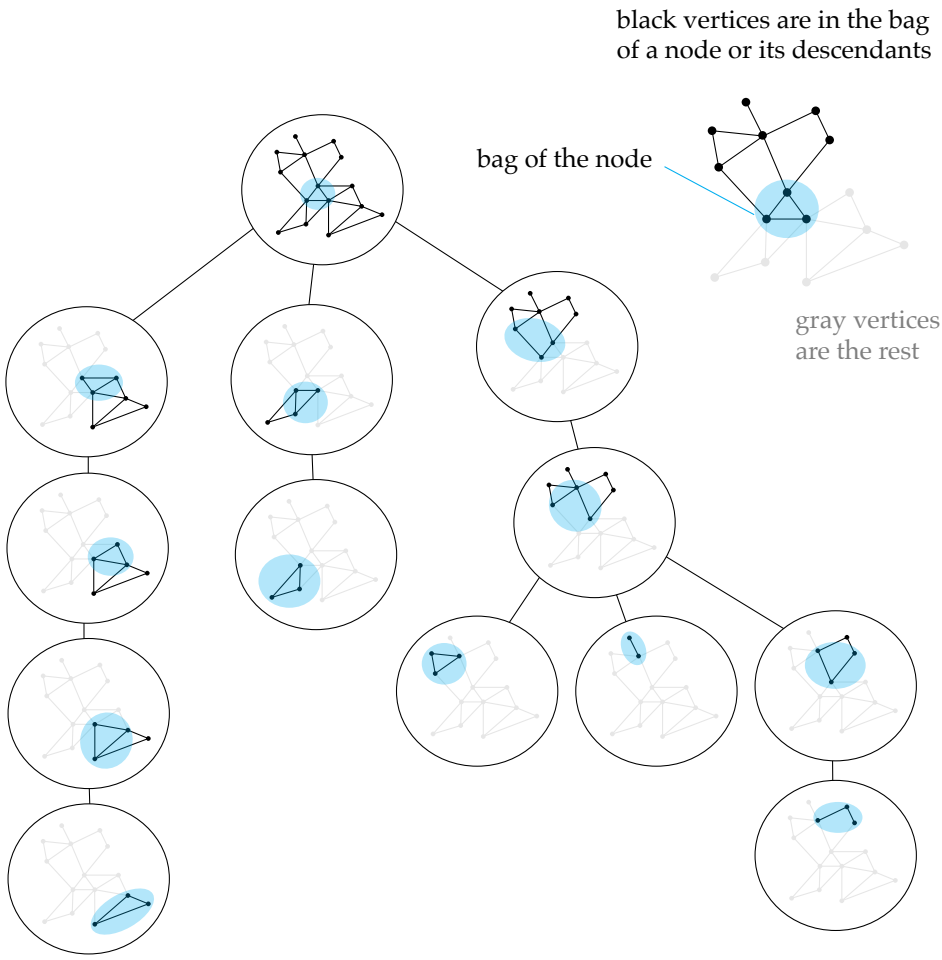
In this section, we define tree decompositions and treewidth.

Treewidth. Consider an undirected graph G. Define a *tree decomposition* of G to be a tree t together with a labelling which maps every node x of the tree to a set of vertices in the graph, called the *bag of* v, such that the conditions (1) and (2) depicted in the following picture are satisfied:



Define the *width* of a tree decomposition to be the maximal size of a bag minus one. In the example above, the width is 2, because the maximal bag size is 3. (If you don't like doing minus one, an alternative view is that the width is the maximal intersection of bags connected by the child relation, i.e. tree width is the maximal size of an *adhesion* in the tree decomposition.) 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:



Fact 7.1. *If a graph has treewidth k, then the number of edges is at most $k \cdot (k+1)$ 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 is at most the number of vertices. Each edge

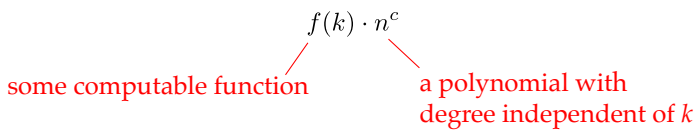
must be present in some node, and each node can have at most $k \cdot (k+1)$ edges. The bound is achieved by a clique over $k+1$ vertices. ∎

Computing a tree decomposition. We present an algorithm that computes tree decompositions of approximately optimal width (at most 4 times worse) and which runs in quadratic time when the treewidth is fixed. The algorithm is from Robertson and Seymour, see also [15, Theorem 7.18].

Theorem 7.2. *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 theorem is not optimal: one can improve quadratic time to linear time, and one can compute tree decompositions of optimal width (i.e. have $< k$ instead of $< 4k$ in the output of the algorithm), see [4]. The function $f(k)$ is exponential, and there is little hope for improvement, because the following problem is NP-complete [2]: 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 following form:

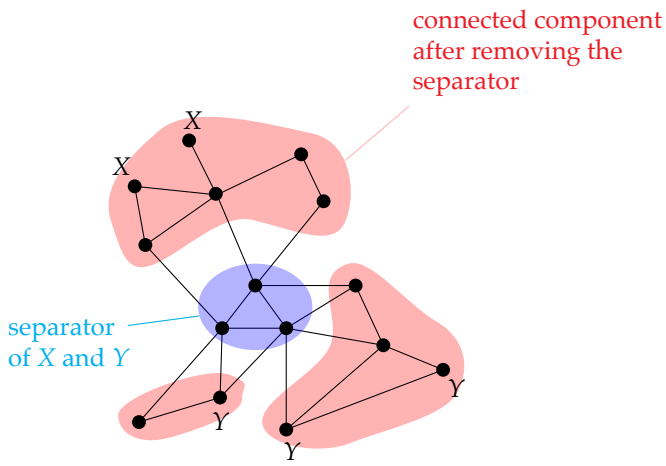


The algorithm uses the following lemma on computing separators.

Lemma 7.3. *Given a graph G and disjoint sets of vertices X, Y, one can compute a separator of minimal size in time*

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

Recall that a separator 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:

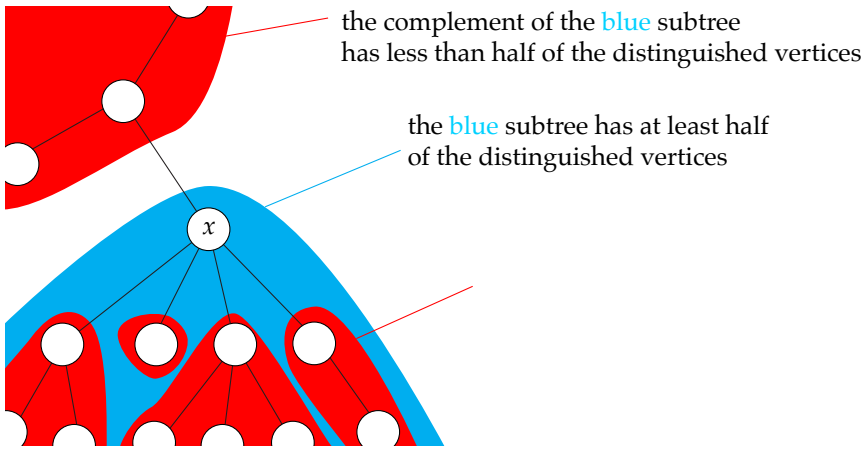


We do not prove the above lemma, it can be shown using the Ford-Fulkerson algorithm for computing maximum flow, see the discussion in [15, p. 198]. When the treewidth is fixed, the number of edges is linear in the number of vertices, and the size of the separator is bounded by a function of k (see the proof of Lemma 7.4), and therefore the running time of the algorithm is linear. The main step in proving Theorem 7.2 is the following lemma.

Lemma 7.4. *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 must succeed on graphs of treewidth $< k$. We enumerate all possible partitions of the distinguished vertices into three parts X, Y, Z, such that Z has size at most k, and each of X, Y has size at most $2k$. The idea is that Z is the distinguished vertices in the separator S. 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 of X and Y in the graph $G - Z$, and report success if the combined size of S and Z is at most k. 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 7.1.
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 at most 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:



Define S to be the bag of x. The size of S is $< k$. By choice of x we know that every connected component of $G - S$ has at most half the distinguished vertices. For each connected component of $G - S$, we count the number of distinguished nodes in that compoment; this is a number that is at most half of $3k$.

Claim 7.5. *Let $n_1 \geq n_2 \geq \cdots \geq n_p$ be numbers in $\{1, \ldots, 2k\}$ with sum $\leq 3k$. Then*

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

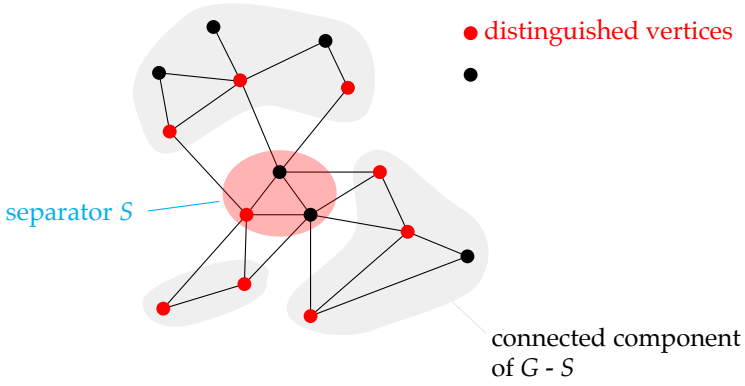
Proof. Take the first i such that the sum of the first i elements is $\geq k$. ∎

Apply the above claim to the numbers of distinguished vertices in the connected components of the graph $G - S$, yielding the lemma. ∎

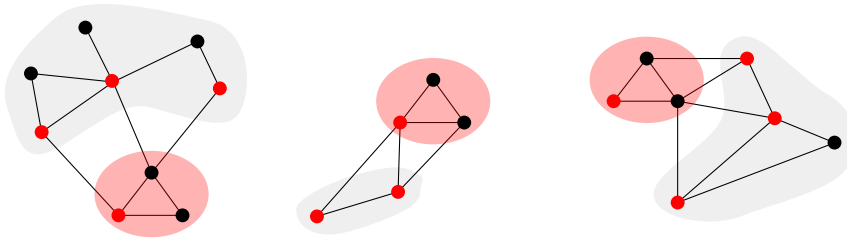
Proof of Theorem 7.2. We use a more detailed of statement of the algorithm, as described below.

- **Input** k and a graph with at $\leq 3k$ 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.

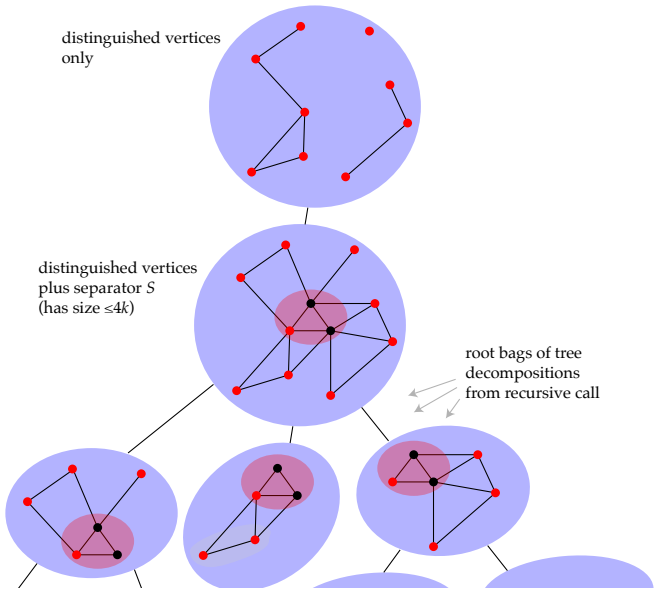
Suppose that G is the graph. If there are less than $3k$ distinguished vertices, we add some arbitrary vertices to make the set have size exactly $3k$. Apply Lemma 7.4, computing S, X and Y. If the input graph has treewidth $< k$ then the algorithm from the lemma must succeed. Find all connected components of the graph $G - S$. We know that 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 that were in G_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:



It is not difficult to check that this is a tree decomposition of G. The algorithm does a linear computation, followed by recursive calls to smaller instances; and therefore its running time is quadratic. ∎

7.2 *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 [13, Theorem 4.4].

Lemma 7.6. *For every $k \in \mathbb{N}$ and every formula of MSO φ on graphs, there is a linear time algorithm which does the following:*

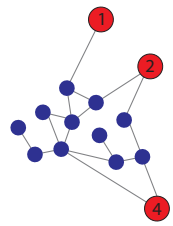
- **Input.** *A graph together with a tree decomposition of width $\leq k$;*
- **Question.** *Does the graph satisfy φ?*

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 [4]):

Theorem 7.7 (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 7.6. 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.

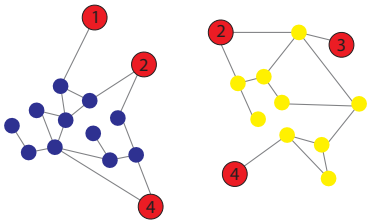
The algebra of tree decompositions. Define a *sourced graph* to be a graph with some but not necessarily all vertices being assigned natural numbers. The vertices with numbers are called the *sources* and the numbers are called the *source names*. We assume that the each source name is used for at most 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. 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:



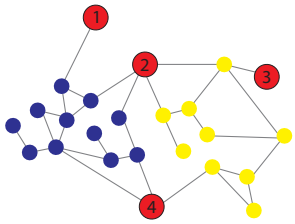
The purpose of sourced graphs is to fuse them. The operation inputs two sourced graphs, and it outputs their disjoint union with each pair of sources

that have the same name being merged together into a single vertex, as in the following picture

two sourced graphs

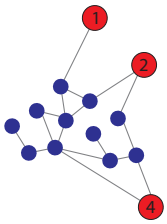


their fusion

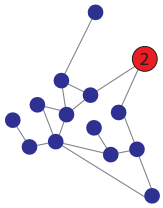


A second operation is forgetting a subset of source names, illustrated below:

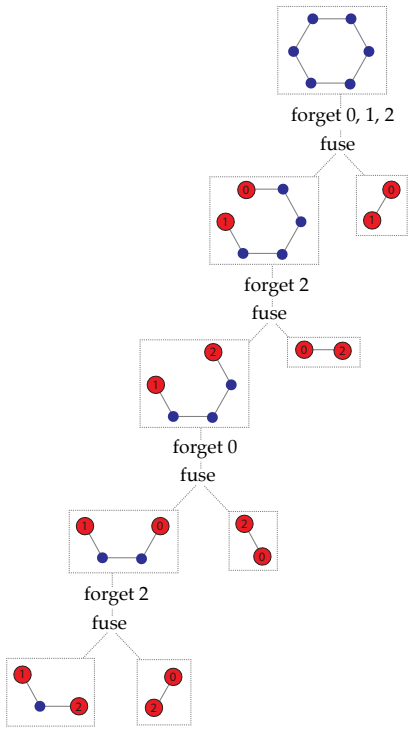
a sourced graph



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, 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:



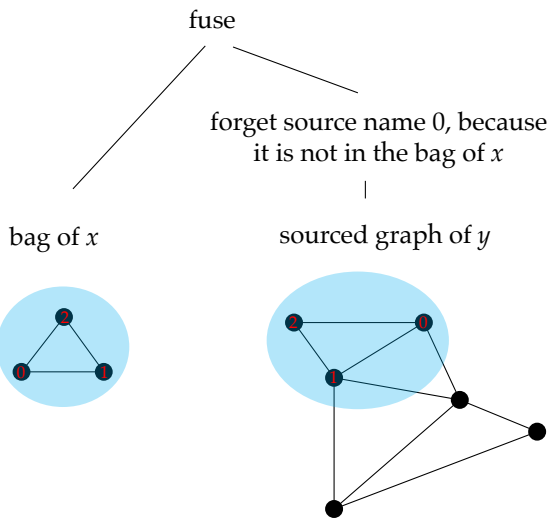
Fact 7.8. *A graph has treewidth k if and only if (when viewed as a sourced graph without any sources) it can be generated by a term in the algebra of width k sourced graphs, starting with constants that have at most $k+1$ vertices.*

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 7.8 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".

