An Automata Toolbox

Version of December 30, 2025

Mikołaj Bojańczyk

Preface

THESE are lecture notes for a course on advanced automata theory, given at the University of Warsaw starting in 2015. The material was chosen to highlight interesting constructions; with a smaller emphasis on the theoretical and bibliographical context.

Mikołaj Bojańczyk

Acknowledgments

I would like to thank Wojciech Czerwiński, who was heavily involved in the first version of the course, especially with the choice of exercises. I would also like to thank several generations of students, for finding mistakes and typos. You are invited to correct any mistakes directly in the github version. If the issue is bigger, and you are not sure if your correction is correct, then let me know directly. At any rate, once you make a change, please add yourself to this list.

- Krzysztof Rogowski
- Arkadiusz Czarkowski

Contents

1
Presburger arithmetic

In this chapter, we show an algorithm that tells us which formulas are true the structure $(\mathbb{N}, +)$ of natural numbers with equality. This structure is called *Presburger arithmetic*, because the original algorithm, which is a quantifier elimination procedure, was proposed by Mojżesz Presburger in 1929 [45]. Another name for the theory is *linear integer arithmetic*, which is popular in the verification community, where it used as part of automatic theorem provers. We are interested in the first-order theory of the natural numbers with addition, i.e. in the formulas that are true in this structure, and are constructed using variables (ranging over natural numbers), addition, quantification (over natural numbers), and the Boolean connectives $\wedge$, $\vee$, $\neg$. Here, we treat addition as a function that takes two arguments, and not as a relation that takes three arguments, although the two views are equivalent.

Example 1. [Zero] The formula

$$\exists z \forall x \quad x + z = x$$

says that addition has a neutral element (zero). Having thus shown how to define zero, we can now start using it in formulas, since each formula that uses zero can be rewritten into an equivalent formula that does not use zero, by replacing each occurrence of zero with its definition. In a similar way, we can use constants for other natural numbers, such as 1 or 2. For example, 1 is the only element such that by adding something to it, we can obtain every nonzero number. □

Example 2. [Linear inequalities] Although we do not have access to multiplication, we can use multiplication by a fixed constant. For example, instead of $3x$ we can write $x + x + x$. We can express any linear inequalities, possibly using negative coefficients, such as

$$3x - 2y + 4z - 7 \geq 0$$

This is done by putting all the negative terms on the other side of the expression, and then expressing the order in terms of addition. □

Example 3. [Divisibility] We can also talk about divisibility by a fixed constant. For example, the formula

$$\exists y \quad x = y + y + y$$

says that x is divisible by 3. □

Decidability. The main result of this chapter is to show that Presburger arithmetic has a decidable theory, i.e. there is an algorithm that inputs a first-order formula, and tells us if it is true in the structure. (The formulas have no free variables, since otherwise they would not have a meaningful truth value without specifying which numbers are used for the free variables.)

Theorem 1.1. *The first-order theory of Presburger arithmetic is decidable.*

We follow the ideas of the original proof, which shows that one can transform every formula into an equivalent one that does not use any quantifiers. If we want to avoid quantifiers, we will need a slightly larger vocabulary. For this purpose, we add some new relations, which can be expressed in terms of addition, but not without using quantifiers.

Definition 1.2. *Define the* extended vocabulary *of Presburger arithmetic to be the following infinite family of relations:*

- *We can write any divisibility constraint, such as*

  $$x \equiv 5 \mod 36$$

  where the remainder of division is computed. There is one such constraint for every remainder and modulus. Note that the numbers 5 *and* 36 *are constants which are hard-coded into the vocabulary, and they cannot be quantified.*

- *We can write any linear inequality such as*

  $$3x - 2y + 4z - 7 \geq 0.$$

  Again, the constants in the inequality are hard-coded and cannot be quantified, since otherwise we would have multiplication.

In the above definition, the divisibility constraints are relations with one argument, while the linear inequalities can take any number of arguments. Thanks to the extended vocabulary, there are more quantifier-free formulas. As it turns out, there are enough quantifier-free formulas to describe all other formulas.

Theorem 1.3 (Quantifier Elimination)**.** *For every formula of Presburger arithmetic, possibly with free variables, one can compute an equivalent formula that is quantifier-free over the extended vocabulary.*

Once we have proved quantifier elimination, decidability of the first-order theory follows immediately. Indeed, if we want to decide if a formula without free variables is true, we simply apply the quantifier elimination procedure to it. Since there are no free variables, the resulting quantifier-free formula can only use trivial linear inequalities like $5 \geq 3$ (which is true) or $3 \geq 5$ (which is false), and its truth can easily be decided. It remains to prove quantifier elimination.

Proof. In the proof, we consider the extended vocabulary without addition as a function symbol, i.e. the vocabulary contains only relations. We use the name *atomic formula* for a relation from the vocabulary applied to some variables. Atomic formulas are either divisibility constraints or linear inequalities.

We prove the theorem by induction on the structure of formulas. In the induction basis, we have atomic formulas, which are already quantifier-free. Since quantifier-free formulas are closed under the Boolean operations, and $\forall$ is complementary to $\exists$, it is enough to show that a single existential quantifier can be eliminated, i.e. if

$$\varphi(\underbrace{x_1, \ldots, x_k}_{\text{call this } \overline{x}}, y)$$

is a quantifier-free formula, then the formula $\exists x\ \varphi(\overline{x}, y)$ can be expressed using a quantifier-free formula that does not mention the existentially quantified variable y. Like any quantifier-free formula, we can rewrite φ into an equivalent formula in disjunctive normal form, i.e.

$$\bigvee_{i \in I} \bigwedge_{j \in J_i} \varphi_{ij}(\overline{x}, y)$$

where each φ_{ij} is an atomic formula or its negation. We also do not need the negations of atomic formulas, since the negation of a linear inequality is also a linear inequality, while the negation of a divisibility condition is a finite disjunction of other divisibility conditions. Furthermore, since disjunction distributes over the existential quantifier, i.e.

$$\exists y \bigvee_{i \in I} \bigwedge_{j \in J_i} \varphi_{ij}(\overline{x}, y) \qquad \Leftrightarrow \qquad \bigvee_{i \in I} \exists y \bigwedge_{j \in J_i} \varphi_{ij}(\overline{x}, y),$$

it is enough to show that an existential quantifier can be removed from a formula that uses only a conjunction of atomic formulas. This is the content of the following lemma, which completes the proof of the theorem.

Lemma 1.4. *Let $\varphi(\overline{x}, y)$ be a conjunction of atomic formulas. Then $\exists y\ \varphi(\overline{x}, y)$ is equivalent to a quantifier-free formula that uses only the variables $\overline{x}$.*

Proof. Some of the atomic formulas in φ are linear inequalities, some are divisibility constraints. Let us first look at the divisibility constraints. Some of these constraints concern the free variables $\overline{x}$. Since they do not depend on the choice of y, we can move them out of the formula, and assume without loss of generality that there are no divisibility constraints in the formula that talk about the free variables $\bar{x}$. Let us now look at the divisibility constraints for the quantified variable y. There might be several such constraints, e.g. we could have

$$(y \equiv 3 \mod 6) \land (y \equiv 2 \mod 15).$$

Using the Chinese remainder theorem, we can replace these constraints with a single constraint

$$y \equiv r \mod a$$

which is equivalent to the conjunction of the original constraints. Also, by replacing the quantified variable y with $y + r$ in all linear inequalities, we can assume without loss of generality that r is zero, i.e. the divisibility constraint is that y is divisible by a. Summing up, we can assume without loss of generality that in the conjunction $\varphi(\bar{x}, y)$ from the assumption of the lemma, there is only one divisibility constraint, and it says that the quantified variable y is divisible by some fixed a.
Let us now consider the atomic formulas that are linear inequalities. In each such linear inequality, we put the variable y on one side of the inequality, and all other variables and the constant on the other side, leading to an inequality which has the same solutions, but has one of the following two forms:

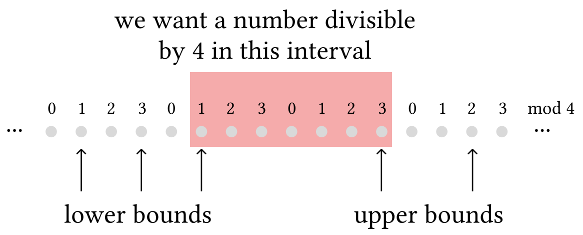
$$\underbrace{by \geq b_0 + b_1x_1 + \cdots + b_nx_n}_{\text{lower bound on } by} \quad \text{or} \quad \underbrace{by \leq b_0 + b_1x_1 + \cdots + b_nx_n}_{\text{upper bound on } by}.$$

We have the two kinds of inequalities because we insist on the coefficient b being positive, hence we can multiply the inequality by -1 if necessary. Furthermore, we want all inequalities to have the same coefficient b next to y, which can be achieved again by multiplying both sides with suitable scaling factors.
Summing up, we can assume that the formula $\varphi(\bar{x}, y)$ is a conjunction of lower and upper bounds on by, for some fixed a that is used in all bounds, and the

requirement that the quantified y to be divisible by b. In this case $\exists y \varphi(\bar{x}, y)$ is equivalent to:

(*) some number divisible by ab lies between the upper and lower bounds.

To complete the proof of the lemma, we will express (*) using a quantifier-free formula. To answer the question in (*), it is enough to know how the lower and upper bounds are ordered, and if there is enough space between them to fit a number divisible by a. Here is a picture of the situation, where $ab = 4$:


we want a number divisible by 4 in this interval
0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 mod 4
...
...
lower bounds
upper bounds

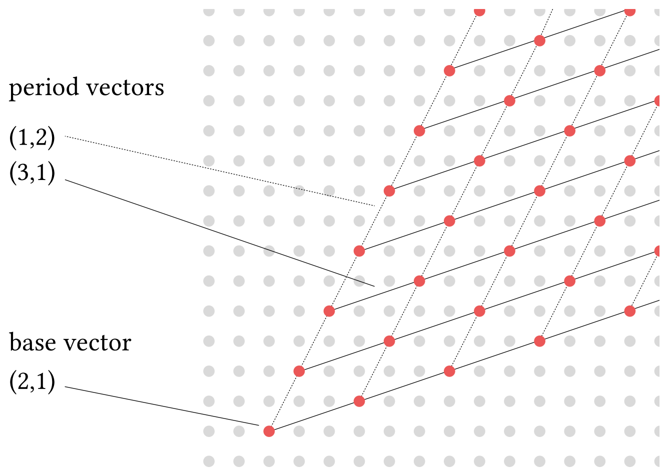
To answer question (*), it is enough to know the remainders of the upper and lower bounds modulo ab, and the differences between the upper and lower bounds up to a threshold $ab - 1$. (This particular threshold is used because $ab - 1$ is the biggest difference between the lower and upper bounds that is possible without necessarily containing a number divisible by ab.) Each such constraint can be expressed in a quantifier-free way, thus completing the proof of the lemma. ■

■

1 Semilinear sets

We now show an alternative, more geometrical, characterization of the sets that can be defined in Presburger arithmetic. This characterization uses the notion of linear and semilinear sets, which are a variant of periodic sets.
The idea behind a linear set is that it is obtained by taking some fixed base vector in $\mathbb{N}^d$, and adding any number of copies of certain period vectors. For

example, consider dimension $d = 2$, and a base vector of (2,1). To this base vector we add any number of copies of the periods (1,2) and (3,1), resulting in the red points of the following picture:



These ideas are formalized in the following definition.

Definition 1.5 (Semilinear sets). *A* linear *subset of $\mathbb{N}^d$ is any set of the form*

$$\{\bar{a} + \beta_1\bar{b} + \cdots + \beta_k\bar{b} \mid \beta_1, \ldots, \beta_k \in \mathbb{N} \quad \},$$

for some choice of base vector $\bar{a} \in \mathbb{N}^d$ and period vectors $\bar{b}_1, \ldots, \bar{b}_k \in \mathbb{N}^d$. A semilinear *set is any finite union of linear sets (in the same dimension).*

The following theorem tells us that the semilinear sets are exactly the sets that can be defined using first-order formulas in Presburger arithmetic. When defining a subset of $\mathbb{N}^d$, we use a formula that has d free variables.

Theorem 1.6. *A subset of $\mathbb{N}^d$ is semilinear iff it is definable in Presburger arithmetic.*

Proof. Clearly every semilinear set is definable in Presburger arithmetic, by simply formalizing the defining condition in logic. The interesting part is that all sets definable in Presburger arithmetic are semilinear. To prove this, we will use the quantifier elimination result from Theorem 1.3, and the following lemma about solutions of systems of linear equations.

Lemma 1.7. *Let A be a $d \times n$ matrix with integer entries. Then for every $\overline{b} \in \mathbb{N}^n$, the following set is semilinear*

$$\{\overline{x} \in \mathbb{N}^d \mid A\overline{x} = \overline{b}\}$$

Proof. Let us begin with the special case when the system is *homogeneous*, which means that $\overline{b}$ is the zero vector, i.e. it has the form

$$A\overline{x} = \overline{0}. \tag{1.1}$$

Consider the set of minimal solutions to such an system, i.e. the set of solutions $\overline{x}$ which are minimal coordinatewise. By Dickson's Lemma, see Claim 5.4, the set of minimal solutions, call it S, is finite. We now claim that every other solution can be obtained by coordinatewise addition of minimal solutions, and therefore the set of solutions forms a linear set, where the period vectors are the minimal solutions. This is an immediate consequence of the definition of minimality: a non-minimal solution can be obtained by taking some minimal solution, and adding some vector to it. The added vector is also a solution to the homogeneous system, and therefore an inductive argument can be used to decompose it into a sum of minimal solutions.
Consider now the general case of the lemma, where $\overline{b}$ is not necessarily the zero vector. Again, consider the set of minimal solutions, which is finite. Every other solution is obtained by taking a minimal solution, and adding to it some solution of the homogeneous version of the system as in (1.1). Since the homogeneous solutions form a linear set, we get the result. ∎

We now complete the proof of the theorem. Take any formula of Presburger arithmetic. By the quantifier elimination result from Theorem 1.3, we can assume that this formula is quantifier-free. We can also assume that the

formula is in disjunctive normal form, i.e. a disjunction of conjunctions of atomic formulas. Since semilinear sets are closed under union by definition, it is enough to consider the case of a quantifier-free formula $\varphi(x_1, \dots, x_n)$ that is a conjunction of atomic formulas. We will reduce this formula to a system of linear inequalities, as in the above lemma, possibly by introducing new variables.
We first want to get rid of the divisibility constraints, such as

$$x \equiv 5 \mod 36.$$

To eliminate this constraint, we introduce a new variable y, and a linear equality

$$x = 36y + 5.$$

Similarly, we can replace inequalities using equalities. For example, a linear inequality

$$3x_1 - 2x_2 + 4x_3 - 7 \geq 0$$

can be eliminated by adding a new variable y, and writing the linear equality

$$3x_1 - 2x_2 + 4x_3 - 7 = u.$$

This works, because all variables range over natural numbers. After this elimination, we have created a new formula

$$\psi(x_1, \dots, x_n, y_1, \dots, y_m)$$

which is a conjunction of linear equalities (and is therefore subject to Lemma 1.7), such that the original formula is equivalent to

$$\exists y_1 \cdots \exists y_m \quad \psi(x_1, \dots, x_n, y_1, \dots, y_m).$$

By Lemma 1.7, the formula ψ defines a semilinear set, in dimension $d = n + m$. We now want to project this set onto the first n coordinates. This is very straightforward to do, since semilinear sets are easily seen to be closed under projection – simply eliminate the unused coordinates from all vectors. ∎

Problem 1. Show that there is a formula $\varphi_n(x)$ of Presburger arithmetic, using addition only, which has size polynomial in n, but is true for only one x, namely 2^n.

Problem 2. Consider a regular language L over an alphabet with k letters. Define its *Parikh image* to be the set

$$\{(n_1, \ldots, n_k) \in \mathbb{N}^k \mid \begin{array}{l}\text{there exists a word } w \in L \text{ such that } n_i \text{ is the} \\ \text{number of occurrences of the } i\text{-th letter in } w\end{array}\}.$$

Show that for every regular language, its Parikh image is semilinear.

Problem 3. Show that every formula of Presburger arithmetic $\varphi(x)$ defines an *ultimately periodic set*, i.e. there exists $p > 0$ such that

$$\varphi(x) \quad \Leftrightarrow \quad \varphi(x+p) \qquad \text{for all sufficiently large } x.$$

Problem 4. Show there is no formula of Presburger arithmetic $\varphi(x, y, z)$ that defines multiplication, i.e. it is equivalent to $x \cdot y = z$.

Problem 5. Show there is no formula of Presburger arithmetic $\varphi(x, y)$ that defines divisibility, i.e. it is equivalent to $x|y$.

Problem 6. (*) Show that if we extend Presburger arithmetic with a binary relation $x|y$ for divisibility, then we can define multiplication.

Problem 7. Show that there is no an algorithm deciding the theory of $(\mathbb{N}, +, \times)$. (This theory is simply called *arithmetic*). Hint: show that if arithmetic would be decidable, then one could decide the theory of strings with concatenation $(\{0,1\}^*, \cdot)$, and the latter theory is undecidable.

2 First-order theory of the reals

Consider the real numbers, equipped with binary functions for addition, subtraction, multiplication, constants for zero and one, and a binary relation for the ordering:

$$(\mathbb{R}, +, -, \times, 0, 1, <).$$

The goal of this section is to prove a Theorem of Alfred Tarski, which says that the first-order theory of this structure is decidable, i.e. there is an algorithm which inputs a sentence of first-order logic like

$$\forall x \exists y \; y \times y + x = 0$$

and says if the sentence is true in the real numbers. Remarkably, this theorem represents a nontrivial algorithm dealing with first-order logic that was found before the notions of "first-order logic" and "algorithm" were well defined in the modern sense. Although apparently proved earlier, the result was published only after the war [56].

There is some freedom in the choice of vocabulary. For example we could add division because it is definable in first order logic by

$$x/y = z \qquad \overset{\text{def}}{=} \qquad z \times y = x.$$

Conversely, we could remove subtraction, because it is definable in terms of addition, or we could remove the ordering, because it is defined by

$$x \leq y \qquad \overset{\text{def}}{=} \qquad \exists z \; x + z \times z = y,$$

thus

$$x < y \qquad \overset{\text{def}}{=} \qquad x \neq y \land \exists z \; x + z \times z = y.$$

Also, we could remove the constants for 0 and 1, because 0 is the unit for addition and 1 is the unit for multiplication. Summing up, as long as we care about first-order logic, the vocabulary could be reduced to have only $+$ and $\times$. Nevertheless, some of the above definitions are not quantifier-free, and we will care about quantifier-free formulas, as expressed in the following theorem.

Theorem 2.1. *For every formula of first-order logic over*

$$(\mathbb{R}, +, -, \times, 0, 1, <)$$

possibly with free variables, one can compute an equivalent (over the real numbers) formula that is quantifier-free. In particular, one can decide if a sentence, i.e. a formula without free variables, is true over the real numbers.

Before proving the theorem, we discuss its relation to decidability of first-order logic for other fields and rings, such as integers or rationals.

Example 4. The first-order theory of the integers with addition, subtraction and multiplication

$$(\mathbb{Z}, +, -, \times, 0, 1, <)$$

is undecidable. This is the original undecidability result, which was proved by Church [17, Corollary 2] and Turing [60, Section 11]. It follows that there is no first-order formula $\varphi(x)$ which defines the integers inside the real numbers. (We will see another reason why the integers cannot be defined later on.) In contrast, every single integer can be defined.

A celebrated theorem of Julia Robinson [48, Theorem 3.1] says that there is a first-order formula that defines the integers inside the rational numbers, and therefore also the rational numbers cannot be defined inside the real numbers. □

Example 5. A complex number can be coded as two real numbers, its real and imaginary parts. Addition, subtraction and multiplication of complex numbers can be reduced to analogous operations on the real and imaginary parts. It follows that the first-order theory of the complex numbers

$$(\mathbb{C}, +, -, \times, 0, 1)$$

is decidable (note that order is not mentioned above, since it is not defined on the complex numbers). □

The rest of this chapter is devoted to proving Theorem 2.1. The second part of the theorem (deciding which sentences are true) is an immediate consequence of the first part (effective quantifier elimination). Indeed, if we want to know if a sentence is true in the real numbers, we eliminate all quantifiers using the first part of the theorem, arriving at a formula which is a Boolean combination of inequalities that involve values which are obtained form the constants 0 and 1 by applying the operations $+, \times, -$. Here is an example of such a formula:

$$((1+1) \times (1+1+0) - 0 \times (1+0) > 0) \vee (1 = 0).$$

A straightforward evaluation leads to the desired true/false answer. It is therefore enough to prove the effective quantifier elimination. Here, it is enough to show that a single existential quantifier can be eliminated, since multiple quantifiers can then be eliminated one by one, starting with the innermost quantifiers (and using closure of quantifier-free formulas under Boolean operations).

Therefore, the essence of Theorem 2.1 is showing that for every formula

$$\exists x \qquad \underbrace{\varphi(x, y_1, \ldots, y_n)}_{\text{quantifier-free formula using } +, \times, -, 0, 1, <}$$

there exists, and can be computed, an equivalent one that is quantifier-free over the same vocabulary, and which talks only about the free variables $y_1, \ldots, y_n$.

Example 6. Consider the formula

$$\exists x \; y_1 x^2 + y_2 x + y_3 = 0.$$

This formula says that the quadratic polynomial with coefficients y_1, y_2, y_3 has a real root. As we all remember from high school this is the same as saying that the discriminant Δ is non-negative, i.e.

$$y_2^2 - 4y_1 y_3 \geq 0.$$

The new condition is quantifier-free, as required. When proving the general result, we will also obtain similar quantifier-free formulas for polynomials of higher degree. The reader might feel uneasy about this, because we know that there is no general "formula" for roots of polynomials of degree five or higher. However, note that we do not need to express the roots themselves, but only the existence of roots, which is a much easier task, and can be done in a quantifier-free way. $\square$

We first observe that the formula φ can be without loss of generality assumed to be a Boolean combination of formulas of the form

$$p(x, y_1, \ldots, y_n) > 0$$

where each p is a polynomial with integer coefficients and variables $x, y_1, \ldots, y_n$. So the goal is to understand the behaviour of the polynomials p, once the arguments $y_1, \ldots, y_n$ have been fixed, as a function of the quantified parameter x. To understand this behaviour, we will use basic operations from calculus and algebra, like differentiation and dividing polynomials with remainders, and then observe that the effect of these operations can be formalised using quantifier-free formulas.

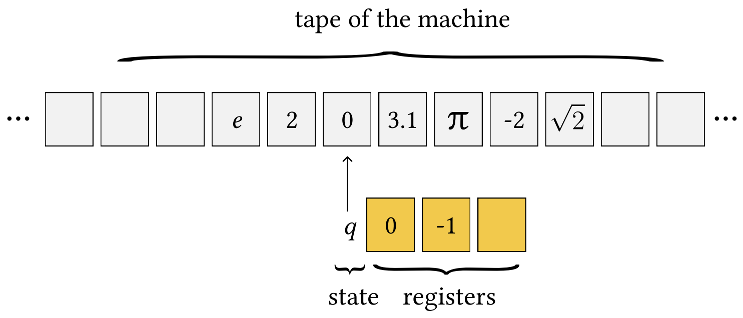
Instead of working directly with quantifier-free formulas, we introduce an intermediate computation model for the reals and show that (a) computation in this model can be simulated using quantifier-free formulas; and (b) quantifier elimination can be done using the computation model. The point of using the computation model is that when proving (b), we can appeal to algorithmic intuitions, like loops and conditionals, which are more cumbersome to formalise when working directly with quantifier-free formulas. The results (a) and (b) are presented in Sections 2.1 and 2.2 below.

2.1 *Computation on the reals*

Consider the following variant of a Turing machine, which we call a BSS *machine*, standing for Blum Shub and Smale, who introduced the model. The purpose of a BSS machine is to compute a partial function of type $\mathbb{R}^* \to \mathbb{R}^*$, i.e. a partial function that inputs and output lists of real numbers. The general idea is that the machine operates on reals using the arithmetic operations

$$x + y \quad x - y \quad x \times y \quad x/y,$$

and it is allowed to compare numbers to zero (are they zero, or positive, or negative?). Observe that we allow division, even though our intended application for the machines is to prove quantifier elimination without division. The machine has a single tape, infinite in both directions (i.e. indexed by integers), whose cells store real numbers plus a fixed number of registers that also store real numbers. Cells of the tape and registers can be undefined. Here is a picture of a configuration of the machine:



At any given moment, the machine is in a state from a finite set of control states, and has a single head which points to one of the cells on the tape. In the initial configuration, the tape stores the input of the function (if the input has n letters, then cells $\{1, \ldots, n\}$ store these letters, and the remaining cells are undefined), the registers are all undefined, the state is a designated initial state and the head points to the first cell of the tape. Based on the answers to the following questions (the second question is actually several questions, one for each of the finitely many registers):

- what the current state?
- what are the signs of the registers (negative, zero, positive, undefined)?

the transition function of the machine indicates deterministically a new state and an operation from the following set:

$$\text{accept} \qquad \text{move the head by } i \in \{-1, 1\} \qquad r := 1 \qquad r := \underbrace{s \ \mathsf{op}\ t}_{\substack{\text{op is one of the four}\\ \text{arithmetic operations}}}$$

where r, s, t range over registers or the contents of the cell under the head. For example, the effect of a transition can be that the contents of registers r, s are multiplied and the result is stored in the cell under the head. If the computation never performs the accept operation then the output of the computed function is undefined. Otherwise, the output of the computed function is defined to be the contents of the defined cells, read from left to right. If a division by zero is performed, then the computation is not aborted, but the cell/register r storing the result becomes undefined. We use this feature to erase cells, which is necessary for functions where the output is shorter than the input. The running time of a computation is defined to be the number of transitions that it uses. As defined above, a BSS machine computes a partial function. We can also view BSS machines as computing languages, i.e. yes/no properties of tuples of reals, by assuming that the answer is "yes" if the function is defined and "no" otherwise. The following lemma shows that for languages of bounded dimension and running computation time, the BSS model can only compute quantifier-free properties.

Lemma 2.2. *If $S \subseteq \mathbb{R}^n$ is computed by a* BSS *machine which uses at most k computation steps in every accepting run, then S is definable by a quantifier-free formula using $+, -, \times, 0, 1, <$. Furthermore, this formula can be computed given the machine and k.*

Proof. Observe that the machine in the assumption of the lemma can use division, while the quantifier-free formula in the conclusion does not use division. By induction on the length of the computation, we can show that the control state, head position, and signs of the registers can be determined by asking quantifier-free queries to the input. In the induction step, we observe that, assuming that the history (sequence of transitions) of the computation is known, then the contents of each register/cell can be described by a term that uses the n input numbers, integer constants, and arithmetic operations (addition, subtraction, multiplication and addition). Furthermore, if t is such a term, then a straightforward induction on the size of t shows that the truth sign (negative, zero, positive) of t can be expressed using a quantifier-free formula as in the statement of the lemma. (Note that the term t is allowed to use division, but the quantifier-free formula is not.) For example,

$$\frac{x * y}{x - y + 1} > 0$$

is equivalent to the quantifier-free formula

$$\underbrace{(x \times y > 0) \wedge (x - y + 1 > 0)}_{\text{enumerator and denominator are positive}} \quad \vee \quad \underbrace{(x \times y > 0) \wedge (x - y + 1 > 0)}_{\text{enumerator and denominator are negative}}$$

■

2.2 *Quantifier elimination*

In Section 2.1 above we have shown that any bounded time procedure in the BSS model can be simulated using quantifier-free formulas. Therefore, to complete the proof of Theorem 2.1, it is enough to show that the truth values of formulas with one quantifier can be decided using the BSS model. In principle, the BSS model manipulates sequences of reals, but we will also use it to manipulate more structured entities, such as polynomials or finite sets of polynomials, implicitly using straightforward encodings as tuples of real numbers.

Lemma 2.3. *For every quantifier-free formula $\varphi(x, y_1, \ldots, y_n)$ there is a BSS machine which computes the following:*

- **Input.** *Real numbers $a_1, \ldots, a_n$.*
- **Output.** *Is $\exists x \varphi(x, a_1, \ldots, a_n)$ true in the real numbers?*

Furthermore, the running time of the algorithm is bounded by a number k that depends only on (and can be computed from) the formula φ, and does not depend on the numbers $a_1, \ldots, a_n$.

Together with Lemmas 2.2, the above lemma completes the proof of Theorem 2.1, and therefore the rest of this chapter is devoted to proving the above lemma. Let φ and $a_1, \ldots, a_n$ be as in the lemma. Without loss of generality we assume that φ is a Boolean combination of formulas of the form

$$p(x, y_1, \ldots, y_n) > 0 \tag{2.1}$$

where each p is a polynomial with integer coefficients and $n+1$ variables. Given a tuple of real numbers $a_1, \ldots, a_n$ as in the input of the problem from Lemma 2.3, define P to be the finite set

$$\{p(x, a_1, \ldots, a_n) : p(x, y_1, \ldots, y_n) > 0 \text{ appears in } \varphi\}. \tag{2.2}$$

The set P contains polynomials with one variable x and real coefficients. The coefficients depend on the parameters $a_1, \ldots, a_n$, in a way that can be computed in the BSS model. Define the *sign* of a real number to be one of the three results (negative, zero, or positive) of comparing the number to 0. The truth value of the comparison in (2.1) depends only on the sign of the left side, and therefore in order to determine if a real number a satisfies $\varphi(a, a_1, \ldots, a_n)$, it is enough to look at the sign of $p(a)$ for all polynomials $p \in P$. This observation motivates the following definition. In the following definition, $\mathbb{R}[x]$ stands for polynomials with real coefficients and one variable x, and a nonzero polynomial is one with at least one nonzero coefficient.

Definition 2.4 (Sign table). *Let $P \subseteq \mathbb{R}[x]$ be a finite set of nonzero polynomials, and let*

$$r_1 < r_2 < \cdots < r_n$$

be all the real numbers that are a root of at least one polynomial in P. (There are finitely many such numbers, because a nonzero polynomial has finitely many roots.) Define the sign table *of P to be the following information:*

- *the number n;*
- *for each $p \in P$ and $i \in \{0, \ldots, n+1\}$, the sign of $p(r_i)$, where the sign in*

  $$r_0 \overset{def}{=} -\infty \qquad r_{n+1} \overset{def}{=} +\infty$$

  is defined by taking the limit in the natural way.
- *for each $p \in P$ and $i \in \{0, \ldots, n\}$, what is the sign of p on the interval $(r_i; r_{i+1})$.*

From the discussion before the above definition, it follows that the truth of

$$\exists x \varphi(x, a_1, \ldots, a_n)$$

can be (effectively) determined by looking at the sign table of the nonzero polynomials in P. Therefore, Lemma 2.2 and thus also the Tarski Theorem will follow from the following lemma.

Lemma 2.5. *The sign table of a finite set $P \subseteq \mathbb{R}[x]$ of nonzero polynomials can be computed by a BSS machine in time bounded by a function of the sum of degrees of P.*

Proof. The proof is essentially the observation that the usual method of plotting polynomials that is taught in high school can be formalised in the BSS model[1]. We begin by observing that the BSS model can run the Euclidean algorithm.

[1]Tarski taught mathematics in a Warsaw high school for girls. A lesser man would complain about having to teach calculus, Tarski proved that it could be automated.

Claim 2.6. *There is a* BSS *algorithm which inputs polynomials*

$$p, q \in \mathbb{R}[x] \qquad \textit{with} \qquad \textit{degree of } q \leq \textit{degree of } p$$

and outputs a polynomial $r \in \mathbb{R}[x]$ such that

$$\textit{degree of } r < \textit{degree of } q \qquad \textit{and} \qquad p = q \times s + r \quad \textit{for some } s \in \mathbb{R}[x].$$

Proof. Define a finite sequence of polynomials

$$p = p_0, p_1, \ldots, p_n = r$$

subject to the following invariant: (a) the degrees in the sequence strictly decrease; and (b) q divides $p - p_n$ for every n. We begin with $p_0 = p$. Suppose that p_i has been defined. If the degree of p_i is strcitly smaller than the degree of q, then we output $r = p_i$; otherwise we define p_{i+1} to be the difference

$$p_i - \frac{\text{leading coefficient of } p_i}{\text{leading coefficient of } q} x^{(\text{degree of } p_i)\text{- (degree of } q)} \cdot q$$

which preserves the invariant. This procedure can clearly be implemented in the BSS model. ∎

Apart from computing remainders using the Euclidean algorithm, we also use derivatives of polynomials (in the usual sense of calculus). The derivative of a polynomial $p \in \mathbb{R}[x]$ can clearly be computed in the BSS model. By repeatedly applying derivation and the Euclidean algorithm, we can extend any finite set of polynomials to a bigger one that is saturated in the following sense: it is closed under derivations and applying the Euclidean algorithm. Since computing a sign table can only get harder after adding polynomials, in order to show the lemma, it suffices to show that sign tables can be computed for sets P that are saturated.
Suppose then that P is a finite saturated set of polynomials. Take some $p \in P$ of maximal degree. The set $P - \{p\}$ is also saturated, because p has maximal degree, and both derivation and the Euclidean algorithm decrease degrees. Therefore, we can use induction to compute the sign table of $P - \{p\}$. Let

$$r_0 = -\infty \qquad r_1 < \cdots < r_n \qquad r_{n+1} = +\infty$$

be the numbers from the definition of a sign table, as applied to $P - \{p\}$. We do not know the exact values of these numbers, but we do know n and we also know how the signs of the polynomials from $P - \{p\}$ behave in the points r_i and the intervals that separate them. Our goal is to enrich this information to account for the polynomial p.