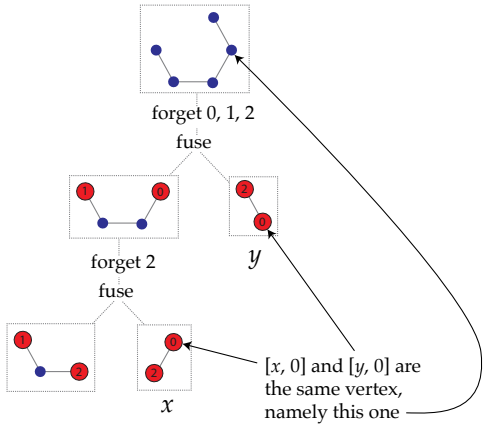
From a width k tree decomposition one compute a corresponding tree over the above alphabet in linear time. By Theorem 6.1, for finite trees over alphabet Σ_k,

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 7.6 will follow once we prove the following lemma.

Lemma 7.9. *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*

$$t \text{ satisfies } \hat{\varphi} \quad \text{iff} \quad \text{the graph of } t \text{ satisfies } \varphi \quad \text{for every } t \text{ over } \Sigma_k.$$

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 tree 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 be 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 [14, Section 1.7]. ∎

Problem 33. 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 34. Using the grid theorem (if a class of graphs has unbounded treewidth, then it has square grids of arbitrarily large dimensions as minors), show that if a class $\mathcal{C}$ of graphs has unbounded treewidth, then the following problem is undecidable: given an MSO formula φ, decide if it is true in some graph from $\mathcal{C}$.

8
Weighted automata over a field

In the following we write $\mathbb{Q}$ to denote any field, but the example we have in mind is the rational numbers. Fix a field $\mathbb{Q}$ for the rest of the lecture.

Definition 8.1 (Weighted automaton over $\mathbb{Q}$)**.** *A weighted automaton consists of the following ingredients:*

1. *an input alphabet, which is a set Σ;*
2. *a set Q of states, which is a vector space over $\mathbb{Q}$;*
3. *an initial state $q_0 \in Q$;*
4. *for each letter $a \in \Sigma$, a linear transition function $\delta_a : Q \to Q$;*
5. *a linear output function $F : Q \to \mathbb{Q}$.*

An automaton is called *finite* if the input alphabet has finitely many elements and the vector space has finite dimension[1]. The semantics of a weighted automaton is a function $\Sigma^* \to \mathbb{Q}$ defined as follows. When given an input word, the automaton begins in the initial state q_0. Then for every new input

[1]One could consider a more general definition, where the input alphabet is also a vector space, and the transition function is a bi-linear map $Q \times \Sigma \to Q$. The case of finite input alphabets would be recovered by viewing an input letter as one of the base vectors in the vector space $\mathbb{Q}^\Sigma$ of finite dimension.

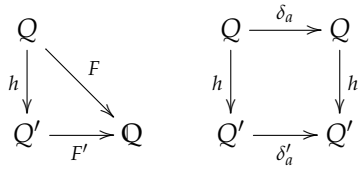
letter a, it applies the transition function δ_a to the current state, yielding a new state. Finally, it applies the output function F to the state at the end, yielding an element of the underlying field.

Example 9. A normal deterministic finite automaton with n states can be viewed as a special case of a weighted automaton. The set of stats will be $\mathbb{Q}^n$, and the reachable ones will only be vectors which have zero on all coordinates except the coordinate corresponding to the current state. □

8.1 Minimisation of weighted automata

We begin by proving a Myhill-Nerode style theorem for weighted automata, which says that for every weighted automaton, there is a canonical one that recognises the same language, and is minimal in a certain sense. Our notion of minimality is based on homomorphisms of weighted automata, as defined below.

Homomorphisms of weighted automata. Suppose that we have two weighted automata $\mathcal{A}$ and $\mathcal{A}'$ over the same input alphabet Σ. A *homomorphism* from $\mathcal{A}$ to $\mathcal{A}'$ is defined to be a linear function h from the state space of $\mathcal{A}$, call it Q, to the state space of $\mathcal{A}'$, call it Q', which is consistent with the structure of the two automata, in the following sense. The initial state of $\mathcal{A}$ is mapped to the initial state of $\mathcal{A}'$, and the following diagrams commute for every $a \in \Sigma$



where F, F' are the output functions of the respective automata, and δ_a, δ'_a are the transition functions. If there is such a homomorphism, then the functions computed by the two automata are the same, as can be shown by induction on the length of the input word. An isomorphism is a homomorphism which has

an inverse that is also a homomorphism. It is not difficult to show that if a homomorphism is a bijection, as a function on state spaces, then it is an isomorphism.
We now state the minimisation theorem for weighted automata. Call an automaton *reachable* if every state in its state space is a finite linear combination of reachable states, i.e. states that can be reached by reading input words.

Theorem 8.2. *Let $f : \Sigma^* \to \mathbb{Q}$ be a function computed by a weighted automaton. There exists a weighted automaton, called the syntactic automaton of f, which computes f and such that every reachable weighted automaton computing f admits a homomorphism into the syntactic automaton.*

Proof. The proof is essentially the same as for the classical Myhill-Nerode theorem.
Define the *continuation* of a word $w \in \Sigma^*$ to be the function

$$[w] : \Sigma^* \to \mathbb{Q} \qquad v \mapsto f(wv).$$

Before defining the syntactic automaton, we define a bigger automaton, called the *continuation automaton*. States of the continuation automaton are vectors in $\Sigma^* \to \mathbb{Q}$, which is a vector space, albeit of infinite dimension Σ^*. The initial state is the continuation of the empty word. The output function maps a state to its value on the empty word. The transition function

$$\delta_a : (\Sigma^* \to \mathbb{Q}) \to (\Sigma^* \to \mathbb{Q})$$

is simply a permutation of coordinates:

$$\delta_a(q)(v) = q(av).$$

It is easy to see that this function is linear, and that it maps a continuation $[w]$ to the continuation $[wa]$. Note how the choice of the function f only plays a role in the definition of the initial state of the automaton.
Define the syntactic automaton to be the continuation automaton with the state space obtained by restricting $\Sigma^* \to \mathbb{Q}$ to finite linear combinations of continuations. We need to show that this automaton is well-defined, i.e. the

transition functions stay within the state space. This is because the transition functions are linear, and they map continuations to continuations.
We now show that every reachable weighted automaton recognising f admits a homomorphism into the syntactic automaton. Let $\mathcal{A}$ be be such a weighted automaton, and let Q be its states. We first show that there is a homomorphism into the continuation automaton, and then we show that the image of the homomorphism is actually the syntactic automaton. Define a function

$$h : Q \to (\Sigma^* \to \mathbb{Q})$$

which maps a state q to the function that maps w to the value of the automaton $\mathcal{A}$ after reading w, assuming that the initial state was changed to q. This is clearly a linear function, and it is not difficult to see that it is a homomorphism. It remains to show that the image of h is actually the syntactic automaton. Since any homomorphism maps reachable states to reachable states, it follows that the image of h is included in the states of the syntactic automaton (because of the assumption that $\mathcal{A}$ was reachable). Finally, the homomorphism h has the property that if w is an input word, then the state of $\mathcal{A}$ after reading w is mapped to the continuation $[w]$, and therefore h is surjective. ∎

8.2 *Algorithms for equivalence and minimisation*

Here we study weighted automata which are finite, in the sense that the input alphabet is finite and the state space is of finite dimension. We also assume that the field is the field of rational numbers.

Representing finite automata. The state space is a finite dimensional vector space, and therefore it must be isomorphic to $\mathbb{Q}^n$ for some $n \in \mathbb{N}$, since these are the vector spaces of finite dimension. Therefore, it suffices to store n. For each input letter a, the transition function is a linear function

$$\delta_a : \mathbb{Q}^n \to \mathbb{Q}^n$$

which can be stored as a matrix. (Here we assume that the entries of the matrix can be represented, which means either that we restrict to rational numbers, or use some computation model that can deal directly with elements of the field.)

Computing reachable states. Let us begin with a simple algorithm for finite weighted automata: computing linear combinations of reachable states. (Computing the actual reachable states, and not their linear combinations, is a different story, as we will see below.) This is a simple saturation procedure. We begin with $Q_0 \subseteq Q$ being the singleton of the initial state. Then, assuming that Q_i has already been defined, we define Q_{i+1} to be the vector space spanned by

$$Q_i \cup \bigcup_{a \in \Sigma} \delta_a(Q_i)$$

This way we get a growing chain of linear subsets

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

Since the dimension cannot grow indefinitely, this sequence must stabilise at some point, and this point is the set of reachable states. Furthermore, if the original automaton is given by a matrix representation, then one can compute the sets Q_i.
Here is a corollary of the reachability algorithm described above.

Theorem 8.3. *The following problem is in polynomial time:*

- **Input.** *Two weighted automata.*
- **Question.** *Do they compute the same function* $\Sigma^* \to \mathbb{Q}$*?*

Proof. We can define the product automaton, with states being pairs of stats from the two automata, and the output function being defined as

$$(q_1, q_2) \qquad \mapsto \qquad F_1(q_1) - F_2(q_2)$$

with F_1, F_2 being the output functions of the two original automata. In the product automaton we compute the linear combinations of reachable states.

The automata were equivalent if and only if, when restricted to those states, the new output function is zero everywhere. The latter can be tested by computing the linear combinations of reachable states in the product automaton, and test if they all belong to the subspace that gives output zero. ∎

Computing the minimal automaton. Here we show that minimisation is effective, i.e. if one gets a finite weighted automaton on input, one can produce (even in polynomial time) the minimal automaton. Observe that if a function is recognised by a finite dimensional weighted automaton, then its syntactic automaton has finite dimension. This is because there is always a surjective homomorphism onto the syntactic automaton, and homomorphisms, which are linear functions, cannot increase dimension.
Consider a weighted automaton $\mathcal{A}$ with a state space Q of finite dimension. For a number $n \in \mathbb{N}$, we define states q, p to be n-equivalent if

$$qw = pw$$

holds for all input words of length at most n, where qw if the output of the automaton after reading word w assuming that the initial state was changed to q. 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, i.e. it is closed under $+$ and multiplying by scalars. Therefore we have a sequence of subsets

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

which are linear. Since each subset has a dimension, which is at most double the dimension of Q, the sequence above must stabilise at some equivalence relation, call it E_*, which is the Myhill-Nerode equivalence relation. Furthermore, a representation of this E_* can be computed in time polynomial in the dimension of the original automaton. The remaining description is essentially book-keeping: we prove that there is a well-defined quotient

automaton, and that a matrix representation of it can be computed based on a matrix representation of the original automaton.

Let $[Q]$ be the set of equivalence classes of Q with respect to E_*, and let

$$h : Q \to [Q]$$

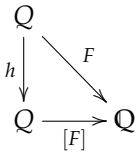
be the function which maps a state to its equivalence class. Because E_* is an equivalence relation and a linear set, the set $[Q]$ is a vector space, and h is a linear function. What is the dimension of $[Q]$ and how do we represent it and the function h, assuming that $Q = \mathbb{Q}^n$ for some n? One solution is the following. Begin with B being some basis of Q, e.g. the n vectors which have 1 on a unique coordinate. Then, iterate the following: check if there is some $b \in B$ which is equivalent under E_* to a linear combination of other elements of B. If there is no such b, then return B, otherwise remove one such b from B and continue the process. At the end we get a subset $B \subseteq Q$, such that every element of Q is equivalent under E_* to a linear combination of vectors from B. In particular, $[Q]$ is isomorphic to $\mathbb{Q}^B$. Out of this process we also get a matrix representation of the function h, seen as a linear function $\mathbb{Q}^n \to \mathbb{Q}^B$.

If we take two states that are equivalent under E_*, and apply to them a transition function δ, then the results are also equivalent. This means that for every input letter a there exists a function $[\delta]_a$ which makes the following diagram commute:

$$\begin{array}{ccc} Q & \xrightarrow{\delta_a} & Q \\ {\scriptstyle h}\downarrow & & \downarrow{\scriptstyle h} \\ [Q] & \xrightarrow[[\delta]_a]{} & [Q] \end{array}$$

The function $[\delta]_a$ is also linear, because its graph, as a subset of $[Q] \times [Q]$, is simply the image of the graph of δ_a under h applied coordinatewise. In particular, we can compute a matrix representing each function $[\delta]_a$. A similar argument proves that there is a linear function $[F]$ which makes the following

diagram commute



and a matrix representing it can be computed. This finishes the description of the algorithm for minimising finite weighted automata.

8.3 Undecidable emptiness

In Theorem 8.3, we showed that equivalence of weighted automata is decidable, in fact 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.

Theorem 8.4. *The following problem is undecidable:*

- **Input.** *a weighted automaton over the field of rational numbers* $\mathbb{Q}$*;*
- **Question.** *is some word mapped to* 0*?*

Section 8.3 is devoted to proving the above undecidability result theorem. There are two basic ingredients in the proof: hashing words as numbers, and composing weighted automata with sequential transducers. For the latter, it is convenient to use an equivalent version of weighted automata, which allows control states from a finite set. These ingredients are described below.

Hashing. The archetypical function that can be computed by a weighted automaton is mapping 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 8.5. *For every alphabet* Σ *there is a finite weighted automaton which computes an injective function to the strictly positive rational numbers*

$$f : \Sigma^* \to \mathbb{Q}_{>0}.$$

Proof. Without loss of generality, assume that $\Sigma = \{0, \ldots, n-1\}$. The idea is to treat an input $a_1 \cdots a_i$ as a fraction in base n:

$$\frac{a_1}{n} + \frac{a_2}{n^2} + \cdots + \frac{a_n}{n^i}$$

The only problem with this solution is that trailing zeros are ignored. Therefore, the automaton adds an imaginary 1 to the end of the input. To implement this procedure by an automaton, we do the following. The state space is $\mathbb{Q}^2$. The idea is that the first coordinate stores the value of the input seen so far, while the second stores $1/n^{i+1}$ where i is the length of the input read so far. The initial vector is $(0, 1/n)$. For $a \in \Sigma$, the state update function is the linear function

$$\delta_a(x, y) = (x + a \cdot y, \frac{y}{n})$$

The output function is

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

which corresponds to adding the imaginary 1 at the end of the output. ∎

Weighted automata with states. We now describe *weighted automata with states*, which are a more convenient version of weighted automata to work with. The syntax of such an automaton consists of an input alphabet Σ, a dimension n, a finite set Q of *control states*, as well as transition functions and output functions defined below. A configuration of the automaton is a pair in $Q \times \mathbb{Q}^n$, the automaton also comes with a designated initial configuration. To update the configuration, for each input letter a we have a state update function

$$\delta_a : Q \to Q$$

and to update the vector, for each input letter a and each state q we have a linear vector update function

$$\delta_{a,q} : \mathbb{Q}^n \to \mathbb{Q}^n.$$

We first update the vector, i.e. $\delta_{a,q}$ is applied with q being the state before letter a. The output of the automaton is computed as follows. We begin in some

designated initial configuration. Next, for each letter of the input, we update the configuration using the transition functions. Assuming that the configuration after reading the whole input is (q, v), we get the output by applying a designated linear function

$$F_q : \mathbb{Q}^n \to \mathbb{Q}$$

to the vector v.

Lemma 8.6. *Weighted automata and weighted automata with states recognise the same functions.*

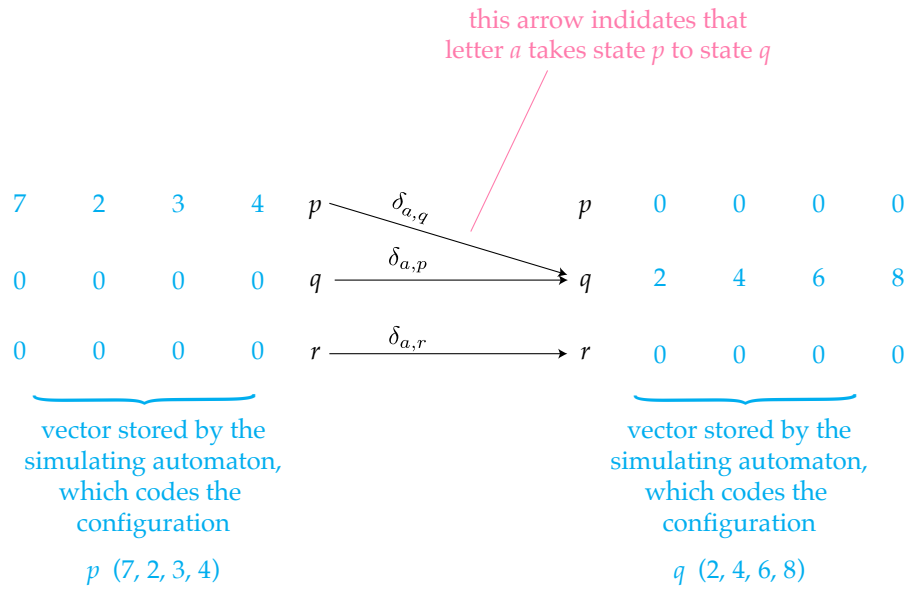
Proof. We need to show how a weighted automaton with states is converted into one without states. Consider an weighted automaton with states $\mathcal{A}$ which has control states Q and dimension n. We will simulate $\mathcal{A}$ by a weighted automaton $\mathcal{B}$ without states, which will have dimension $n \times Q$. For a state $q \in Q$, define two linear maps

$$\mathbb{Q}^n \underset{\pi_q}{\overset{\iota_q}{\rightleftarrows}} \mathbb{Q}^{n \times Q}$$

in the natural way, i.e. ι_q maps coordinate i to coordinate (q, i) and leaves other coordinates at zero, while π_q projects coordinate (q, i) to coordinate i. To prove the lemma, we design a weighted automaton $\mathcal{B}$ with dimension $Q \times n$ such that the following invariant is preserved: (*) for every input word, if the configuration of $\mathcal{A}$ after reading it is (q, v), then the state of $\mathcal{B}$ after reading the same input is $\iota_q(v)$. The initial state of $\mathcal{B}$ is defined by applying the invariant to the initial configuration of $\mathcal{A}$. It remains to define the transition function. Let a be an input letter. For a control state q, define f_q to be the linear map obtained by taking the following composition:

$$\mathbb{Q}^{n \times Q} \xrightarrow{\pi_q} \mathbb{Q}^n \xrightarrow{\delta_{a,q}} \mathbb{Q}^n \xrightarrow{\iota_{\delta_a(q)}} \mathbb{Q}^{n \times Q} .$$

The transition function of the automaton $\mathcal{B}$ over input letter a is defined to be the sum of the linear functions f_q, with q ranging over all states. It is not difficult to see that this definition preserves the invariant; here is the picture:

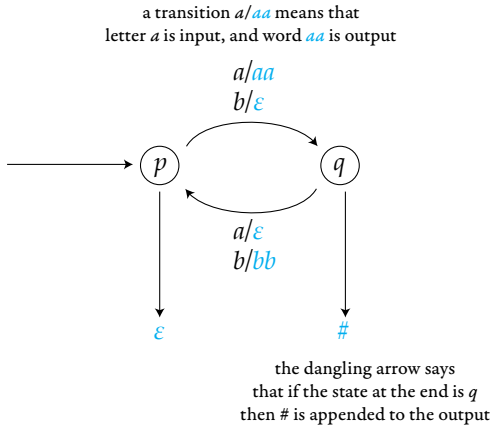


■

Sequential transducers. To give a high-level description of the undecidability proof, it will be convenient to use sequential transducers (we have already used a variant of this model, for ω-words, in Chapter 1). A *sequential transducer* is a type of word-to-word function

$$f : \Sigma^* \to \Gamma^*$$

which is described by an automaton as follows. The syntax consists of a deterministic finite automaton with input alphabet Σ, plus a labelling of each transition by a word (possibly empty) over the output alphabet Γ. Furthermore, instead of distinguishing a subset of final states, we give an *end of word* function from states to Γ^*. Here is an example:



The automaton produces an output as follows: run it on the input word, and output all words that label the transitions; at the end add the value of the end of word function applied to the last state.
The following lemma summarises the closure properties of weighted automata that will be used in the undecidability proof.

Lemma 8.7. *If $f, g : \Gamma^* \to \mathbb{Q}$ are recognised by finite weighted automata, then also the following functions are also recognised by finite weighted automata:*

1. *the weighted sum sum $a \cdot f + b \cdot g$, for $a, b \in \mathbb{Q}$;*
2. *the composition $f \circ s : \Sigma^* \to \mathbb{Q}$, for a sequential transducer $s : \Sigma^* \to \Gamma^*$;*
3. *the function which is defined as f on L and as g outside L, for regular $L \subseteq \Gamma^*$.*

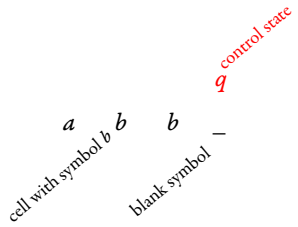
Proof. The weighted sum is immediate. For items 2 and 3, it is convenient to use an extended model, The closure properties in items 2 and 3 of the lemma are easy to do for extended weighted automata, and therefore the lemma follows from the above claim. ∎

Equipped with Lemmas 8.5 and 8.7, we can now prove the undecidability result from Theorem 8.4.

Proof of Theorem 8.4. A reduction from the halting problem. For a Turing Machine M, we will define a weighted automaton $\mathcal{A}$ such that $\mathcal{A}$ can output zero if and only if M has some halting computation. Suppose that Σ is the work alphabet of the machine M. We encode a configurations of the machine as a word over the alphabet

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

in the natural way, here is a picture:



To make the encoding unique, 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. It is not difficult to see that there is a sequential transducer $s : \Delta^* \to \Delta^*$ such that if w encodes a configuration, then $s(w)$ encodes the same configuration after one computation step. Define L to be the set of words of the form

$$w_1 \# w_2 \# \cdots \# w_{n-1} \# w_n \tag{8.1}$$

where $w_1, \ldots, w_n$ represent configurations, with w_1 being initial and w_n being final, and the letters # and # are separator symbols (note how the red separator is used only once, at the end). The language L is regular. Define

$$f_1, f_2 : (\Delta + \# + \#)^* \to \mathbb{Q}$$

to be functions that map a word as in (8.1) to numbers that represent (according to Lemma 8.5) the words

$$s(w_1) \# s(w_2) \# \cdots \# s(w_{n-1}) \qquad \text{and} \qquad w_2 \# w_2 \# \cdots \# w_n$$

respectively (it is not important what the functions do for inputs outside L). The first function applies the successor function to the first $n-1$ configurations (the red separator is used to not copy the last configuration), while the second functions copies the last $n-1$ configurations. The functions f_1, f_2 are recognised by weighted automata thanks to Lemmas 8.5 and 8.7. From the construction, it follows that the Turing machine has a halting computation if and only if

$$f_1(w) - f_2(w) = 0 \qquad \text{for some } w \in L.$$

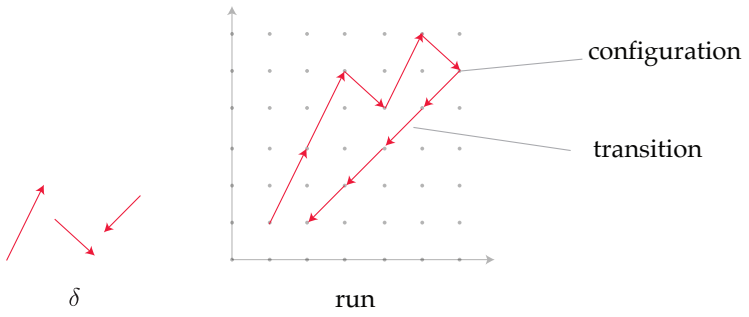
This problem can be reduced to checking if some weighted automaton can produce zero, thanks to the closure properties in Lemma 8.7. ∎

Problem 35. Show that emptiness is undecidable already for probabilistic automata, i.e. automata 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 $F \subseteq \{1, \ldots, d\}$. Here emptiness is the question: is there some input word which produces output at least $1/2$?

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*, i.e. a pair $x \to y$ with $y - x \in \delta$. Here is a picture in 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 [21], although the computational

complexity of the problem remains unknown, see [26]. 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 can be solved using integer linear programming, see Excercise 39.
In this chapter, we present a simple algorithm for a different problem, called coverability.

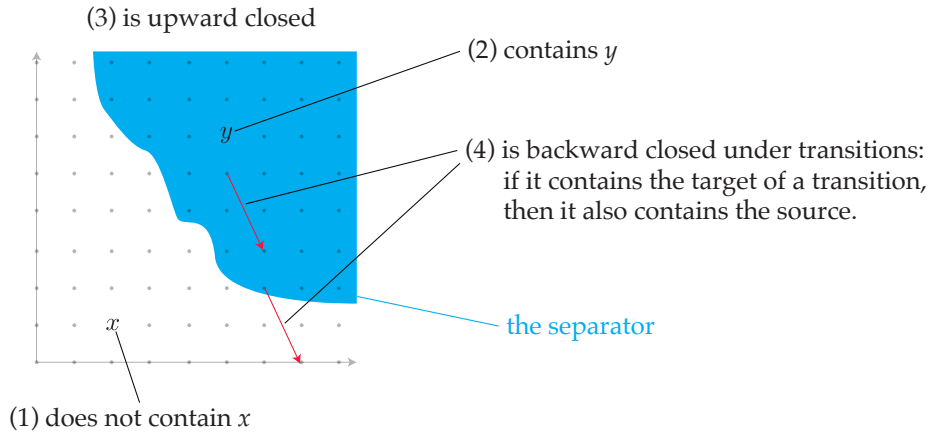
Theorem 9.2. *The following problem, called coverability, is decidable:*

- **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 [27] 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 a 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



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, 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 $a + c \to a + c$ is a transition, for every $c \in \mathbb{N}^d$. 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, again using the induction assumption.

An alternative proof would use the Ramsey theorem. ∎

By Dickson's Lemma shown above, 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 36. 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 37. 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 38. 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 39. Define an $\mathbb{Z}$*-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 40. 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 41. 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.

10
Polynomial grammars

In this section, we show that one can decide if a polynomial grammar – a type of grammar that generates rational numbers – has its language contained in $\{0\}$. The key tool is the Hilbert Basis Theorem.

Polymoial grammars. 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

$$p : \mathbb{Q}^n \to \mathbb{Q}^k$$

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 rational numbers (as opposed to words) and uses polynomial functions in the rules (as opposed to concatenation). Also, nonterminals are allowed to generate tuples of rational numbers, although we require the starting nonterminal to generate only numbers (this restriction is not important).

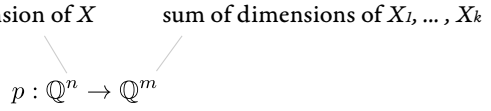
Definition 10.1 (Polynomial grammar)**.** *A* polynomial grammar *consists of*

- *a set X of nonterminals, each one with a* 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



is a polynomial function.

If a nonterminal has dimension n, then it generates a subset of $\mathbb{Q}^n$, 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.

Example 10. 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 $(a,b) \mapsto a$. $\square$

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 10.2. Define a *solution* to a grammar to be a function η which associates to each nonterminal X a set of vectors of rational numbers 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 10.2. *One can decide if the language of a polynomial grammar is contained in* $\{0\}$.

The nonzeroness problem 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 crucial ingredient is having a finite witness 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. Let us write $\mathbb{Q}[X]$ for the set polynomials with variables in X and coefficients in $\mathbb{Q}$. Define an *ideal* to be a set $I \subseteq \mathbb{Q}[X]$ with the following closure properties:

$$\underbrace{p, q \in I \quad \Rightarrow \quad p+q \in I}_{\text{addition inside } I} \qquad \underbrace{p \in I, q \in \mathbb{Q}[X] \quad \Rightarrow \quad pq \in I}_{\text{multiplication by arbitrary polynomials}}.$$

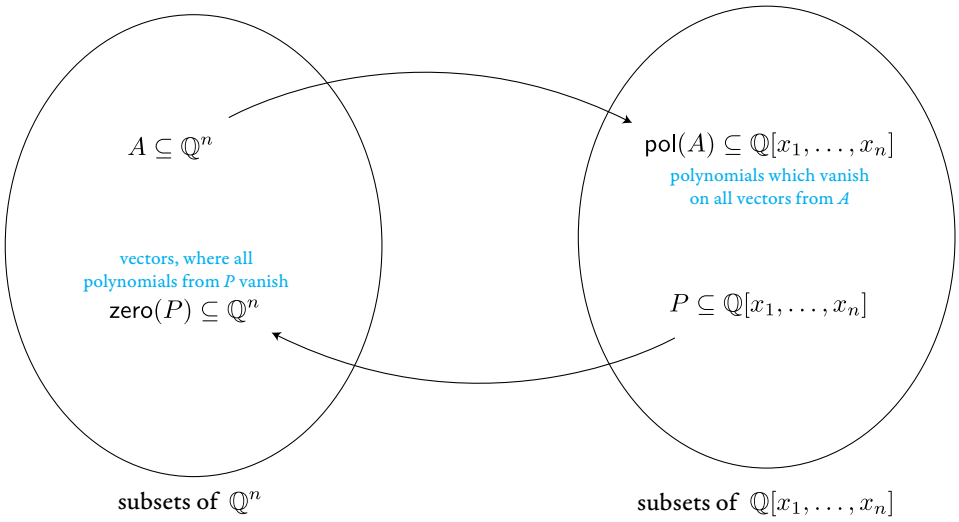
If $P \subseteq \mathbb{Q}[X]$ is a set of polynomials, then the ideal generated by P, denoted by $\langle P \rangle$, is the set of polynomials of the form

$$p_1 q_1 + \cdots + p_n q_n \qquad \text{where } p_i \in P, q_i \in \mathbb{Q}[X].$$

This is the inclusionwise least ideal that contains P. We are now ready to state the Hilbert Basis Theorem (together with a computational remark on deciding membership in ideals).