Claim 2.7. *For each $i \in \{0, 1, \dots, n+1\}$ we can compute the sign of $p(r_i)$.*

Proof. For $i = 0$ and $i = n+1$ we simply look at the sign of the leading coefficient of p, and the parity of its degree. This information determines the sign of p in $\pm\infty$. We are left with the case of $i \in \{1, \dots, n\}$. By definition of r_i, there must be some polynomial $q \in P - \{p\}$ which has a root in r_i, and we can use the sign table to find that polynomial. Since P is closed under applying the Euclidean algorithm, there must be some $r \in P - \{p\}$ such that

$$p = q \times s + r \qquad \text{for some } s \in \mathbb{R}[x].$$

Since r_i is a root of q, the sign of $p(r_i)$ is the same as the sign of $r(r_i)$, and the latter sign is stored in the sign table. ∎

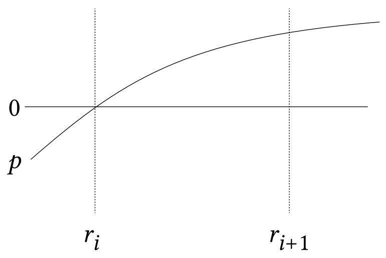
We now proceed to investigate the behaviour of p in the intervals separating the roots r_i. The important observation is that p does not have any turning points in these intervals (a turning point is one where the polynomial changes behaviour between increasing/decreasing). The reason is that a turning point is also a root of the derivative, and all such roots are in the points $r_1, \dots, r_n$ because the derivative of p is belongs to $P - \{p\}$. This observation yields the following claim:

Claim 2.8. *For every $i \in \{0, \dots, n\}$, p has at most one root in the interval $[r_i; r_{i+1}]$.*

Proof. Otherwise there would be a turning point between r_i and r_{i+1}. ∎

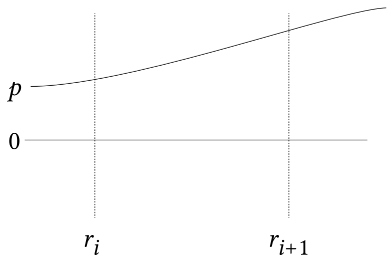
Using the above claim, we can describe the behaviour of p in an interval of the form $(r_i; r_{i+1})$. This is done using the following case analysis, which can readily be formalised in the BSS model.

- The sign of p is zero on one of the endpoints of the interval $[r_i; r_{i+1}]$. By Claim 2.8, at most one of the endpoints is zero, and there are no roots inside the interval, as in the following picture



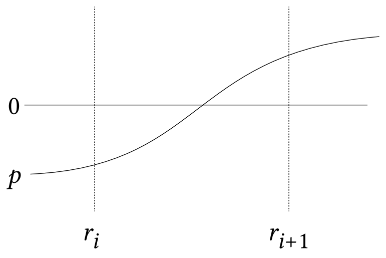
  It follows that the sign inside this interval is the same as for the nonzero endpoint.

- The signs of p are the same on both endpoints of the interval $[r_i; r_{i+1}]$, like this:



  If p would have a root inside the interval, then it would also have a turning point in this interval, and this cannot happen. Therefore, p has no roots in this interval, and its sign in the interval is the same as in either one of its endpoints.

- The signs of $p(r_i)$ and $p(r_{i+1})$ are nonzero and different, like this:



In this case p has exactly one root in the interval, which splits the interval into two parts; on the left part the sign of p is as in $p(r_i)$ and on right part the sign is as in $p(r_{i+1})$.

Doing the above case analysis for all $i \in \{0, \ldots, n\}$, we fill in the sign table for P. This completes the proof of Lemma 2.5, and therefore also of the Tarski Theorem. ■

Problem 8. Show that adding the function $\sin(x)$ to the real numbers yields an undecidable theory.

Problem 9. Show that the following structure has a decidable first-order theory: the universe consists of subsets of the Euclidean plane $\mathbb{R}^2$ that are points, lines or circles, and there is a binary predicate for inclusion.

Problem 10. Show that every $X \subseteq \mathbb{R}$ definable by a first-order formula with one free variable is a finite union of points and open intervals.

3
Zero-one laws

Suppose that we want to randomly select a graph with n vertices. One way to do this is as follows: independently for every pair of vertices, flip a coin to decide if there is an edge or not. We are working with undirected graphs without loops, and therefore if the graph has n vertices, then we need to flip the coin $\binom{n}{2}$ times, once for each possible edge. For example, the probability that the graph is a clique is

$$\frac{1}{2^{\binom{n}{2}}}$$

because there are $\binom{n}{2}$ edges to choose randomly, and all of them need to be selected. We will be interested in the probability that a graph has some property, under this particular distribution.[1]

Example 7. [Connectivity] Let us show that the probability that a graph is connected tends to one, as n tends to infinity. Indeed, for each pair of vertices v and w, the probability that they are connected using some fixed third vertex u is equal to $1/4$, since we need both of the edges vu and wu. However, if we want to avoid such a connection for all possible choices of u, this will happen with probability at most $(3/4)^n$, which is exponentially small in n. Therefore,

[1]There are other distributions, for example the probability of having an edge could be $1/n$. For example, we could have $p = 1/n$. Such distributions on graphs, with various edge probabilities like $1/2$, $1/n$ or $1/n^2$, are known as Erdös-Rényi random graphs [28]. In this chapter, we use $p = 1/2$.

the probability of two vertices being disconnected is exponentially small, and since a pair can be chosen in quadratically many ways, there is also an exponentially small probability that the graph is not connected. □

Example 8. [Parity] The probability that there is an even number of edges is exactly $1/2$, independently of n. This is because the absence or presence of the last edge (in some predetermined order of edges) determines the parity. □

As the two examples show, the probability of a graph having some property can have some limit, such as zero or one half. There are also properties that do not have a limit, e.g. the property "the number of vertices is even" (which does not depend on the choice of edges), oscillates between zero and one without any convergence. The goal of this chapter is to show that for properties which can be defined in first-order logic, there will be a limit, and this limit will be zero or one.

Let us begin by explaining first-order logic on graphs. The idea is to use a formula that is built using the quantifiers $\forall$ and $\exists$, the logical connectives $\land$, $\lor$, $\neg$, and a binary relation for the edges. Here are some examples of first-order formulas, and the corresponding limiting probabilities.

Example 9. [Apex] The formula

$$\exists x\ \forall y \qquad x \neq y \ \Rightarrow\ \text{edge}(x,y)$$

says that the graph has an apex vertex, i.e. some vertex x that is connected to every other vertex y. For any given vertex, the probability that it is an apex is equal to $1/2^{n-1}$. Even if we sum this probability across all n possible choices of the apex, we will still get a probability that tends to zero with n. Therefore, the limiting probability of having an apex vertex is zero. □

Example 10. [Triangle] The formula

$$\exists x\ \exists y\ \exists z \qquad \text{edge}(x,y) \land \text{edge}(y,z) \land \text{edge}(z,x)$$

says that the graph contains a triangle as an induced subgraph. If we fix a particular triple of vertices, then the probability that this triple is a triangle will be $1/8$. However, if we look at $n/3$ disjoint triples, then the probability that none of them is a triangle will be exponentially small. Therefore, the limiting probability of having a triangle is one. □

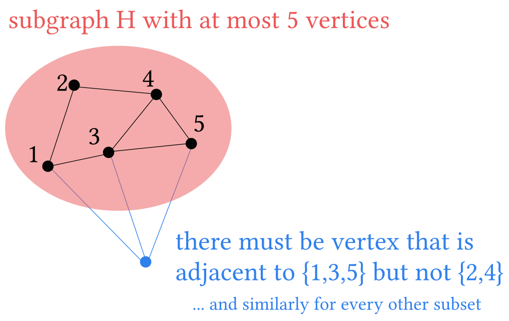
As in the above examples, we define the *limiting probability* of a graph property to be the limit, with $n \to \infty$, of the probability that a graph with n vertices satisfies the property. The main result of this chapter is the following theorem, which shows that the limiting probability exists and is necessarily zero or one for properties that can be defined in first-order logic.

Theorem 3.1. *Let φ be a first-order formula that defines a property of graphs, then its limiting probability exists and is equal to zero or one.*

Proof. The proof will be based on a certain property of graphs, which we call the *extension property*. This property formalizes the idea that the graph contains many different induced subgraphs.

Definition 3.2 (Extension property)**.** *Let $k \in \{0, 1, \ldots\}$. A graph G has the* k-extension property *if for every induced subgraph $H \subseteq G$ with at most k vertices, and every partition of the vertices of H into two parts V_1 and V_2, there is some vertex in $G - H$ that is adjacent to all vertices in V_2 and non-adjacent to all vertices in V_2.*

Here is a picture of the 5-extension property:


subgraph H with at most 5 vertices
1
2
3
4
5
there must be vertex that is adjacent to {1,3,5} but not {2,4}
... and similarly for every other subset

The first observation is that for every k, sufficiently large random graphs have the k-extension property with high probability.

Lemma 3.3. *For every fixed k, the probability that a graph with n vertices has the k-extension property tends to one as n tends to infinity.*

Proof. Let $n \in \mathbb{N}$. Consider the probability that some particular k-tuple vertices in the graph with n vertices is a violation of the k-extension property. This probability is exponentially small in n, for the same reasons as in Example 7. Since the number of k-tuples is polynomial in n, once k has been fixed, it follows that the property that some k-tuple is a violation must tend to zero. ■

We now show that for every formula of first-order logic, there is some k such that all graphs with the k-extension property agree on this formula.

Lemma 3.4. *Let φ be a formula of first-order logic. Then there is some k such that either all graphs with the k-extension property satisfy φ, or all graphs with the k-extension property do not satisfy φ.*

Proof. The formula in the statement of the lemma does not have free variables, since otherwise it would not be meaningful to say that a graph satisfies or does not satisfy it, without specifying the values of the free variables. However, in the proof we will discuss a slightly stronger statement that involves free variables. To evaluate a formula with ℓ free variables, we need a graph together with a list of ℓ distinguished vertices. We use the name *ℓ-pointed graph* for such an object. The lemma follows immediately from the following claim in the case of $\ell = 0$.

(*) Let φ be a formula that has ℓ free variables. There is some k with the following property: if two ℓ-pointed satisfy the same quantifier-free formulas, and the underlying graphs both have the k-extension property, then φ is true in both or none of them.

The claim is proved by induction on the structure of the formula. The only interesting case is the induction step for an existential quantifier (or dually, a universal quantifier), where we use the extension property to find a corresponding vertex in the other graph. ■

Combining the above two lemmas, we get the theorem. Indeed, we take some formula, and apply Lemma 3.4 to find some k. Either the formula holds in all graphs with the k-extension property, or it fails in all of these graphs. By Lemma 3.3, the probability that a random graph has the k-extension property tends to one, so the limit in the theorem exists and is either zero or one. ■

3.1 *The infinite random graph*

In this section, we discuss a variant of random graphs, in which the set of vertices is the infinite set

$$\omega = \{1, 2, \ldots\}.$$

We use the same distribution as previously, i.e. for each (unordered) pair of vertices, we flip a coin to decide if there is an edge or not. The main result of this section is that, rather remarkably, we get the same graph with probability one.

Theorem 3.5. *There is some countably infinite graph G, such that if we randomly select a graph with vertices ω, then with probability one, the selected graph is isomorphic to G.*

Thanks to the above theorem, it makes sense to talk about *the infinite random graph*, since it is unique up to probability one.

Proof of Theorem 3.5. The proof is also based on the extension property.

Lemma 3.6. *For every k, with probability one a randomly selected graph on vertices ω has the k-extension property.*

Proof. Same proof as previously: the probability is zero that any particular subgraph on k vertices is a violation of the k-extension property. Since there are countably many such subgraphs, the probability is still zero that at least one of them is a violation. ■

Since an intersection of countably many events with probability one must also have probability one, it follows that with probability one, a randomly selected graph on vertices ω will have the k-extension property for every k. To complete the proof of the theorem, we will show that any two such graphs will necessarily be isomorphic.

Lemma 3.7. *If two graphs on vertices ω have the k-extension property for every k, then they are isomorphic.*

Proof. We define an isomorphism piece by piece. Define a *partial isomorphism* between two graphs to be an isomorphism between two induced subgraphs. To construct the complete isomorphism from the lemma, we will use the following observation on partial isomorphisms.

Claim 3.8. *Consider two graphs G_1 and G_2 as in the assumption of the lemma. We will show that for every partial isomorphism between them, and every vertex v_1 in G_1, there is some vertex v_2 in G_2 such that the partial isomorphism can be extended with the pair (v_1, v_2).*

Proof. We apply the k-extension property for G_2, where k is the number of vertices that participate in the partial isomorphism (on either one of the two sides). This allows us to find some vertex v_2 that has the same connections as v_1 with the vertices that are already in the partial isomorphism. This completes the proof of the claim. ∎

Using the claim, and a symmetric version where the graphs G_1 and G_2 are swapped, we can iteratively construct a sequence of partial isomorphisms, which is growing with respect to extension, and which eventually covers every vertex in G_1 and G_2. To construct this sequence, in even-numbered steps we add a vertex from G_1, and in odd-numbered steps we add a vertex from G_2. The limit of this sequence is a complete isomorphism, which completes the proof of the lemma. ∎

Lemmas 3.6 and 3.7 give us the theorem, since the first one says that almost all graphs have the property that guarantees isomorphism in the second lemma. ∎

We finish this chapter with the observation that the random graph has quantifier elimination.

Theorem 3.9. *For every first-order formula, possibly with free variables, there is a quantifier-free formula that is equivalent to it in the random graph, i.e. it is equivalent for all possible assignments of the free variables.*

Proof. We will prove the following lemma.

Lemma 3.10. *For every two tuples $(x_1, \dots, x_k)$ and $(y_1, \dots, y_k)$ of vertices in the random graph, the following conditions are equivalent:*

1. *the two tuples satisfy the same quantifier-free formulas;*
2. *the two tuples satisfy the same first-order formulas;*
3. *$\bar{x} \mapsto \bar{y}$ can be extended to an isomorphism of the random graph with itself.*

Proof. Clearly we have 2 ⇒ 1, since quantifier-free formulas are a special case of first-order formulas. Also, we have 3 ⇒ 2, since isomorphism does not affect the values of first-order formulas. Finally, 1 is the same as saying that $\bar{x} \to \bar{y}$ is a partial isomorphism, and this we know by Claim 3.1 can be extended to a full isomorphism. ∎

The theorem is an immediate consequence of the above lemma, and the fact that there are finitely many quantifier-free non-equivalent formulas, once the number of free variables has been fixed. ∎

Problem 11. Show that the following conditions are equivalent for every first-order formula:

1. its limiting probability is one for finite graphs;
2. it is true in the infinite random graph.

Problem 12. Show that the quantifier elimination in Theorem 3.9 is decidable, i.e. the equivalent quantifier-free formula not only exists, but can be effectively computed.

Problem 13. Prove that the infinite random graph is connected.

Problem 14. For $n \in \{1, 2, \ldots\}$ consider the following graph. The vertices are vectors $x \in \{0, 1\}^n$, and two vertices are connected by an edge if their scalar product is odd (i.e. nonzero if we work in the two-element field). Show that this graph has the k-extension property when $k = n^2$.

Problem 15. Show that the zero-one law from Theorem 3.1 fails when, instead of first-order logic, we use monadic second-order logic. The latter is the extension of first-order logic where we also quantify over sets of vertices. (But not sets of pairs of vertices, or more complicated objects.)

4
Weighted automata over a field

This chapter is about automata which input words and output rational numbers. The original definition comes from Schützenberger [51]. We show that these automata can be minimised (even in polynomial time) and can be tested for equivalence (again, in polynomial time), but the following version of the emptiness problem is undecidable:

is the output 0 for some input?

Note that the dual problem,

is the output 0 for every input?

is a special case of the equivalence problem, and is therefore decidable in polynomial time. We use the field of rational numbers, but most results would work for other fields.

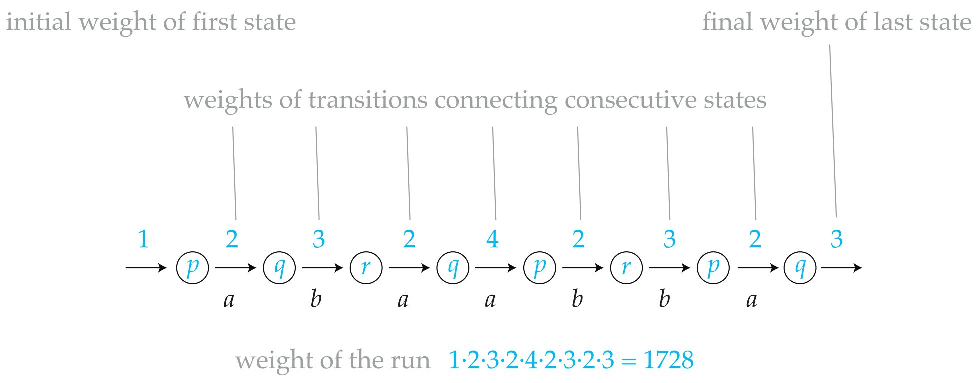
The automata discussed in this chapter can be viewed in two ways: as a nondeterministic device with states from a finite set (we call these weighted automata) and as a deterministic device with states from a vector space (we call these vector space automata). Both views are useful, so we present both of them.

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

Definition 4.1 (Weighted automaton)**.** *A* weighted automaton *consists of:*

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

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


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

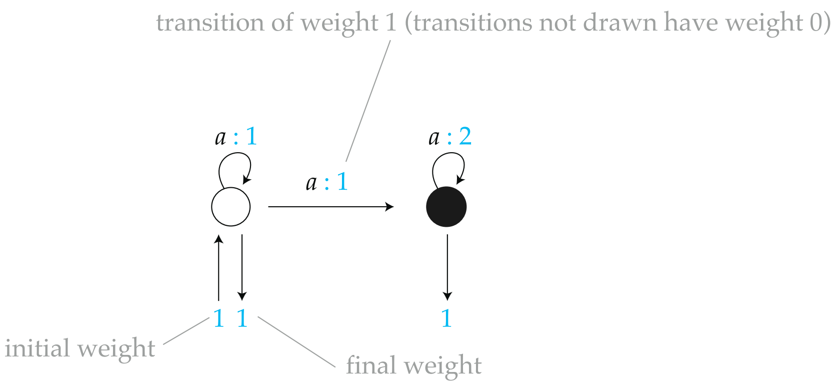
Define the weight *of a word to be the sum of the weights of all runs. The function* recognised *by the automaton is the function that maps a word to its weight.*

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

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

then we recover distance automata as discussed in Chapter 13. For this chapter, however, it will be important that we use the rational numbers, or more generally a field, so that we can use linear algebra.

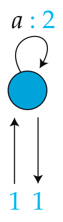
Example 11. Consider the following weighted automaton with three states.


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

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

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

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

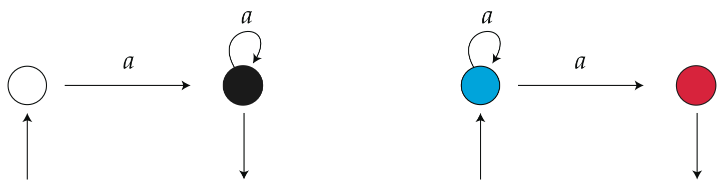


As we will see later in this chapter, weighted automata can be minimised. The second automaton is in fact the minimal automaton – how could it be smaller? – and there exists an automaton homomorphism (see later in the chapter for the definition) from the first automaton to the second one, namely the function

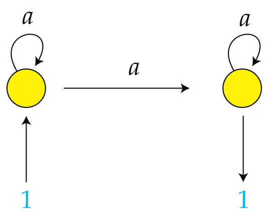
$$x \cdot \circ + y \cdot \bullet \qquad \mapsto \qquad (x+y) \cdot \bullet.$$

□

Example 12. [Running example] We describe two weighted automata which will be used as the running example in this chapter. We can view a nondeterministic automaton as a special case of a weighted automaton, by assuming that every arrow (including dangling arrows that indicate initial and final states) has weight 1. In this view, the semantics of weighted automata will map an input word to the number of accepting runs. Consider the following two nondeterministic automata over input alphabet $\{a\}$



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



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

Vector space automata. We now present a deterministic view on weighted automata. In this view, the automaton has a state space that is a vector space, and each letter deterministically updates the state using a linear function. This definition is almost the same as the original definition of Schützenberger [51, Definition 1], except that the original definition also allowed control states from a finite set. We do not use control states, because they do not contribute to expressive power of the model (although they make constructions easier), see the proof of Lemma 4.12.

Definition 4.2 (Vector space automaton). *A* vector space automaton[1] *consists of:*

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

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

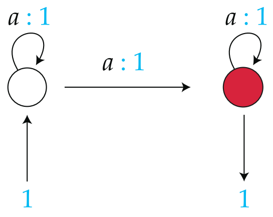
The automaton begins in the initial state, and when reading a letter $a \in \Sigma$, it updates its state using the transition function from item 4. After reading all the letters of the input word, the output function is applied to the last state, yielding the output of the automaton.

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

Example 13. Without increasing the expressive power of the model, we could allow affine functions in the transitions of a vector space automaton. The construction is illustrated on the following example.
Consider the length function over a one letter alphabet $\{a\}$. The most natural approach to recognise this function would be to have the one dimensional vector space $\mathbb{Q}$ as the state space and use the affine function $q \mapsto q + 1$ as the transition function. However, Definition 4.2 requires linear transition functions, so we use a workaround. The state space is $\mathbb{Q}^2$ and the initial state is $(1, 0)$. When reading a letter a, the automaton applies the function

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

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



If we ignore the weights, the above picture shows a nondeterministic automaton, which has exactly n accepting runs on a word of the form a^n, and thus using the weighted semantics, we get the length function. (This is the weighted automaton at the end of Example 12.) □

Equivalence of the models. A closer inspection of the vector space automaton and the weighted automaton used in Example 13 shows that these are actually the same automaton, only drawn using different pictures. This sameness is formalised in the following lemma.

Lemma 4.3. *Weighted automata and vector space automata recognise the same functions.*

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

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

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

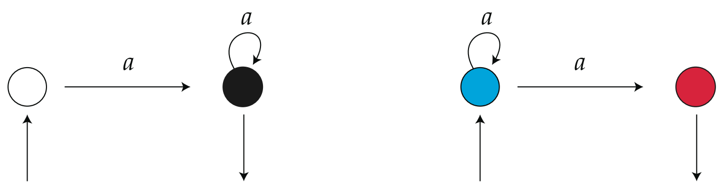
- The state space of $\mathsf{vec}\mathcal{A}$ is $\mathbb{Q}^P$, where P is the states of $\mathcal{A}$.
- The initial state of $\mathsf{vec}\mathcal{A}$ assigns to each state its initial weight.
- The output function of $\mathsf{vec}\mathcal{A}$ multiplies each coordinate by its final weight.
- For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

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

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

■

Example 14. [Running example] Recall the two weighted automata:



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

$$(x \cdot ○ + y \cdot ●) \cdot a = (x+y) \cdot ● \qquad (x \cdot 🔵 + y \cdot 🔴) \cdot a = x \cdot 🔵 + x \cdot 🔴.$$

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

4.1 *Minimisation of weighted automata*

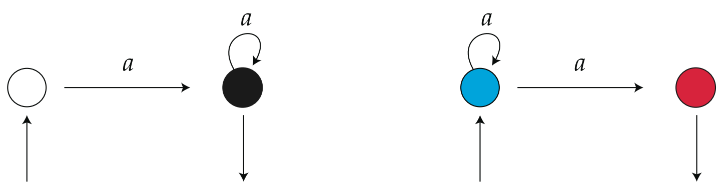
In this section, we prove a Myhill-Nerode style theorem on the existence of a minimal automaton, which is unique up to isomorphism (although the notion of isomorphism is a bit more involved than usual).

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

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

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

Example 15. [Running example] Recall these weighted automata



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

$$x \cdot \bigcirc + y \cdot \bullet \qquad \mapsto \qquad (x+y) \cdot {\color{cyan}\bullet} + y \cdot {\color{red}\bullet}.$$

Note that h is an isomorphism between the two state spaces. Clearly h maps the initial state $\bigcirc$ of the first automaton to the initial state ${\color{cyan}\bullet}$ of the second automaton. The following diagram shows that h is consistent with the transition functions:

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

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

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

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

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

Theorem 4.4. *Let $f : \Sigma^* \to \mathbb{Q}$ be a function recognised by a vector space automaton. There exists a vector space automaton, called the minimal automaton of f, which recognises f and such that every reachable vector space automaton recognising f admits a homomorphism into the minimal automaton.*

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

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

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

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

Let f be a function as in the statement of the theorem. Define the minimal automaton of f as follows. The state space, which is a subspace of the infinite dimensional space $\Sigma^* \to \mathbb{Q}$, is all finite linear combinations of functions of the form fw for $w \in \Sigma^*$. The initial state is f. The transition function is defined using the right action $q \mapsto qa$ defined above. The output function takes a state q to its value on the empty word. The automaton clearly recognises the function f. We will justify below why the state space has finite dimension, and therefore the automaton is indeed a vector space automaton as per Definition 4.2. We now show that every reachable vector space automaton recognising f admits a surjective homomorphism onto the minimal automaton defined above. Let $\mathcal{A}$ be a vector space automaton recognising f. For a state q of $\mathcal{A}$, define $[q] : \Sigma^* \to \mathbb{Q}$ to be the function recognised by the vector space automaton obtained from $\mathcal{A}$ by changing its initial state to q.

Claim 4.5. *The function $[\_]$ is a surjective homomorphism from $\mathcal{A}$ onto the minimal automaton.*

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

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

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

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

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

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

Example 16. [Running example] The function recognised by the automata in the running example is the characteristic function of the set of nonempty words. This function is not recognised by any vector space automaton with a one dimensional state space (equivalently, by any weighted automaton with one state) because if the state space has one dimension, then the recognised function is of the form

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

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

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

4.2 *Algorithms for equivalence and minimisation*

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

Lemma 4.6. *Assume that rational numbers are represented in binary notation, linear subspaces of $\mathbb{Q}^d$ are represented using a basis, and linear maps are represented using matrices. The following operations on linear subspaces can be done in polynomial time: (a) test for inclusion, (b) compute the subspace spanned by a union of two subspaces, (c) compute the image under a linear map.*

Equivalence. We begin with a simple algorithm for finite vector space automata: computing linear combinations of reachable states. Computing the actual reachable states, and not their linear combinations, is a different story and leads to undecidability, as we will see in Section 4.3. To compute linear combinations of reachable states we use a simple saturation procedure. We begin with $Q_0 \subseteq Q$ being the vector space spanned by the singleton of the initial state, i.e. this is the one dimensional vector space whose basis is the initial state. Then, assuming that a vector space $Q_i \subseteq Q$ has already been defined, we define Q_{i+1} to be the vector space spanned by

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

A representation of Q_{i+1} can be computed in polynomial time from a representation of Q_i, using the toolkit from Lemma 4.6. We also use the following observation: the coefficients in the basis for $Q_i \cdot a$ can only grow, as compared with the coefficients for Q_i, by a constant amount depending on the linear map $q \mapsto qa$. This way we get a growing chain of linear subspaces

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

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

Theorem 4.7. *The following problem is in polynomial time, assuming that field operations have unit cost:*

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

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

Computing the minimal automaton. We show that the minimal automaton from Theorem 4.4 can be computed in polynomial time from any vector space automaton recognising the function f.

Theorem 4.8. *The following problem is in polynomial time:*

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

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

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

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

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

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

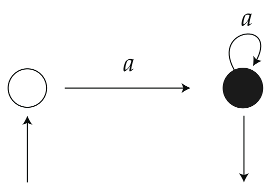
$$E_{n+1} = \bigcap_{a \in \Sigma} (f_a)^{-1}(E_n) \qquad \text{where } f_a \text{ is the linear map } (p, q) \mapsto (pa, qa).$$

We have a sequence of linear subspaces

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

By the same arguments as in the equivalence algorithm, the sequence above must stabilise at some equivalence relation, call it E_*, which can be computed in polynomial time. This stable equivalence relation E_* is the Myhill-Nerode equivalence relation, which identifies states if they produce the same outputs on all inputs. In the terminology of the proof of Theorem 4.4, two states are equivalent under E_* if and only if they have the same image under the function $[\_]$. The quotient of Q under E_* is therefore the minimal automaton; and this quotient can be computed in polynomial time, see Exercise 21. ■

Example 17. [Running Example] To finish the running example, we run the minimisation algorithm on the vector space automaton that corresponds to



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

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

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

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

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

4.3 *Undecidable emptiness*

In Theorem 4.7, we showed that equivalence of vector space automata (and therefore also of weighted automata) is decidable in polynomial time. A corollary is that one can decide if a weighted automaton maps all inputs to zero. We now show that a dual problem, namely mapping some word to zero, is undecidable. For the undecidability proof, it will be more convenient to use the syntax of weighted automata and not that of vector space automata.

Theorem 4.9. *The following problem is undecidable:*

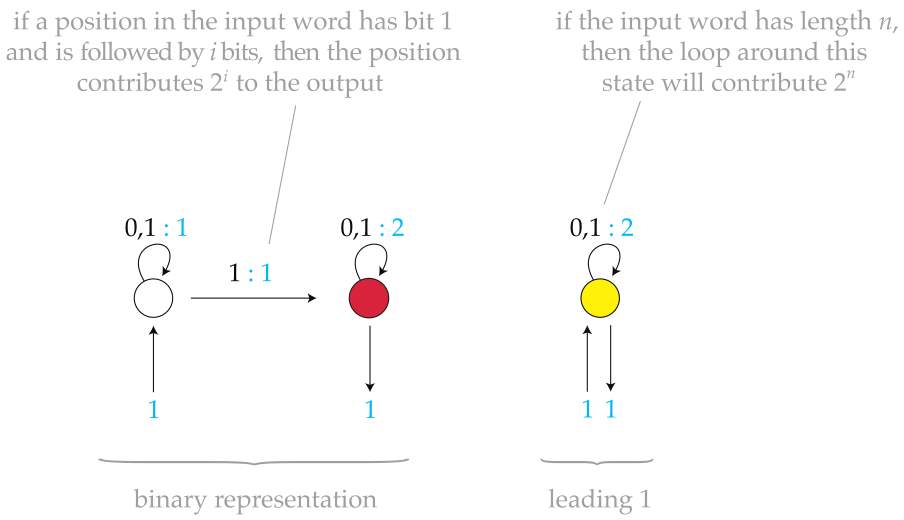
- **Input.** *A weighted automaton.*
- **Question.** *Is some word mapped to* 0*?*

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

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

Lemma 4.10. *For every alphabet* Σ *there is a weighted automaton which computes an injective function from* Σ^* *to the strictly positive rational numbers.*

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


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

■

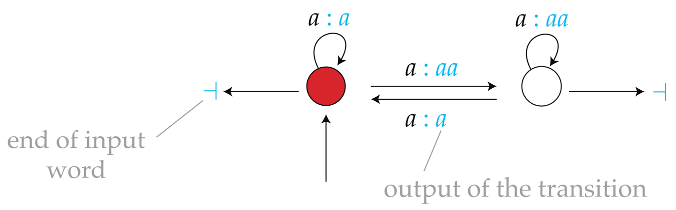
Composition with NFA's. To give a high-level description of the undecidability proof, it will be convenient to compose weighted automata with a certain kind of word-to-word functions.

Definition 4.11. *An* NFA with output *consists of:*

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

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

Example 18. Consider the following NFA with output where the input alphabet is $\{a\}$ and the output alphabet is $\{a, \dashv\}$.