Theorem 10.3. *If X is a finite set of variables, then every ideal in $\mathbb{Q}[X]$ is finitely generated, i.e. of the form $\langle P \rangle$ for some finite set of polynomials. Furthermore, given a polynomial p and a finite set of polynomials P, one can decide if p belongs to the ideal $\langle P \rangle$.*

Algebraic closure. We say that a polynomial function $p : \mathbb{Q}^n \to \mathbb{Q}$ *vanishes* on a set $X \subseteq \mathbb{Q}^n$ 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:



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 $A \subseteq \mathbb{Q}^n$, define its *closure* as follows:

$$\overline{A} \quad \stackrel{\text{def}}{=} \quad \mathsf{zero}(\mathsf{pol}(A)).$$

Example 11. Suppose that the dimension is $n = 1$. Then $\mathsf{pol}(A)$ contains only the constant zero polynomial whenever A is infinite. This means that the algebraic closure of any infinite set $A \subseteq \mathbb{Q}$ is the whole space $\mathbb{Q}$. 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 $x^2 + y^2 = 1$. $\square$

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. By the Hilbert Basis Theorem, the algebraic closure can be represented by giving a finite basis for the ideal $\mathsf{pol}(A)$.

The following lemma is the key to deciding if a grammar generates only zero.

Lemma 10.4. *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, (10.1) and (10.2), which show how closure interacts with polynomial images and Cartesian products. The first inclusion is about polynomial images:

$$p(\overline{A}) \subseteq \overline{p(A)} \qquad \text{for every } A \subseteq \mathbb{Q}^n \text{ and polynomial } p : \mathbb{Q}^n \to \mathbb{Q}^m. \quad (10.1)$$

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:

$$\overline{A} \times \overline{B} \subseteq \overline{A \times B} \qquad \text{for every } A \subseteq \mathbb{Q}^n \text{ and } B \subseteq \mathbb{Q}^m. \tag{10.2}$$

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 (10.2).
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.

$$\begin{array}{rcl}
p(\overline{\eta}(X_1), \ldots, \overline{\eta}(X_n)) & = & \text{by definition of } \overline{\eta} \\
p(\overline{\eta(X_1)}, \ldots, \overline{\eta(X_n)}) & \subseteq & \text{repeated application of (10.2)} \\
p(\overline{\eta(X_1) \times \cdots \times \eta(X_n)}) & \subseteq & \text{by (10.1)} \\
\overline{p(\eta(X_1) \times \cdots \times \eta(X_n))} & \subseteq & \text{because } \eta \text{ is solution and closure is monotone} \\
\overline{\eta(X)} & = & \text{by definition of } \overline{\eta} \\
\overline{\eta}(X) & &
\end{array}$$

This completes the proof of the lemma. Note that the proof is not very specific to polynomials and rational numbers, and it would work for more abstract notions of algebra, as will be defined in Section 10.1. ∎

We now complete the proof of Theorem 10.2. Recall that by enumerating derivations, we have 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 10.4, the generated language is contained in $\{0\}$ if and only if there is a an assigment 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 $\{0\}$ or the empty set.

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. For this, we use the following lemma, which follows from decidability of membership $p \in \langle P \rangle$.

Lemma 10.5. *Assume that a closed set A is represented by a finite basis of the ideal $\mathsf{pol}(A)$. If A, B are closed sets, then the following are also closed, and their representations can be computed:*

$$A \cup B \qquad A \times B \qquad p(A) \qquad \text{where } p \text{ is a polynomial.}$$

Furthermore, one can decide equality on closed sets.

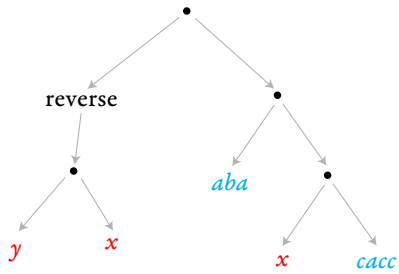
10.1 Application to equivalence of register automata

We now use Theorem 10.2 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

$$(\mathbb{Q}, +, \times) \qquad (\{a, b\}^*, \cdot).$$

We adopt the convention that boldface letters like $\mathbf{A}$ and $\mathbf{B}$ range over algebras, and the universe (the underlying set) of an algebra $\mathbf{A}$ is denoted by A. Define a polynomial with variables X in an algebra $\mathbf{A}$ 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



We write $\mathbf{A}[X]$ for the set of polynomials over variables X in the algebra $\mathbf{A}$. 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 12. If the algebra is $(\mathbb{Q}, +, \times)$, 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 $\mathbb{Q}$ with the operations being addition and the family of scalar multiplications $\{x \mapsto ax\}_{a \in \mathbb{Q}}$, then the polynomials are exactly the affine functions. □

Definition 10.6 (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

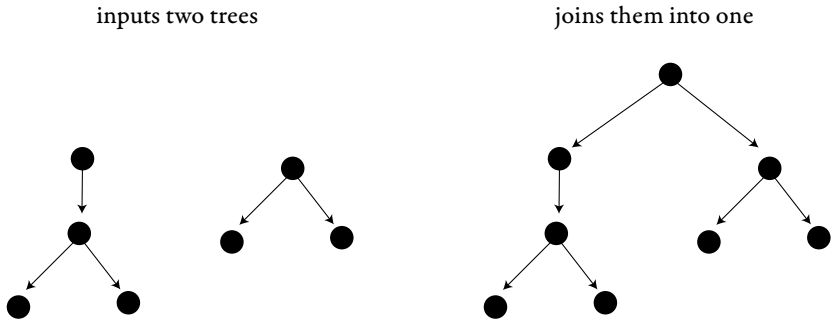
$$(q, v) \overset{a}{\mapsto} (p, f(v)) \qquad \text{where } \delta(q, a) = (p, f).$$

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 13. 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 14. 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 1 and one state. When it reads a letter a, it executes the register update given by the polynomial $a \cdot 1$. The output function is the identity. □

Example 15. Consider an algebra $\mathbf{A}$ 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 $\mathbf{A}$, with input alphabet $\{a\}$, which maps a word a^n to a balanced binary tree of depth n. $\square$

The following result is a direct corollary of Theorem 10.2. The result was first shown in [3, Theorem 4], were the proof was also based on the Hilbert Basis Theorem.

Theorem 10.7. *The following problem is decidable:*

- **Input.** *Two functions* $\Sigma^* \to \mathbb{Q}^n$ *given by register automata over* $(\mathbb{Q}, +, \times)$*;*
- **Question.** *Are the functions equal?*

The above theorem generalises Theorem 8.3, because weighted automata can be viewed as a special case of register automata over $(\mathbb{Q}, +, \times)$ where only linear maps are allowed as the register updates, instead of arbitrary polynomials, as allowed in Theorem 10.7. The generalisation of Theorem 8.3 is only in terms of decidability, since the running time of the algorithm for Theorem 10.7 is not estimated in any way, not to mention polynomial time. In fact, a lower bound of Ackermann time is given in [3, 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

$$\underbrace{Q}_{\text{dimension } n} \quad + \quad \underbrace{1}_{\text{dimension } 1} \quad .$$

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 $\overline{a} \in \mathbb{Q}^n$ such that configuration $(q, \overline{a})$ 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 10.2. ∎

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 10.7, we showed that equivalence is decidable for register automata over the algebra $(\mathbb{Q}, +, \times)$. From this we can infer decidability for some other algebras, by coding them into rational numbers according to the following definition.

Definition 10.8. *Let* $\mathbf{A}$ *and* $\mathbf{B}$ *be algebras. We say that* $\mathbf{A}$ *can be simulated by polynomials of* $\mathbf{B}$ *(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* $\mathbf{A}$*, there is a polynomial* g *of* $\mathbf{B}$ *which makes the following diagram commute*

$$\begin{array}{ccc} A^m & \xrightarrow{(\alpha,\ldots,\alpha)} & B^{m\times n} \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle g} \\ A & \xrightarrow[\alpha]{} & B^n \end{array}$$

It is easy to see that if $\mathbf{A}$ can be simulated by polynomials of $\mathbf{B}$, then decidability of equivalence of register automata over $\mathbf{B}$ implies decidability equivalence of register automata over $\mathbf{A}$.

Corollary 10.9. *For every finite alphabet* Σ*, the equivalence problem is decidable for register automata over the algebra* $(\Sigma^*, \cdot)$.

Proof. By Theorem 10.7, it suffices to show that $(\Sigma^*, \cdot)$ can be simulated by polynomials of $(\mathbb{Q}, +, \times)$. Assume without loss of generality that Σ is $\{0, \ldots, n-1\}$. We use the coding:

$$a_i a_{i-1} \cdots a_0 \in \Sigma^* \qquad \overset{\alpha}{\mapsto} \qquad (n^i, \quad \underbrace{a_i n^i + \cdots a_0 n^0}_{\text{input as a number in base } n} \quad) \in \mathbb{Q}^2$$

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 10.7, 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 [28].

■

Problem 42. 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.

11
Parsing in matrix multiplication time

In this chapter we present a parsing algorithm of Valiant, which parses context-free languages in approximately the same time as (Boolean) matrix multiplication. Another view on Boolean matrix multiplication is that it is the problem of computing composition of binary relations:

- **Input.** Two binary relations $R, S \subseteq \{0, \ldots, n\}^2$;
- **Output.** The composition

$$R \circ S = \{(i,k) : (i,j) \in R \text{ and } (j,k) \in S \text{ for some } k\}.$$

The naive algorithm for the above 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 $\mathcal{O}(n^{2.3729})$.

Theorem 11.1. *Assume that multiplication of $n \times n$ Boolean matrices can be computed in time $\mathcal{O}(n^\omega)$ for $\omega \in \mathbb{Q}$. Then membership in a context-free language can be decided in time at most*

$$\text{poly}(\mathcal{G}) \cdot n^\omega \cdot \log^2(n) \qquad \textit{where } \mathcal{G} \textit{ is the grammar and } n \textit{ is the length of the input.}$$

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 a is a nonterminal). A grammar can be converted into Chomsky Normal Form in polynomial time, so this assumption can be made without loss of generality. Define an 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 $\{1, \ldots, n\}$. The intuition is that $I \in M_X$ means that the nonterminal X generates the infix corresponding to I. For parse matrices M, N of same dimension, define their product $M \circ N$ by

$$(M \circ N)_X \stackrel{\text{def}}{=} \bigcup_{X \to YZ} M_Y \circ M_Z$$

where the union ranges over rules of the grammar and $M_Y \circ M_Z$ is 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.

Lemma 11.2. *For length n parse matrices, product can be computed in time $\mathcal{O}(n^\omega)$.*

Proof. Composition of binary relations is the same as multiplying Boolean matrices. ∎

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 11.3. *There is an algorithm which runs in time $T(n)$ of order at most*

$$\text{poly}(\mathcal{G}) \cdot \log(n) \cdot n^\omega$$

and 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 11.1.

Proof of Theorem 11.1. 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. This parse matrix can be computed in time linear in n. It is easy to see that w is generated by the grammar if and only if the closure M^* contains $\{1, \ldots, n\}$ on the component corresponding to the starting nonterminal. It suffices therefore to compute 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 matrixs for the first and second halves of w (using a recursive procedure), and then combine these using the algorithm from Proposition 11.3. The running time of this algorithm is at most

$$T(n) + 2T(\frac{n}{2}) + \cdots + 2^k T(\frac{n}{2^k}). \tag{11.1}$$

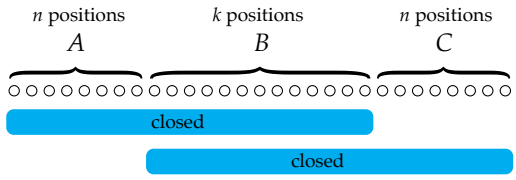
Because $T(n)$ is at least linear, it follows that

$$2^i T(\frac{n}{2^i}) \leq T(n),$$

which shows that the running time (11.1) is at most $\log n$ times slower then $T(n)$, thus proving the theorem, given the bounds on $T(n)$ from Proposition 11.3. ∎

It remains to prove the proposition. We use the following lemma.

Lemma 11.4. *Suppose that M is a length $k + 2n$ parsing matrix that is closed on the intervals $A \cup B$ and $B \cup C$ as depicted below:*



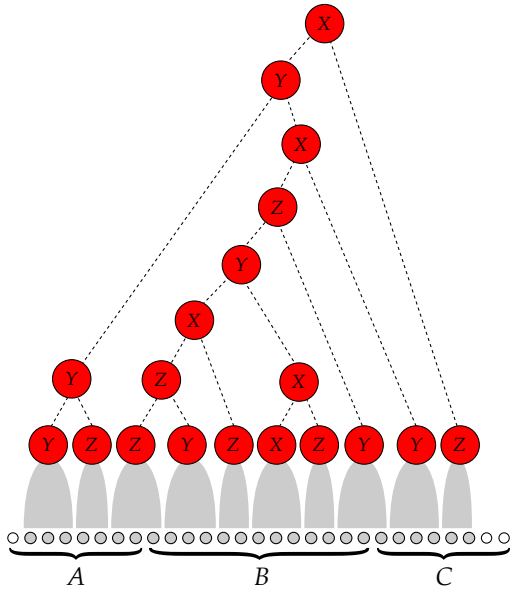
Then the closure M^* can be computed in time $\text{poly}(\mathcal{G}) \cdot n^\omega + T(n)$.

Proof. Define N to be $M \cup M \circ M$ restricted intervals that contain B. The main observation in the lemma is

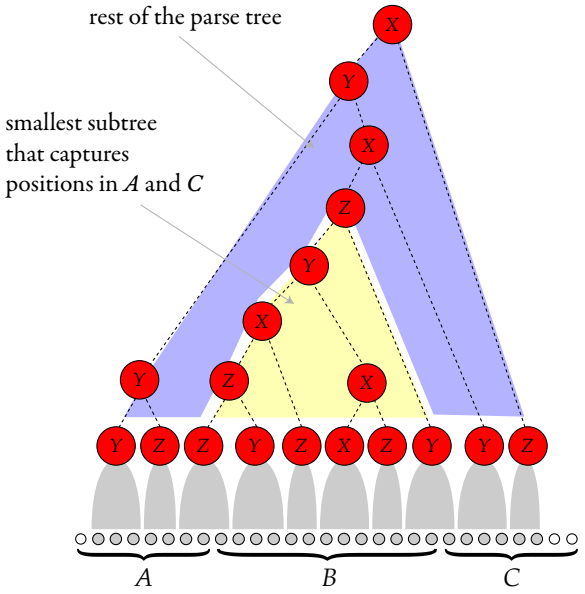
$$M^* = M \cup N^*. \tag{11.2}$$

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 11.2, N can be computed in time $\text{poly}(\mathcal{G}) \cdot n^\omega$. Because the matrix M was closed over intervals A and C, it follows N is also closed over these intervals. Since all entries of N contain 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 (11.2). 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:

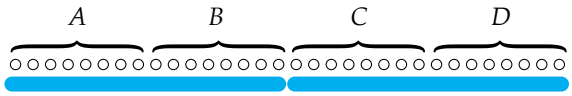


The leaves in the picture correspond to intervals from M, and the inner nodes correspond to intervals that appear when computing the closure M^*. Consider the smallest subtree of the parse tree which captures positions from both B and C, decomposing the parse tree into two parts (yellow and blue):



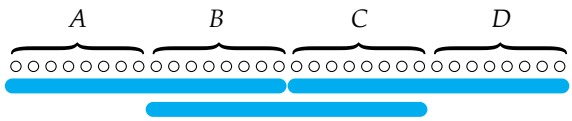
By minimality, the left subtree of the yellow subtree captures positions only in $A \cup B$, and the right subtree of the yellow subtree captures only positions in $B \cup C$. Therefore, the root of the yellow subtree corresponds to the matrix $M \circ M$. It follows that all leaves of the blue part of the parse tree correspond to the matrix N; and thus the root of the blue part corresponds to the matrix N^*. ∎

Proof of Proposition 11.3. 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:

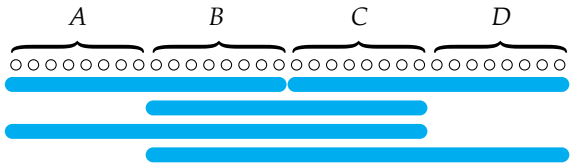


As in Lemma 11.4, the blue rectangles indicate the intervals which are closed.