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

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

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

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

Lemma 4.12. *If $\mathcal{A}$ is an NFA with output, which has input alphabet Σ and output alphabet Γ, and $\mathcal{B}$ is a weighted automaton with input alphabet Γ, then the function $\mathcal{B} \cdot \mathcal{A}$ defined by*

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

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

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

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

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

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

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

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

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

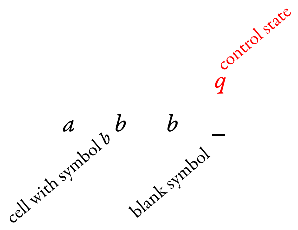
The multiset semantics of NFA's with output were chosen so that the proof above works. In our undecidability proof below, we use the above lemma in the special case when $\mathcal{A}$ has at most one run over every input word, and so the multiset semantics do not play a role. Equipped with Lemmas 4.10 and 4.12, we prove the undecidability result from Theorem 4.9.

Proof of Theorem 4.9. We first introduce some notation and closure properties for weighted automata. If $\mathcal{A}, \mathcal{A}'$ are weighted automata, then we write $\mathcal{A} + \mathcal{A}'$ for the disjoint union of the automata; on the level of recognised functions this corresponds to addition of outputs. We write $-\mathcal{A}$ for the weighted automaton obtained from $\mathcal{A}$ by multiplying all initial weights by -1, on the level of recognised functions this corresponds to multiplying the output values by -1. We also write $\mathcal{A} - \mathcal{A}'$ instead of $\mathcal{A} + (-\mathcal{A}')$. Finally, if L is a regular language, then there is a weighted automaton, call it $\mathsf{char}(L)$ which recognises the characteristic function of L, i.e. maps words from L to 1 and other words to 0.

The proof of the theorem is by reduction from the halting problem for Turing machines. For a Turing Machine M, we define a weighted automaton which outputs 0 on at least one input word if and only if M has at least one halting computation. Suppose that Σ is the work alphabet of the machine M, which includes the blank symbol. We encode a configuration of the machine as a word over the alphabet

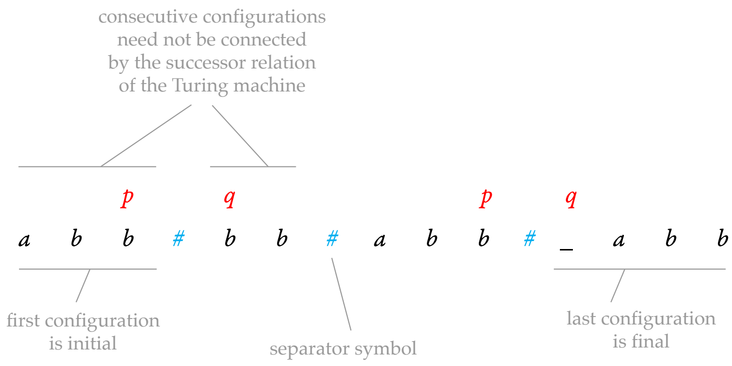
$$\Delta \quad \stackrel{\text{def}}{=} \quad \Sigma + \Sigma \times Q$$

in the natural way, here is a picture:



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

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


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

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

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

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

$$w_2 \# w_3 \# \cdots \# w_n \qquad w'_1 \# w'_2 \# \cdots \# w'_{n-1}$$

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

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

where $\mathcal{H}$ is the hashing automaton from Lemma 4.10 and the product operation $\cdot$ is as in Lemma 4.12. ■

Problem 16. Construct weighted automata over unary alphabet, which for a word of length n output

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

Problem 17. Show that for weighted automata with 2 states over a unary alphabet, it is decidable whether the automaton assigns value 0 to some word. *Remark:* for weighted automata over a unary alphabet with an arbitrary number of states, this is an important open problem, called the Skolem Problem in [43].

Problem 18. A probabilistic automaton is a vector space automaton where the initial state $q \in \mathbb{Q}^d$ is a probability distribution on $\{1, \ldots, d\}$, the linear updates are such that they preserve probability distributions, and the output function sums the coordinates corresponding to some accepting subset $F \subseteq \{1, \ldots, d\}$. Show that the following questions are undecidable for probabilistic automata:

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

Problem 19. Show that the following question is decidable for probabilistic automata: is there some input word which produces output equal exactly 0?

Problem 20. Show that for every weighted automaton there is an isomorphic (using the notion of isomorphism inherited from vector space automata) one which has one initial and one final state.

Problem 21. Let $E \subseteq \mathbb{Q}^n \times \mathbb{Q}^n$ be a linear subspace which is an equivalence relation. Let $f_1, \ldots, f_k : \mathbb{Q}^n \to \mathbb{Q}^n$ be linear maps which respect the equivalence relation, i.e. if inputs are equivalent, then also outputs are also equivalent. Show that one can compute in polynomial time linear maps

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

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

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

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

Problem 22. Consider a more symmetric model of NFA with output, as in Definition 4.11, where there is also a *start of input* word associated to each initial state, and the output of a run begins with the start of input word for its first state. Show that this model has the same expressive power as in Definition 4.11.

Problem 23. Call an NFA *unambiguous* if for every input there is at most one accepting run. Show that equivalence – i.e. are the same input words accepted – for unambiguous automata can be decided in polynomial time.

Problem 24. Construct an NFA with n states such that shortest rejected word rejected has length exponential wrt. n.

Problem 25. Show that if an NFA with n states is unambiguous and rejects at least one word, then it rejects some word of length at most $n - 1$.

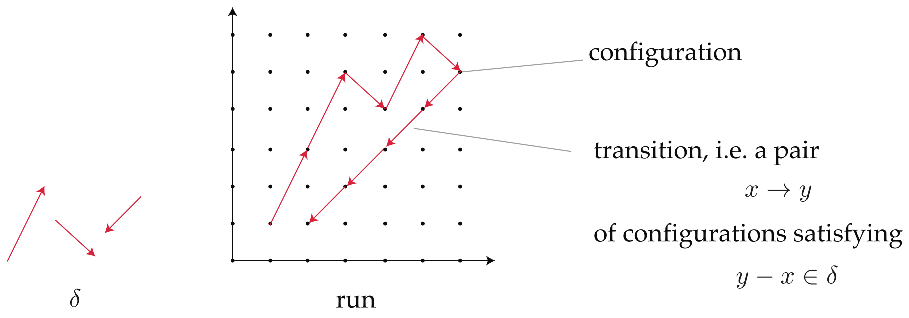
Problem 26. Show a polynomial time algorithm that decides if an NFA is unambiguous.

Problem 27. Give a more direct proof of Theorem 4.9 which uses the Post Correspondence Problem. Recall that the Post Correspondence Problem is the question: given two homomorphisms $f, g : \Sigma^* \to \Gamma^*$, decide if there is some nonempty word w such that $f(w) = g(w)$. This problem is undecidable.

5
Vector addition systems

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

Definition 5.1 (Vector Addition System)**.** *The syntax of a* vector addition system *consists of a dimension $d \in \{1, 2, \ldots\}$ and a finite set $\delta \subseteq \mathbb{Z}^d$. A run of the system is a finite sequence of vectors in $\mathbb{N}^d$ (called* configurations*) such that every consecutive configurations in the run form a* transition *as explained in the following picture for dimension $d = 2$:*



The most famous problem for vector addition systems is *reachability*, i.e. given two configurations, decide if they can be connected by a run. Reachability is

decidable, which was first shown by Mayr in [37], although the computational complexity of the problem remains unknown, see [49]. The reachability algorithm is complicated and beyond the scope of this book. It is crucial that configurations are vectors of natural numbers; for configurations that are integer vectors the reachability problem and related problems become much simpler, see Exercise 37.
In this chapter, we present a simple algorithm for a different problem, called coverability.

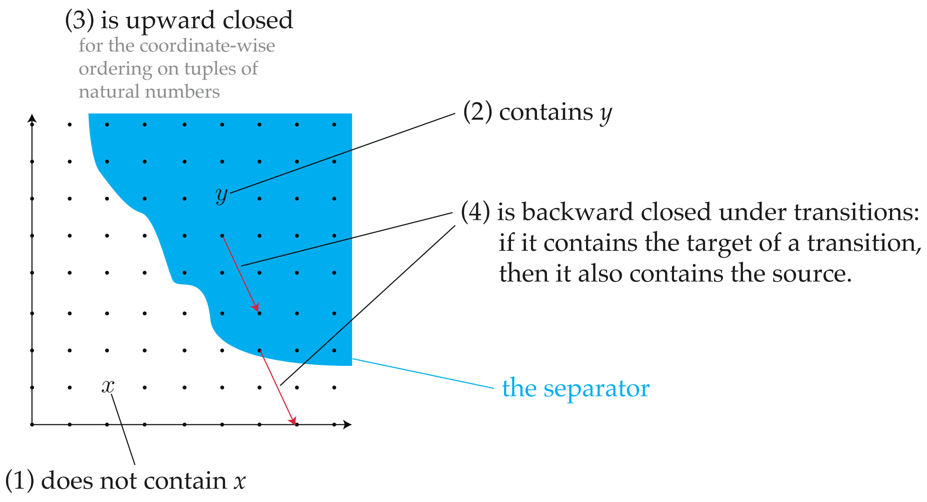
Theorem 5.2. *The following problem, called coverability, is decidable:*

- **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 [50] 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 5.2, by giving a semi-algorithm for the complement of the coverability problem.

Lemma 5.3. *There is a semi-algorithm deciding non-coverability, i.e. an algorithm that inputs a vector addition system with configurations x, y and terminates with success if and only if there is no run from x to any configuration $\geq y$.*

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


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

We claim that the following conditions are equivalent:

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

2. there is a separator for x and y.

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

can be represented in a finite way using its minimal elements. First, every element in the separator is above some minimal element, because the order $\leq$ on $\mathbb{N}^d$ is well-founded. Second, every set has finitely many minimal elements, because minimal elements form an antichain (i.e. are pairwise incomparable with respect to $\leq$) and antichains are finite according to the following claim.

Claim 5.4 (Dickson's Lemma)**.** *Antichains in $\mathbb{N}^d$ are finite.*

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

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

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

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

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

An alternative proof would use the infinite Ramsey theorem, see Problem 31. ■

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

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

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

Problem 28. 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 29. Which of the following ordered sets are wqo's?

1. $\mathbb{N}^2$ with lexicographic order;
2. $\{a, b\}^*$ with lexicographic order;
3. $\mathbb{N}$ with divisibility order, i.e. x smaller than y if $x \mid y$;
4. Σ^* with prefix order;
5. Σ^* with infix order;
6. line segments with an order: $[a, b]$ smaller than $[c, d]$ if $(b < c) \vee (a = c \wedge b \leq d)$;
7. graphs with subgraph order (remove some edges and some vertices);

8. trees with subtree order (remove some nodes, but keep the descendant ordering).

Problem 30. Show that if $(X, \leq_X)$ and $(Y, \leq_Y)$ are both wqos then also $(X \times Y, \leq)$ is wqo, where $(x, y) \leq (x', y') \Leftrightarrow x \leq_X x' \wedge y \leq_Y y'$.

Problem 31. Prove the Infinite Ramsey Theorem: in every infinite clique, with edges coloured on finitely many colours there is an infinite monochromatic subgraph, i.e. subgraph such that all the edges in it are coloured by the same colour.

Problem 32. Let $(X, \preceq)$ be a wqo. Show that there is no infinite growing sequence of upward-closed subsets X, i.e. no sequence

$$U_1 \subsetneq U_2 \subsetneq \ldots,$$

s.t. for all $i \in \mathbb{N}$ set $U_i \subseteq X$ is upward-closed wrt. $\preceq$.

Problem 33. Show that given a d-dimensional VAS and $s \in \mathbb{N}^d$, one can compute the set of all configurations from which s is coverable. Hint: use Problem 32.

Problem 34. Show that given a vector addition system with a distinguished source configuration, one can decide if the set of configurations reachable from the source is finite.

Problem 35. 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 36. 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 37. Define a *$\mathbb{Z}$-vector addition 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 38. 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 $(p, \bar{0})$ to some configuration with state q.

Problem 39. Find a vector addition system, say of dimension d, where the reachability relation

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

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

Problem 40. Find a family of vector addition systems with states, say of dimension d (the dimension does not need to be fixed for the family), where the reachability set

$$\{v : \text{there is a run from from the origin to } v\} \subseteq \mathbb{N}^d$$

is finite, but

1. of doubly exponential size,
2. of tower size

with respect to the number of transitions.

6
Polynomial automata

In this chapter, we introduce an extension of vector space automata, in which linear updates are extended to polynomial ones. We will show that equivalence remains decidable, but instead of using linear algebra, we will need to use (non-linear) algebra, namely Hilbert's Basis Theorem. The application of the Hilbert Basis Theorem to problems in automata theory dates back at least to the solution of the Ehrenfeucht Conjecture by Albert and Lawrence [2]. The presentation here is inspired by the more recent results from [53] and [6].

Polynomial automata. We begin by defining the automaton model. Fix some field $\mathbb{F}$. Instead of linear maps, we use *polynomial maps*, which are functions

$$f : \mathbb{F}^{d_1} \to \mathbb{F}^{d_2}$$

where each of the d_2 output coordinates is defined by a polynomial with d_1 variables corresponding to the input coordinates. Here is an example with $d_1 = 3$ and $d_2 = 2$:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \mapsto \begin{pmatrix} x^2 + 3yz + 7 \\ 2x + y^3 \end{pmatrix}.$$

The definition of polynomial automata is the same as the definition of vector space automata in Definition 4.2, except that polynomial maps are used instead of linear ones.

Definition 6.1 (Polynomial automaton)**.** *A* polynomial automaton *over a field* $\mathbb{F}$ *consists of:*

1. *an input alphabet, which is a finite set* Σ*;*
2. *a set of states, which is of the form* $Q = \mathbb{F}^d$ *for some* $d \in \{0, 1, \ldots\}$*;*
3. *an initial state* $q_0 \in Q$*;*
4. *for each letter* $a \in \Sigma$*, a polynomial map from* Q *to itself, denoted by* $q \mapsto qa$*;*
5. *a polynomial map from* Q *to* $\mathbb{F}$*, called the* output function.

The semantics of the automaton, which is a function of type $\Sigma^* \to \mathbb{F}$, is defined like for vector space automata. We have defined the model for fields, but the definition makes sense also in the more general case of rings or even semirings, since we do not use division in the definition of the model. Nevertheless, the techniques that we will apply will not work for general rings, as was the case in Chapter 4.
By definition, polynomial automata generalise vector space automata. The generalisation is strict. One reason is that polynomial automata can produce bigger numbers, as explained in the following example.

Example 19. Consider a polynomial automaton over the field of rational numbers. The input alphabet is a, and the dimension is $d = 1$. The automaton begins in state 2. Upon reading an input letter, it squares its state, via the polynomial x^2, and the final function outputs the state. Upon reading a word with n letters, the output will be 2^{2^n}, which is doubly exponential in the input length. This function cannot be computed by a vector space automaton, since vector space automata can only produce numbers that are singly exponential in the input length. $\square$

6.1 The equivalence algorithm

This chapter is devoted to showing that equivalence is decidable for polynomial automata, assuming that the field is the rational numbers.

Theorem 6.2. *The following problem is decidable:*

- **Input.** *Two polynomial automata $\mathcal{A}, \mathcal{B}$ over the field of rationals.*
- **Question.** *Do they compute the same function $\Sigma^* \to \mathbb{Q}$?*

We do not give any bounds on the complexity of the algorithm, since the proof will use Hilbert's Basis Theorem in a non-constructive way. Most of the proof will work for any field, but toward the end we will need to use specific properties of the field of rationals. In fact, the theorem also holds for any other field, assuming that field elements can be represented in a finite way and the field operations are computable. However, as we explain later in this chapter, certain things become simpler if we use the rational numbers.
We begin as we did for vector space automata, by reducing the equivalence problem $\mathcal{A} = \mathcal{B}$ to the zeroness problem $\mathcal{A} = 0$. This is done checking zeroness of the difference $\mathcal{A} - \mathcal{B}$. Once we start working with the zeroness problem, the proof starts to bear more similarities with the coverability algorithm for vector addition systems from Chapter 5. As in the coverability algortihm, we decide zeroness for polynomial automata by using two semi-algorithms: (a) one which terminates with success if and only if the automaton is nonzero; and (b) one which terminates with success if and only if the automaton is zero. The first semi-algorithm is easy: enumerate all input words, and stop upon finding a word that produces a nonzero output. The interesting part is the second semi-algorithm, which amounts to finding a finite witness for zeroness. Similarly to the coverability algorithm, this finite witness will be a kind of invariant, i.e. a set of states which: (1) contains the initial state; (2) is closed under transitions; (3) contains only states on which the output function is zero; and (4) can be represented in a finite way. In the coverability algorithm, condition (4) was ensured by using downward closed sets. In this chapter, a similar role will be played by algebraic varieties, as we describe next.

Algebraic varieties. An algebraic variety is a polynomial generalisation of a linear subspace. To see this, it is convenient to think of a linear subspace as being defined by a system of linear equalities, e.g.

$$\begin{aligned} 3x + 2y - z &= 0 \\ 2x - y &= 0. \end{aligned}$$

In an algebraic variety, one uses systems of polynomial equalities, e.g.

$$\begin{aligned} x^2 + yz - 1 &= 0 \\ 2x - y^3 &= 0. \end{aligned}$$

The formal definition is given below.

Definition 6.3 (Algebraic variety). *A subset $V \subseteq \mathbb{F}^d$ is called an* algebraic variety *if there is some set of polynomials $P \subseteq \mathbb{F}[x_1, \ldots, x_d]$ such that a vector belongs to V if and only if it is a root of all polynomials from P, i.e.*

$$(x_1, \ldots, x_d) \in V \quad \Leftrightarrow \quad \forall p \in P \quad p(x_1, \ldots, x_d) = 0.$$

The polynomials in the set P correspond to the equalities, and therefore we will also refer to P as a system of polynomial equalities. The definition does not stipulate that the set P is finite, but as we will see later, it can always be made finite.

Example 20. Suppose that the dimension is $d = 1$, and the field is the reals. The full set of all reals is an example of a variety, as witnessed by the empty system of polynomial equalities. (If one insists on a nonempty system, we can throw in the equation $0 = 0$.). Also, every finite set $\{a_1, \ldots, a_n\}$ of reals is a variety, as witnessed by a single polynomial equality

$$(x - a_1) \cdots (x - a_n) = 0.$$

These are all the varieties, i.e. in dimension one a variety is either full or finite. Indeed, if the system of polynomial equalities contains at least one nonzero polynomial, then it this polynomial will have only finitely many roots, which will mean that the set of solutions is finite. □

Example 21. Let us stay with the field of reals. In dimension two, one can have infinite varieties that are not full, e.g. the unit circle which is defined by the polynomial equality

$$x^2 + y^2 - 1 = 0.$$

Observe that all examples of varieties have used one equation only. This is not a coincidence, if we use the field of reals. This is because a finite system (and all systems can be made finite, as we will soon see) of polynomial equalities

$$p_1 = 0, \dots, p_n = 0$$

can be replaced by a single polynomial equality

$$p_1^2 + \cdots + p_n^2 = 0.$$

This trick fails in the complex numbers. □

The following lemma shows that systems can always be assumed to be finite.

Lemma 6.4. *For every infinite system of polynomial equalities, there is a finite system which defines the same variety.*

Proof. Consider a possibly infinite set of polynomials $P \subseteq \mathbb{F}[x_1, \dots, x_d]$. Define the *ideal generated by P*, denoted by $\langle P \rangle$ to be the set of polymials of the form

$$q_1 \cdot p_1 + \cdots + q_n \cdot p_n,$$

where $p_1, \dots, p_n$ are polynomials from P and $q_1, \dots, q_n$ are arbitrary polynomials. In other words, these are finite linear combinations of polynomials from P that use other polynomials as coefficients. This set is an ideal, which means that it is closed under addition inside itself and multiplication by arbitrary polynomials. It is also easily seen to be the inclusion-wise least ideal that contains P.

If we replace P be its generated ideal, then the defined variety does not change. Indeed, if a vector $x \in \mathbb{F}^d$ is a root of all polynomials from P, then it is also a root of all polynomials from the ideal $\langle P \rangle$. Therefore, the lemma will be a corollary of the following result, which is known as Hilbert's Basis Theorem: every ideal in $\mathbb{F}[x_1, \ldots, x_d]$ is finitely generated. Although Hilbert's Basis Theorem is a standard result in algebra, we give a self-contained proof later in this chapter. ∎

As mentioned before, we will use algebraic varieties to define invariants which witness zeroness of polynomial automata.

Lemma 6.5. *Consider a polynomial automaton, where the state space is $Q = \mathbb{F}^d$. The automaton is zero if and only if there is an* invariant, *which is defined to be a variety $V \subseteq \mathbb{F}^d$ such that:*

1. *the initial state belongs to V;*
2. *V is closed under applying the transition functions of the automaton;*
3. *the output function is zero on all points from V.*

Proof. Clearly if there is an invariant, then the automaton must be zero. Let us prove the converse implication: assuming that the autmaton is zero, we will construct an invariant. This invariant is rather straightforward: the correspnding equalities are those which are satisfied by all reachable states. More formally, let $P \subseteq \mathbb{F}[x_1, \ldots, x_n]$ be the polynomials that vanish on all reachable states, i.e. they give zero for every reachable state in the automaton. Define the invariant V to be the corresponding variety,

$$V = \{\bar{x} \in \mathbb{F}^d \mid \bar{x} \text{ is a root of all polynomials in } P \ \}.$$

Let us argue that V satisfies the three required properties.

1. By definition, V contains all reachable states, in particular the initial state.
2. Consider a transition function $a : \mathbb{F}^d \to \mathbb{F}^d$ of the automaton. The set P is closed under precomposition with a. Indeed, if a polynomial vanishes on

all reachable states, then the same will be true after precomposing it with a, because reachable states are closed under applying a. This in turn implies that the variety V is closed under a.

3. Since the automaton is zero, P contains the polynomial defining the output function. Therefore, the output function is zero on all points from V.

■

To complete the proof of Theorem 6.2, it remains to show that we can enumerate through candidates for an invariant, as in the above lemma, and we can check if a candidate satisfies the three conditions from the above lemma. So far, we have not used any assumptions on the field, and such assumptions will only appear now. The first assumption, which is needed to write down candidates for invariants, is that

(i) we can represent field elements in a finite way.

This assumption rules out the reals. Using this assumption, we can enumerate through all varieties, since every variety is described by a finite system of equations thanks to Lemma 6.4. It remains to show how to check if a given variety satisfies the three conditions that define an invariant in Lemma 6.5. The first condition says that the variety contains the initial state. In order to check this condition, all we need to do is to evaluate the polynomials from the system on the initial state, for which we need the following assumption:

(ii) the field operations are computable.

Let us now look at the second condition, which says that the invariant is closed under transitions. This means that for every transition function a, we have the inclusion

$$a(V) \subseteq V.$$

Assume that the variety V is defined by polynomials $p_1, \ldots, p_n$. The above inclusion says that if we take a point that is a root of these polynomials, and we apply a to this point, then the new point must also be a root of these polynomials:

$$\forall \overline{x} \in \mathbb{F}^d \qquad \bigwedge_{i \in \{1,\ldots,n\}} p_i(\overline{x}) = 0 \quad \Rightarrow \quad \bigwedge_{i \in \{1,\ldots,n\}} p_i(a(\overline{x})) = 0.$$

In other words, the variety defined by $p_1, \ldots, p_n$ is contained in the variety defined by $p_1 \circ a, \ldots, p_n \circ a$. Therefore, in order to check if the second condition is satisfied, a sufficient assumption for our algorithm would be:

(iii) inclusion on varieties is decidable.

Assumption (iii) is also enough for checking the last condition in the definintion of an invariant, which says that V is contained in the variety defined by the output function. Summing up, if the field satisfies assumptions (i), (ii) and (iii), then we can enumerate through candidates for invariants, and check if they satisfy the three required conditions, and thus equivalence is decidable for polynomial automata, as required by Theorem 6.2.
It remains to check if the rational numbers satisfy the assumptions (i), (ii) and (iii). The first two assumptions are clearly satisfied. One can show that the third assumption is spurious, i.e.

$$\text{(i) and (ii)} \quad \Longrightarrow \quad \text{(iii)}.$$

However, showing this implication would require more algebraic background, namely Hilbert's Nullstellensatz and Groebner bases, which are beyond the scope of this book. Therefore, in the interest of a self-contained presentation, we will take a different approach to (iii), by drawing on the decidability of the first-order theory of the reals from Theorem 2.1. We will show that the rational numbers can be extended to a field which satisfies conditions (i), (ii) and (iii). This will complete the proof of the theorem. Indeed, if equivalence of polynomial automata is decidable over this bigger field, then it is also decidable over the smaller field of rational numbers, since polynomial automata over a smaller field are a special case of polynomial automata over a bigger field. Therefore, it remains to prove the following lemma.

Lemma 6.6. *There is a field which contains the rational numbers and satisfies assumptions (i), (ii) and (iii).*

Proof. We intend to use the decidability of the first-order theory of the field of real numbers from Theorem 2.1. We cannot simply use the field of reals, since these cannot be finitely represented, as required by assumptions (i) and (ii). However, there is a simple work-around, which is to consider only reals that can be represented in a finite way in first-order logic. More formally, a real number is called *definable* if there is a formula of first-order logic $\varphi(x)$ over the reals which is true for this number and no other numbers. For example, $\sqrt{2}$ is defined by the formula

$$x^2 = 2 \wedge x > 0,$$

and hence $\sqrt{2}$ is a definable real. It is easy to see that the definable reals are a field – i.e. they are closed under addition, multiplication and inverses. (One can also show without much difficulty that the definable reals are the same as the algebraic numbers which are real, i.e. have no complex part. This, however, is not needed here.) By definition, a definable real has a finite representation, namely the formula that defines it. There might be several different representations, but using decidability of the theory of the reals, we can check if two representations define the same number. Summing up, the definable reals satisfy assumptions (i) and (ii).
It remains to show that the definable reals also satisfy assumption (iii). As explained before, inclusion on varieties can be phrased as a formula of the form

$$\forall \bar{x} \in \mathbb{F}^d \quad \bigwedge_{p \in P} p(\bar{x}) = 0 \quad \Rightarrow \quad \bigwedge_{q \in Q} q(\bar{x}) = 0,$$

for some finite sets of polynomials P and Q. This is a first-order formula, and therefore we can appeal to Theorem 2.1 to decide if this formula is true. There is, however a subtle point: the quantification over $\bar{x}$ in our intended application refers to definable reals, while Theorem 2.1 decides formulas where quantification ranges over reals which are not necessarily definable. The following claim shows that this is not a problem, since the change of quantification does not affect the truth value, i.e. the same sentences are true in the reals as in the definable reals.

Claim 6.7. *The definable reals have the same first-order theory as the reals.*

Proof. The main observation is that if a formula $\varphi(x)$ of first-order logic over the reals is true for some real number x, then it is also true for some x that is a definable real. Indeed, by the quantifier elimination result in Theorem 2.1, the set of reals x which make the formula $\varphi(x)$ true is a finite union of intervals. If one of these intervals is an isolated point, then this point is definable (as one of the finitely many isolated points which make the formula true). Otherwise, one (in fact, every one) of these intervals contains a rational number, and rational numbers are definable.
Using the observation from the previous paragraph, we show that the truth value of a formula does not change if it is interpreted over the reals or over the definable reals. To prove the statement by induction, we extend it to formulas with free variables. We show that for every first-order formula $\varphi(x_1, \ldots, x_n)$ and definable reals $a_1, \ldots, a_n$, we have

$$\mathbb{R} \models \varphi(a_1, \ldots, a_n) \quad \Leftrightarrow \quad \underbrace{\mathbb{R}_{\text{def}}}_{\text{definable reals}} \models \varphi(a_1, \ldots, a_n).$$

The only interesting case in the induction is the case of a quantifier, where we use the observation from the previous paragraph. ∎

The above lemma completes the proof of Theorem 6.2. For the sake of clarity, we recapitulate the proof. First, we reduce the equivalence problem to zeroness, by subtracting the two automata. Next, we think of the polynomial automaton as being over the field of definable reals, which does not change the answer to the zeroness problem, since the outputs produced do not change. To solve the zeroness problem, we search for either: (a) an input word that witnesses non-zeroness; or (b) an invariant that witnesses zeroness. Each of these is a finite object, and therefore a witness of one of the two kinds must exist. For witnesses of the second kind, we use decidability of the first-order theory of the reals to check if a variety is an invariant. This completes the proof of Theorem 6.2. ■

6.2 *A proof of Hilbert's Basis Theorem*

We finish this chapter with a self-contained proof of Hilbert's Basis Theorem, which was used in the proof of Lemma 6.4.

Theorem 6.8 (Hilbert's Basis Theorem)**.** *Let $\mathbb{F}$ be a field. Every ideal in $\mathbb{F}[x_1, \ldots, x_d]$ is finitely generated.*

Proof. Recall that a monomial is a product of variables with exponents, e.g. $x_1^3 x_2^2 x_3^5$ is a monomial in three variables. We consider two orders on monomials.

1. The first one is the *divisibility order*, in which a monomial is increased by multiplying it by another monomial, i.e. increasing the exponents. If we think of monomials as being vectors in $\mathbb{N}^d$, then the divisibility order is the same as the coordinate-wise order that was used in Chapter 5.

2. The second order is called the *monomial order*. In this order, we first order monomials by the total degree, i.e. the sum of all exponents, then we order them by the exponent next to x_1, then by the exponent next to x_2, and so on. Here is an example:

$$x_1^3 x_2^2 x_3^5 \quad > \quad x_1^2 x_2^1 x_3^7 \quad > \quad x_1^5 x_2^1 x_3^1.$$

This divisibility order is not total, since there can be incomparable monomials. The monomial order is total, and it refines the divisibility order, i.e. it has more comparable pairs. For a polynomial, define its *leading monomial* to be the biggest monomial with respect to the monomial ordering. Here is an example:

$$3\underbrace{x_1^3 x_2^2 x_3^5}_{\text{leading monomial}} + 7x_1^2 x_2^1 x_3^7 - 2x_1^5 x_2^1 x_3^1$$

Using the above orders, we prove the theorem. Let I be an ideal. Define a sequence of polynomials

$$p_1, p_2, \ldots \in I$$

as follows. Suppose that $p_1, \ldots, p_n$ have already been defined. If these polynomials generate the ideal I, then we stop, since we have shown that I is finitely generated. Otherwise, take some polynomial $p_{n+1} \in I$ that is not in the ideal generated by $p_1, \ldots, p_n$, and choose it so that its leading monomial is the least possible in the monomial ordering. We will show that the sequence defined this way stops at some point, and so the ideal is finitely generated. Toward a contradiction, suppose that the sequence is infinite. Since the divisibility order is a well quasi-ordering, there must be some $i < j$ such that the leading monomial of p_i is less than or equal to the leading monomial of p_j in the divisibility order. This means that there is some monomial m such that

$$m \cdot (\text{leading monomial of } p_i) \quad = \quad \text{leading monomial of } p_j.$$

Since multiplying by m preserves the monomial ordering, it follows that

$$(\text{leading monomial of } m \cdot p_i) \quad = \quad \text{leading monomial of } p_j.$$

This means that leading monomial of $p_j - m \cdot p_i$ is strictly smaller than the leading monomial of p_j, and thus $p_j - m \cdot p_i$ should have been used instead of p_j. ■

Problem 41. Which of the following structures are rings:

1. $(\mathbb{N}, +, \cdot, 0)$;
2. $(\mathbb{Z}, +, \cdot, 0)$;
3. $(\mathbb{R}, +, \cdot, 0)$;
4. $(\{0\}, +, +, 0)$;
5. $(\mathbb{Z}[x_1, \ldots, x_n], +, \cdot, 0)$;

6. $(\mathbb{Q}[x_1, \ldots, x_n], +, \cdot, 0)$;
7. $(\mathbb{Z}, \max, +, 0)$;
8. $(\mathbb{N}, \max, +, 0)$;
9. $(S_n, \cdot, \cdot, id)$;
10. $(\mathbb{Z}_n, +, \cdot, 0)$ for $n \in \mathbb{N}$.

Problem 42. Let $(R, +, \cdot, 0)$ be a ring. A subset $I \subseteq R$ is called an *ideal* if the following two conditions hold: 1) for all $i, j \in I$ it holds $i + j \in I$, 2) for all $i \in I, r \in R$ it holds $i \cdot r, r \cdot i \in I$. Find all ideals in the following rings:

1. $(\mathbb{Z}, +, \cdot, 0)$;
2. $(\mathbb{Q}, +, \cdot, 0)$.