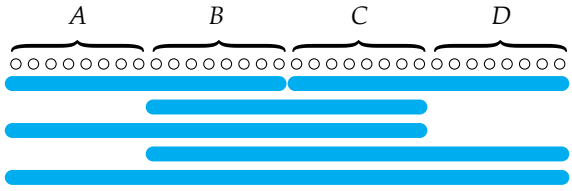
1. By induction, compute the closure of the interval $B \cup C$:



2. Using Lemma 11.4 twice, compute the closures of $A \cup B \cup C$ and $B \cup C \cup D$:



3. Using Lemma 11.4, 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 11.1, 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

$$2^{i} \cdot (\frac{n}{2^{i}})^{\omega} \leq n^{\omega},$$

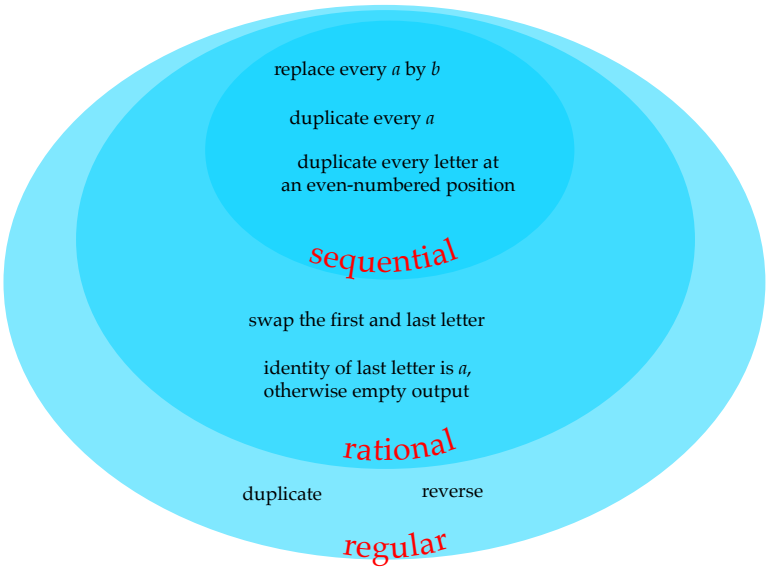
which gives the bound in the proposition. ∎

Problem 43. Show that the operation $M \circ N$ is not associative.

12

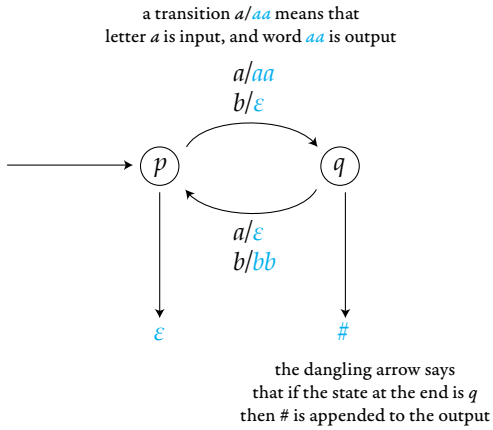
Two-way transducers

In this part of the lecture, we talk about automata which define functions from words to words. Such automata are called *transducers*. In this chapter, we cover three types of transducers, which are called *sequential*, *rational* and *regular*. Here is the transducer map:



12.1 *Sequential transducer*

We have already seen sequential transducers in Section 8. These are deterministic finite automata, where transitions are labelled by possibly empty words over the output alphabet, together with an end-of-input function which says what should be added to the output as a function of the last state that was seen. Recall the following example from Section 8:



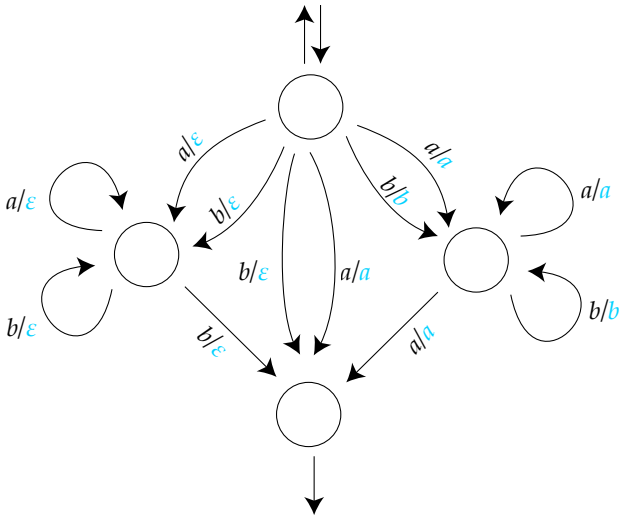
12.2 *Rational transductions*

There are many ways to define rational transductions, of which we mention three: functional NFA with output, lookahead DFA with output, and Eilenberg bimachines.

Functional NFA with output. An NFA *with output* is a nondeterministic finite automaton over the input alphabet, where each transition is labelled by a possibly empty word over the output alphabet. When given an input word over the input alphabet, the automaton produces all those words over the output alphabet which can be found by taking an accepting run, and reading from left to right the words on the transitions. Such an automaton is called *functional* if

for every input word, there is exactly one output word (but possibly realised through several different accepting runs). Such an automaton is called *unambiguous* if for every input word, it has a unique accepting run; this implies being functional.

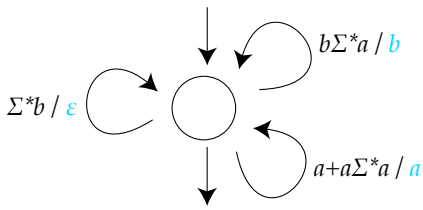
Example 16. The following unambiguous NFA with output realises the function "identity if the last letter is a, otherwise erase the entire word".



□

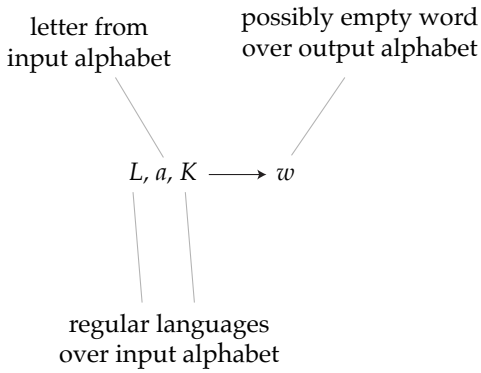
Lookahead DFA with output. An *lookahead* NFA *with output* is defined like an NFA with output, except that transitions are labelled by pairs (regular language over the input alphabet, possibly empty word over the output alphabet). A transition labelled by a pair (L, w) can be applied if the unread part of the input belongs to L; the effect of using such transition is that w gets added to the output. A *lookahead* DFA *with output* is the special case where for every state q and word u over the input alphabet, there is exactly one transition that is applicable when the automaton is in state q and the unread part of the word is u.

Example 17. The following lookahead DFA with output realises the function "identity if the last letter is a, otherwise erase the entire word":

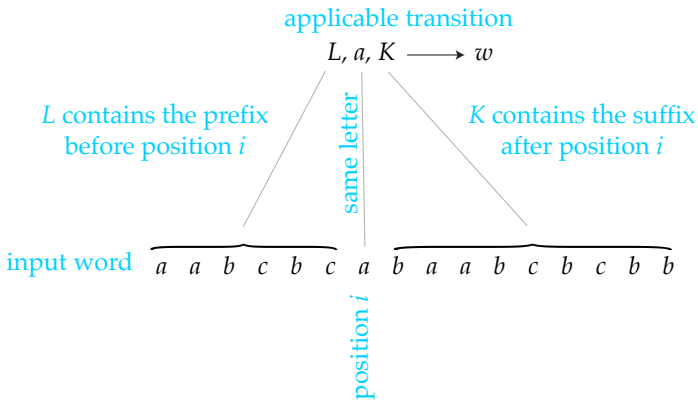


In general, there might be a need to use control states, e.g. for the function "swap the first and last letter". □

Eilenberg bimachine. An *Eilenberg bimachine* is set of four-tuples of the form



We require that for every word over the input alphabet and every position i in that word, there is a unique transition that is applicable to position i, in the sense illustrated below:



Given an input word, the output of the bimachine is defined choosing the applicable transition for each input position, and concatenating the output words used in these transitions, in the order given by the input positions.

Rational functions. We show that the models described above are all equivalent. We use the name *rational function* for an word-to-word function that is defined by any one of these equivalent models.

Theorem 12.1. *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.*

Proof sketch.

$1 \subseteq 2$ The simulating 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

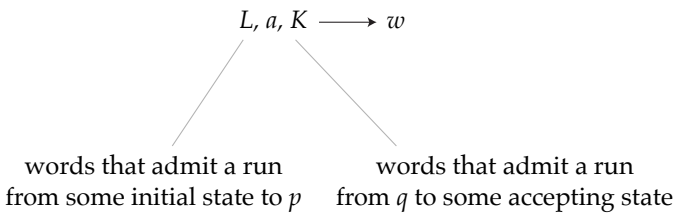
more than one transition can be chosen, some arbitrary tie-breaking mechanism is used.

$2 \subseteq 3$ Consider some lookahead DFA with output $\mathcal{A}$. Let $\mathcal{B}$ be a deterministic right-to-left DFA 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 $\mathcal{A}$. For each transition

$$p \xrightarrow{a/w} q$$

of the automaton $\mathcal{A}$, the simulating Eilenberg bimachine has a quadruple of the form



Because $\mathcal{A}$ was unambiguous, for each position there is a unique applicable quadruple.

$4 \subseteq 1$ Consider an Eilenberg bimachine. Let $\mathcal{A}$ be a left-to-right DFA (i.e. a usual DFA) which recognises all languages that appear on the first coordinate (the L languages) of a rule in the machine, and let $\mathcal{B}$ be a right-to-left DFA which recognises all languages that appear on the third coordinate (the K languages) of a rule in the machine. The simulating functional NFA with output has states which are pairs (state of $\mathcal{A}$, state of $\mathcal{B}$), and the

transitions are of the form

$$(p, aq) \xrightarrow{a/w} (pa, q)$$

where aq applies the transition function of $\mathcal{B}$, pa applies the transition function of $\mathcal{A}$, and w is the unique letter that appears in a rule (L, a, K, w) where L intersects (and therefore contains) the language of state p and K intersects (and therefore contains) the language of state q.

■

Fact 12.2. *Every rational function is a composition of: (a) a reverse sequential function (i.e. reverse, a sequential function, reverse again); followed by(b) a sequential function.*

Proof. The function (a) computes the lookahead of a lookahead DFA, and the function (a) runs the lookahead DFA itself. ■

12.3 Regular transducers

We now introduce the most powerful class of transducers, namely *regular transducers*. In this chapter, we use a definition in terms of deterministic two-way automata with output. In the next chapter, we present an equivalent model, which uses registers.

Two-way transducers. A *deterministic two-way transducer* consists of:

- an input alphabet Σ;
- an output alphabet Γ;
- a set of states Q with a distinguished initial state and final state;
- a transition function

$$\delta : Q \times (\Sigma \cup \{\vdash, \dashv\}) \to Q \times \{-1, 0, 1\} \times \Gamma^*$$

The semantics of the transducer are defined similarly to Turing machines, as follows. (Actually, the model is equivalent to a Turing machine where there is one read-only input tape and one write-once output tape.) If the input word is $a_1 \cdots a_n$, the we embellish it with start and end markers as follows:

$$\vdash a_1 \cdots a_n \dashv$$

The head of the automaton is then placed over the start marker $\vdash$ with the 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, yielding a new state, a direction to move the head, and some output letters that to be 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. doing a -1 move on the start marker $\vdash$, or doing a 1 move on the end marker $\dashv$), 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^*. If the automaton does enter the final state, the computation stops.
Typical things that can be done using a two-way transducer are duplication or reversing the input. Define a *regular transducer* to be any (partial) function from words to words that is recognised by a deterministic two-way transducer.

Composing with rational functions. We show that regular functions are closed by pre- and post-compositions with rational ones.

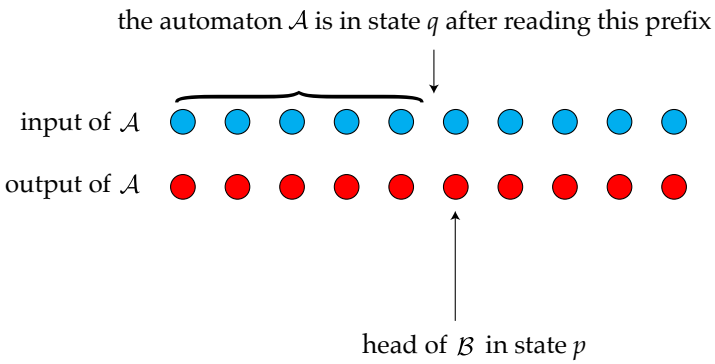
Theorem 12.3. *If f is a regular function and g is rational (with suitable input and output alphabets), then $f \circ g$ and $g \circ f$ are regular.*

By Fact 12.2, it suffices to consider the case when g is sequential, or reverse sequential. By symmetry of the two-way model, it suffices to consider the case when g is sequential. For post-composition $g \circ f$, the obvious construction

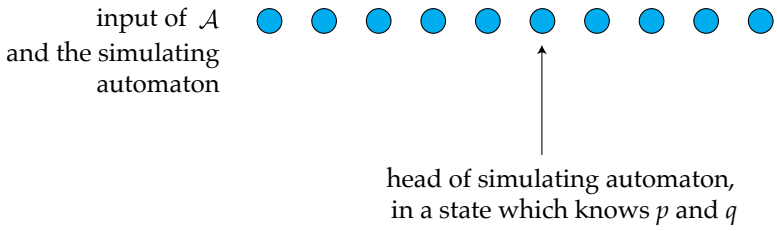
works: the simulating automaton keeps ints state the state of the automaton for g. Pre-composition $f \circ g$ is much more interesting, and shown in the following lemma.