Generalize the second case to any field.

Problem 43. A ring congruence is an equivalence relation $\equiv$ such that $x_1 \equiv y_1$, $x_2 \equiv y_2$ implies $x_1 * x_2 \equiv y_1 * y_2$ for $* \in \{+, \cdot\}$. For an ideal $I \subseteq R$ and $r_1, r_2 \in R$ we say that $r_1 \equiv_I r_2$ if $r_1 - r_2 \in I$. In case of the ring of integers all the $\equiv_I$ are actually $\equiv_n$, so in particular ring congruences. Show that for every ideal I, the relation $\equiv_I$ is a ring congruence.

Problem 44. Show that every ideal in $\mathbb{Q}[x]$ is generated by one element.

Problem 45. Is every ideal in the following rings generated by one element:

1. $\mathbb{Z}[x]$?
2. $\mathbb{Q}[x, y]$?

Problem 46. Is there a constant $c \in \mathbb{N}$ such that every ideal in $\mathbb{Z}[x]$ is generated by at most c elements?

Problem 47. Show that for every ring R the following conditions are equivalent:

1. every ideal in R is finitely generated;
2. every growing sequence of ideals $I_1 \subsetneq I_2 \subsetneq \ldots$ is finite.

Problem 48. Prove the Hilbert's Basis Theorem in the following formulation: if R is a ring where every ideal in R is finitely generated, then also every ideal in $R[x]$ is finitely generated.

Problem 49. Show that for every set $A \subseteq \mathbb{F}$ the set $\mathsf{pol}(\mathsf{A})$ is an ideal.

Problem 50. Consider the closure operation from $\mathbb{P}(\mathbb{Q})$ to $\mathbb{P}(\mathbb{Q})$ defined for $A \subseteq \mathbb{Q}$ as $\overline{A} = \mathsf{zero}(\mathsf{pol}(\mathsf{A}))$. Show that the following conditions are true for every $A, B \subseteq \mathbb{Q}$:

- $A \subseteq \overline{A}$;
- if $A \subseteq B$ then $\overline{A} \subseteq \overline{B}$;
- $\overline{A} = \overline{\overline{A}}$.

Problem 51. Consider the field of rational numbers $\mathbb{Q}$. Show that for every finite set of variables X and every ideal $I \subseteq \mathbb{Q}[X]$, there is an ideal $J \subseteq \mathbb{Q}[x]$ generated by a single polynomial such that $\mathsf{zero}(I) = \mathsf{zero}(J)$.

Problem 52. Assume that a closed set $A \subseteq \mathbb{F}^n$ is represented by a finite basis for the ideal $\mathsf{pol}(A)$. Show that closed sets are effectively closed under products and inverse images of polynomials, i.e. if A, B are closed and p is a polynomial, then the sets $A \times B$ and $p^{-1}(A)$ are closed, and their representations can be computed.

Problem 53. 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.

7
Orbit-finite automata

In this chapter, we study regular languages over infinite alphabets.

The simplest possible idea for a regular language over an infinite alphabet is that it is recognised by a finite automaton with finitely many states. This is not an interesting idea, since this kind of model cannot have non-trivial access to the infinity of the alphabet. More formally, we consider two input letters a and b equivalent if

$$q \xrightarrow{a} p \text{ is a transition} \qquad \Leftrightarrow \qquad q \xrightarrow{b} p \text{ is a transition}$$

holds for all states p and q. There are finitely many possible equivalence classes of letters, and equivalent letters are indistinguishable by the automaton. Therefore, up to all intents and purposes, the input alphabet is also finite.

In this chapter, we study a more interesting notion of regularity for infinite alphabets. It is based on the idea that the automata have access to equality on the alphabet, but nothing else[1]. We fix for the rest of this chapter some infinite alphabet, which we denote by $\mathbb{A}$. We think of the letters in the alphabet as

[1]This idea dates back to the work of Kaminski and Francez [35], and has been developed in many subsequent papers, see e.g. the lecture notes [11].

being names[2], such as

$$\mathbb{A} = \{\text{John}, \text{Eve}, \text{Tom}, \ldots\}.$$

Examples of languages that we care about include

$$\{w \in \mathbb{A}^* \mid \text{ the first letter is equal to the last letter } \}. \tag{7.1}$$

$$\{w \in \mathbb{A}^* \mid \text{ some letter appears at least twice } \}. \tag{7.2}$$

To recognise such languages, we use automata that have some kind of finite memory, and which can store letters from $\mathbb{A}$. For example, the language (7.1) is recognised by an automaton which loads the first letter into its state, and then toggles acceptance depending on comparison of the state with the current input letter. The language (7.2) is recognised by an automaton which nondeterministically guesses a position, stores its letter, and then waits for this letter to appear again. These automata will be described in more detail in Examples 22 and 23.

Equivariance. Before giving a definition of the automata involved, we begin by a formal definition of what it means for a language to "depend only on equality". The general idea is that if a language contains a word such as the following three-letter word

$$\text{John} \cdot \text{Eve} \cdot \text{John},$$

then it must also contain any other word which has the same equality patterns, such as the following one:

$$\text{Tom} \cdot \text{John} \cdot \text{Tom}.$$

This idea is formalised by applying permutations of the alphabet, since choosing different names amounts to applying a permutation on the alphabet.

[2] Although we use names of people as examples, a more relevant application for computer science is variable names in programs. Based on this analogy, the sets discussed in this chapter are called *nominal sets* in the literature on semantics of programming languages [44].

Definition 7.1 (Equivariant language)**.** *A language $L \subseteq \mathbb{A}^*$ is called* equivariant *if*

$$w \in L \quad \Leftrightarrow \quad \pi(w) \in L$$

holds for every permutation π of the alphabet $\mathbb{A}$.

The goal of this chapter is to define certain automata models that recognise equivariant languages. Let us now describe in more detail two automata for the languages that were mentioned before.

Example 22. Consider the language from (7.1), i.e. the set of words where the first letter is equal to the last letter. We describe a deterministic automaton that recognises this language. The state space of this automaton will be

$$\underbrace{\mathbb{A} \;+\; \mathbb{A}}_{\text{two disjoint copies}} \;+\; \underbrace{\{\varepsilon\}}_{\substack{\text{initial}\\ \text{state}}}.$$

In the above, and also elsewhere in this chapter, we use $+$ to represent disjoint union. Elements of the first disjoint copy of $\mathbb{A}$ will be denoted by $1(a)$ with $a \in \mathbb{A}$, while elements of the second disjoint copy will be denoted by $2(a)$. The transitions of the automaton are as follows. When reading a letter $a \in \mathbb{A}$ in the initial state ε, the automaton stores this letter in its state, using the first copy, as explained in the following set of transitions:

$$\varepsilon \xrightarrow{a} 1(a) \qquad \text{for every } a \in \mathbb{A}.$$

From now on, this atom will be kept in the state forever. However, the automaton will toggle between $1(a)$ and $2(a)$, depending on whether the current input letter is equal to the stored atom or not:

$$i(a) \xrightarrow{b} \begin{cases} 1(a) & \text{if } a = b \\ 2(a) & \text{if } a \neq b \end{cases} \qquad \text{for every } i \in \{1,2\} \text{ and } a, b \in \mathbb{A}.$$

Therefore, the accepting states are all states of the form $1(a)$, for $a \in \mathbb{A}$. □

Example 23. Let us now consider the language from (7.2), i.e. the set of words where some letter appears at least twice. We describe a nondeterministic automaton that recognises this language. The state space of this automaton will be

$$\underbrace{\mathbb{A}}_{\substack{\text{candidate}\\\text{for repetition}}} + \underbrace{\{\varepsilon, \top\}}_{\substack{\text{initial and final}\\\text{states}}}.$$

When the automaton is in the initial state, it can choose to ignore the input letter, which means that it expects the repeated letter to appear later. This is represented by transitions of the form

$$\varepsilon \xrightarrow{a} \varepsilon \qquad \text{for every } a \in \mathbb{A}.$$

Eventually, the automaton guesses nondeterministically that the current input letter a is the one that will appear twice, and stores it in its state:

$$\varepsilon \xrightarrow{a} a \qquad \text{for every } a \in \mathbb{A}.$$

Once this choice has been made, the behaviour of the automaton is deterministic:

$$a \xrightarrow{b} \begin{cases} \top & \text{if } a = b \\ a & \text{if } a \neq b \end{cases} \qquad \text{for every } a, b \in \mathbb{A}.$$

The only accepting state is $\top$. $\square$

7.1 Orbit-finite sets

The automata in Examples 22 and 23 use their state space to store some finite information (e.g. one of the two copies in Example 22) and a constant number of letters (in the above examples, the state stores at most one letter, but two or more letters could also be stored). This idea could be formalised using an automaton model that has a state of the form (element of a finite set, tuple of letters from $\mathbb{A}$); this is the approach used in [35]. In this chapter, we choose a slightly more abstract approach, following [11]. Instead of defining a new notion of automaton, we define a new notion of "finite set", and then the automata are defined to be those that use this new notion.

Definition 7.2 (Orbit-finite sets)**.** *An* orbit-finite set[3] *is any set of the form*

$$\mathbb{A}^{d_1} + \cdots + \mathbb{A}^{d_n},$$

for some natural numbers $d_1, \ldots, d_n \in \{0, 1, \ldots\}$.

For example, the state space in Example 22 can be seen as an orbit-finite set of the form

$$\mathbb{A}^1 + \mathbb{A}^1 + \mathbb{A}^0,$$

since the set $\{\varepsilon\}$ can be seen as a singleton set, just like $\mathbb{A}^0$.
Let us explain the name "orbit-finite". The name is chosen because, from a mathematical point of view, an orbit-finite set is equipped with an action of the group of atom permutations, and it will have finitely many orbits under this action. Let us explain this in more detail, and using elementary terminology. For an orbit-finite set as in Definition 7.2, an element can be written as

$$i(a_1, \ldots, a_{d_i}),$$

where $i \in \{1, \ldots, n\}$ identifies the summand $\mathbb{A}^{d_i}$, and $a_1, \ldots, a_{d_i}$ are the atoms used in the corresponding tuple of atoms. To such an element we can apply a permutation of the atoms: the summand i will remain unchanged, but the atoms will be changed according to the permutation. We say that two elements of an orbit-finite set are in *the same orbit* if one can be obtained from the other by applying a permutation of the atoms. Being in the same orbit is an equivalence relation, and therefore each orbit-finite set is partitioned into orbits. As we will see in a moment, this partition has finitely many parts. First, let us look at some examples of orbits.

Example 24. Let us have a look at some examples of orbits. Consider first an orbit-finite set without any atoms, which is achieved by using the copies of $\mathbb{A}^0$, as in the following example:

$$\mathbb{A}^0 + \mathbb{A}^0 + \mathbb{A}^0.$$

[3]The notion of orbit-finiteness used in [11] is slightly stronger than the one used here. However, the weaker notion used here (which is called *polynomial orbit-finite sets* in [11]) is simpler while retaining many interesting applications, so we stick to it in this chapter.

This set contains three elements $1(), 2(), 3()$, and each of these elements is in its own orbit, since there are no atoms to permute, and summands are not changed when applying atom permutations. Therefore, this set has three orbits, each one with a single element. This way, we can view finite sets as a special case of orbit-finite sets. □

Example 25. Let us now consider the orbits in a set of the form $\mathbb{A}^d$ with $d > 0$.

1. In the set $\mathbb{A}^1$, all elements are in the same orbit, since any letter can be mapped to any other letter by an appropriate permutation of the atoms.
2. In the set $\mathbb{A}^2$, there are two orbits, which are represented by

$$\underbrace{(\text{John}, \text{John})}_{\text{equal pairs}} \quad \underbrace{(\text{John}, \text{Eve})}_{\text{non-equal pairs}}.$$

3. In the set $\mathbb{A}^3$, there are five orbits, which are represented by

$$\underbrace{(\text{John}, \text{John}, \text{John})}_{\text{all equal}} \quad \underbrace{(\text{John}, \text{Eve}, \text{Mary})}_{\text{all different}}$$
$$\underbrace{(\text{John}, \text{John}, \text{Eve}) \quad (\text{John}, \text{Eve}, \text{John}) \quad (\text{Eve}, \text{John}, \text{John})}_{\text{two equal one different}}.$$

□

Lemma 7.3. *An orbit-finite set has only finitely many orbits.*

Proof. In a general orbit-finite set

$$\mathbb{A}^{d_1} + \cdots + \mathbb{A}^{d_n},$$

the number of orbits is the sum of the number of orbits in each summand $\mathbb{A}^{d_i}$, because applying an atom permutation does not change the summand. Therefore, it remains to show that each summand has finitely many orbits. As explained in Example 25, the number of orbits in $\mathbb{A}^d$ is equal to the number of possible equality patterns on d elements, which is given by the d-th Bell number. In particular, this number is finite. ■

Equivariance. We will use orbit-finite sets instead of finite sets when defining automata. In order to make the definition meaningful, the way that these sets are manipulated by the automata must also be compatible with the idea that only equality is relevant. This is captured by the following definition.

Definition 7.4. *A subset Y of an orbit-finite set X is called* equivariant *if*

$$x \in Y \quad \Leftrightarrow \quad \pi(x) \in Y \qquad \text{for every } x \in X \text{ and atom permutation } \pi.$$

An equivariant set is the same as some union of orbits, since if it contains one element of an orbit, then it must contain all elements of this orbit. It follows that an orbit-finite set can have only finitely many equivariant subsets. For example, the set $\mathbb{A}^3$ has $32 = 2^5$ equivariant subsets, since it has five orbits, and each of these orbits can either be included in the subset, or not.
We can generalise the notion of equivariance to relations

$$(x_1, \ldots, x_n) \in R \quad \Leftrightarrow \quad (\pi(x_1), \ldots, \pi(x_n)) \in R.$$

In fact, this is essentially a special case of the original notion, since a relation can be seen as a subset of the product $X_1 \times \cdots \times X_n$, and orbit-finite sets are closed under finite products. We can also generalise equivariance to functions

$$f(x) = y \quad \Leftrightarrow \quad f(\pi(x)) = \pi(y).$$

This again is not a new notion, since a function can be seen as a special kind of relation, in which every input has a unique output.

Example 26. There is only one equivariant function of type $\mathbb{A} \to \mathbb{A}$, which is the identity function. Indeed, consider some function that is not the identity, i.e. there exist $a \neq b$ such that $f(a) = b$. By equivariance, the output b could be replaced by any other atom that is not equal to a, which would contradict functionality. Using the same kind of argument, one can show that if $f : X \to Y$ is an equivariant function between two orbit-finite sets, then every atom which appears in the output $f(x)$ must also appear in the input x. $\Box$

7.2 Orbit-finite automata

As mentioned in the previous section, our automata will use orbit-finite sets instead of finite ones, with all structure being equivariant, as in the following definition.

Definition 7.5 (Orbit-finite automata)**.** *A* nondeterministic orbit-finite automaton *is defined in the same way as a nondeterministic finite automaton, except that all sets are orbit-finite, and all subsets and functions are equivariant:*

$$\overbrace{\underbrace{Q}_{\textit{states}} \quad \underbrace{\Sigma}_{\substack{\textit{input}\\\textit{alphabet}}}}^{\textit{orbit-finite}} \qquad \overbrace{\underbrace{I \subseteq Q}_{\substack{\textit{initial}\\\textit{states}}} \quad \underbrace{F \subseteq Q}_{\substack{\textit{final}\\\textit{states}}} \quad \underbrace{\Delta \subseteq Q \times \Sigma \times Q}_{\textit{transitions}}}^{\textit{equivariant}}.$$

A deterministic orbit-finite automaton *is the special case which has exactly one initial state, and where the transition relation is a function.*

The automata in Examples 22 and 23 were examples of deterministic and nondeterministic orbit-finite automata, respectively. As one can imagine, this approach can be applied to other notions, e.g. one can have orbit-finite graphs, orbit-finite Turing machines, orbit-finite systems of equations, etc. Such ideas are discussed at length in [11].

As we have mentioned before, orbit-finite sets generalise finite sets. A set with n letters can be seen as the orbit-finite set

$$\underbrace{\mathbb{A}^0 + \cdots + \mathbb{A}^0}_{n \text{ times}}.$$

Since sets of this form do not contain any atoms, the equivariance condition is trivial, and any relation or function between such sets is equivariant. Therefore, orbit-finite automata that use such input alphabets and state spaces are equivalent to ordinary finite automata.

7.3 Determinisation

In some ways, orbit-finite automata are similar to ordinary finite automata. Some constructions transfer without difficulty to the orbit-finite setting, such as the product construction $\mathcal{A} \times \mathcal{B}$ which is used to show that languages recognised by automata are closed under intersections and unions. This is because the product of two orbit-finite sets is again orbit-finite.
However, some constructions do not transfer. An important example is complementation for nondeterministic automata. This is usually proved by determinisation, which itself is proved using a powerset construction. The powerset construction fails for orbit-finite automata, and in fact no construction works, as stated in the following theorem.

Theorem 7.6.

1. *Languages recognised by nondeterministic orbit-finite automata are not closed under complementation.*

2. *Deterministic orbit-finite automata are strictly less expressive than nondeterministic orbit-finite automata.*

Proof. The first item implies the second one, since deterministic orbit-finite automata are closed under complementation. It remains to prove the first item. The witnessing language is

$$\{w \in \mathbb{A}^* \mid \text{ some letter appears twice } \}.$$

As we have seen in Example 23, this language is recognised by a nondeterministic orbit-finite automaton. Let us show that its complement is not. Suppose toward a contradiction that there is a nondeterministic orbit-finite automaton $\mathcal{A}$ that recognises the complement language, i.e. the words where each letter appears at most once. Let $d \in \{0, 1, \dots\}$ be the maximal number of atoms that can be stored in a state of this automaton. Consider an input word with $2d + 2$ pairwise different letters, and let w_1 and w_2 be its first and second halves, respectively. The automaton $\mathcal{A}$ must accept $w_1 w_2$. In the witnessing run, consider the first and last states, as well as the state that is used between the two halves:

$$p \xrightarrow{w_1} q \xrightarrow{w_2} r.$$

The middle state q can store at most d atoms, and therefore we can find atoms $a_1, a_2 \in \mathbb{A}$ such that (1) a_1 appears in w_1 but not in q, and (2) a_2 appears in w_2 but not in q. Let π be an atom permutation that swaps a_1 with a_2. Since neither a_1 nor a_2 appear in the middle state, this permutation does not affect q. Since the automaton is invariant under atom permutations, we can apply the permutation π to the second part of the run, but not the first one, and we will still get a valid run, because the middle state is the same:

$$p \xrightarrow{w_1} q = \pi(q) \xrightarrow{\pi(w_2)} \pi(r).$$

Therefore, the automaton also has an accepting run on the input word $w_1\pi(w_2)$. This input contains a repetition, since a_1 appears both in w_1 and in $\pi(w_2)$. The desired contradiction follows. ∎

7.4 *Emptiness*

In this section, we show that the emptiness problem is decidable for orbit-finite automata. This theorem is meant to illustrate the idea that orbit-finite sets can be manipulated algorithmically, despite being infinite.

Theorem 7.7. *The emptiness problem for orbit-finite automata is decidable.*

Proof. To make the problem well-defined, we have to explain how an orbit-finite automaton is represented as input, which requires a representation of orbit-finite sets and their equivariant subsets.

Representation of orbit-finite sets and equivariant subsets. For orbit-finite sets, Definition 7.2 already comes with a representation: we need to give a list of powers $[d_1, \ldots, d_n]$. For equivariant subsets, there are several possibilities, but in this chapter we use a finite list of example elements, with one example for each orbit. More formally, for an orbit-finite set X and a finite subset $Y \subseteq X$, define

$$\langle Y \rangle = \{\pi(y) \mid \; y \in Y \text{ and } \pi \text{ is an atom permutation } \}.$$

In other words, this is the least equivariant subset of X that contains Y. Every equivariant subset of an orbit-finite set is generated by a finite subset in this way, and thus we can use expressions of the form $\langle Y \rangle$ as representations of equivariant subsets. (This assumes that one can write down individual elements of orbit-finite sets, which in turn assumes that we can write down atoms. This is indeed the case, since we can think of atoms as being natural numbers, or strings over some finite alphabet.) As we have already mentioned, relations and functions can be represented as special kinds of subsets, so these can also be represented. For example, the function of type $\mathbb{A}^2 \to \mathbb{A}$ that projects onto the first coordinate is represented as

$$\langle \quad (\text{Eve}, \text{John}) \mapsto \text{Eve}, \quad (\text{Eve}, \text{Eve}) \mapsto \text{Eve} \quad \rangle.$$

Basic operations on equivariant subsets. Using this representation, we can easily implement a membership test

$$x \in \langle Y \rangle,$$

since this amounts to checking if x is equivalent, up to atom permutations, to one of the finitely many elements in Y. Using the membership test, we can implement an inclusion test

$$\langle Z \rangle \subseteq \langle Y \rangle,$$

since this amounts to checking membership for each of the finitely many representatives in Z. Using inclusions, we can also implement equality on subsets, since this is the same as mutual inclusion.

Emptiness algorithm. Having explained the representation and the basic operations on it, we can now explain the algorithm for emptiness. The idea is similar to the standard algorithm for emptiness of ordinary finite automata, which computes the set of reachable states. Let us write Q_n for the set of states that can be reached by reading some input word of length at most n. Observe that this set is equivariant: if we take a run that witnesses membership $q \in Q_n$, then applying an atom permutation π to this entire run will also witness membership $\pi(q) \in Q_n$. Therefore, each set Q_n can be represented using the representation of equivariant subsets that we have explained above. The algorithm computes these sets as described in the following procedure.

- **Initial step.** The set Q_0 is the set of initial states, which is given in the representation of the automaton.
- **Inductive step.** Suppose that we have computed Q_n. We compute Q_{n+1} as follows. First, we add all states from Q_n to Q_{n+1}. Next, we look at the representatives of the transitions, which are given in the representation of the automaton. For each such transition

  $$p \xrightarrow{a} q,$$

  we check if the source state p is in Q_n, using the membership test (this amounts to checking if p is in the same orbit as one of the finitely many representatives that we have stored for Q_n). If this is the case, then we add the target state q to Q_{n+1}. It is not hard to see that this will give us a list of representatives for all states in Q_{n+1}, since every transition will be considered up to atom permutations.
- **Stopping condition.** The chain is guaranteed to eventually stabilise with $Q_{n+1} = Q_n$. This is because the set of states has finitely many orbits, and therefore there are only finitely many possible equivariant subsets.

Once the chain stabilises, we check if Q_n contains some final state. Since the set of accepting states is equivariant, this boils down to checking if one of the finitely many representatives of Q_n belongs to the final set, which can be done using the membership test. The result of this test gives the answer to the emptiness problem. ∎

The above procedure even runs in polynomial time, assuming the representation used in the proof. This is because the running time is bounded by the number of orbits in the state space and the transition relation, and these numbers are polynomial in the size of the representation. However, the complexity of the algorithm depends on the choice of representations, and the one that we have used is particularly verbose (which makes the running time look quick). For more succinct representations the complexity can go up, e.g. the problem is PSPACE-complete for a representation using registers [35]. Not all problems for orbit-finite automata are decidable. For example, universality is undecidable [11, Section 2.2].

8
Star-free languages

In this chapter, we describe an alternative approach to regular languages, which uses monoids instead of automata. This approach will be illustrated by a characterisation of the languages that can be recognised by monoid which do not contain groups.
Let us begin with the definition of a monoid. This is like a group, except that we do not require inverses.

Definition 8.1 (Monoid). *A monoid is a set M equipped with a binary multiplication operation, which we denote multiplicatively by $m \cdot n$, subject to:*

- *associativity: $(m \cdot n) \cdot k = m \cdot (n \cdot k)$ for all $m, n, k \in M$; and*
- *identity element: there is some $1 \in M$ such that $1 \cdot m = m \cdot 1 = m$ for all $m \in M$.*

We often abuse notation, and use the same letter M to denote both the monoid and its underlying set. For example, we will write $m \in M$ for a monoid to mean that m is an element of the underlying set of the monoid.

Example 27. [Example monoids]

1. For every alphabet Σ, the set of all words Σ^* is a monoid under concatenation, with the empty word as the neutral element. This monoid is called the *free monoid* over alphabet Σ.

2. For every set Q, the set of functions $f : Q \to Q$ is a monoid. The monoid operation is function composition, and the identity element is the identity function.

3. For every set Q, the set of binary relations $R \subseteq Q \times Q$ is a monoid. The monoid operation is relation composition

   $$R \circ S = \{(p,q) \mid \ (p,t) \in R \text{ and } (t,q) \in S \text{ for some } t \in Q \ \}.$$

   The neutral element is the identity relation, which happens to be a function.

Observe that the monoid from item 2 is contained in the monoid from item 3.
□

When describing a monoid, it is not necessary to describe the identity element. This is because if the identity element exists, then it is unique. Indeed, by multiplying two candidates for the identity element, we would get the "real" identity. Therefore, from now on we will no longer mention the identity element when describing monoids.

The appropriate notion of function between monoids is that of a monoid homomorphism, as defined below.

Definition 8.2 (Homomorphism). *A monoid homomorphism $h : M \to N$ between two monoids is a function between their underlying sets which preserves the monoid operation*

$$h(m \cdot n) = h(m) \cdot h(n)$$

and which maps the monoid identity in M to the monoid identity in N.

Example 28. [Parity] Consider the monoid $\mathbb{Z}_2$ where the underlying set is $\{0, 1\}$, and the monoid operation is addition modulo two. An example of a monoid homomorphism is the function

$$h : \Sigma^* \to \mathbb{Z}_2$$

which tells us the parity of occurrences of some chosen letter $a \in \Sigma$. □

Example 29. [Free monoid homomorphisms] Consider an alphabet Σ and a monoid homomorphism

$$h : \Sigma^* \to M.$$

This monoid homomorphism is uniquely defined by its values on the letters, since the letters generate the input monoid. Furthermore, there are no constraints on the values of the letters, i.e. any function of type $\Sigma \to M$ will extend to a monoid homomorphism. This is the reason why the monoid Σ^* is called the free monoid. □

Example 30. [Monoid for a DFA] Consider a deterministic finite automaton with input alphabet Σ and states Q. Define a function

$$h : \Sigma^* \to (Q \to Q),$$

which maps a word w to its corresponding state transformation, i.e. the function that maps each state q to the state reached from q after reading w. This function is easily seen to be a monoid homomorphism, if we view the inputs as the free monoid, and the outputs as the monoid of functions from Q to Q (see Example 27). □

As far as this chapter is concerned, the purpose of monoids is to recognise languages, similarly to automata.

Definition 8.3 (Recognising by a monoid)**.** *We say that a language $L \subseteq \Sigma^*$ is* recognised *by a monoid M if there is a monoid homomorphism*

$$h : \Sigma^* \to M$$

such that membership $w \in L$ depends only on the value $h(w) \in M$. In other words, there is some accepting subset $F \subseteq M$ such that

$$w \in L \Leftrightarrow h(w) \in F \qquad \text{for all } w \in \Sigma^*.$$

Example 31. [Recognising b^*] Take the finite monoid M where the underlying set is $\{0, 1\}$, the monoid operation is minimum, and the identity element is 1. Consider the monoid homomorphism

$$h : \{a, b\}^* \to M,$$

which maps a to 0 and b to 1. If we take the accepting set to be $\{1\}$, then the recognised language is b^*. □

The following theorem shows that finite monoids can be used instead of finite automata to define regular languages.

Theorem 8.4. *A language is regular if and only if it is recognised by a finite monoid.*

Proof. In Example 30, we have shown how a deterministic finite automaton yields a homomorphism into a finite monoid, thus proving implication $\Rightarrow$ in the theorem. For the converse implication $\Leftarrow$, if we take a monoid homomorphism

$$h : \Sigma^* \to M,$$

then we can construct a deterministic finite automaton whose states are the elements of M, the initial state is the monoid identity, the transition function is defined by $(m, a) \mapsto m \cdot h(a)$. ■

8.1 Green's relations

One of the advantages of using monoids instead of automata is that monoids have a better structure theory than automata. This structure is based on Green's relations, which generalise the natural notions of prefix, suffix, and infix to monoids, as described in the following definition.

Definition 8.5 (Green's relations)**.** *Consider two elements m, n in a monoid M.*

- *We say that m is a* prefix *of n if $mx = n$ for some $x \in M$.*
- *We say that m is a* suffix *of n if $ym = n$ for some $y \in M$.*
- *We say that m is an* infix *of n if $xmy = n$ for some $x, y \in M$.*

All three relations defined above define pre-orders, i.e. they are reflexive and transitive, which is easy to see. In a free monoid Σ^*, these relations are anti-symmetric, i.e. one cannot have non-trivial comparisons in both directions. However, in general monoids these relations need not be anti-symmetric. For example, in a group, all elements are prefixes of each other, and likewise for infixes and suffixes

A prefix is a special case of an infix. Therefore, every prefix class is contained in some infix class. One could imagine that it is possible to grow in the prefix ordering while staying in the same infix class. As the following lemma shows, this is not possible in a finite monoid, and therefore all prefix classes contained in the same infix class must be incomparable in the prefix ordering.

Lemma 8.6. *Consider a finite monoid M, and let $m, n \in M$ be such that m and n are infix equivalent, and m is a prefix of n. Then m and n are prefix equivalent.*

Proof. By the assumption that m is a prefix of n, one can obtain n from m by multiplying to the right by some $x \in M$. By the assumption that n is an infix of m, one can obtain m from n by multiplying to the right by some $y \in M$ and multiplying to the left by some $z \in M$. By iterating this process in a loop, we see that

$$z^i m(xy)^i = m \quad \text{and} \quad z^i m(xy)^i x = n \qquad \text{for all } i \in \{0, 1, \ldots\}.$$

Since the monoid is finite, there must be some $i < j$ such that $(xy)^i = (xy)^j$. Therefore,

$$n = z^i m(xy)^i x \quad \text{is a prefix of} \quad z^j m(xy)^i = z^j m(xy)^j = m.$$

This proves that n is a prefix of m, and therefore they are prefix equivalent. ■

There are other results about Green's relations, which identify a very well-behaved structure inside each infix class. A description of some of this structure can be found in [10, Section 1]. For the purposes of this chapter, the above lemma will be sufficient.

8.2 Star-free languages and aperiodic monoids

To illustrate the power of monoids, we will use them to describe star-free languages. These are defined by using regular expressions which cannot use the Kleene star operation, but which are allowed to use complementation.

Definition 8.7 (Star-free language). *A language $L \subseteq \Sigma^*$ is called* star-free *if it can be generated from finite languages (i.e. languages containing finitely many words) using concatenation $L \cdot K$, as well as the Boolean operations of union, intersection, and complement.*

Example 32. The full language Σ^* is star-free, since it is the complement of the empty language, which is finite. Also, if we take some letter $a \in \Sigma$, then the language a^* is star-free, since it can be obtained as the complement of the language

$$\Sigma^* \cdot (\Sigma - \{a\}) \cdot \Sigma^*.$$

□

Example 33. As we will see later in this chapter, the language $(aa)^*$ is not star-free. The intuitive reason is that this language requires counting modulo two, which goes beyond the capabilities of star-free expressions. □

The main result of this chapter is the following theorem, which characterises star-free languages in terms of a monoid property called aperiodicity.

Theorem 8.8. *A language $L \subseteq \Sigma^*$ is star-free if and only if it is recognised by a finite monoid M which has the following* aperiodicity *property:*

$$\exists \omega \in \{1, 2, \ldots\} \; \forall m \in M \quad m^\omega = m^{\omega+1}.$$

Before proving the theorem, let us discuss some of its consequences.
The first consequence is an alternative description of aperiodicity, which is expressed in terms of groups. We say that a monoid *M contains a group G* if the group is a subset of the monoid and the group operation is inherited from the

monoid operation. Importantly, the group identity does not need to be inherited from the monoid operation. For example, if we take a group G, and we adjoin a new neutral element to it (different from the group identity in G), then the new monoid will contain G.