Lemma 12.4. *If $g : \Sigma^* \to \Gamma^*$ is a sequential function (i.e. recognised by a* DFA *with output) and $f : \Gamma^* \to \Delta^*$ is recognised by a deterministic two-way transducer, then the composition $f \circ g : \Sigma^* \to \Delta*$ is also recognised by a deterministic two-way transducer.*

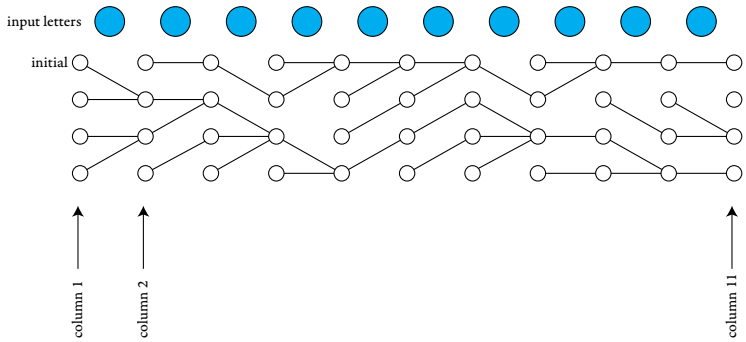
Proof. The idea for this proof comes from Hopcroft and Ullman. Suppose that g is recognised by a DFA $\mathcal{A}$ with states Q, and f is recognised by a two-way automaton $\mathcal{B}$ with states P. To make notation simpler, we assume that $\mathcal{A}$ is letter-to-letter, i.e. each transition outputs exactly one letter. The general idea for the simulation is straightforward. Suppose that the simulated automata are in a configuration like this:



Then the simulating automaton knows the current configuration of $\mathcal{B}$, plus the state of $\mathcal{A}$ on the prefix to the left of the head:



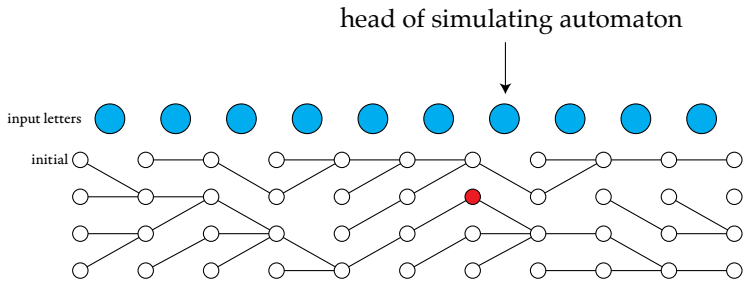
The question is how to maintain this information, especially when the simulated automaton $\mathcal{B}$ wants to move its head to the left. The key insight is to consider the graph which describes the states of $\mathcal{A}$ and how they are updated by the transition function. This graph looks likes this:



The vertices of the graph are configurations of $\mathcal{A}$, i.e. pairs (state of $\mathcal{A}$, column between positions in the word), and the edges correspond to transitions of the automaton. We number the columns beginning with 1. Because $\mathcal{A}$ is deterministic, the graph is a forest.

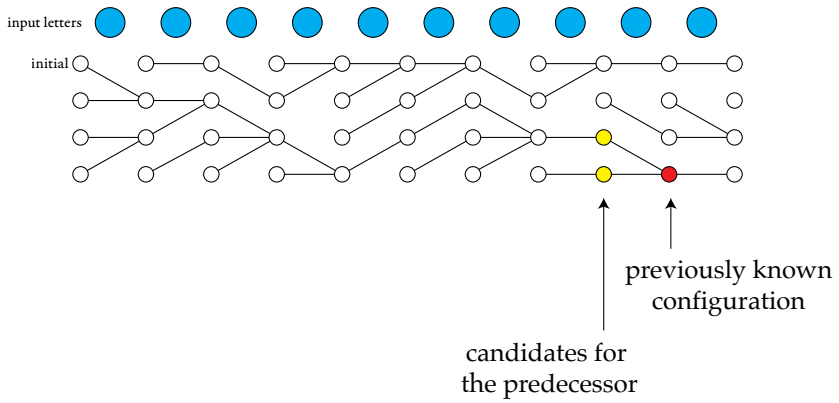
Let us define q_i to be the state of $\mathcal{A}$ when in the i-th column, i.e. after reading the first $i-1$ letters of the input word. The simulating two-way automaton uses the state q_i to get the i-th letter in the output $f(w)$. Suppose that the head of the simulating two-way automaton is over 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:



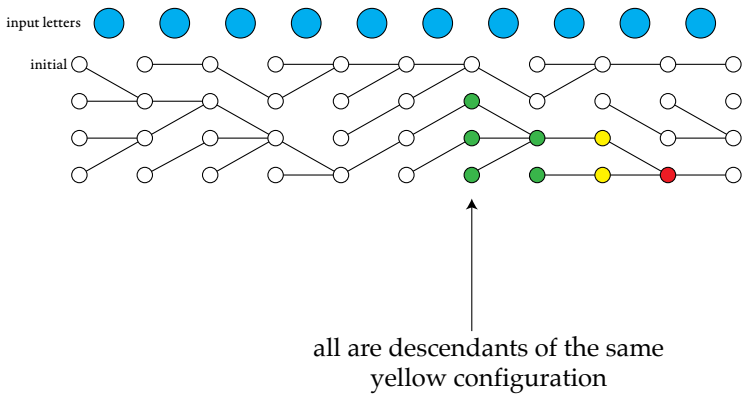
We want to show that the state of $\mathcal{A}$ can be maintained when simulating one transition of the two-way automaton $\mathcal{B}$. If the transition of the two-way automaton $\mathcal{B}$ does not move the head, there is no problem. If the transition of the two-way automaton moves the head to the right, there is also no problem, since the transition function of $\mathcal{A}$ can be simply applied to the known state q_i.

The issue is when the simulated two-way automaton $\mathcal{B}$ 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, the state q_i corresponds to the configuration (q_i, i) in the forest. By inspecting the subtree this configuration, which goes in the left direction of the picture, we will find the unique predecessor of q_i that is connected to the initial configuration. 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:



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 colour all descendants of the yellow configuration (and therefore of the red configuration as well) using a green colour.

Two things may happen. In one case, we might reach a moment when all green configurations are descendants of a unique yellow configuration, as in this picture:



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: since we have stopped in the first column when the yellow configuration was uniquely identified, in the previous column there were at least two candidates for the yellow configurations. Therefore, if we remember the previous column, we can start moving to the right until the first time that the whole subtree reaches a single node, this way we will reach the red configuration again. But in our state we can keep the yellow configuration that was the actual predecessor.
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. ■

Problem 44. Show that deterministic two-way automata (seen as acceptors of words) can be complemented with polynomial blowup.

13
Streaming string transducers

Two-way automata are easy to describe, but a pain to work with. For example, it is difficult to show that they cannot do something, or it is difficult to show that they are closed under composition. Finally, it would be nice to have a one way model for efficiency reasons (the buzzword here is streaming). This is where register automata come in, a model also known as streaming transducers. In the end, we will prove that it is equivalent to deterministic two-way automata with output.

A register transducer consists of the following ingredients: • an input alphabet Σ; • an output alphabet Γ; • a set of states Q with a distinguished initial state $q_0 \in Q$; • a set of registers R; • a transition function

$$\delta : Q \times \Sigma \to Q \times \underbrace{(R \to (R \cup \Gamma)^*)}_{\text{register update}}$$

• an output function

$$out : Q \to (R \cup \Gamma)^*$$

The automaton is run as follows. The registers store words over the output alphabet. Initially, every register stores the empty word. When the automaton is in state q and it reads a letter a, transition function is applied to (q, a), yielding a new state p and a register update

$$f : R \to (R \cup \Gamma)^*$$

Then, in parallel, each register r is set to $f(r)$, with the register names replaced by their contents. For example if

$$f(r) = rar$$

with a being an input letter, then register r is replaced by its previous contents, then a, and then its previous contents again.

After the entire word has been processed, with the last state used being q, then the automaton outputs $out(q)$, with register names replaced by their contents.

Copyless restriction. The model, as defined above, goes beyond two-way automata with output. For example, one could initially set a register to a, and then duplicate its contents in every step, thus creating an output of exponential length. To avoid this, one introduces the following copyless restriction. We say that a register update

$$f : R \to (R \cup \Gamma)^*$$

is copyless if every register $r \in R$ appears in at most one word $f(s)$, and in that word it appears at most once. The intuition is that the register contents are physical objects and can only be moved around and not duplicated.

From now on, when we say register automaton, we assume that all register updates in the transition function are copyless. The output function need not be copyless, since it is applied only once, and requiring it to be copyless does not weaken the model.

Future oracle. A register automaton with a future oracle is defined as follows. The future oracle is a an automaton over the input alphabet, which is deterministic from right to left. If the states of the future oracle are P, then the transition function of the register automaton using this oracle is the same as in the definition of a register automaton, except that instead of seeing the current input letter from Σ, one sees the state of the future oracle in P, as obtained by reading the remaining input, beginning at the end of the input and ending in the first position that has not been read yet.

We now show that future oracles can be eliminated. In this particular model, past oracles do not make much sense, since they can be simulated by the state of the automaton.

Theorem. For every register transducer with a future oracle, there is register transducer (without any oracle), which defines the same function from words to words.

13.1 *Equivalence with two-way automata*

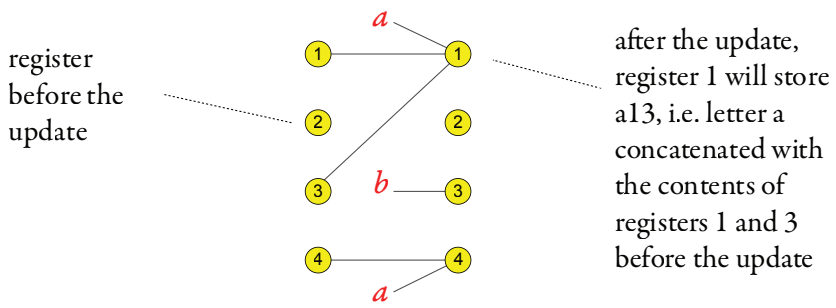
Here we show that two-way automata and register automata describe the same functions from words to words.

Theorem. The following devices recognise the same partial functions from words to words:

deterministic two-way automaton with output deterministic register automaton with regular lookahead. Note that we have already claimed (but not proved) that for deterministic register automata, the regular lookahead can be removed.

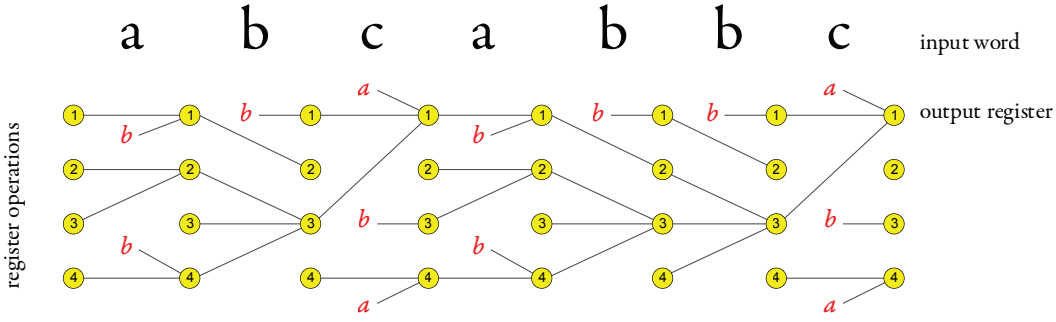
From register automata to two-way automata

Let us show the implication from 2 to 1, i.e. how to convert a deterministic register automaton with regular lookahead. Recall that in a register automaton with registers R and output alphabet Γ, a register update is a function $R \to (R \cup \Gamma)^*$. We draw a register update like this:



In the above picture, we assume that the inputs to the registers are read from top to bottom, formally speaking we would need to specify an ordering on the incoming edges for every register name at the right. When it reads an input

word, the register automaton executes a sequence of register updates, as in the following picture:



In the above picture, we assume that there is a single designated output register, and the output produced by the automaton is simply the contents of this output register at the end of the run. This assumption can be made when regular lookahead is available. Let Δ be the finite set of register operations that can be triggered by individual transition. Therefore a run of the register autompicaton can be viewed as transforming an input word into a word over Δ^*.

Lemma 1. There is an NFA with output which takes an input word to the sequence of register updates that it triggers.

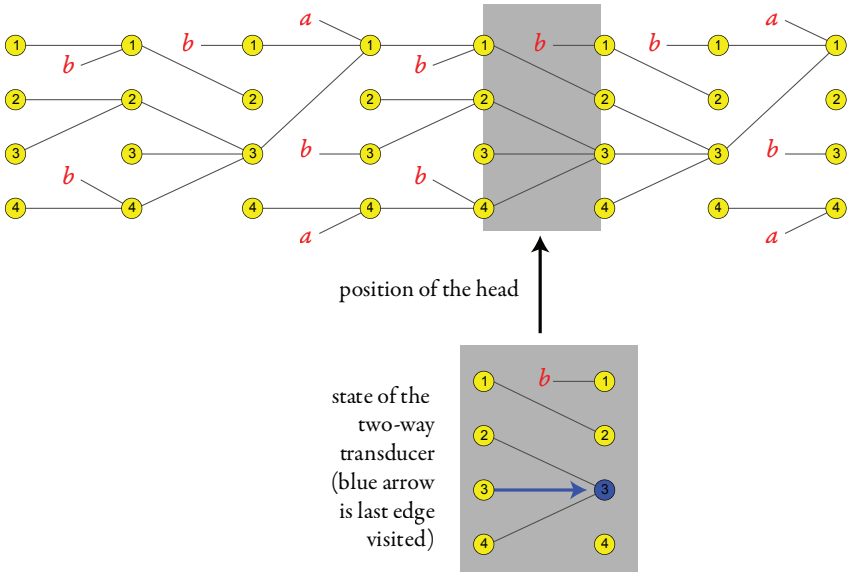
Proof. Just by simulating the automaton and using nondeterminism to guess the run of the regular lookahead. □

Lemma 2. If $f : \Sigma^* \to \Delta^*$ is recognised by an NFA with output and $g : \Delta^* \to \Gamma^*$ is recognised by a deterministic two-way automaton, then their composition $g \circ f$ is recognised by a deterministic two-way automaton.