Fact 8.9. *A finite monoid is aperiodic if and only if it does not contain any non-trivial group (i.e. a group with more than one element).*

Proof. For the implication $\Rightarrow$, take some hypothetical group G contained in the monoid M. We know that

$$g^{\omega} = g^{\omega+1},$$

which by using group cancellation implies that g is the identity of the group. Therefore, the group is trivial. Consider now the opposite implication $\Leftarrow$. Take some monoid element $m \in M$, and consider its powers

$$m^1, m^2, m^3, \ldots.$$

Since the monoid is finite, these powers start to cyclically repeat, which means that some tail of the sequence of powers defines a cyclic group. This group must be trivial, and therefore the powers stabilise. Therefore, for every m there is some ω_m such that

$$m^{\omega} = m^{\omega_m+1}.$$

By taking ω to be the maximum of all ω_m, we obtain the aperiodicity. ∎

Thanks to the above fact, an alternative statement of Theorem 8.8 is that a language is star-free if and only if it is recognised by a finite monoid that does not contain any non-trivial group. This is particularly pleasing, since it relates a language property (star-freeness) to a purely algebraic property (absence of non-trivial groups). Things get even better, since star-freeness is also equivalent to other conditions, such as definability in first-order logic or the temporal logic LTL, as explained in [10, Chapter 2.2], but we do not discuss these other characterisations here.

Let us state another consequence of Theorem 8.8, which is that star-freeness is a decidable property of regular languages. On its own, Theorem 8.8 does not give such a decision procedure, since it might seem that we need to check all possible recognising monoids for a given language, and some of these monoids might not be aperiodic. However, as the following corollary shows, it is sufficient to check one monoid only, which leads to decidability.

Corollary 8.10. *Given a regular language, one can decide if it is star-free.*

Proof of Corollary 8.10. Suppose that we are given a regular language, given by a monoid homomorphism

$$h : \Sigma^* \to M$$

and an accepting set $F \subseteq M$. Let us begin by minimising this homomorphism. Define an equivalence relation $\sim$ on the monoid M by identifying two monoid elements $m, m' \in M$ if

$$xmy \in F \Leftrightarrow xm'y \in F \quad \text{for all } x, y \in M.$$

This equivalence relation can be computed. If this is a non-trivial equivalence relation, then we can reduce the size of the monoid, by fusing equivalent monoid elements, as in the Myhill-Nerode Theorem. Therefore, we can assume that the monoid M is minimal, i.e. no two distinct monoid elements are equivalent under $\sim$. Also, as in the proof of the Myhill-Nerode Theorem, the minimal monoid is unique up to isomorphism, and it can be obtained from any other recognising monoids by a fusion process as described above. Since aperiodicity is preserved under minimisation, it follows that

$$\text{some recognising monoid is aperiodic} \implies \text{minimal one is aperiodic.}$$

This leads to an algorithm: minimise the monoid, and then check if it is aperiodic. This algorithm runs in polynomial time, assuming that we use monoids as the representation of regular languages. ■

For example, if we take the language $(aa)^*$, then its minimal monoid is $\{0, 1\}$ with addition modulo 2, which is not aperiodic. Therefore, the language is not star-free, as we have asserted in Example 33.

Proof of Theorem 8.8. We begin with the implication

$$\text{star-free} \quad \Longrightarrow \quad \text{aperiodic monoid.}$$

The proof is by induction on the structure of the star-free expression. It is not hard to see that finite languages are recognised by aperiodic monoids, and that the class of languages recognised by aperiodic monoids is closed under union, intersection, and complement. For the complement, we simply change the accepting set, which does not affect the monoid, and for union and intersection we take the product of two monoids, which is an operation that preserves aperiodicity. The only interesting part is concatenation.

Lemma 8.11. *The class of languages recognised by aperiodic monoids is closed under concatenation.*

Proof. We will prove this by using a certain monoid construction, which corresponds to concatenation of languages, and which preserves aperiodicity. Consider two languages $L_1, L_2 \subseteq \Sigma^*$, which are recognised by two homomorphisms

$$h_i : \Sigma^* \to M_i, \quad i = 1, 2,$$

into two finite aperiodic monoids M_1 and M_2. To recognise the concatenation, we will use a new monoid, which stores the following information: (a) the values in the original two monoids; and (b) the set of possible pairs of values, ranging over factorisations into two parts. More precisely, consider the function

$$h : \Sigma^* \to \underbrace{M_1 \times M_2 \times \mathcal{P}(M_1 \times M_2)}_{\text{call this set } M}$$

which maps an input word to its values under M_1 and M_2 as well as the set of pairs $(h_1(w_1), h_2(w_2))$, ranging over factorisations $w_1 w_2$ of the input. It is not hard to see that the set M can be equipped with a monoid structure such that h is a homomorphism. The monoid operation is

$$(m_1, m_2, S) \cdot (n_1, n_2, T) = (m_1 n_1, m_2 n_2, S n_2 \cup m_1 T),$$

where Sn_2 appends n_2 to the second component of each pair in S, and similarly for m_1T. The homomorphism h recognises the concatenation L_1L_2, with the accepting set consisting of monoid elements where the set component contains some pair which is accepting in both M_1 and M_2.
The construction described above works for the concatenation of any two regular languages, without any assumptions on aperiodicity. We will now show that if the original two monoids were aperiodic, then the new monoid is also aperiodic. Consider then some element $m = (m_1, m_2, S) \in M$. We want to show that the powers

$$m^1, m^2, m^3, \ldots$$

stabilise at some value. Clearly these powers stabilise on the first two components, by assumption on aperiodicity of M_1 and M_2. Consider now the last component, which is equal to

$$\{(m_1^{i_1}x, ym_2^{i_2}) \mid (x,y) \in S \text{ and } i_1 + i_2 + 1 = i \}.$$

By aperiodicity of M_1 and M_2, there is a number $\omega \in \{1, 2, \ldots\}$ that witnesses aperiodicity in both monoids. If we take i to be sufficiently large, then the decomposition in $i_1 + i_2 + 1$ can have at most one number that is smaller than ω, and the other number will be equal to ω. Therefore, the third component will also stabilise for sufficiently large i. ■

We now turn to the implication

$$\text{aperiodic monoid} \implies \text{star-free},$$

which is the heart of the proof. Consider a homomorphism into an aperiodic monoid

$$h : \Sigma^* \to M.$$

We will show that every language recognised by this homomorphism is star-free. The interesting case is when the accepting set is a single monoid element $m \in M$, in which case the recognised language is

$$L_m = \{w \in \Sigma^* \mid h(w) = m \}.$$

In the following lemma, we will show that this language is star-free. This will extend to all other languages recognised by the homomorphism, thus completing the proof of Theorem 8.8, since all languages recognised by the homomorphism are finite unions of languages of the form L_m.

Lemma 8.12. *For every $m \in M$, the language L_m is star-free.*

Proof. The lemma is proved by induction on the position of m in the infix ordering. Suppose that we want to prove the lemma for some $m \in M$, and we have already proved it for all monoid elements that are proper prefixes of m. Define K_m to be the set of words which:

1. have a prefix whose value in the monoid is in the prefix class of m; and
2. have a suffix whose value in the monoid is in the suffix class of m.

We begin by showing that this property is star-free.

Claim 8.13. *The language K_m is star-free.*

Proof. Consider first the special case when m is in the prefix class of the monoid identity. In this case, K_m is the set of all words, since the corresponding prefixes and suffixes in the definition of K_m can be taken to be the empty word.
We are left with the case when m is not equivalent to the monoid identity, which means that the monoid identity is a proper prefix of m. For a word $w \in K_m$, consider the shortest prefix of w whose value in the monoid is in the prefix class of m. Since we have assumed that m is not prefix equivalent to the identity, this shortest prefix is non-empty, and therefore it is of the form ua, where a is a letter and u has a value in the monoid that is strictly smaller than m in the prefix ordering. Thanks to Lemma 8.6, the value of u is smaller not only in the prefix ordering, but also in the infix ordering. Therefore, we can use the induction assumption. Summing up, the property "some prefix is in the prefix class of m" can be expressed using the star-free expression

$$\bigcup_{n,a} L_n \cdot a \cdot \Sigma^*,$$

where n ranges over monoid elements that are strictly smaller than m in the infix ordering, and a ranges over letters such that $n \cdot h(a)$ is in the prefix class of m. A similar construction works for suffixes. Intersecting the two, we get the desired star-free expression K_m. ■

In the following claim, we show that the language K_m is closely related to the language L_m that we want to describe in the present lemma.

Claim 8.14. *The language K_m satisfies the following inclusions*

$$L_m \quad \subseteq \quad K_m \quad \subseteq \quad L_m \cup L_{>m},$$

where $L_{>m}$ consists of words whose value is strictly bigger than m in the infix ordering.

Proof. Clearly the first inclusion holds, since every word with value m has a prefix (e.g. the entire word) whose value is in the prefix class of m, and similarly for suffixes.
Consider now the second inclusion. Take some $w \in K_m$. Since w has a prefix whose value in the monoid is in the prefix class of m, the value of w is either equal to m, or strictly bigger than m in the prefix ordering. By Lemma 8.6, if the value is strictly bigger than m in the prefix ordering, then it is also strictly bigger in the infix ordering. Using this observation and a similar one for suffixes, we know that the value of $w \in K_m$ is either: (a) equivalent to m in both the prefix and suffix orderings; or (b) strictly bigger than m in the infix ordering. In case (b), we have membership in $L_{>m}$. It remains to prove that in case (a), the value is actually equal to m, and not some other element of its prefix class and suffix class. This is because there are no other such elements, as we now show:

(*) If $n \in M$ is both prefix and suffix equivalent to m, then $n = m$.

To prove (*), we use aperiodicity of the monoid. (The statement fails for general monoids, e.g. in a group all elements are prefix and suffix equivalent to each other.) If n is prefix equivalent to m, then there exist $x, y \in M$ such that

$$mx = n \quad \text{and} \quad yn = m.$$

This means that if we start with m, and then we multiply in alternation by x to the right and by y to the left, we will alternate in values between m and n:

$$y^i m x^i = m \quad \text{and} \quad y^i m x^{i+1} = n.$$

By aperiodicity of the monoid, at some point there will be no difference between x^i and x^{i+1}, and hence $m = n$. ■

Thanks to the above lemma, in the language K_m we have all the words with value m, and some extra words with value strictly bigger than m in the infix ordering. The following claim shows what these extra words look like.

Claim 8.15. *The following inclusion holds:*

$$K_m - L_m \quad \subseteq \quad \bigcup_{n,a} K_n \cdot a \cdot \Sigma^*,$$

where $n \in M$ in the union ranges over monoid elements that are prefixes of m, and $a \in \Sigma$ ranges over letters such that $n \cdot h(a)$ is not a prefix of m.

Proof. Consider a word in the left-hand side of the inequality. By Claim 8.14, this word is in $L_{>m}$, and therefore it has some prefix (for example, the entire word) whose value is strictly bigger than m in the infix ordering. If we take the shortest such prefix, then it will be in $L_n \cdot a$ for some n and a that satisfy the conditions described in the claim. Since $L_n \subseteq K_n$, we are done. ■

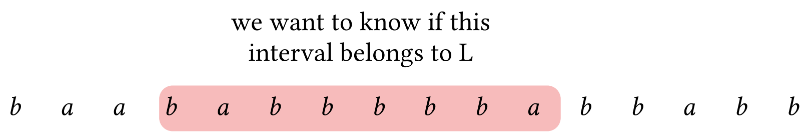
We are now ready to complete the proof of the lemma, by defining L_m using a star-free expression. Consider the expression on the right-hand side of the inequality from the above claim. This is a star-free expression, since we have proved that each K_n is a star-free expression. Formally speaking, we have proved it only for K_m, but in the same step we can prove it for all K_n with n being a prefix – and even infix – of m, since these monoid elements occupy the same or lower position in the induction order. Therefore, by removing the right-hand side of Claim 8.15 from K_m, we get a star-free expression for L_m, as required in the statement of the lemma. ■

■

9
The factorisation forest theorem

In this chapter, we prove the Factorisation Forest Theorem of Imre Simon [55], which is a powerful tool for describing how a word can be evaluated in a finite monoid.

An algorithmic motivation. The theorem can be viewed as describing a certain data structure, which is similar to a binary interval tree. Before presenting this data structure, we motivate it from an algorithmic perspective. Consider the following problem. We fix a regular language $L \subseteq \Sigma^*$. Given an input word, we want to build a data structure which allows us to answer efficiently the following query: given an interval of positions in the input word, does the corresponding infix belong to the language L? Here is a picture of such a query

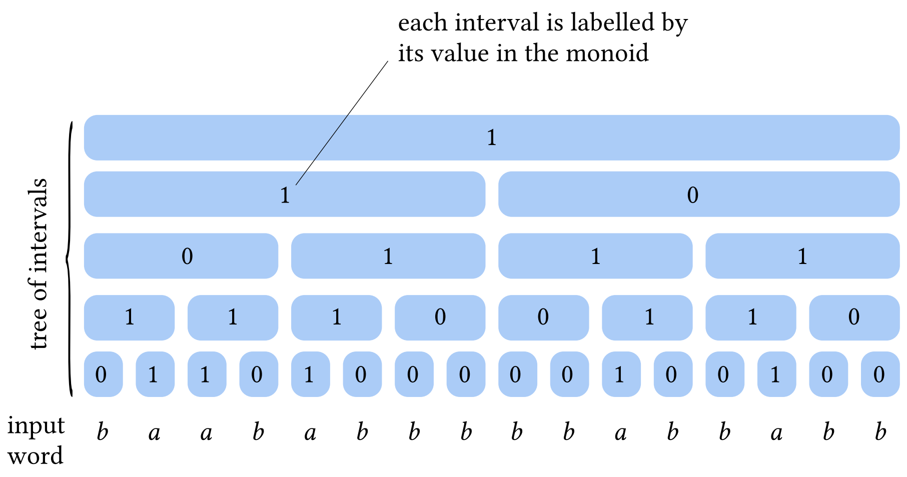

we want to know if this interval belongs to L
b a a b a b b b b b a b b a b b

Example 34. Suppose that the regular language is $\Sigma^* a \Sigma^*$. The corresponding queries ask if the interval contains the letter a. This can be solved by the following data structure: for each position in the input word, we store the number of occurrences of the letter a up to this position. Then, to answer a query for an interval, we just need to check if the numbers at the endpoints of the interval differ. The data structure can be built in linear time, and each query can be answered in constant time. The same data structure works for the language "even number of appearances of a". □

Let us begin with a straightforward divide-and-conquer solution to this problem, which works in the general case of any regular language. In this solution, as elsewhere in this chapter, we will use monoids instead of automata. Suppose that the language is recognised by a monoid homomorphism

$$h : \Sigma^* \to M$$

into a finite monoid, as explained in Chapter 8.2. Let us build an interval tree over the input word, as explained in the following picture (the picture uses the monoid homomorphism that counts occurrences of a modulo 2):

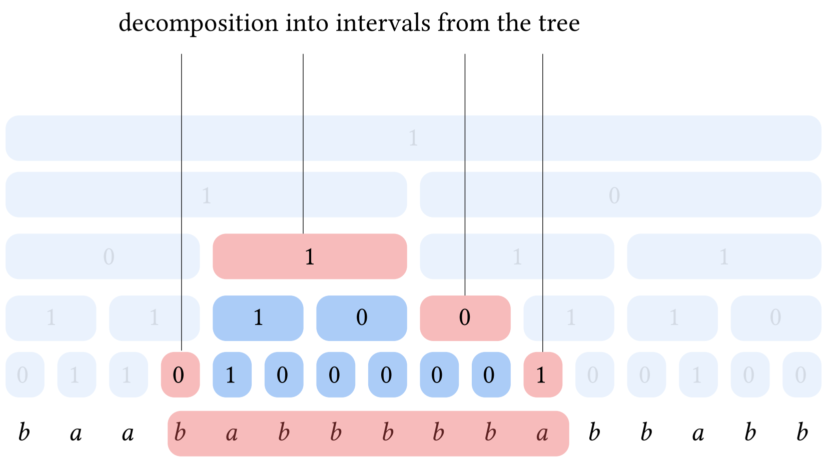


As usual for such pictures, it is convenient to assume that the length of the input word is a power of two, but this is not necessary. Formally speaking, an interval tree is defined as follows.

Definition 9.1 (Interval tree)**.** *An* interval tree *for a word w is a family of intervals with the following properties:*

1. *all singleton intervals are in the family; and*
2. *the full interval is in the family; and*
3. *every two intervals are either disjoint or one contains the other.*

The intervals in an interval tree can be seen as nodes in a tree, and we will adopt tree terminology when talking about them: child, sibling, parent, leaf, root etc. For the moment, we are interested in an interval tree that is binary, i.e. each node has either zero or two children. Such a tree can be built in linear time, with logarithmic height. For each node in the tree, we store the value of the homomorphism h on the infix corresponding to this node. This data structure can be built in linear time. Once we have the data structure, we can answer the queries in time $\mathcal{O}(\log n)$, by multiplying the semigroup elements from nodes of the tree that correspond to the given interval, as explained in the following picture:


decomposition into intervals from the tree
1
1
0
0
1
1
1
1
1
1
0
0
1
1
0
0
1
1
0
1
0
0
0
0
1
0
0
1
0
0
b a a b a b b b b b a b b a b b

9.1 Simon trees

As we have seen above, using a binary interval tree, we can achieve logarithmic query time. The purpose of this chapter is to improve this to constant time. This data structure was found by Imre Simon [55, Theorem 6.1], and for this reason we call it a *Simon tree*. A Simon tree is also a tree of intervals, however it is no longer binary and it can have nodes of unbounded degree. The general idea is that such nodes will be restricted to intervals whose value in the monoid is an idempotent, i.e. an element e satisfying $e^2 = e$. Here is the formal definition.

Definition 9.2 (Simon tree)**.** *Consider a monoid homomorphism*

$$h : \Sigma^* \to M$$

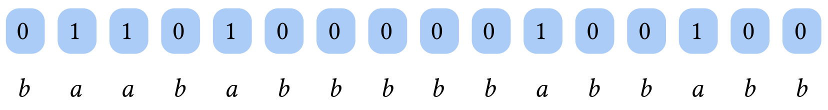
into a finite monoid. A Simon tree *for an input word $w \in \Sigma^*$ is an interval tree with the following property: if a node has $n > 2$ children, then all these children are mapped by h to the same monoid element, and this element is an idempotent.*

The main result of this chapter, see Theorem 9.3, will be that we can find a Simon tree whose height is bounded by a constant that depends only on the monoid homomorphism, and not on the input word. Before stating and proving the theorem, let us begin with an example of how the tree is constructed, and also with an explanation of how it can be used in our algorithmic application.

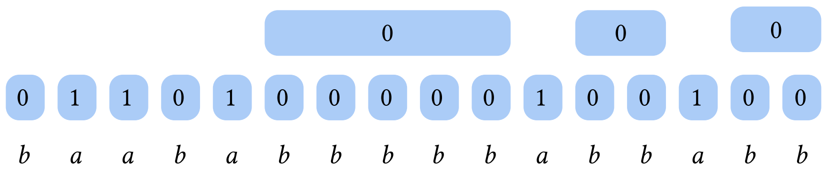
Example 35. Consider the monoid homomorphism

$$h : \{a, b\}^* \to \{0, 1\}$$

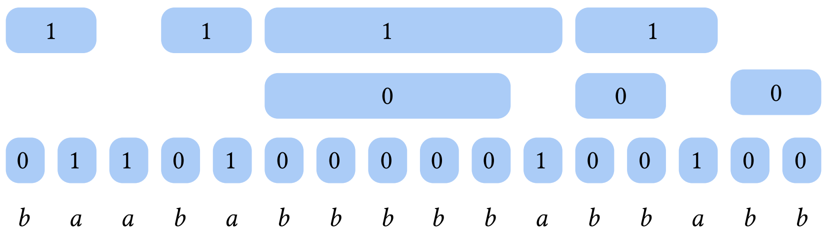
which counts the occurrences of a modulo 2. We begin by replacing each letter by its value in the monoid:



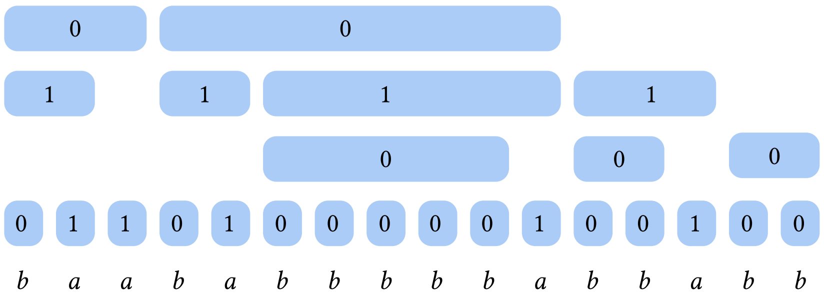
In this monoid, the only idempotent is 0, and therefore idempotent nodes can only be used to group intervals with value 0. This is what we do now: we consider groups of consecutive 0's which have length at least two. Each such group can be grouped into a single node, as in the following picture:



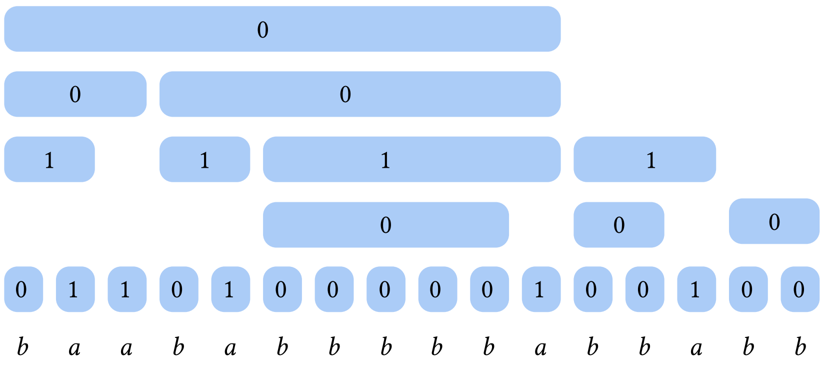
In the next step, we use binary nodes to group each 1 with the preceding block of 0's:



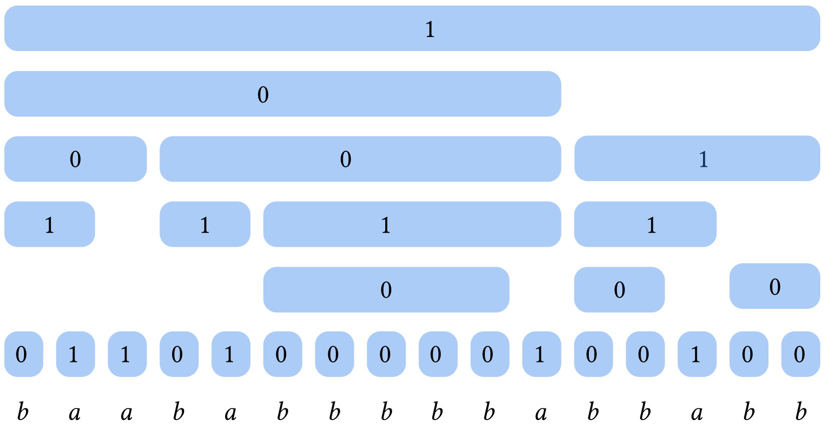
At this stage, the maximal intervals all have value 1, except for possibly a last interval with value 0. We group the maximal intervals with value 1 into pairs, so that they get value 0:



The intervals created in the previous step all have value 0, and therefore they can be grouped using an idempotent step into a single interval:

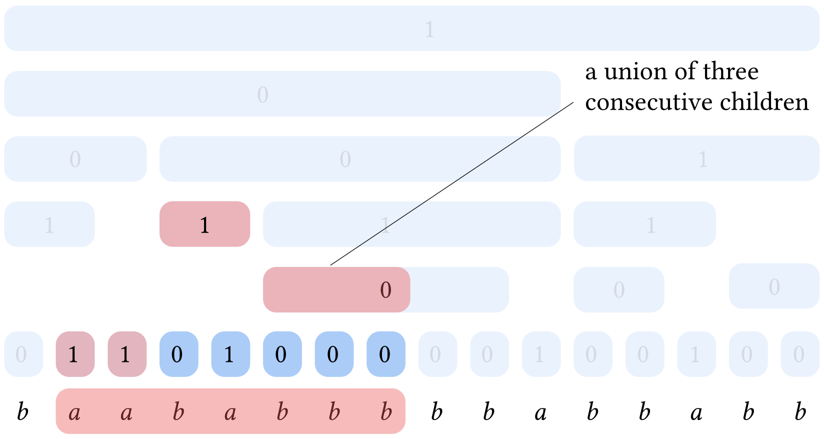


At this stage, we have at most three top level intervals: one big interval on the left that has value 0, possibly followed by intervals with values 1 and 0. The latter two can be joined without increasing the height of the tree, and then finally we can create a single root node:



All in all, we have built a Simon tree of height at most 6, for an arbitrary input word. □

The algorithmic application of a Simon tree is the same as for binary trees. Suppose that we have access to a Simon tree, and that we are given an interval query. Similarly to the binary case, the interval can be decomposed into intervals that correspond to the tree. The difference with respect to the binary tree is that now the decomposition can use two kinds of intervals: (a) nodes in the tree; or (b) unions of consecutive siblings in the tree. (In fact, the first kind is a special case of the second one.) The number of intervals in this decomposition is bounded by the height of the tree. Here is a picture:


a union of three consecutive children
1
0
0
0
1
1
1
1
1
0
0
0
0
1
1
0
1
0
0
0
0
0
1
0
0
1
0
0
b a a b a b b b b b a b b a b b

For the intervals that are in the tree, their value is stored in the tree. For the unions of consecutive children, the interesting case is when there are at least three of them. In this case, the values of these intervals must all be the same idempotent e, and therefore the value of their union will also be the same idempotent e. Summing up, using the Simon tree we can compute the value of each interval in time that is proportional to the height of the tree.
It remains to prove that Simon trees of bounded height exist.

Theorem 9.3 (Factorisation Forest Theorem, Simon [55]). *Consider a monoid homomorphism*

$$h : \Sigma^* \to M$$

into a finite monoid. There exists a constant $k \in \{1, 2, \ldots\}$ such that for every input word $w \in \Sigma^*$ there exists a Simon tree of height at most k.

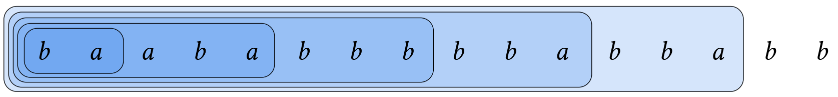
Proof. The proof will also come with an algorithm for constructing the tree in linear time, which is relevant for the algorithmic applications.
We work with Green's relations, which were introduced in Chapter 8.2. There will be three steps: (1) we first prove the theorem in the special case when the monoid is a group; (2) then we prove the theorem in the special case where all

intervals have a value in the same infix class; and (3) finally we prove the general case.

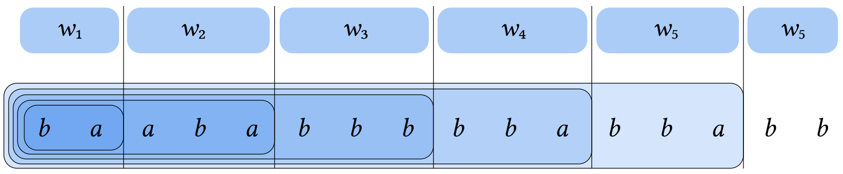
Step 1. Groups. We begin by proving the theorem in the special case when the monoid is a group. In this step, the Simon tree for an input word w is constructed based on the size of the set

$$P(w) = \{h(v) \mid \ v \text{ is a non-trivial prefix of } w \ \},$$

where a non-trivial prefix is defined to be one that is neither empty nor full. Since this set is a subset of the group, the height of the induction will be bounded by the size of the group. Each induction step will increase the height of the tree by at most 3, thus leading to a Simon tree whose height is at most 3 times the size of the group.
The induction base is when the set $P(w)$ is empty. This means that there are no non-trivial prefixes, i.e. the word has length at most one. In this case a trivial Simon tree with zero or one node suffices.
Consider now the induction step. Choose some element $g \in P(w)$. Consider all the non-trivial prefixes of the input word whose value is equal to g, as in the following picture:



We can cut the word along the ends of these prefixes, leading to a factorisation as in the following picture, with the corresponding infixes being called $w_1, \ldots, w_n$:



The following claim shows that we can apply the induction assumption to each of these infixes.

Claim 9.4. *For each $i \in \{1, \ldots, n\}$, the set $P(w_i)$ is strictly smaller than $P(w)$.*

Proof. For the first infix w_1, this is clearly true, since it is a prefix of w, and it was chosen so that it does not have any non-trivial prefixes with value g. For the remaining infixes, we have a similar situation, except that the group element needs to be prepended:

$$g \cdot P(w_i) \subseteq P(w) - \{g\}.$$

Since multiplying to the left by g does not change the size of a set, it follows that each $P(w_i)$ is strictly smaller than $P(w)$. ■

The Simon tree for w is constructed as follows. For each of the infixes $w_1, \ldots, w_n$, the above claim shows that we use the induction assumption to build a Simon tree. Then, we combine these trees three additional levels as follows. For each $i \in \{2, \ldots, n-1\}$, the value $h(w_i)$ must be the group identity, since it satisfies

$$g \cdot h(w_i) = g.$$

In particular, this value is an idempotent. Therefore, we can group the intervals corresponding to $w_2, \ldots, w_{n-1}$ using a single idempotent node. The first and last intervals w_1 and w_n can then be added using two binary nodes, thus adding three levels to the height of the tree from the induction assumption.

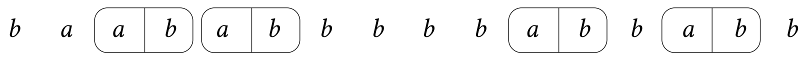
Step 2. Single infix class. Recall the notion of infix class that was used in Chapter 8.2. In this step, we consider the special case when the input word satisfies the following assumption:

(*) all nonempty intervals in the input word have their value in the same infix class, call it $J \subseteq M$.

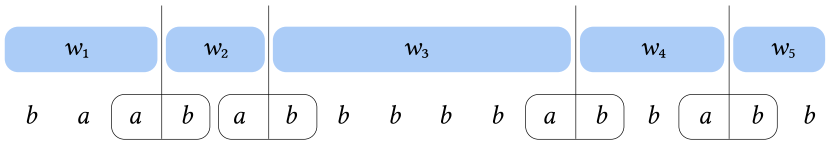
In this case, the proof is by induction on the number of windows of size two:

$$\{(a,b) \in M^2 \mid a \text{ and } b \text{ are values of two consecutive letters in } w \}$$

As previously, the induction basis is when there are no such windows, in which case the word has length at most one, and a trivial Simon tree suffices. Let us consider the induction step. Choose some window (a, b) in the input word. Consider all occurrences of this window, as in the following picture:



We can cut the word along the middle of each window, which leads to a factorisation as in the following picture:



Let $w_1, \ldots, w_n$ be the infixes which arise from the factorisation defined above. All of these infixes have a smaller induction parameter, since they avoid the window (a, b), and therefore we can find Simon trees for them. We now need to put them together. For this, the crucial observation is that if we ignore the first infix w_1 and the last infix w_n, then the remaining infixes have values in a group, and we can use the construction from the previous step. Indeed, each of the infixes $w_2, \ldots, w_{n-1}$ has b as a prefix and a as a suffix. The following claim shows that these infixes form a group.

Claim 9.5. *Let J be an infix class, and let $a, b \in J$ be such that $ba \in J$. Then*

$$G = \{m \in J \mid m \text{ has } b \text{ as a prefix and } a \text{ as a suffix} \}$$

is a group, with multiplication inherited from M, but with some possibly new choice of identity element.

Proof. We prove progressively finding more and more group structure in G.

1. **Sub-semigroup.** Let us first show that the set G is a sub-semigroup of M, i.e. it is closed under multiplication. Consider two elements $f, g \in G$. Since all elements in G are infix equivalent and a is a suffix of f, we can use Lemma 8.6 from the previous chapter to conclude that f is suffix equivalent to a. In particular, there is some $x \in M$ such that $x \cdot f = a$. Similarly, we can find some $y \in M$ such that $g \cdot y = b$. It follows that $f \cdot g$ belongs to J since it sits infix-wise between f and

   $$x \cdot f \cdot g \cdot y = a \cdot b \in J.$$

   Since $f \cdot g$ begins with b and ends with a, we conclude that it belongs to G.

2. **Idempotent.** We now show that G contains some idempotent. This follows from a more general result: every finite semigroup contains an idempotent. Indeed, we take any semigroup element $g \in G$, and consider its powers g^n for $n \in \{1, 2, \ldots\}$. By finiteness, there must be some repetition, i.e. n and $n + k$ will give the same power for some $k > 0$. This remains true if we increase n to any number in the set $\{n, n+1, n+2, \ldots\}$, and if we replace k by some multiple of k. Therefore, we can choose n and k so that they are equal to each other. As a result, g^n will be an idempotent.