Proof. Recall that NFA with output are equivalent to Eilenberg bimachines. An Eilenberg bimachine is the same thing as an oracle in a deterministic two-way automaton, and therefore $g \circ f$ is recognised by a deterministic two-way automaton with an oracle. Then, the oracle can be removed. □

Combining Lemmas 1 and 2 above, in order to show that a register automaton can be simulated by a deterministic two-way automaton, it suffices to show that

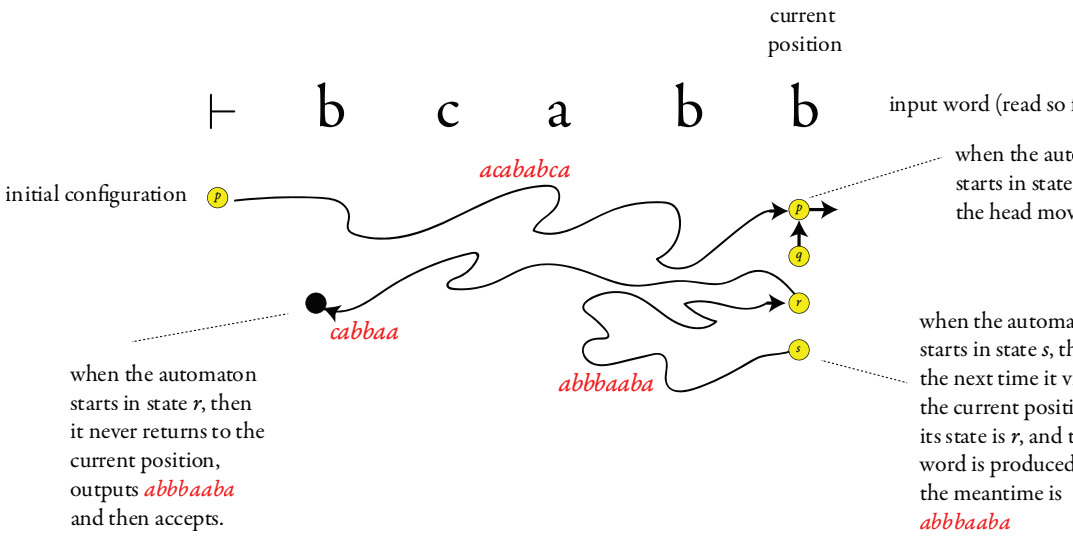
there is a deterministic two-way automaton which inputs a sequence of register updates, and which outputs the contents of the designated output register at the end. The automaton views the register operations are a graph, where the vertices are pairs (x, r) where x is a position in the input word and r is a register name. This graph is a tree, thanks to the copylesss assumption. The two-way automaton begins by going to (x, r) where x is the last position and r is the designated output register. Then it does a depth-first search traversal through the tree, outputting letters during this run. In order to do this, the automaton needs to only remember the current register r and the edge of the tree that was previously processed, as in the following picture:



From two-way automata to register automata

The idea is classical, dating back to the proof of Rabin and Scott that two-way automata (as accept/reject devices) can be made one way. Suppose that we have read a prefix of the input word. We need to remember what are the

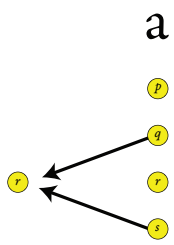
outputs that are produced by the two-way automaton on: a) the unique from the initial configuration to the first time that it reaches the current position; and b) runs that begin and end in the current position. Here is a picture:



More formally, one can describe the situation after reading a prefix w of the input by a function

$$f_w : Q \cup \{\text{initial}\} \to Q \cup \{\text{reject}\} \cup (\Gamma^* \times (Q \cup \{\text{accept}\}))$$

whose definition is according to the picture above. The natural idea would be to store, for each argument of the above function, the value of the function. The state of the transducer would be used to store the finite part of the function, and the registers (one for each argument) would be used to store the output words used in the function. The problem with this idea is that it would lead to register updates that are not copyless. This problem would arise if there would be two transitions that move to the left and lead to the same target state, as in this picture:



In such a situation, the register corresponding to state r would need to be used twice in the register update. The solution to this observation is that in an actual accepting run of the two-way automaton, only one of the transitions would be used. The reason is that if both transitions would be used, then the automaton would enter a loop. To use this observation, we use the following lemma, whose straightforward proof is omitted.

Lemma. Consider a deterministic two-way automaton with input alphabet Σ and states Q. Consider the function $g : \Sigma^* \to (\Sigma \times \mathsf{P}(Q))^*$ which additionally labels each input position by the set of states used by (the unique) accepting run of the automaton in that position. Then g is recognised by an NFA with output.

Register automata with regular lookahead are closed under precomposing with functions computed by NFA with output; for this one convert the NFA with output into a lookahead DFA with output, and then use the regular lookahead of the register automaton. Therefore, we can assume that a register automaton has access to the states in the accepting run (possibly none if there is no accepting states). It can then compute the function f_w with its domain restricted to states used in the accepting run; and here the natural construction works because it is copyless.

14
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 [1]. 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, as 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 14.1. *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 14.1, 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 14.2. *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. ∎

Exercise solutions

1
Determinisation of ω-automata

Problem 1. *Are the following languages regular:*

1. *prefix of v belongs infinitely often to the fixed regular language of finite words $L \subseteq \Sigma^*$;*
2. *word v contains infinitely many infixes of the form $ab^p a$, where p is prime;*
3. *word v contains infinitely many infixes of the form $ab^p a$, where p is even.*

Solution. Solutions of the points:

1. YES. Let $\mathcal{A}$ be an automaton for L. We make the same automaton with the same final states and Büchi acceptance condition.
2. NO. Assume that yes and $\mathcal{A}$ is an automaton for this language. Let $\mathcal{A}$ has n states and let $p > n$ be some prime number. The word $(b^p a)^\omega$ belongs to L, so $\mathcal{A}$ has an accepting run on it. Note that in every block b^p some two states are the same. We pump the part b^k between them p times, so that the block has now the length b^{p+kp}, which is not prime. The now word is accepted by $\mathcal{A}$, because it has an accepting run (which came out from the pumping of an old run), but no block has prime length.
3. YES. It suffices to count length of the block b^k modulo 2 and go into an accepting state on a which finishes such a block.

■

Problem 2. *Let UP be the set of* ultimately periodic words, *i.e.* $UP = \{uv^\omega : u, v \in \Sigma^*\}$. *Let L be a regular language of infinite words. Show that if $L \cap UP = \emptyset$ then $L = \emptyset$.*

Solution. Towards contradiction assume that $L \neq \emptyset$. Then there is some infinite word $w \in L$. Word w visits infinitely many times accepting state (in automaton for L), so in particular visits some concrete accepting state q infinitely many times. In particular $w = xyz$ such that x goes from initial state to state q and y goes from state q to itself. Then also word xy^ω belongs to L and is ultimately periodic. ■

Problem 3. *Let K and L be regular languages of infinite words. Show that if $L \cap UP = K \cap UP$ then $K = L$.*

Solution. Assume towards contradiction that $K \neq L$. Without loss of generality assume that $K - L \neq \emptyset$ (symmetric case would be $L - K \neq \emptyset$). Clearly $K - L$ is regular. Thus $(K - L) \cap UP \neq \emptyset$, which implies that $K \cap UP \neq L \cap UP$. Contradiction. ■

Problem 4. *Are the following languages regular:*

1. *word v contains arbitrary long infixes in the fixed regular language of finite words L;*

2. *prefix of v belongs infinitely often to the fixed language of finite words $L \subseteq \Sigma^*$ (not necessarily regular).*

Solution. Solutions of the points:

1. NO. Consider $L = ab^*a$. This language contains no ultimately periodic word, so it would have to be empty to be regular.

2. NO. Let us fix some infinite word u, which is not ultimately periodic. Let L be all its prefixes. When prefix of v belongs infinitely often to L if and only if $v = u$. However language $\{u\}$ is not regular. By the way note that the language $\{w\}$ is regular iff w is ultimately periodic.

■

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

Solution. By reading a^k for any $k \in \mathbb{N}$ we cannot get blocked. Therefore word a^ω also cannot get blocked, which means that it is accepted by the considered automaton, contradiction. ■

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

Solution. Assume towards a contradiction that it is accepted by some automaton with n states. Let $w = (ab^n)^\omega$. For any prefix of it of the form $(ab^n)^k$ there should be an accepting state among the last $n+1$ states. Indeed, otherwise a word $(ab^n)^k b^\omega$ would not be accepted. Therefore a run over w visits infinitely many times an accepting state, which means that w is accepted. On the other hand it does not belong to the language, as it has infinitely many letters a. Contradiction. ■

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

Solution. We have an automaton $\mathcal{A}$ with Muller condition, we will be trying to make an automaton $\mathcal{A}'$ with Büchi condition such that $L(\mathcal{A}') = L(\mathcal{A})$. For every $S \in \mathcal{F}$ we will do a separate gadget in automaton $\mathcal{A}'$ such that acceptance in this gadget is if and only if $\inf(\rho) = S$. At the beginning we just make a copy of $\mathcal{A}$, but such that no states are accepting. Beside that for every $S \in \mathcal{F}$ we add a gadget $\mathcal{A}_S$. The idea is that automaton $\mathcal{A}'$ will jump into the gadget $\mathcal{A}_S$ if it wants to choose that $\inf(\rho) = S$ and now is exactly this moment from which on only states from S will occur. Let $S = \{q_1, \ldots, q_k\}$.
Observe now that if $\inf(\rho) = S$ and there are no states outside of S in ρ then states occurring in ρ have also an infinite subsequence of the form $(q_1 q_2 \cdots q_k)^\omega$. Thus we can just investigate whether there exist such a subsequence. Gadget

$A_§$ will be the following. It contains $|S|$ copies of $\mathcal{A}$: $\mathcal{A}_{S,1}, \ldots, \mathcal{A}_{S,|S|}$. The copy $\mathcal{A}_{S,i}$ has only one accepting state: q_i. In the copy $\mathcal{A}_{S,i}$ transitions are like in $\mathcal{A}$ with the only exception that from state q_i we go to the next copy: $\mathcal{A}_{S,(i+1) \mod |S|}$. Now we can easily observe that ρ visits infinitely many times accepting state iff it infinitely many times changes a copy. Therefore it has a subsequence of states of the form $(q_1 q_2 \cdots q_k)^\omega$, so indeed $\inf(\rho) = S$. ∎

Problem 8. *Assume that we have changed the acceptance condition into such which investigates which sets of transitions are visited infinitely often. Does it affect the expressivity of automata? How it is for Büchi acceptance condition? And how for Muller acceptance condition?*

Solution. In both cases (Büchi and Muller) expressivity does not change. Therefore we have to prove four facts: (1) condition with states can be implemented on transitions, (2) condition with transitions can be implemented on states, both points for both Büchi and Muller acceptance conditions.
Let us start

1. We first implement states on transitions for Büchi condition. It is very easy, simply these transitions are final which finish into previously final states.

2. Now we implement transitions on states for Büchi condition. Let language L be accepted by automaton $\mathcal{A}$ with Büchi acceptance condition on transitions. We make an automaton $\mathcal{A}'$, which has two copies of every state in $\mathcal{A}$. To one copy go all the accepting transitions, while to another one go all the non accepting ones. The outgoing transitions are identical in both copies, the same as in $\mathcal{A}$. All the copies with accepting incoming transitions are final, while the other not (some of the final states may not be reachable, but this is not the problem).

3. Now we implement states on transitions for Muller acceptance condition. This is also easy, set of transitions is accepting iff the set of states into which they go is accepting.

4. Now we implement transitions on states for Muller acceptance conditions. Let automaton $\mathcal{A}$ with Muller condition on transitions accept $L = L(\mathcal{A})$. We make $\mathcal{A}'$ as follows. Every state of $\mathcal{A}$ is split into as many copies in $\mathcal{A}'$ as it has incoming transitions in $\mathcal{A}$. The state of $\mathcal{A}'$ is accepting if the incoming transition in $\mathcal{A}$ was accepting.

■

Problem 9. *Show that nonemptiness is decidable for automata with Muller acceptance condition.*

Solution. It is enough to check whether for some $S \in \mathcal{F}$ there exist a run ρ of an automaton in which $\inf(\rho) = S$, where $\inf(\rho)$ is the set of states which occur infinitely often in ρ. We do this separately for every $S \in \mathcal{F}$. We check whether we graph with only states from S is strongly connected and whether some state from S is reachable from some initial state (now in the situation where we have all the states). ■

Problem 10. *Let us define a metric on infinite words: $d(u,v) = \frac{1}{2^{\mathrm{diff}(u,v)}}$, where $\mathrm{diff}(u,v)$ is the smallest index on which u and v differ. Language L jest open (in this metric) if for every $w \in L$ there exist some open ball centered in w which is included in L (so the standard definition). Prove that the following conditions are equivalent for a regular language L:*

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

Solution. We will first show equivalence of (1) and (2), so two implications. First $(1) \Rightarrow (2)$. If L is open then around every $w \in L$ there is a ball L_w with positive radius such that $L_w \subseteq L$. Therefore $L = \bigcup_{w \in L} L_w$. Observe however that $L_w = f(w)\Sigma^\omega$, where $f(w)$ is some finite prefix of w.Thus $L = \bigcup_{w \in L} f(w)\Sigma^\omega$, czyli $L = K\Sigma^\omega$ for $K = \bigcup_{w \in L} f(w)$.

Now (2) $\Rightarrow$ (1). Let $w \in L = K\Sigma^\omega$. Therefore $w = uv$, where $u \in K$. Thus for any $v' \in \Sigma^\omega$ we have $uv' \in L$. Therefore the ball centered in w and radius equal $\frac{1}{2^{|u|+1}}$ is included in L.
Now we will show equivalence of (1) and (3). Implication (3) $\Rightarrow$ (1) works the same as (2) $\Rightarrow$ (1), so we will focus on implication (1) $\Rightarrow$ (3).
Let us consider an automaton $\mathcal{A}_L$ of language L, where $\mathcal{A}_L$ is a deterministic automaton with Muller acceptance condition. Let $Q_U \subseteq Q$ be a subset of states such that $q \in Q_U$ iff $L(q) = \Sigma^\omega$. Let $L' \subseteq \Sigma^*$ be the set of finite words accepted by automaton $\mathcal{A}_L$, where final states are Q_U. We will show that $L = L'\Sigma^\omega$. We have $L \supseteq L'\Sigma^\omega$, because it is easy to find a run in $\mathcal{A}_L$ for every word from $L'\Sigma^\omega$. We simply first go like in automaton for L' and then arbitrarily and this is an accepting run for $\mathcal{A}_L$. On the other hand let us take $w \in L$ and its run. After some its prefix we will already be in the ball L_w, which means that not depending on the suffix everything is accepted, so we are in the state from Q_U. This gives us a division into prefix from L' and the rest. ∎

Problem 11. *Consider a transducer, which defines a function $f : \Sigma^\omega \to \Gamma^\omega$ and metrics on Σ^ω and Γ^ω defined as $d(u, v) = \frac{1}{2^{\text{diff}(u,v)}}$. Show that such an f is continuous.*

Solution. It is easy to do this from definition. Let us take some two words, which are close to each other, so they agree on some prefix, say of length n. Then their images will also agree on the prefix of length n. Therefore by definition: if we want that images agree on prefix of length n it is enough to take arguments which agree on prefix of length n. ∎

Problem 12. *We will look for a candidate for Myhill-Nerode relation for infinite words, i.e. an equivalence relation $\sim_L$ such that $\sim_L$ has finite index iff L is regular. Check whether this fact is true for the following relations*

1. $\sim_L \subseteq \Sigma^* \times \Sigma^*$ *such that* $u \sim_L v$ *iff for all* $w \in \Sigma^\omega$ *it holds* $uw \in L \Leftrightarrow vw \in L$;
2. $\sim_L \subseteq \Sigma^\omega \times \Sigma^\omega$ *such that* $u \sim_L v$ *iff for all* $w \in \Sigma^*$ *it holds* $wu \in L \Leftrightarrow wv \in L$;
3. $\sim_L \subseteq \Sigma^* \times \Sigma^*$ *such that* $u \sim_L v$ *iff for all* $w \in \Sigma^\omega$ *and* $s, t \in \Sigma^*$ *it holds* $uw \in L \Leftrightarrow vw \in L$ *and* $s(ut)^\omega \in L \Leftrightarrow s(vt)^\omega \in L$.

Solution. We will show that none of these relations has finite index iff L is regular. In all cases if L is regular then $\sim_L$ has a finite index. We will first show this. Consider an automaton $\mathcal{A}$ with Büchi condition for L. In $\sim_L$ we just have to remember

1. which states of $\mathcal{A}$ one can reach via a word u;
2. from which states of $\mathcal{A}$ there exists an accepting run via a word u;
3. as in 1. and in 2. and additionally for which pair of states $p, q \in Q(\mathcal{A})$ there exists a run via a word u which goes from p to q and a) has an accepting state b) has no accepting state.

Now we show that there exists nonregular languages L such that $\sim_L$ has finite index.

1. every prefix independent language, for example: language from exercises 1.2 (u contains infinitely many infixes of the form $ab^p a$, where p is prime). For such a language $\sim_L$ has only one equivalence class;
2. also every prefix independent language, as language from 1.2 works. For such a language relation $\sim_L$ has two classes ($wu \in L$ or $wu \notin L$, this does not depend on w);
3. language similar like from exercise 1.4 works (u contains arbitrary long infixes of the form ba^*b). It is not regular. Definitely $uw \in L$ does not depend in u. It is enough to know whether u contains any b (if yes then $s(vt)^\omega \notin L$) and additionally if not whether u is empty (it matters for empty t).

In general there exists no reasonable relation with this property. ∎

2
Infinite duration games

Problem 13. *We say that game is* finite *if its game tree is finite. Prove that every finite game is determined.*

Solution. ■

Problem 14. *Show that a finite reachability game can be solved in polynomial time.*

Solution. ■

Problem 15. *We will now show an example of a game, which is not determined. We will construct this example by a sequence of a few exercises. First consider a following riddle. There are infinitely many players (countable many), every one has a hut: either white or black. Everybody sees the color of everybody else hut, but not of his own. Everybody should say what is the color of his hut, such that only finitely many players will make a mistake. They can fix a strategy before, but they cannot tell anything to each other after they will see the huts. What is the winning strategy?*

Solution. Hint: consider the following relation on infinite 0-1 sequences $w \sim w'$ iff they differ on finitely many places. We can treat a hut configuration as an infinite 0-1 sequence. Relation $\sim$ is an equivalence relation. A strategy is to fix one element in every equivalence class. Everybody can easily recognize what is the equivalence class. Then everybody says the corresponding number of this one distinguished element in the equivalence class. Only finitely many players will make a mistake. ■

Problem 16. *Define a function* xor : $\{0,1\}^\omega \to \{0,1\}$*, called an* infinite xor*, such that changing one bit or an argument changes the result.*

Solution. In every equivalence class C of $\sim$ choose one element v_C and put $\mathrm{xor}(v_C) = 0$. For every $v \in C$ put $\mathrm{xor}(v) = \text{fin-xor}(v \otimes v_C)$, where $\otimes$ is the standard bit xor and fin-xor outputs 0 iff an argument has even number of 1 (argument of fin-xor has to have a finite number of ones). It is easy to verify that xor is indeed an infinite xor. ∎

Problem 17. *Consider the following two player game. 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 have to pick on still existing piece and eat with piece together with everything which is towards the left and bottom from the picked piece (the picked piece is right upper corner of the currently eaten part). The one who has to eat rotten piece loses. Determine who has a winning strategy.*

Solution. The aim of this exercise is to present strategy stealing argument on a simple example. We show that first player has a winning strategy for all n, k (beside the trivial case $n = k = 1$) without finding this strategy. As chocolate game is a finite game we know that it is determined. So for every position one of the players has a winning strategy. Fix $n, k \in \mathbb{N}$ such that $n > 1$ or $k > 1$. Assume towards contradiction that second player has a winning strategy from the initial position. To initial position is losing (as first player will move from it). Consider a move of first player, which eats just one piece, the leftmost and bottommost. The new position has to be winning. Then second player moves, eating some rectangle $n_1 \times k_1$ at the leftmost and bottommost corner. Notice that this is the only move second player can perform from this position. This position has to we losing, as we assumed that second player used a winning strategy. However, this position can be reached from already from the initial position by a single move of first player. This means that actually initial position was winning, as the owner can choose a move, which leads to a losing position. Contradiction. Thus indeed it has to be the first player who has a winning strategy from the initial position. ∎

Problem 18. *Define a non determined game.*

Solution. Hint 1: use an infinite xor. Hint 2: the game is as follows. There are two players, they construct an infinite 0-1 sequence. Player One wins if xor of constructed element is 1, otherwise Zero wins. Zero and One construct this infinite sequence w by delivering in an alternating manner a finite 0-1 sequences and appending it to the currently constructed prefix of w. Assume that Zero starts.
Observe now that this game is not determined. We will show that no player has a winning strategy. Assume first that One has a winning strategy σ. We will show how Zero can sometimes win against this strategy. Consider two plays P_1 and P_2. Let Zero play 0 in play P_1 and the response of One in P_1 be v_0. Then Zero plays $1v_0$ in P_2 and response of One in P_2 is some v_1. Then Zero plays v_1 in P_1 and the response of One in P_1 is some v_2. Then Zero plays v_2 in P_2 and the response of One in P_2 is some v_3. In that way Zero copies responses of One to another play. Then in P_1 the constructed play will be of the form $0v_1v_2v_3\cdots \in \{0,1\}^\omega$ and in P_2 it will be of the form $1v_1v_2v_3\cdots \in \{0,1\}^\omega$. So the different player will win these plays, contradiction with the assumption that σ is a winning strategy for One.
Similarly we can get a contradiction with the assumption that Zero has a winning strategy. Zero starts with some v both in P_1 and in P_2. Then One plays 0 in P_1, the response of Zero is v_1 and One plays $1v_1$ in P_2. Then he copies Zero's responses as before and the results of these plays will be different in P_1 and P_2. This leads to the contradiction.
So indeed this xor-game is not determined. ∎

3
Parity games in quasipolynomial time

Problem 19. *Consider the following variant of the automaton from Lemma 3.2. Only odd numbers are kept in the registers, and the update function is the same as in Lemma 3.2 when reading an odd number. When reading an even number a, the automaton erases all registers store values $< a$. Show that this automaton does not satisfy the properties required in Lemma 3.2.*

Solution. ∎

Problem 20. *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.*

Solution. ∎

Problem 21. *Show that there is no safety automaton with $< n$ states which satisfies the properties required in Lemma 3.2.*

Solution. ∎

Problem 22. *A probabilistic reachability automaton is defined like a finite automaton, except that each transition is assigned a probability (a number in $[0;1]$) such that for every state, the sum 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.*

Solution. ∎

Problem 23.

Solution. ∎

Problem 24. *Show that there is no safety automaton with* $< n$ *states which satisfies the properties required in Lemma 3.2.*

Solution. ∎

4
Distance automata

Problem 25. *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.)*

Solution. ■

Problem 26. *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.*

Solution. ■

5
Tree-walking automata

Problem 27. *Show that deterministic top-down branching tree automata do not recognise the language "at least one leaf has label a", assuming that the alphabet has a binary letter and at least two leaf letters.*

Solution. ■

Problem 28. *Following [20]. Consider a model of tree-walking automata where the automaton sees only the label and not the view (i.e. it does not know if it is in the root, or what is the child number). Show that this model, even in the nondeterministic variant, cannot recognise the language "some leaf has label a".*

Solution. ■

Problem 29. *(Answer unknown) Prove or disprove: for every nondeterministic tree-walking automaton, there is a deterministic bottom-up branching tree automaton that recognises the same language.*

Solution. ■

Problem 30. *Show that for every deterministic top-down tree automaton, there is a deterministic tree-walking automaton that recognises the same language.*

Solution. ■

Problem 31. *Show the following generalisation of the Rotation Lemma: every two homogeneous patterns of same arity are equivalent.*

Solution. Let us use the name *rotation* for the operation which replaces one of the patterns in the Rotation Lemma with the other one. The exercise follows from the following two observations: (a) every homogeneous pattern of arity ≥ 2 is equivalent to one that is built only using the pattern t_2; (b) every two patterns of same arity that are constructed only using t_2 can be transformed into each other using a sequence of rotations. ∎

6
Monadic second-order logic

Problem 32. *Show that the set* $\mathbb{N}^*$ *equipped with the prefix relation has decidable* MSO *theory.*

Solution. ■

7
Treewidth

Problem 33. *Show that the following problem is decidable: given an* MSO *formula* φ *and* $k \in \{1, 2, \dots, \}$*, decide if* φ *is true in some graph of treewidth at most* k*.*

Solution. ∎

Problem 34. *Using the grid theorem (if a class of graphs has unbounded treewidth, then it has square grids of arbitrarily large dimensions as minors), show that if a class* $\mathcal{C}$ *of graphs has unbounded treewidth, then the following problem is undecidable: given an* MSO *formula* φ*, decide if it is true in some graph from* $\mathcal{C}$*.*

Solution. ∎

8
Weighted automata over a field

Problem 35. *Show that emptiness is undecidable already for probabilistic automata, i.e. automata 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 $F \subseteq \{1, \ldots, d\}$. Here emptiness is the question: is there some input word which produces output at least $1/2$?*

Solution. ∎

9
Vector addition systems

Problem 36. *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.

Solution. We prove the equivalences $1 \Leftrightarrow 2$ and $2 \Leftrightarrow 3$. The implications $1 \Rightarrow 2$ and $3 \Rightarrow 2$ are straightforward, so only prove the converses.

- $2 \Rightarrow 1$. Take some infinite sequence of elements in the quasi-order. By the Ramsey Theorem, there is an infinite subsequence where either: (a) elements are strictly decreasing; (b) elements are pairwise incomparable; or (c) elements are (not necessarily strictly) increasing. Item 2 rules out cases (a) and (b), so we are left with case (c).

- 2 $\Rightarrow$ 3. Take some upward closed set U, and consider the minimal elements. Because the quasi-order is well founded (by item 2), every element of U is above some minimal element, and therefore U is the upward closure of its minimal elements. If we take one minimal element for each equivalence class (i.e. equivalence in the sense of both bigger and smaller), then we are left with an antichain, and this antichain is necessarily finite (by item 2).

■

Problem 37. *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.*

Solution. ■

Problem 38. *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 $a \to \varepsilon$.*

Solution. ■

Problem 39. *Define an $\mathbb{Z}$-*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.*

Solution. ■

Problem 40. *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.

Solution. ∎

Problem 41. *Find a vector addition system, say of dimension d, where the reachability relation*

$$\{(x, y) : \textit{there is a run from from } x \textit{ to } y\} \subseteq \mathbb{N}^{2d}$$

is not semilinear. Hint: use states and try to simulate exponentiation.

Solution. ∎

10
Polynomial grammars

Problem 42. *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.*

Solution. ■

11
Parsing in matrix multiplication time

Problem 43. *Show that the operation $M \circ N$ is not associative.*

Solution. ∎

12
Two-way transducers

Problem 44. *Show that deterministic two-way automata (seen as acceptors of words) can be complemented with polynomial blowup.*

Solution. ■

13
Streaming string transducers

14
Learning automata

Bibliography

[1] Dana Angluin. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[2] 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.

[3] Michael Benedikt, Timothy Duff, Aditya Sharad, and James Worrell. Polynomial automata - Zeroness and applications. *LICS*, pages 1–12, 2017.

[4] Hans L Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. *STOC*, pages 226–234, 1993.

[5] Mikolaj Bojanczyk. Star Height via Games. *LICS*, pages 214–219, 2015.

[6] Mikolaj Bojanczyk and Thomas Colcombet. Tree-walking automata cannot be determinized. *Theor. Comput. Sci.*, 350(2-3):164–173, 2006.

[7] Mikolaj Bojanczyk and Thomas Colcombet. Tree-Walking Automata Do Not Recognize All Regular Languages. *SIAM J. Comput.*, 38(2):658–701, 2008.

[8] J Richard Buchi. Weak Second-Order Arithmetic and Finite Automata. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 6(1-6):66–92, 1960.

[9] J Richard Buchi. State-Strategies for Games in F G. *J. Symb. Log.*, 48(04):1171–1198, 1983.

[10] J Richard Buchi and Lawrence H Landweber. Solving Sequential Conditions by Finite-State Strategies. *Transactions of the American Mathematical Society*, 138:295, April 1969.

[11] Cristian S Calude, Sanjay Jain, Bakhadyr Khoussainov, Wei Li, and Frank Stephan. Deciding parity games in quasipolynomial time. *STOC*, pages 252–263, 2017.

[12] Alonzo Church. Logic, Arithmetic, and Automata. pages 21–35, 1962.

[13] Bruno Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Information and Computation*, 85(1):12–75, March 1990.

[14] Bruno Courcelle and Joost Engelfriet. *Graph Structure and Monadic Second-Order Logic*. 2012.

[15] Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Daniel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, July 2015.

[16] Calvin C Elgot. Decision problems of finite automata design and related arithmetics. *Transactions of the American Mathematical Society*, 98(1):21–21, January 1961.

[17] Yuri Gurevich and Leo Harrington. *Trees, automata, and games*. ACM, May 1982.

[18] K Hashiguchi. Limitedness theorem on finite automata with distance functions. *Journal of Computer and System Sciences*, 24(2):233–244, April 1982.

[19] Kosaburo Hashiguchi. Algorithms for Determining Relative Star Height and Star Height. *Inf. Comput.*, 78(2):124–169, 1988.

[20] Tsutomu Kamimura and Giora Slutzki. Parallel and two-way automata on directed ordered acyclic graphs. *Information and Control*, 49(1):10–51, 1981.

[21] Ernst W Mayr. An Algorithm for the General Petri Net Reachability Problem. *SIAM J. Comput.*, 13(3):441–460, 1984.

[22] Robert McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9(5):521–530, 1966.

[23] David E Muller and Paul E Schupp. Alternating automata on infinite trees. *Theoretical Computer Science*, 54(2-3):267–276, 1987.

[24] Anca Muscholl, Mathias Samuelides, and Luc Segoufin. Complementing deterministic tree-walking automata. *Inf. Process. Lett.*, 99(1):33–39, 2006.

[25] Michael O Rabin. Decidability of Second-Order Theories and Automata on Infinite Trees. *Transactions of the American Mathematical Society*, 141:1, July 1969.

[26] Sylvain Schmitz. The complexity of reachability in vector addition systems. *SIGLOG News*, 2016.

[27] 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.

[28] 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.

[29] Claude E Shannon. A mathematical theory of communication, Part I, Part II. *Bell Syst. Tech. J.*, 27:623–656, 1948.

[30] Michael Sipser. Halting Space-Bounded Computations. *Theor. Comput. Sci.*, 10(3):335–338, 1980.

[31] Wolfgang Thomas. Languages, Automata, and Logic. In *Handbook of Formal Languages*, pages 389–455. Springer, Berlin, Heidelberg, Berlin, Heidelberg, 1997.

[32] Boris A Trakthenbrot. Finite automata and monadic second order logic. . *Siberian Mathematical Journal*, 3:103–131, 1962.

[33] Moshe Y Vardi. Logic and Automata - A Match Made in Heaven. *ICALP*, 2719(Chapter 6):64–65, 2003.

[34] Wieslaw Zielonka. Infinite Games on Finitely Coloured Graphs with Applications to Automata on Infinite Trees. *Theor. Comput. Sci.*, 200(1-2):135–183, 1998.