3. **Identity element.** Let e be the idempotent found above. We will show that this is an identity in the semigroup G, which in particular implies that it is unique. Since e belongs to G, it has b as a prefix. Therefore, also b has e as a prefix, and by Lemma 8.6, and therefore every element of G has an e as a prefix. This implies that e is a left identity, since $e \cdot g$ holds for every element g that has e as a prefix, by idempotence of e. A symmetric argument shows that e is also a right identity.

4. **Inverses.** Finally, we show that every element of G has an inverse. Consider some $g \in G$. Since g is in the prefix class of e, as we have just

shown, it follows that there is some $x \in M$ such that $g \cdot x = e$. We can also improve this x so that it falls into G, by pre- and post-multiplying it with e. This gives us a right inverse for g. Similarly, we get a left inverse. Finally, in a group the left and right inverses must coincide, since

$$g_2 = e \cdot g_2 = \underbrace{g_1}_{\substack{\text{left}\\\text{inverse}}} \cdot g \cdot \underbrace{g_2}_{\substack{\text{right}\\\text{inverse}}} = g_1 \cdot e = g_1.$$

■

Thanks to the above claim, we can use the construction from Step 1 to combine the Simon trees for $w_2, \ldots, w_{n-1}$. The first and last infixes w_1 and w_n can then be added using two binary nodes, thus adding three levels to the height of the tree from the induction assumption.

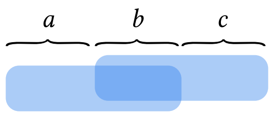
Step 3. General case. We now turn to the general case, where we do not have any assumptions on the input word. In the proof, we no longer distinguish between input letters and monoid elements. In other words, we think of the letters as being monoid elements, with the homomorphism being multiplication in the monoid. This is no different from the general case, since the only relevant information about a letter is its value under the homomorphism. The Simon tree is constructed by induction on the following parameter for a word $w \in M^*$:

$$\{m \in M \mid \text{ some letter in } w \text{ is an infix of } m \ \}.$$

Choose some infix class J that is represented by some letter in the input word, and choose this infix class to be minimal, i.e. there is no input letter that is a proper infix of J. Define a J-interval to be an interval that has value J. By minimality of J, we know that all sub-intervals in a J-interval are also J-intervals. The key observation is in the following claim.

Claim 9.6. *If two J-intervals overlap, then their union is a J-interval.*

Proof. Consider two overlapping J-intervals, and the following three elements of the monoid:



We know that all three monoid elements belong to the infix class J. By Lemma 8.6, we know that $a \cdot b$ is in the suffix class of b, and therefore there is some x such that $x \cdot a \cdot b = b$. It follows that $x \cdot a \cdot b \cdot c = b \cdot c$, and therefore $a \cdot b \cdot c$ belongs to the infix class J. ■

Thanks to the above claim, the maximal inclusionwise J-intervals are pairwise disjoint. In each such interval, we can use the construction from Step 2 to build a Simon tree, since all of its sub-intervals have value in J. We can now collapse these maximal intervals into single letters, and as a result they will have one letter only. These letters can then be paired with the next letter using binary nodes, and as a result there is no longer any occurrence of the infix class J in the input word, and only bigger infix classes remain. The induction assumption can then be used to build a Simon tree for the remaining word. ■

10
Determinisation of ω-automata

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

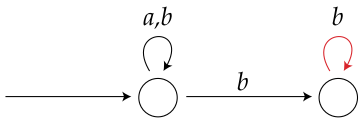
$$a_1 a_2 a_3 \cdots$$

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 [58].

10.1 Automata models for ω-words

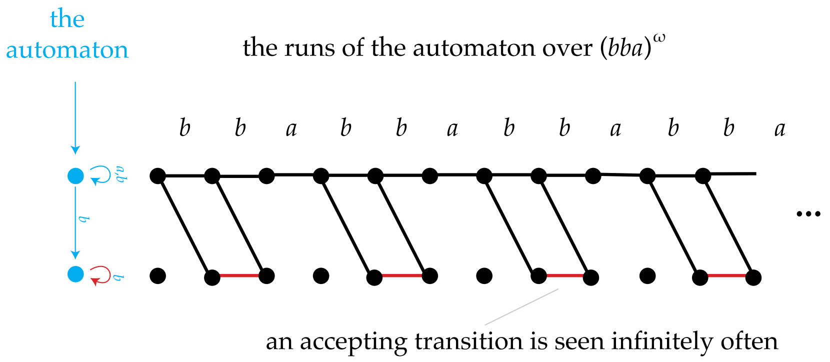
A *nondeterministic Büchi automaton* is a type of automaton for ω-words. Its syntax is typically defined to be the same as that of a nondeterministic finite automaton: a set of states, an input alphabet, initial and accepting subsets of states, and a set of transitions. For our presentation it is more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

Example 36. 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 36:


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

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

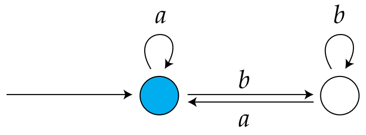
Fact 10.1. *Nondeterministic Büchi automata recognise strictly more languages than deterministic Büchi automata.*

Proof. Take the automaton from Example 36. 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. ■

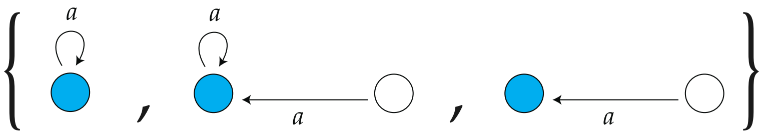
The above fact shows that if we want to determinise automata for ω-words, we need something more powerful than the Büchi condition. One solution is called the Muller condition, and is described below. Later we will see another (equivalent) solution, which is called the parity condition.

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

Example 37. Consider this automaton



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



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

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

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

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

Theorem 10.2 (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 60. It follows from the above discussion that

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

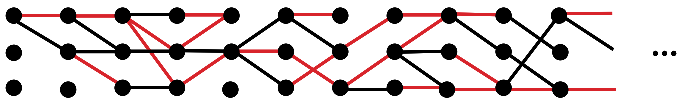
have the same expressive power, but deterministic Büchi automata are weaker. Theorem 10.2 was first proved by McNaughton in [38]. The proof here is

similar to one by Muller and Schupp [41]. An alternative proof method is the Safra Construction, see e.g. [58].
The proof strategy is as follows. We first define a family of languages, called universal Büchi languages, and show that the McNaughton's theorem boils down to recognising these languages with deterministic Muller automata. Then we do that.

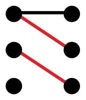
The universal Büchi language. For $n \in \mathbb{N}$, define a width n dag to be a directed acyclic graph where the nodes are pairs $\{1, \ldots, n\} \times \{1, 2, \ldots\}$ and every edge is of the form

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

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



In the pictures, we adopt the convention that the i-th column stands for the set of vertices $\{1, \ldots, n\} \times \{i\}$. The top left corner of the picture, namely the vertex $(1, 1)$ is called the *initial vertex*.
The essence of McNaughton's theorem is showing that for every n, there is a deterministic Muller automaton which inputs a width n dag and says if it contains a path that begins in the initial vertex and visits infinitely many red (accepting) edges. In order to write such an automaton, we need to encode as a width n dag as an ω-word over some finite alphabet. This is done using an alphabet, which we denote by $[n]$, where the letters look like this:



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

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

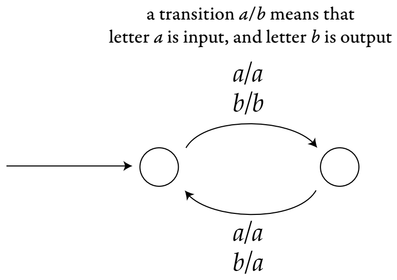
Define the *universal n state Büchi language* to be the set of words $w \in [n]^\omega$ which, when treated as a width n dag, contain a path that starts in the initial vertex and visits accepting edges infinitely often. The key to McNaughton's theorem is the following proposition.

Proposition 10.3. *For every $n \in \mathbb{N}$ there is a deterministic Muller automaton recognising the universal n state Büchi language.*

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

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

Example 38. Here is a picture of a sequential transducer which inputs a word over $\{a, b\}$ and replaces letters on even-numbered positions by a.


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

□

Lemma 10.4. *Languages recognised by deterministic Muller automata are closed under inverse images of sequential letter-to-letter transducers, i.e. if $\mathcal{A}$ in the diagram below is a deterministic Muller automaton and f is a sequential transducer, there is a deterministic Muller automaton $\mathcal{B}$ which makes the following diagram commute:*

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

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

$$p \overset{a/b}{\to} p',$$

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

$$q \overset{b}{\to} q'.$$

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

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

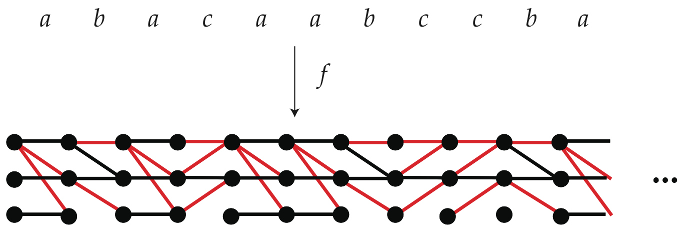
Note how each transition in $\mathcal{B}$ corresponds to two transitions, one in f and one in $\mathcal{A}$. The Muller condition is inherited from the automaton $\mathcal{A}$, i.e. a set of transitions in $\mathcal{B}$ is accepting if the corresponding set of transitions in $\mathcal{A}$ is accepting.

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

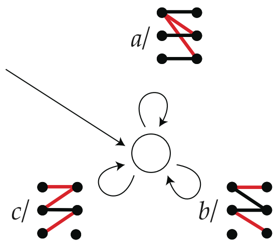
Let us continue with the proof of McNaughton's theorem. We claim that every language recognised by a nondeterministic Büchi automaton reduces to a universal Büchi language via some transducer. Let $\mathcal{A}$ be a nondeterministic Büchi automaton with input alphabet Σ. We assume without loss of generality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

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

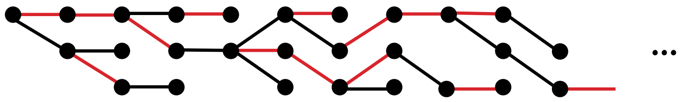
such that a word $w \in \Sigma^\omega$ is accepted by $\mathcal{A}$ if and only if $f(w)$ contains a path from the initial vertex with infinitely many accepting edges. Here is a picture:



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



Using Lemma 10.4, we compose the transducer with the automaton from Proposition 10.3, getting a deterministic Muller automaton equivalent to $\mathcal{A}$. It now remains to show the proposition, i.e. that the n state universal Büchi language can be recognised by a Muller automaton. The proof has two steps. The first step is stated in Lemma 10.5 and says that a deterministic transducer can replace an arbitrary width n dag by an equivalent tree. Here we use the name *tree* for a width n dag, where every non-isolated node other than (1,1) has exactly one incoming edge. Here is a picture of such a tree, with the isolated nodes not drawn:



Lemma 10.5. *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 10.6. *There exists a deterministic Muller automaton with input alphabet $[n]$ such that for every $w \in [n]^\omega$ that is a tree, the automaton accepts w if and only if w contains a path from the root with infinitely many accepting edges.*

Combining the two lemmas using Lemma 10.4, we get Proposition 10.3, and thus finish the proof of McNaughton's theorem. Lemma 10.5 is proved in Section 10.2 and Lemma 10.6 is proved in Section 10.3.

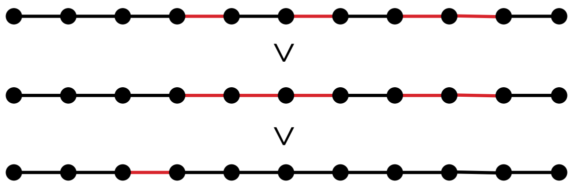
10.2 *Pruning the graph of runs to a tree*

We begin by proving Lemma 10.5, which says that a sequential transducer can convert a width n dag into a tree, while preserving the existence of a path from the initial vertex with infinitely many accepting edges. The transducer is simply going to remove edges.

Profiles. For a path π in a width n dag, define its *profile* to be the word of same length over the two-letter alphabet

$$\{\text{accepting}, \text{non-accepting}\}$$

which is obtained by replacing each edge with its appropriate type. We order profiles lexicographically, with "accepting" smaller than "non-accepting".



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

Lemma 10.7. *There is a sequential transducer*

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

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

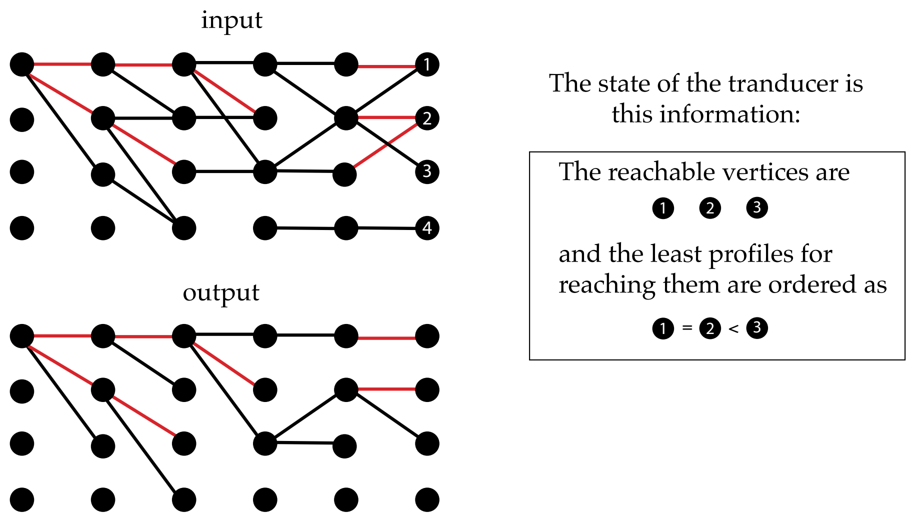
Proof. The key observation is that the prefix of a profile optimal path is also profile optimal. Therefore, if we want to do find a profile optimal path that leads to a vertex (q, i), we need to do the following. Consider all paths from the initial vertex to (q, i), decomposed as $\pi \cdot e$ where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

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

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

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

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



The state of the tranducer is this information:

The reachable vertices are

❶ ❷ ❸

and the least profiles for reaching them are ordered as

❶ = ❷ < ❸

■

Lemma 10.8. *Let f be the sequential transducer from Lemma 10.7. If the input to f contains a path with infinitely many accepting edges, then so does the output.*

Proof. Assume that the dag w, which is an input for the transducer f, contains a path with infinitely many accepting edges. We use the name *accepting path* for such a path. Our goal is to show that the tree $f(w)$ also contains an accepting path.

For $i \in \{0, 1, \dots\}$, define P_i to be the length i prefixes of profiles of accepting paths in the dag w. We know that this set is nonempty, since there is an accepting path. Let p_i be the lexicographically minimal element of P_i. As defined, the profile p_i is the profile of some finite path in the original run dag, before pruning it to a tree. However, because the pruned tree $f(w)$ stores paths with optimal profiles, it follows that for every i, the tree $f(w)$ has some path with profile p_i.

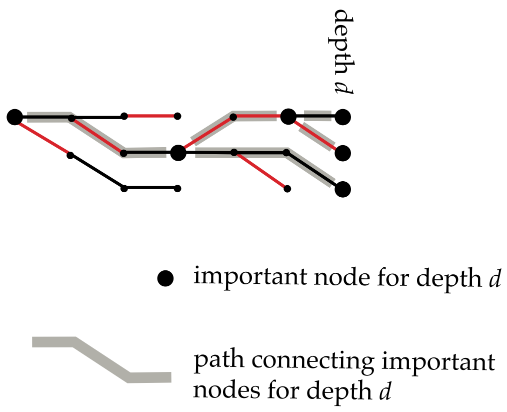
Using the definition of the lexicographic ordering, one can see that p_i is a prefix of p_j when $i < j$. Therefore, the profiles p_i have some infinite limit, call it p. We will now show that the pruned tree $f(w)$ contains an infinite path with the limit profile p. We will do this using the König lemma, which says that every finitely branching tree with arbitrarily long paths contains an infinite path. Indeed, as we have argued in the previous paragraph, the pruned tree $f(w)$ contains paths with every profile p_i. Therefore, if we prune it even more, so that it only contains paths consistent with the profile p, we will get a finitely branching tree, which has arbitrarily long paths. Therefore, by the König lemma, it contains some infinite path.

It remains to prove that the limit profile p has infinitely many accepting edges, and therefore the infinite path from the previous paragraph is accepting. We will show that for every i, the limit profile contains an accepting edge which is later than i. Indeed, consider the profile p_i. By definition, we know that this profile can be extended to the profile of some accepting path in the original run dag w. This accepting path must use some accepting edge after position i. Therefore, there is some $j > i$ such that P_j contains a profile that extends p_i, and has one more accepting edge. This means that the minimal profile p_j, which extends p_i, also has at least one more accepting edge than p_i. ∎

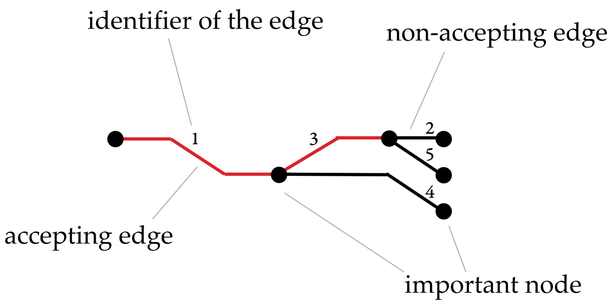
10.3 *Finding an accepting path in a tree graph*

We now show Lemma 10.6, which says that a deterministic Muller automaton can check if a width n tree contains a path with infinitely many accepting edges.

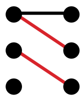
Consider a tree $t \in [n]^\omega$, and let $d \in \mathbb{N}$ be some depth. Define an *important node for depth d* to be a node which is either: the root, a node at depth d, or a node which is a closest common ancestor of two nodes at depth d. This definition is illustrated below (with red lines representing accepting edges, and black lines representing non-accepting edges):


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

Definition of the Muller automaton. We now describe the Muller automaton for Lemma 10.6. After reading the first d letters of an input tree (i.e. after reading the input tree up to depth d), the automaton keeps in its state a tree, where the nodes correspond to nodes of the input tree that are important for depth d, and the edges correspond to paths in the input tree that connect these nodes. This tree stored by the automaton is a tree with at most n leaves, and therefore it has less than $2n$ edges. The automaton also keeps track of a colouring of the edges, with each edge being marked as accepting or not, where "accepting" means that the corresponding path in the input tree contains at least one accepting edge. Finally, the automaton remembers for each edge an identifiers from the set $\{1, \ldots, 2n-1\}$, with the identifier policy being described below. A typical memory state looks like this:

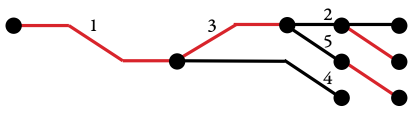


The big black dots correspond to important nodes for the current depth, red edges are accepting, black edges are non-accepting, while the numbers are the identifiers. All identifiers are distinct, i.e. different edges get different identifiers. It might be the case (which is not true for the picture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

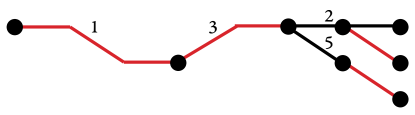


To define the new state, perform the following steps.

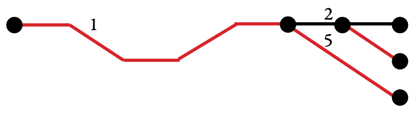
1. Append the new letter to the tree in the state of the automaton. In the example of the tree and letter illustrated above, the result looks like this:



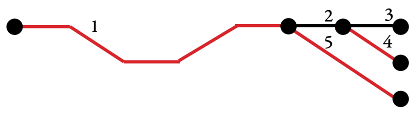
2. Eliminate paths that die out before reaching the new maximal depth. In the above picture, this means eliminating the path with identifier 4:



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



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



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

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

1. **Delete.** An edge can be deleted in step 2 or in step 3 (by being merged with an edge closer to the root). The identifier of such an edge is said to

be deleted in the transition. Since we reuse identifiers, an identifier can still be present after a transition that deletes it, because it has been added again in step 4, e.g. this happens to identifier 4 in the above example.

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

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

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

Lemma 10.9. *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 10.6. 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 10.6.

Proof of Lemma 10.9. The implication from (b) to (a) is straightforward. An identifier in the state of the automaton corresponds to a finite path in the input tree. If the identifier is not deleted, then this path stays the same or grows to the right (i.e. something is appended to the path). When the identifier is refreshed, the path grows by at least one accepting edge. Therefore, if the identifier is deleted finitely often and refreshed infinitely often, there is some path that keeps on growing with more and more accepting states, and its limit is a path with infinitely many accepting edges.

Let us now focus on the implication from (a) to (b). Suppose that the tree t contains some infinite path π that begins in the root and has infinitely many accepting edges. Call an identifier *active* in step d if the path described by this identifier in the d-th state of the run corresponds to an infix of the path π. Let I be the set of identifiers that are active in all but finitely many steps, and which are deleted finitely often. This set is nonempty, e.g. the first edge of the path π always has the same identifier. In particular, there is some step d, such that identifiers from I are not deleted after step n. Let $i \in I$ be the identifier that is last on the path π, i.e. all other identifiers in I describe finite paths that are earlier on π. It is not difficult to see that the identifier i must be refreshed infinitely often by prefixes of the path π. ■

Problem 54. Are the following languages ω-regular (i.e. recognised by nondeterministic Büchi automata)?

1. ω-words which have infinitely many prefixes in a fixed regular language of finite words $L \subseteq \Sigma^*$;
2. ω-words with infinitely many infixes of the form $ab^p a$, where p is prime;
3. ω-words with infinitely many infixes of the form $ab^n a$, where n is even.

Problem 55. Call an ω-word *ultimately periodic* if it is of the form uv^ω for some finite words u, v. Show that if an ω-regular language is nonempty, then it contains an ultimately periodic word.

Problem 56. Let UP be the set of ultimately periodic words. Let K and L be ω-regular languages. Show that if $L \cap UP = K \cap UP$ then $K = L$.

Problem 57. Are the following languages ω-regular?

1. ω-words with arbitrarily long infixes belonging to a fixed regular language of finite words L;

2. ω-words which have infinitely many prefixes in a fixed language of finite words $L \subseteq \Sigma^*$ (not necessarily regular).

Problem 58. Show that the language of words "there exists a letter b" cannot be accepted by a nondeterministic automaton with the Büchi acceptance condition, where all the states are accepting (but possibly transitions over some letters in some states are missing).

Problem 59. Show that the language "finitely many occurrences of letter a" cannot be accepted by a deterministic automaton with the Büchi acceptance condition.

Problem 60. Show that every language accepted by a nondeterministic automaton with the Muller acceptance condition is also accepted by some nondeterministic automaton with the Büchi acceptance condition.

Problem 61. Show that nonemptiness is decidable for automata with the Muller acceptance condition.

Problem 62. Define a metric on ω-words by

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

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

1. is open;

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

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

Problem 63. Which of the following candidates for a Myhill-Nerode congruence indeed have the property: $\sim_L$ has finite index if and only if L is ω-regular

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

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

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

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

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

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

11
Infinite duration games

In this chapter, we prove the Büchi-Landweber Theorem [14, Theorem 1], see also [58, Theorem 6.5], which shows how to solve games with ω-regular winning conditions. These are games where two players move a token around a graph, yielding an infinite path, and the winner is decided based on some property of this path that is recognised by an automaton on ω-words. The Büchi-Landweber Theorem gives an algorithm for deciding the winner in such games, thus answering a question posed in [18] and sometimes called "Church's Problem".

11.1 Games

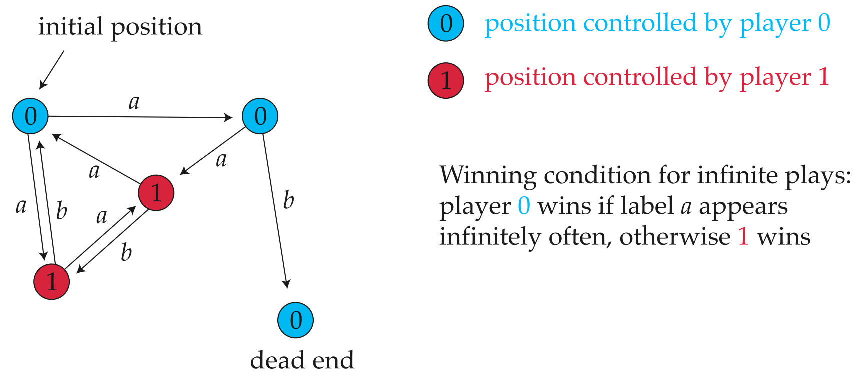
In this chapter, we consider games played by two players (called 0 and 1), which are zero-sum, perfect information, and most importantly, of potentially infinite duration.

Definition 11.1 (Game). *A* game *consists of*

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

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

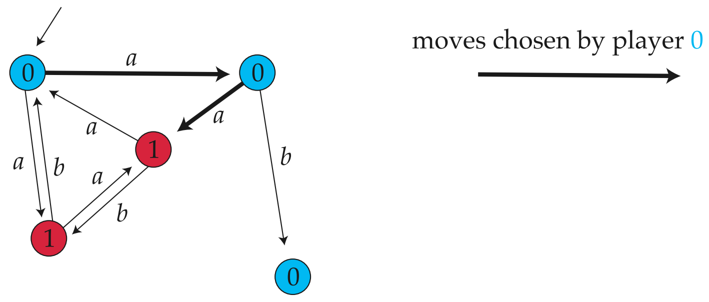
Intuitively speaking, the winning condition inputs a sequence of labels produced in an infinite play, and says which player wins. The definition is written in a way which highlights the symmetry between the two players; this symmetry will play an important role in the analysis. Here is a picture.


initial position
0
0
1
1
0
a
a
a
b
a
b
a
b
dead end
0 position controlled by player 0
1 position controlled by player 1
Winning condition for infinite plays: player 0 wins if label a appears infinitely often, otherwise 1 wins

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

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

Example 39. In the game from the picture above, player 0 has a winning strategy, which is to always select the fat arrows in the following picture.



□

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

Definition 11.2 (Finite memory strategy). *Consider a game where the positions are V. Let i be one of the players. A strategy for player i with memory M is given by:*

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

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

We will apply the above definition to games with possibly infinitely many positions, but we only care about finite memory sets M. An important special case is when the set M has only one element, in which case the strategy is called *memoryless*. For a memoryless strategy, the new position chosen by the player only depends on the current position, and not on the history of the game before that. The strategy in Example 39 is memoryless.

Theorem 11.3 (Büchi-Landweber Theorem). *Let Σ be finite and let*

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

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

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

Definition 11.4 (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 11.5 (Memoryless determinacy of parity games)**.** *For every parity game, one of the players has a memoryless winning strategy.*

In fact, for edge labelled games (which is our choice) the parity condition is the only condition that admits memoryless winning strategies regardless of the graph structure of the game, among conditions that are prefix independent, see [20, Theorem 4].
The above theorem is proved in Section 11.2. The second step of the Büchi-Landweber theorem is a reduction to parity games. This essentially boils down to transforming deterministic Muller automata into deterministic parity automata, which are defined as follows: a parity automaton has a ranking function from states to numbers, and a run is considered accepting if the smallest rank appearing infinitely often is even. This is a special case of the Muller condition, but it turns out to be expressively complete in the following sense:

Lemma 11.6. *For every deterministic Muller automaton, there is an equivalent deterministic parity automaton.*

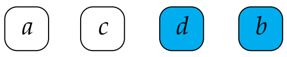
Proof. The lemma can be proved in two ways. One way is to show that, by taking more care in the determinisation construction in McNaughton's Theorem, we can actually produce a parity automaton. Another way is to use a data structure called the later appearance record [32]. The construction is presented in the following claim.

Claim 11.7. *For every finite alphabet Σ, there exists a deterministic automaton with input alphabet Σ, a totally ordered state space Q, and a function*

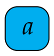
$$g : Q \to \mathcal{P}(\Sigma)$$

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

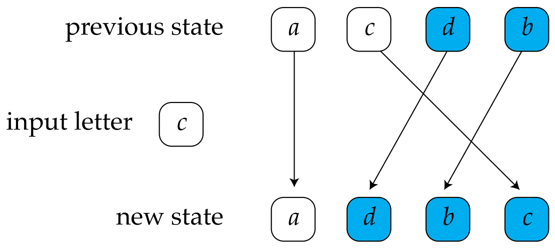
Proof. The state space Q consists of data structures that look like this:



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



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



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

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

Let us now finish the proof of the Büchi-Landweber theorem. Consider a game with an ω-regular winning condition. By Lemma 11.6, there is a deterministic parity automaton which accepts exactly those sequences of edge labels where player 0 wins. Consider a new game, call it the *product game*, where the positions are pairs (position of the original game, state of the deterministic parity automaton). Edges in the product game are of the form

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

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

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

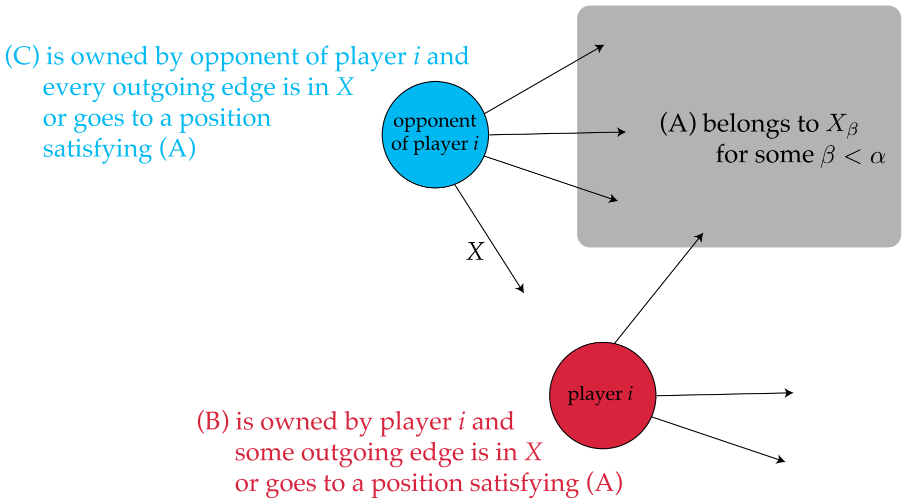
2. player i wins from position (v, q) in the product game, where q is the initial state of the deterministic parity automaton recognising L.

The implication from 1 to 2 crucially uses determinism of the automaton and would fail if a nondeterministic automaton were used (under an appropriate definition of a product game). Since the product game is a parity game, Theorem 11.5 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.

11.2 *Memoryless determinacy of parity games*

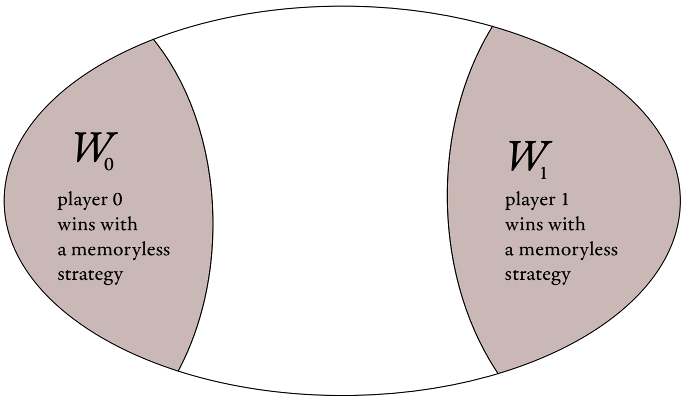
In this section, we prove Theorem 11.5 on memoryless determinacy of parity games. The proof we use is based in [64] and [58]. Recall that in a parity game, the positions are assigned numbers (called ranks from now on) from a finite set of natural numbers, and the goal of player i is to ensure that for infinite plays, the minimal number appearing infinitely often has parity i. Our goal is to show that one of the players has a winning strategy, and furthermore this strategy is memoryless. The proof of the theorem is by induction on the number ranks used in the game. We choose the induction base to be the case when there are no ranks at all, and hence the theorem is vacuously true. For the induction step, we use the notion of attractors, which is defined below.

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


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

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

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

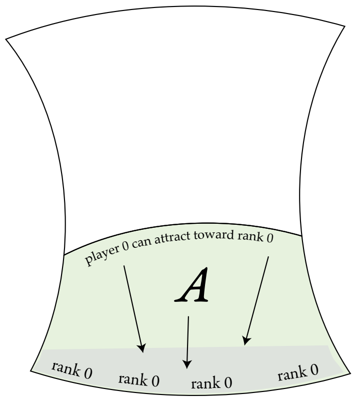


By definition, for every position in $w \in W_i$, player i has a memoryless winning strategy that wins when starting in position w. In principle, the memoryless strategy might depend on the choice of w, but the following lemma shows that this is not the case.

Lemma 11.8. *Let $i \in \{0, 1\}$ be one of the players. There is a memoryless strategy σ_i for player i, such that if the game starts in W_i, then player i wins by playing σ_i.*

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

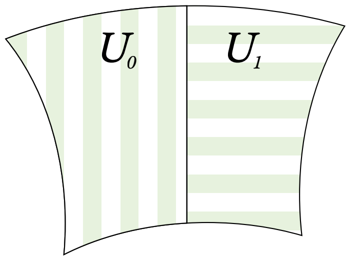
Define the game restricted to U to be the same as the original game, except that we only keep positions from U. In general restricting a game to a subset of positions might create new dead ends. However, in this particular case, no new dead ends will be created: if a position controlled by player i has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



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

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

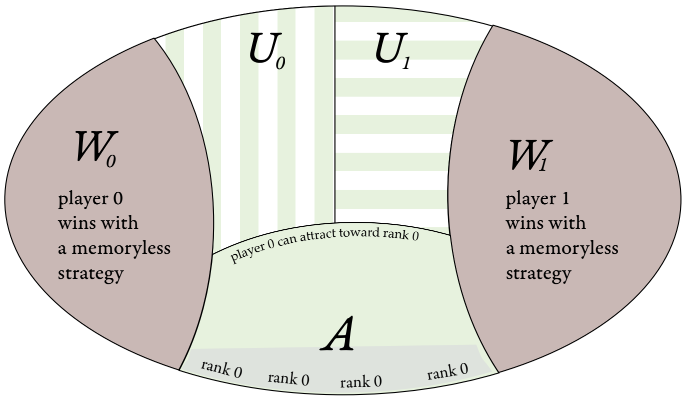
Consider the following game H: we restrict the original game to positions from $U - A$, and remove all edges which have minimal rank 0 (these edges necessarily originate in positions controlled by player 1, since otherwise they would be in A). Since this game does not use rank 0, the induction assumption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



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

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

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


U_0
U_1
W_0
player 0 wins with a memoryless strategy
W_1
player 1 wins with a memoryless strategy
player 0 can attract toward rank 0
A
rank 0
rank 0
rank 0
rank 0

Lemma 11.9. *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 11.8;
- in other positions do whatever.

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

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

- in U_0 use the winning memoryless strategy from the game H;
- in W_0 use the winning memoryless strategy from Lemma 11.8;
- in A use the attractor strategy to reach a rank 0 edge inside U;
- on other positions, i.e. on W_1, do whatever.

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

Problem 64. We say that a game is *finite* if it has no infinite plays, i.e. every play eventually reaches a dead end. Prove that every finite game is determined, i.e. exactly one of the players has a winning strategy.

Problem 65. Show that reachability games played on finite game graphs can be solved in time proportional to the number of edges.

Problem 66. Show that one player parity games can be solved in PTIME.

Problem 67. Show that solving parity games is in NP ∩ coNP.

Problem 68. Consider the following game on a finite game graph V together with function $rank : V \to \mathbb{N}$. At every moment of the play, the owner of the current vertex chooses a next vertex among current vertex successors. This continues until some vertex repeats on the play, i.e. till the first loop is closed. Then depending on the parity of the smallest rank on the loop the winning player is determined. Prove that player i in the described game wins iff player i wins in the parity game on the same arena.

Problem 69. Are Muller games positionally determined?

Problem 70. Show that Büchi games are positionally determined without direct use of the same result for parity games.

Problem 71. Show that the winning condition Muller games is a Borel set, and therefore Muller games are determined by Martin's theorem. (Most of this problem is looking up what Borel sets and Martin's theorem are.)

Problem 72. Show that Muller games on finite arenas are not positionally determined.

Problem 73. Construct an infinite game played on a finite game graph, in which player 0 has a winning strategy, but not a winning finite memory strategy. *Remark:* Notice that by Büchi-Landweber theorem the winning condition in that game cannot be ω-regular.

Problem 74. Consider the following riddle. There are infinitely many dwarfs (countably many). Every dwarf is given a hat, which is either red or green. Every dwarf sees the color of every hat beside his own one. Every dwarf is supposed to tell what is the color of his hat, such that only finitely many dwarfs make a mistake. They can fix a strategy in advance, before getting their hats, but they cannot communicate after getting their hats. Find a winning strategy for dwarfs. *Remark:* Problems 74, 75 and 76 serve as a preparation for the Problem 77.

Problem 75. Show that there is a function inf-xor : $\{0,1\}^\omega \to \{0,1\}$, such that changing one bit of an argument always changes the result. (The solution uses the axiom of choice.)

Problem 76. Consider the following two player game, called Chomp. There is a rectangular chocolate in a shape of $n \times k$ grid. The right upper corner piece is rotten. Players move in an alternating manner, the first one moves first. Any player in his move picks square of the chocolate that is not yet eaten, and eats all pieces that are to the left and to the bottom from the picked piece. The player who eats the rotten piece loses. Determine who has a winning strategy.

Problem 77. Show a game that is not determined.

Problem 78. Consider the following *bisimilarity game* played on a finite game graph with vertices V equipped with a function $rank : V \to \mathbb{N}$. Two players, *Spoiler* and *Duplicator* start from a position $(u, v) \in V \times V$. The play proceeds in rounds. If at the beginning of a round $rank(u) \neq rank(v)$ or u and v belong to different players then Spoiler immediately wins. Otherwise Spoiler makes a move to (u', v) or (u, v') such that $u \to u'$ or $v \to v'$, respectively. Then Duplicator makes a move to (u', v') such that $v \to v'$ or $u \to u'$, respectively. Next round starts from (u', v'). If play continues infinitely long then Duplicator wins. Show that if Duplicator has a winning strategy from position (u, v) then the same player has a winning strategy in the parity game starting from u and in the parity game starting in v.

12
Parity games in quasipolynomial time

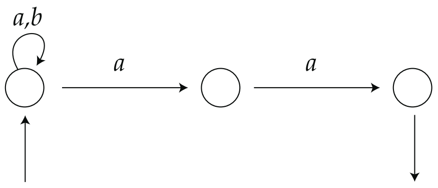
In this chapter, we show the following result.

Theorem 12.1. *Parity games with n positions and d ranks can be solved in time $n^{\mathcal{O}(\log d)}$.*

The time in the above theorem is a special case of *quasipolynomial time* mentioned in the title of the chapter. Whether or not parity games can be solved in time which is polynomial in both n and d is an important open problem. The presentation here is based on the original paper [15], with some new terminology (notably, the use of separation).
Define a *reachability* game to be a game where the objective of player 0 is to visit an edge from a designated subset. (We assume that the designated subset contains all edges pointing to dead ends of player 1, so that winning by reaching a dead end is subsumed by reaching designated edges.) Reachability games can be solved in time linear in the number of edges, as is shown in Exercise 64. Our proof strategy for Theorem 12.1 is to reduce parity games to reachability games of appropriate size.

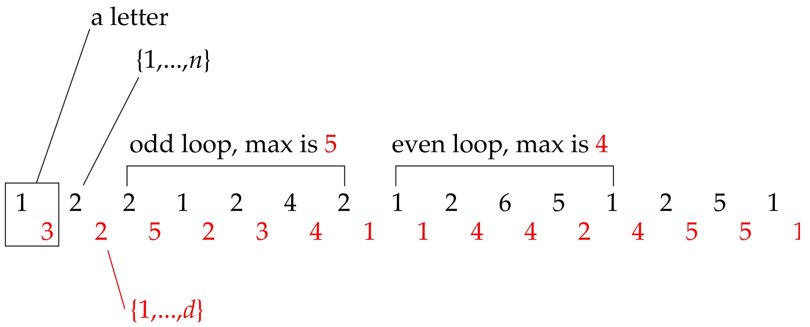
12.1 Reduction to reachability games

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



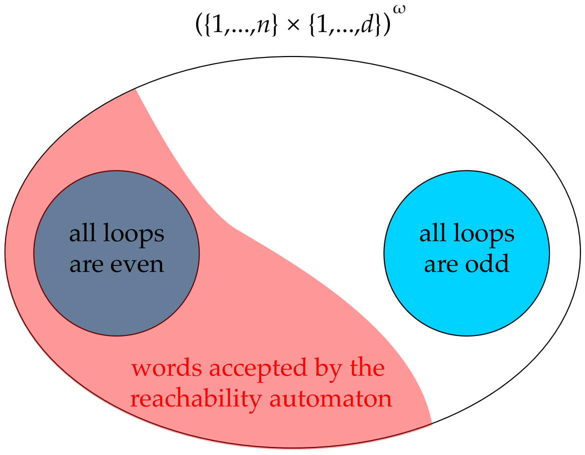
accepts all ω-words over alphabet $\{a, b\}$ which contain two consecutive a's. A reachability automaton is called deterministic if its transition relation is a function.

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



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

Lemma 12.2. *Let $n, d \in \{1, 2, \ldots\}$. Assume that one can compute a deterministic reachability automaton $\mathcal{D}$ with input alphabet $\{1, \ldots, n\} \times \{1, \ldots, d\}$ that accepts every ω-word where all loops are even, and rejects every ω-word where all loops are odd, as in the following picture:*


({1,...,n} × {1,...,d})^ω
all loops are even
all loops are odd
words accepted by the reachability automaton

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

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

Proof. Let G be a parity game with vertices $\{1, \ldots, n\}$ and edges labelled by parity ranks $\{1, \ldots, d\}$. Let $\mathcal{D}$ be an automaton as in the assumption of the lemma. Consider a product game $G \times \mathcal{D}$, as defined on page 149, i.e. the positions are pairs (position of v, state of $\mathcal{A}$) and the structure of the game is inherited from G with only the states being updated according to the parity ranks on edges. Player 0 wins the product game $G \times \mathcal{D}$ if a dead end of player 1 is reached, or if the play is infinite and accepted by $\mathcal{D}$ (in the latter case, by the assumption that $\mathcal{D}$ is a reachability automaton, this is done by reaching an accepting state of $\mathcal{D}$ at some point during the play).

Claim 12.3. *If player $i \in \{0, 1\}$ wins G, then player i also wins $G \times \mathcal{D}$.*

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

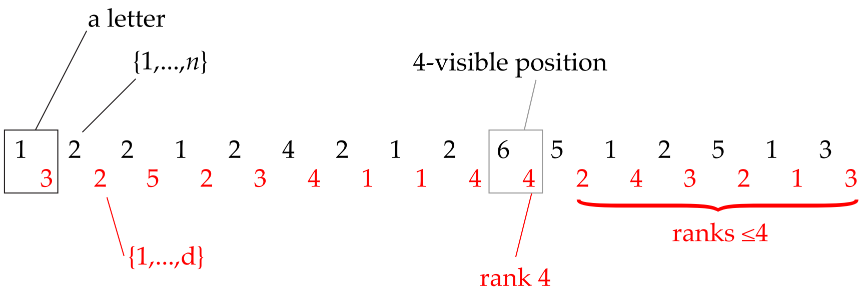
Because $\mathcal{D}$ is a reachability automaton, the product game $G \times \mathcal{D}$ can be solved in time proportional to the number of its edges, which is consistent with the bound in the lemma. ■

12.2 *A small reachability automaton for loop parity*

By Lemma 12.2, to prove Theorem 12.1, it suffices to find a deterministic automaton which separates "all loops even" from "all loops odd", and which has a quasipolynomial state space (and time to compute the automaton). As a warm-up, we present a simpler construction which has $n^{d/2}$ states.

Fact 12.4. *Let $n, d \in \{1, 2, \dots\}$. There is a deterministic reachability automaton with $n^{d/2}$ states which satisfies the properties in Lemma 12.2.*

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



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

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

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

Note that in the above construction, the automaton satisfies a stronger property than required by Lemma 12.2, namely it accepts all words satisfying the parity condition (instead of only those where all loops are even).

Lemma 12.5. *Let $n,d \in \{1,2,\ldots\}$. There is deterministic reachability automaton with $n^{\mathcal{O}(\log d)}$ states which satisfies the assumptions of Lemma 12.2.*

The rest of Section 12.2 is devoted to proving the above lemma. Like in the construction with $n^{d/2}$ states, the automaton will reject all words which violate the parity condition, and not just those where all loops are odd. This stronger property, however, is not used in the proof of Theorem 12.1.

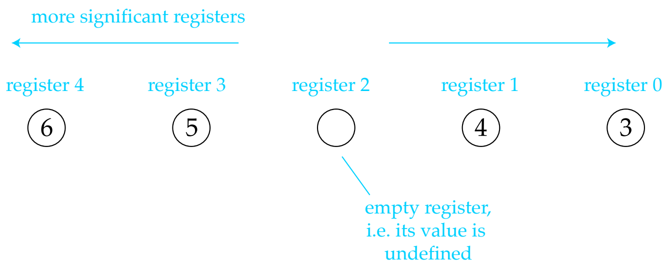
We begin with a *nondeterministic* reachability automaton $\mathcal{A}$ which satisfies the properties in the lemma in the following sense: if all loops are even, then at least one run reaches an accepting state, and if all loops are odd, then all runs avoid accepting states.

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

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

and therefore the number of states in $\mathcal{A}$ is at most $n^{\mathcal{O}(\log d)}$. Our final automaton $\mathcal{D}$ will be obtained by keeping the same states as $\mathcal{A}$ and removing transitions so as to make the automaton deterministic, and hence the size of $\mathcal{D}$ will be as required to make Theorem 12.1 true.

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

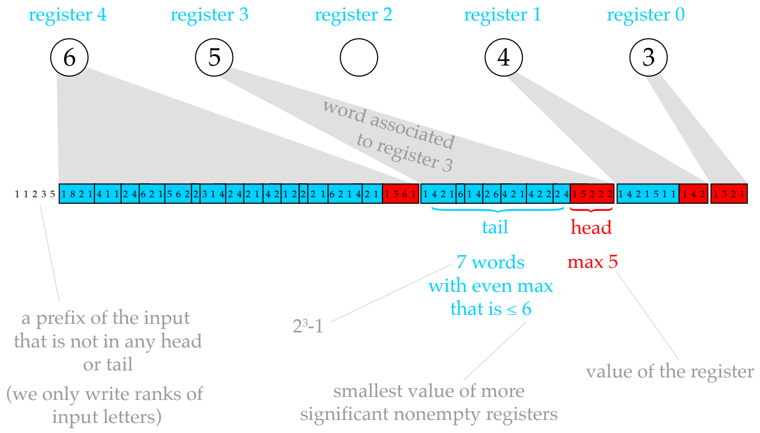

more significant registers
register 4
register 3
register 2
register 1
register 0
6
5
4
3
empty register, i.e. its value is undefined

We design the transition relation to respect following invariant.

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

1. if r is empty then w_r is empty; and
2. the word $w_{k-1}w_{k-2}\cdots w_1w_0$ is a suffix of the input read so far; and
3. if a register r is nonempty and stores $i \in \{1,\dots,d\}$, then:
   (a) all words associated to nonempty registers $< r$ use ranks $\leq i$;
   (b) the word w_r associated to r is a concatenation of two words:
      - TAIL: $2^r - 1$ words with even maximal rank;
      - HEAD: a word with maximal rank exactly i.

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



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

Lemma 12.6. *Assume that the invariant is satisfied, all registers are nonempty and store even ranks, and the input letter has even rank. Then the input contains an even loop.*

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

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

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

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

Lemma 12.7. *Emptying any subset of the registers preserves the invariant.*

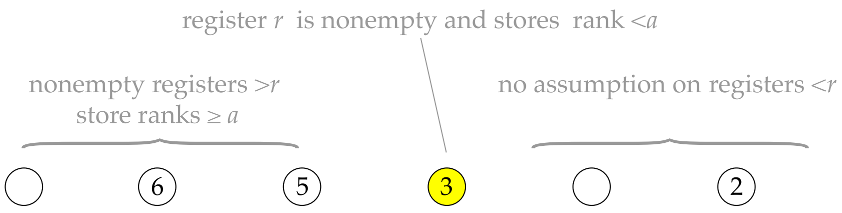
Proof. It is enough to show that emptying any single register r preserves the invariant. If r is the most significant nonempty register, then the word associated to r is put into the prefix of the input that is not assigned to any register. Otherwise, the word associated to r is appended to the head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

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

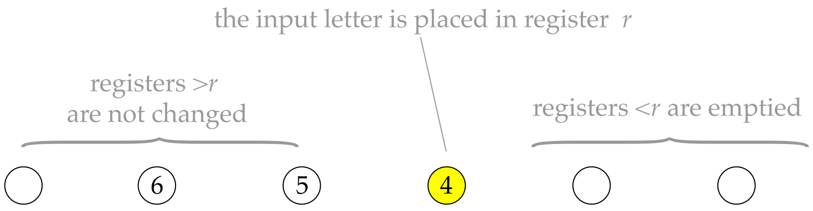
A. Assume that in the current state, all registers store values $\geq a$, which includes the special case when all registers are empty. Under this assumption, the automaton is allowed to do nothing, i.e. not change the state when reading a.

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

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

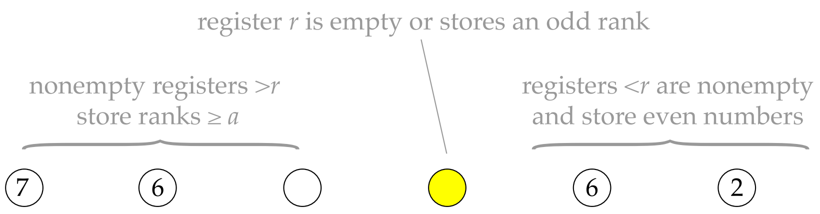

register r is nonempty and stores rank $<a$
nonempty registers $>r$ store ranks $\geq a$
no assumption on registers $<r$
6
5
3
2

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


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

Why the invariant is preserved. We view this transition as a two-step process. First, all registers $< r$ are made empty, which preserves the invariant by Lemma 12.7. Next, the input letter a is appended to the head of the register r (which is now the least significant nonempty register), and therefore becomes the new maximum in this head.

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

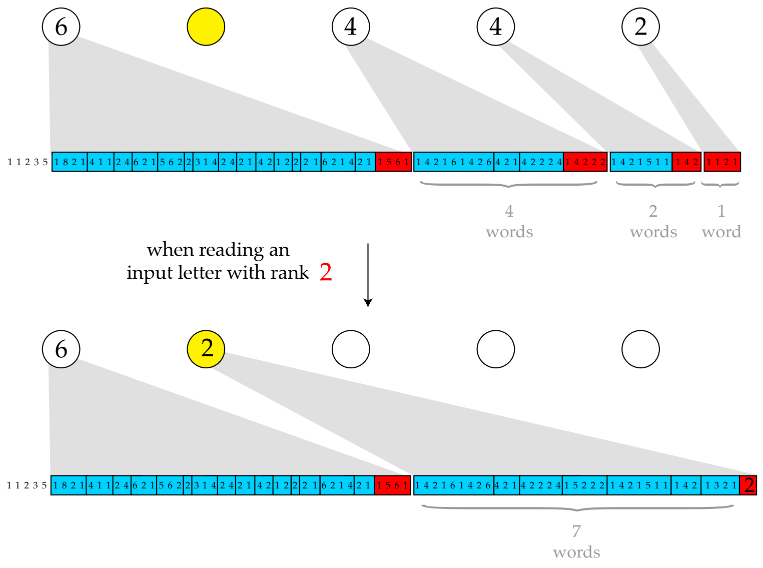

register r is empty or stores an odd rank
nonempty registers $>r$ store ranks $\geq a$
registers $<r$ are nonempty and store even numbers
7
6
6
2

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

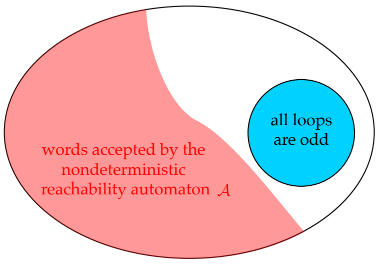
Why the invariant is preserved. We also view this transition as a two-step process. First, we empty register r (but not the smaller ones), which preserves the invariant by Lemma 12.7. Next, all of the words associated to registers $< r$ are concatenated and put into the tail of register r. As explained in the proof of Lemma 12.6, after the update the tail of register r consists of

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

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



Since every transition of $\mathcal{A}$ preserves the invariant, we can use Lemma 12.6 to conclude that if the invariant is preserved and the automaton accepts (which happens when all registers store even ranks and a new even rank is read), then the input contains at least one even loop. This gives the following inclusion:


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

Define $\mathcal{D}$ to be the deterministic reachability automaton which is obtained from $\mathcal{A}$ as follows: if there are several applicable transitions, then choose any transition that maximises the most significant register that is modified. The automaton $\mathcal{D}$ has fewer accepting runs than $\mathcal{A}$, and therefore it still rejects all words that have only odd loops. Therefore, the proof of Lemma 12.5 is completed by the following lemma.

Lemma 12.8. *If the input has only even loops, then $\mathcal{D}$ accepts.*

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

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

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

Problem 79. Consider the following variant of the automaton from Lemma 12.5. Only odd numbers are kept in the registers, and the update function is the same as in Lemma 12.5 when reading an odd number. When reading an even number a, the automaton erases all registers, which store values $< a$. Show that this automaton does not satisfy the properties required in Lemma 12.5.

Problem 80. 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 81. Show that there is no safety automaton with $< \lfloor n/2 \rfloor$ states which satisfies the properties required in Lemma 12.5.

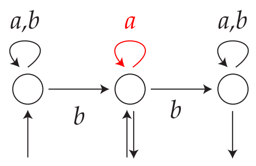
Problem 82. A probabilistic reachability automaton is defined like a finite automaton, except that each transition is assigned a probability – a number in the unit interval – such that for every state, the sum of probabilities for outgoing transitions is 1. The value assigned by such an automaton to an ω-word is the probability that an accepting state is seen at least once. Show that there is a probabilistic reachability automaton over the alphabet $\{1, \ldots, n\}^\omega$, with state space polynomial in n, that:

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

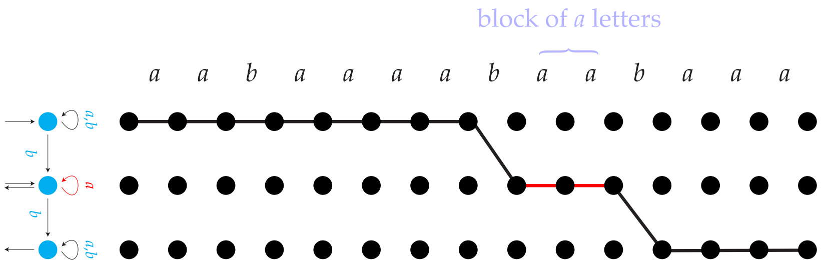
13
Distance automata

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 40. 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 [33]. The theorem was part of Hashiguchi's solution [34] to the star height problem, i.e. the problem of determining what is the least number of nested Kleene stars that is needed to define a given regular language.

Theorem 13.1. *The following problem is decidable:*

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

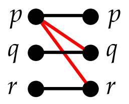
The problem in the above theorem was called *limitedness* in [33]. The algorithm we use, based on [9], uses the Büchi-Landweber Theorem [14] discussed in Chapter 11. The algorithm leads to an ExpTime upper bound on the limitedness problem; the optimal complexity is PSpace, which follows as a special case of [36, Theorem 2.2].

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

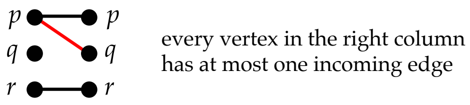
- player Input chooses a letter of the input alphabet;

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

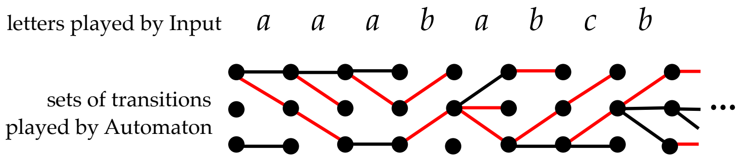
A move of player Automaton in a given round, which is a set of transitions, can be visualised as a bipartite graph, which says how the letter can take a state to a new state, with costly transitions being red and non-costly transitions being black, like below:



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


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

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



The winning condition for player Automaton is the following:

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

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

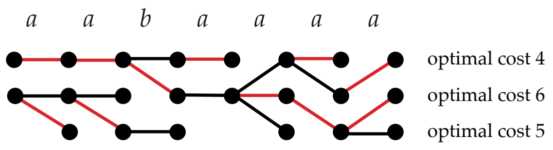
If either of the conditions above is violated, then player Input wins. The following lemma implies the decidability of the limitedness problem.

Lemma 13.2. *For a distance automaton, the following conditions are equivalent, and furthermore one can decide if they hold:*

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

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

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



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

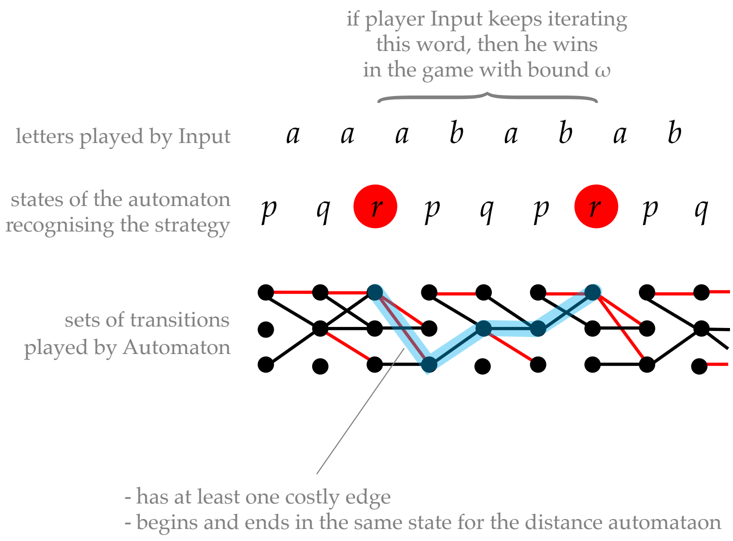
Implication from 3 to 2. Suppose that player Automaton wins the limitedness game with bound ω. We will prove that player Automaton can also win the limitedness game with a finite bound.

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

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

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

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


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

■

Problem 83. 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 84. Let $\mathcal{A}$ be a distance automaton with input alphabet Σ. The problem of *limitedness of $\mathcal{A}$ on regular language $L \subseteq \Sigma^*$* asks whether there exists $n \in \mathbb{N}$ such that for every word $w \in L$ the cost of w with respect to $\mathcal{A}$ is not bigger than n. Show that this problem is decidable.

Problem 85. We say that a regular language L has the *finite power property* if there exists $n \in \mathbb{N}$ such that $L^* = L^0 \cup L^1 \cup \ldots \cup L^n$. Show that one can decide if a regular language has the finite power property. is decidable.

Problem 86. We say that languages $K \subseteq \Sigma^*$ and $L \subseteq \Sigma^*$ are *separated by* language $S \subseteq \Sigma^*$ if $K \subseteq S$ and $L \cap S = \emptyset$. For $u, v \in \Sigma^*$ we say that $u = a_1 \cdots a_k$ is a *subsequence* of v, denoted $u \preceq v$, if $v \in \Sigma^* a_1 \Sigma^* \ldots \Sigma^* a_k \Sigma^*$. A language L is called *upward closed* if for every $u \in L$ and $u \preceq v$ also $v \in L$. Show that deciding whether two given regular languages K and L are separated by some upward closed language is decidable.

Problem 87. Let $\mathcal{F}$ be the class of finite unions of languages of the form $\Sigma^* w_1 \Sigma^* \dots \Sigma^* w_k \Sigma^*$, where all w_i are words from Σ^*. Show that for given regular languages K and L it is decidable whether they are separated by a set from $\mathcal{F}$.
Remark: Note that $\mathcal{F}$ contains all upward closed languages defined in the Problem 86. To see this recall that Higman's Lemma implies that there is no infinite antichain in the $\preceq$ order. Therefore every upward closed language has finitely many minimal elements. Thus every upward closed language is a finite union of languages of the form $\Sigma^* a_1 \Sigma^* \dots \Sigma^* a_k \Sigma^*$, where all $a_i \in \Sigma$.

Problem 88. 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.

14 Monadic second-order logic

In this section we discuss the connection between monadic second-order logic (MSO) and automata, specifically tree automata. The presentation here is largely based on [58]. One of the crowning achievements of logic in computer science is Rabin's Theorem [46], 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 10, and memoryless determinacy of parity games from Chapter 11. It remains only to deploy the appropriate definitions and put the tools to work.

14.1 Monadic second-order logic

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

[1]Büchi says this in [13, page 2]: "Given the statement of this lemma [the complementation lemma for automata on infinite trees], and given McNaughton's handling of sup-conditions by order vectors, and given time, everybody can prove Rabin's theorem."

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

Example 41. Suppose that we view an directed graph as relational structure (i.e. a model as in logic), where the universe is the vertices and there is one binary relation $E(x, y)$ for the edges; this relation is not necessarily symmetric because the graph is directed. The formula

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

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

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

The above formula illustrates all syntactic constructs in MSO: one can quantify over elements, over sets of elements, one can test membership of elements in sets, and one can use the relations available in the input model (in the case of directed graphs, only one binary relation).
Here is another example for graphs. The following MSO formula says that the input graph is three-colourable (in the formula, the direction of the edges plays no role):

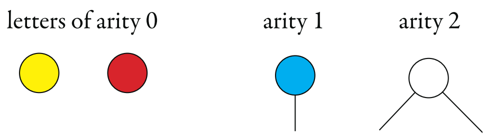
$$\exists X_1 \exists X_2 \exists X_3 \quad \underbrace{\forall x \bigvee_i x \in X_i}_{\text{every vertex is coloured}} \quad \wedge \quad \underbrace{\forall x \forall y \; E(x, y) \Rightarrow \bigwedge_i x \notin X_i \vee y \notin X_i}_{\text{no edge has both endpoints with the same colour}}$$

□

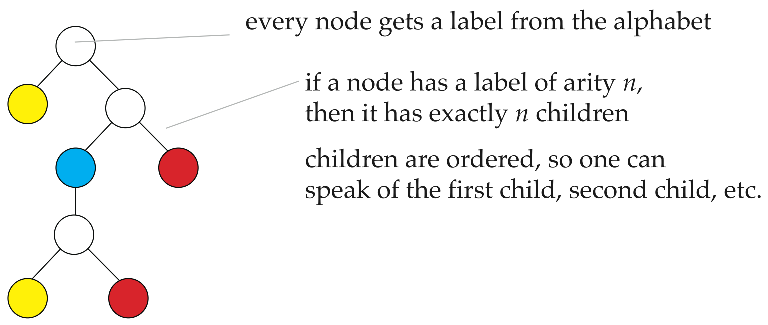
We say that a property of relational structures over some vocabulary (e.g. graphs as in the above example) is MSO *definable* if there is a formula of MSO which is true exactly in those structures which have the property. In this chapter, we use MSO to describe properties of trees (finite and infinite).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

14.2 Finite trees

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



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


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

In this section, Section 14.2, we will be interested only in finite trees. Trees as defined above are sometimes called *ranked and ordered*. One can consider other variants, where the label does not determine the number of children (unranked) or where the siblings are not ordered (unordered). The goal of this section is to show that, over finite trees, automata have the same expressive power as MSO.

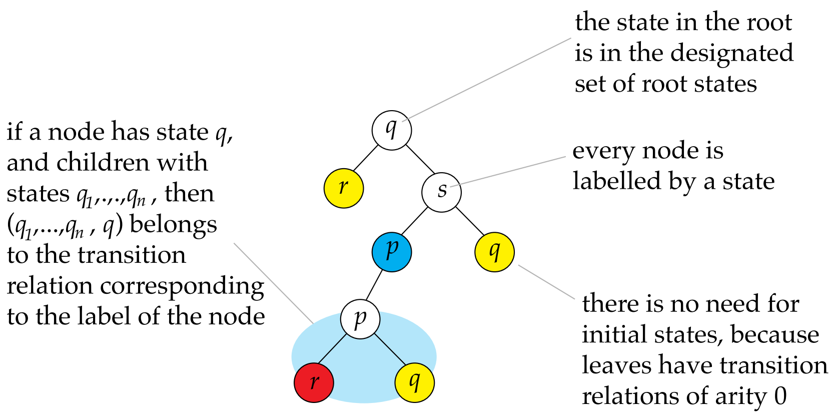
Tree automata. We begin by defining automata for finite trees.

Definition 14.1. *A* nondeterministic tree automaton *consists of:*

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

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

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

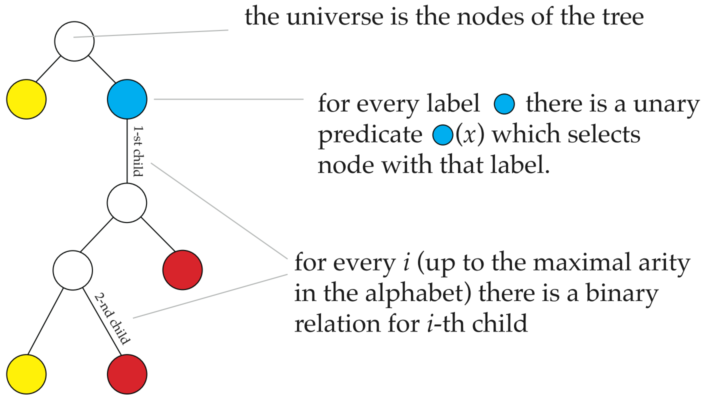

the state in the root is in the designated set of root states
if a node has state q, and children with states $q_1,..,q_n$, then $(q_1,...,q_n, q)$ belongs to the transition relation corresponding to the label of the node
every node is labelled by a state
there is no need for initial states, because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

Lemma 14.2. *Languages recognised by nondeterministic tree automata are closed under union, intersection and complementation.*

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

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

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

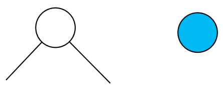

the universe is the nodes of the tree
for every label ● there is a unary predicate ●(x) which selects node with that label.
1-st child
for every i (up to the maximal arity in the alphabet) there is a binary relation for i-th child
2-nd child

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

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

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

Example 42. Suppose that the ranked alphabet is



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

The following theorem shows that for finite trees, tree automata have the same expressive power as monadic second-order logic. The connection of between automata and MSO was originally discovered simultaneously by three authors: Büchi [12], Elgot [26] and Trakthenbrot [59], in their quest to answer a question by Tarski: "is the MSO theory of the natural numbers with successor decidable"? We present below the version of the result for finite trees, which has essentially the same proof as for finite words (a word can be viewed as a tree over a ranked alphabet where all letters have arity zero or one), and was first observed in [57].

Theorem 14.3. *The following conditions are equivalent for every set of finite trees over a finite ranked alphabet:*

1. *definable in* MSO*;*

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

Proof.

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

there exists a labelling of nodes with states

every node has exactly one state

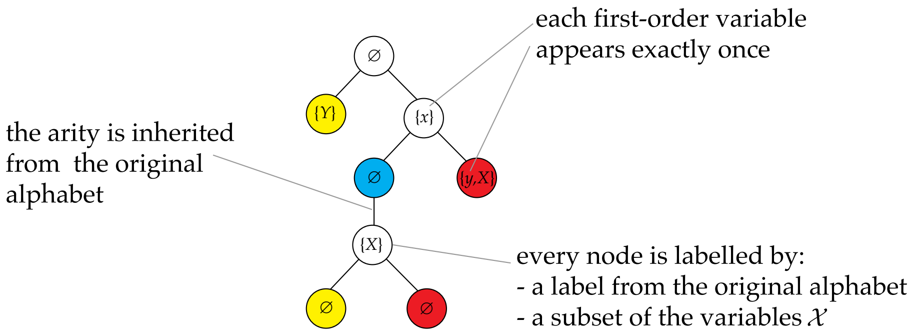
the root has a root state

for every node, a transition of the automaton is used

$$\exists X_1 \cdots \exists X_n \quad \wedge \begin{cases} \forall x \bigvee_{q \in \{1,\ldots,n\}} x \in X_q \wedge \bigwedge_{p \neq q} x \notin X_p \\ \forall x\ \mathrm{root}(x) \Rightarrow \bigvee_{i \in R} x \in X_i \\ \bigwedge_{a \in \Sigma} \forall x\ a(x) \Rightarrow \bigvee_{(q_1,\ldots,q_k,q) \in \delta_a} \left(x \in X_q \wedge \bigwedge_{i \in \{1,\ldots,k\}} \mathrm{child}_i(x) \in X_{q_i} \right) \end{cases}$$

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

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


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

A tree as above is said to satisfy φ if φ is true under the valuation which maps each first-order variable to the unique node that has it in the label, and maps each second-order label to the set of nodes that have it in their label. Define the *language* of φ to be the trees (over the extended alphabet with sets of variables) that satisfy φ. By induction on the size of an MSO formula, we show that its language, as defined above, is recognised by a tree automaton. For Boolean operations we use Lemma 14.2, for existential quantification we use nondeterminism.

■

14.3 *Infinite trees*

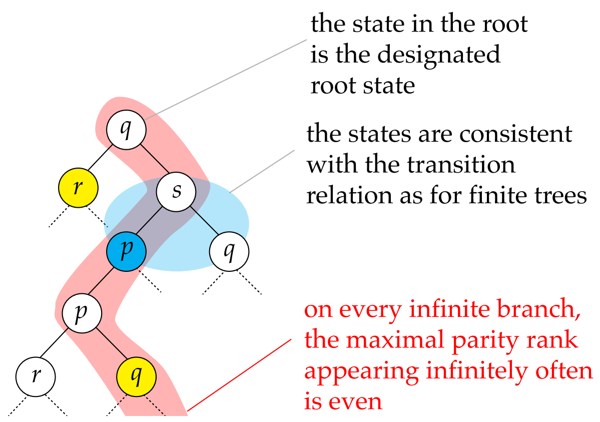
We now move to infinite trees and Rabin's Theorem. For simplicity of notation, we use ranked alphabets where all letters have rank 2. For such alphabets, the set of nodes is always the same, and can be identified with
$\{\text{left child}, \text{right child}\}^*$. For arbitrary alphabets, infinite trees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

Definition 14.4. *The syntax of a nondeterministic parity tree automaton consists of*

- *an input alphabet Σ, which is a finite ranked set where all letters have rank 2;*

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

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



We now state Rabin's Theorem. Rabin's original proof did not use the parity acceptance condition, but what is now called the *Rabin condition*, see [58].

Theorem 14.5 (Rabin's Theorem)**.** *The following conditions are equivalent for every set of (necessarily) infinite trees over a finite ranked alphabet where all letters have arity 2:*

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

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

Lemma 14.6 (Complementation Lemma)**.** *Languages recognised by nondeterministic parity automata are closed under complement.*

The difficulty in the Complementation Lemma is that we use only nondeterministic automata; in fact no deterministic model for infinite trees is known that would be equivalent to MSO. Rabin's Theorem will follow immediately once the Complementation Lemma is proved, so the rest of this chapter is devoted to proving the Complementation Lemma.
A corollary of the statement of Rabin's Theorem as in Theorem 14.5, and of decidability of emptiness for nondeterministic parity tree automata, is that the following logical structure has decidable MSO theory: the universe is the nodes of the complete binary tree, and there are two binary relations for left child and right child. This corollary is the original statement of Rabin's Theorem, see [46, Theorem 1.1.].

Alternating parity tree automata. To show complementation of nondeterministic tree automata, we pass through a more powerful model. The syntax of an *alternating parity tree automaton* is defined the same as in Definition 14.4 for nondeterministic automata, with the following differences: (1) to each state we assign an *owner*, which is either "player 0" or "player 1"; and (2) for each letter a, the transition relation has form

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

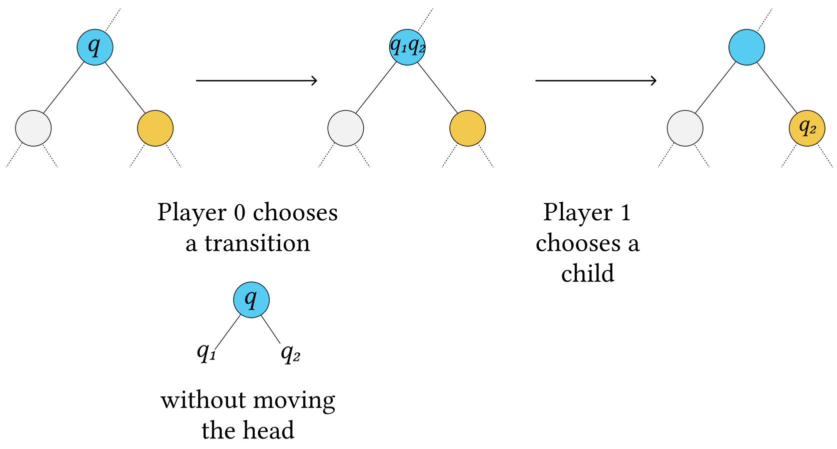
To define whether or not an automaton $\mathcal{A}$ accepts an input tree t over Σ, we consider a parity game $G_{\mathcal{A}}(t)$ defined as follows. The positions of the game are pairs (state of the automaton, node of the input tree). The initial position is (root state, root of the tree). Suppose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

Theorem 14.7 (Dealternation Theorem)**.**

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

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

Proof of Theorem 14.7. For item 1, let $\mathcal{A}$ be a nondeterministic parity tree automaton with states Q. The simulating alternating automaton has states $Q + Q^2$. The initial state is the root state of $\mathcal{A}$, and the transitions are explained in the following picture:



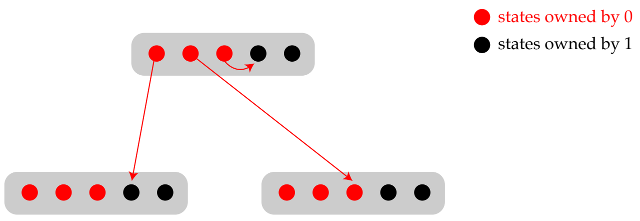
The parity condition for states from Q is inherited from the original nondeterministic automaton, and all states from Q^2 are assigned the least important rank.

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

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

The equivalences $1 \Leftrightarrow 2$ and $4 \Leftrightarrow 5$ are by definition of the language recognised by an alternating automaton. The equivalence $2 \Leftrightarrow 3$ is by construction of $\overline{\mathcal{A}}$.

The equivalence 3 $\Leftrightarrow$ 4 is because $G_{\overline{\mathcal{A}}}(t)$ is a parity game, and it is therefore determined, i.e. one of the players has a winning strategy. The reason why this proof works is that: (a) the parity condition is self-dual, which allows one to define $\overline{\mathcal{A}}$; and (b) games with the parity condition are determined.
It remains to show the last item of the theorem, namely that alternating parity tree automata can be made nondeterministic. Suppose that $\mathcal{A}$ is an alternating parity tree automaton, with states Q and input alphabet Σ. By memoryless determinacy of parity games, it follows that a tree t is accepted if and only if player 0 has a memoryless winning strategy σ_0 in the game $G_{\mathcal{A}}(t)$. We will find a nondeterministic parity automaton on trees which checks this. Define Γ to be an alphabet which consists of functions from states controlled by player 0 to pairs in $Q \times \{\epsilon, 0, 1\}$. Here is a picture of a such a letter:



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

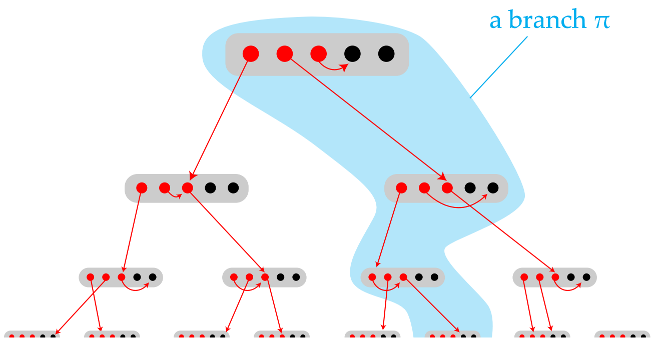
$$\{ \underbrace{(t, \sigma_0)}_{\substack{\text{tree over } \Sigma \times \Gamma \\ \text{representing } t \text{ and } \sigma_0}} : \sigma_0 \text{ is a memoryless strategy for player 0 in } \mathcal{G}_{\mathcal{A}}(t)\} \tag{14.1}$$

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

Claim 14.8. *There is a* nondeterministic *parity automaton $\mathcal{B}$ over ω-words over the alphabet $\Sigma \times \Gamma \times \{0, 1\}$ such that the following conditions are equivalent for every tree t, branch π and memoryless strategy σ_0 for player* 0*:*

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

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



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

Apply the above claim, yielding a nondeterministic parity automaton. By McNaughton's Theorem, see Chapter 10, there exists an equivalent deterministic parity automaton, call it $\mathcal{D}$. It is not difficult to see that a memoryless strategy σ_0 wins in the game $G_{\mathcal{A}}(t)$ if and only if every branch in the tree (t, σ_0) is rejected by the automaton $\mathcal{D}$. This can be checked by a (deterministic top-down) parity automaton on trees, which runs the automaton $\mathcal{D}$ on every branch (and has the acceptance condition complemented). ■

Problem 89. The translation from MSO to automata in Theorem 14.3 does an exponential blowup whenever it determinises the automaton, and therefore an upper bound on the running time is n-fold iteration of exponential, where n is the size of the formula. Here is a matching lower bound. Consider MSO on words, i.e. there is a successor relation and unary predicates for the labels. Show that for every n, there is a formula of MSO (in fact, first-order logic is enough) which has size polynomial in n and is true in a unique word which has length

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

Problem 90. Show that the set $\mathbb{N}^*$ equipped with the prefix relation has decidable MSO theory.

Problem 91. Show that emptiness is polynomial time and universality is ExpTime-complete for nondeterministic tree automata on finite trees.

Problem 92. Show that emptiness for nondeterministic parity tree automata reduces in polynomial time to solving parity games.

Problem 93. Determine whether the following tree languages are regular:

1. trees with an even number of nodes;
2. trees with an even number of a-labelled nodes;
3. trees over leaf alphabet $0, 1$ and internal alphabet $\vee, \wedge$ which evaluate to true when treated as boolean expressions;
4. balanced trees (every leaf is at the same depth).

Problem 94. Determine which of the following four variants of tree automata: deterministic / nondeterministic, top-down / bottom-up tree automata are equivalent.

Problem 95. Define the *yield* of a tree to be the word composed from labels of its leaves written in infix order. Show that for every $L \subseteq \Sigma^*$ the following are equivalent

1. L is context-free;
2. L is the set of yields of some regular tree language.

Problem 96. Show that deterministic top-down tree automata cannot recognize the language "some node has label a".

Problem 97. Show that the language of words of even length is definable in MSO.

Problem 98. Show that the following languages of infinite trees are regular (accepted by some nondeterministic automaton):

1. on every path, the sequence of labels belongs to a given ω-regular language L;
2. some node has label a;
3. in every subtree some node has label a.

Problem 99. In Existential Second Order Logic ($\exists$SO) one can write $\exists_{R_1,\ldots,R_n}\phi$, where R_i are any relations (possibly of arity greater than 1) and ϕ is a first order sentence (which of course may use R_i). Show that the language of words of composite (non-prime) length is expressible in $\exists$SO.

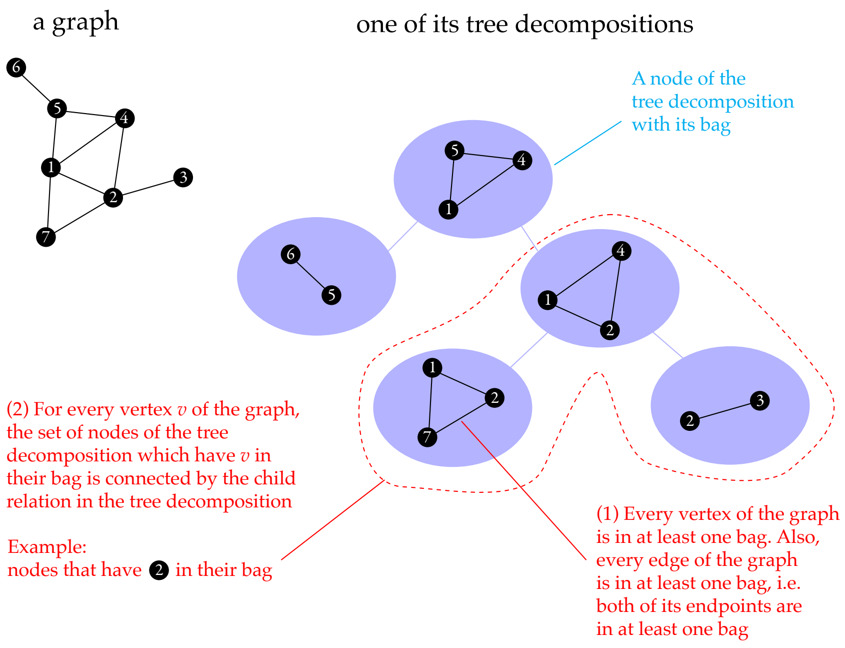
Problem 100. Consider the following game. There are two players *Insider* and *Outsider*. They choose in an alternating manner bits: 0 or 1 and create in that way an ω-word w. If w belongs to a given regular language $W \subseteq \{0,1\}^\omega$ then Insider wins a play, otherwise Outsider wins. Show that it is decidable to check which player has a winning strategy in that game. *Remark:* use MSO logic.

15
Treewidth

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

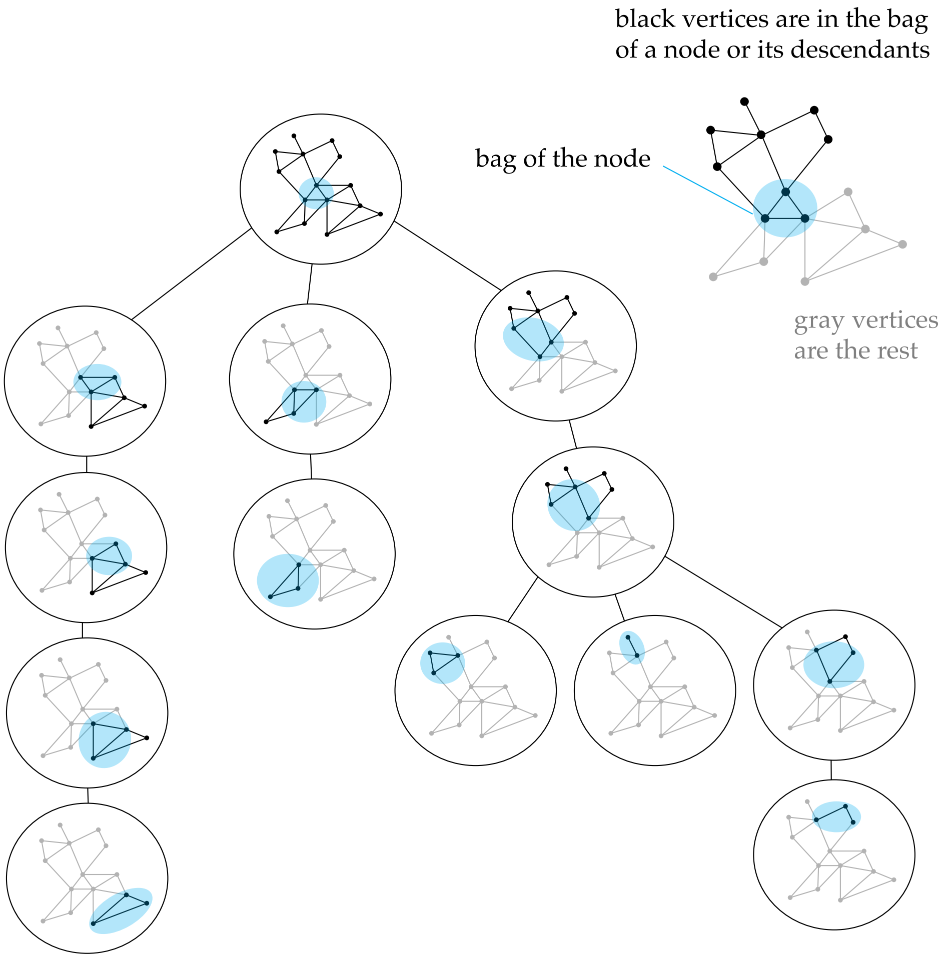
15.1 Treewidth and how to compute it

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


a graph
one of its tree decompositions
A node of the tree decomposition with its bag
(2) For every vertex v of the graph, the set of nodes of the tree decomposition which have v in their bag is connected by the child relation in the tree decomposition
Example: nodes that have 2 in their bag
(1) Every vertex of the graph is in at least one bag. Also, every edge of the graph is in at least one bag, i.e. both of its endpoints are in at least one bag

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

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


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

Fact 15.1. *If a graph has treewidth k, then the number of edges in the graph is at most $k \cdot (k+1)/2$ times the number of vertices.*

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

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

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

Theorem 15.2. *There is a function $f : \mathbb{N} \to \mathbb{N}$ and an algorithm which runs in time $f(k) \cdot n^2$ that approximates tree decompositions in the following sense:*

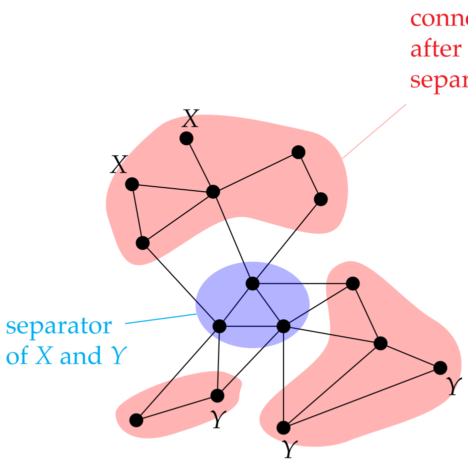
- **Input**. *k and a graph with n vertices;*
- **Output.** *A tree decomposition of the graph which has width $< 4k$, or a certificate that the graph has treewidth $\geq k$.*

The algorithm from the theorem is not optimal. The optimal algorithm, by Bodlaender [8], runs in linear time instead of quadratic time, and computes tree decompositions of optimal width (i.e. $< k$ instead of $< 4k$). The function $f(k)$ is exponential, and there is little hope for improvement, because the following problem is NP-complete [5]: given k and a graph, decide if the graph has treewidth at most k. The theorem gives a (prototypical) example of a an algorithm that is *fixed parameter tractable*, i.e. the input has two parameters k, n and the running time is of the form:

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

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

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



Lemma 15.3. *Given a graph G and disjoint sets of vertices X, Y, one can compute a separator of minimal size in time*

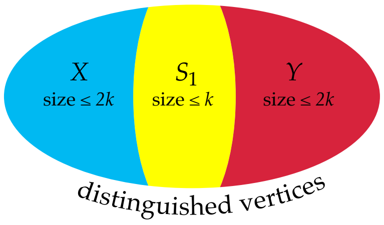
$$\mathcal{O}((\textit{number of edges} + \textit{number of vertices}) \cdot (\textit{size of the separator})).$$

We do not prove the above lemma, it can be shown using the Ford-Fulkerson algorithm for computing maximum flow, see the discussion in [24, p. 198]. When the treewidth is fixed, the number of edges is linear in the number of vertices, and the size of the separator is bounded by a function of k (see the proof of Lemma 15.4), and therefore the running time of the algorithm is linear. The main step in proving Theorem 15.2 is the following lemma.

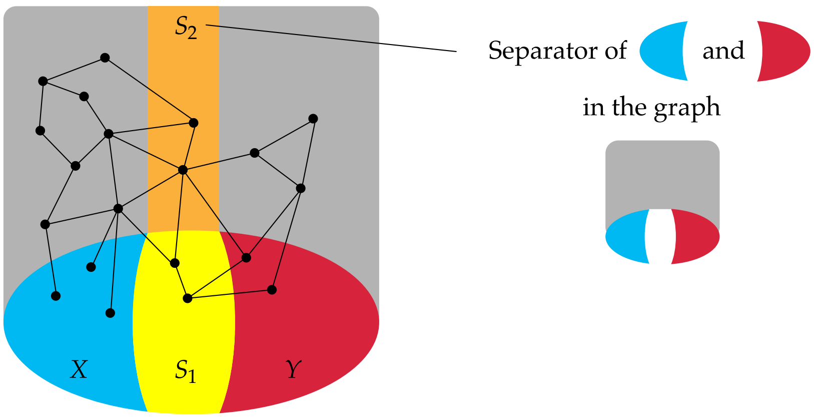
Lemma 15.4. *Let $k \in \mathbb{N}$. There is a linear time algorithm which does this:*

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

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

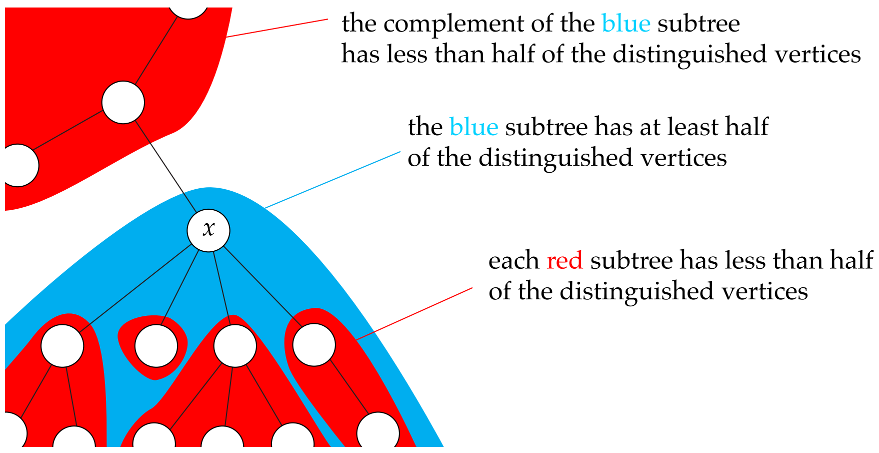


The idea is that S_1 is the intersection of the separator with the distinguished vertices. The number of such partitions is exponential in k, but is a constant if k is assumed to be fixed. For each such partition, compute a minimal size separator S_2 of X and Y in the graph $G - S_1$, as depicted in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



Report success if the size of $S_1 \cup S_2$ is at most k, and return $S_1 \cup S_2$ as the separator. This completes the algorithm. The running time is linear, because the size of the separator is fixed, and the number of edges is linear in the number of vertices by Fact 15.1.

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


the complement of the blue subtree
has less than half of the distinguished vertices
the blue subtree has at least half
of the distinguished vertices
x
each red subtree has less than half
of the distinguished vertices

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

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

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

Claim 15.5. *Let $n_1 \geq n_2 \geq \cdots \geq n_p$ be numbers in $\{1, \ldots, 2k\}$ with sum $\leq 3k$. Then*

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

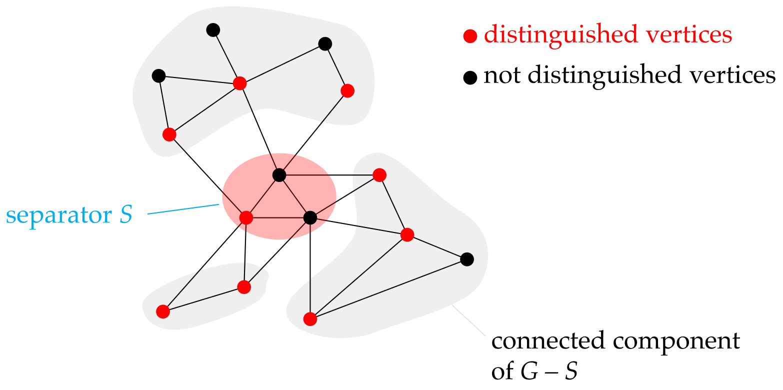
Proof. Take the first i such that the sum of the first i elements is $\geq k$. ∎

∎

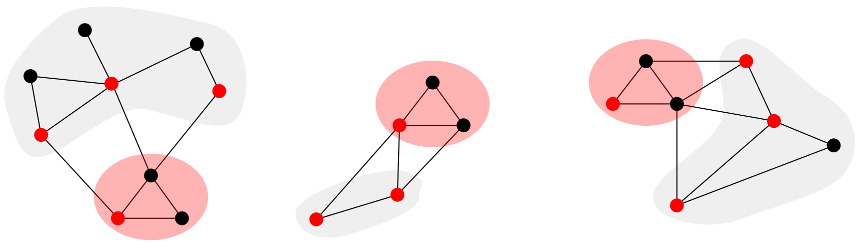
Proof of Theorem 15.2. We use a more detailed statement of the algorithm, as described below.

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

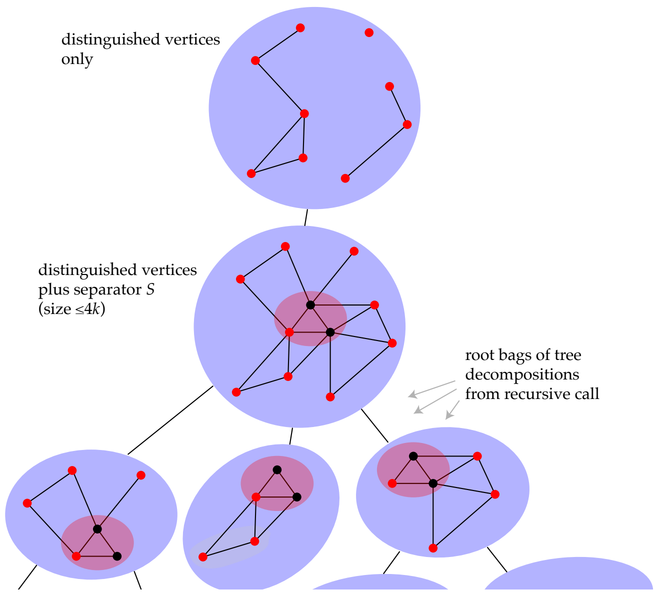
Suppose that G is the graph. If there are $< 3k$ distinguished vertices, we add some arbitrary vertices to make the set have size exactly $3k$. Apply Lemma 15.4, computing S, X and Y. If the input graph has treewidth $< k$ then the algorithm from the lemma must succeed. Find all connected components of the graph $G - S$, of which there are at least two. Each connected component has $\leq 2k$ distinguished vertices. Here is a picture:



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



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


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

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

15.2 *Courcelle's Theorem*

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

Lemma 15.6. *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 $\leq k$;*

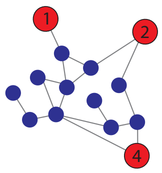
- **Question.** *Does the graph satisfy φ?*

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 [8]):

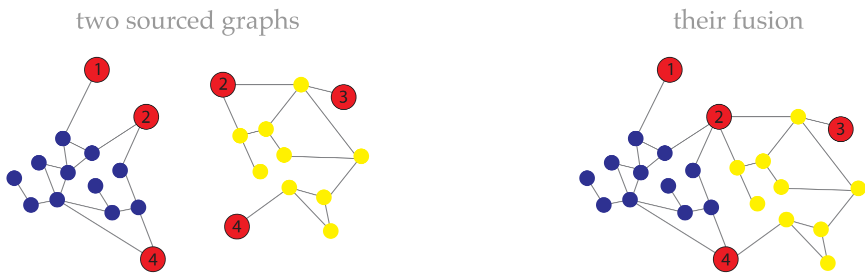
Theorem 15.7 (Courcelle's Theorem)**.** *For every $k \in \mathbb{N}$ and every formula of* MSO *φ on graphs, there is a linear time algorithm evaluates φ on graphs of treewidth $\leq k$.*

The rest of this chapter is devoted to proving Lemma 15.6. To this end, we present a more algebraic way of defining treewidth, so that tree decompositions can be viewed as trees over a finite ranked alphabet.

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



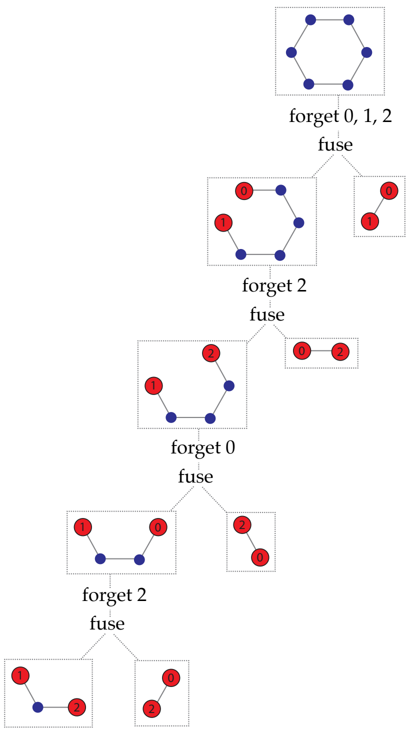
The purpose of sourced graphs is to combine them using the following fusion operation. The fusion operation inputs two sourced graphs, and outputs their disjoint union with each pair of sources that have the same name being merged together into a single vertex, as in the following picture



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

a sourced graph
after forgetting 1, 4.
1
2
4

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



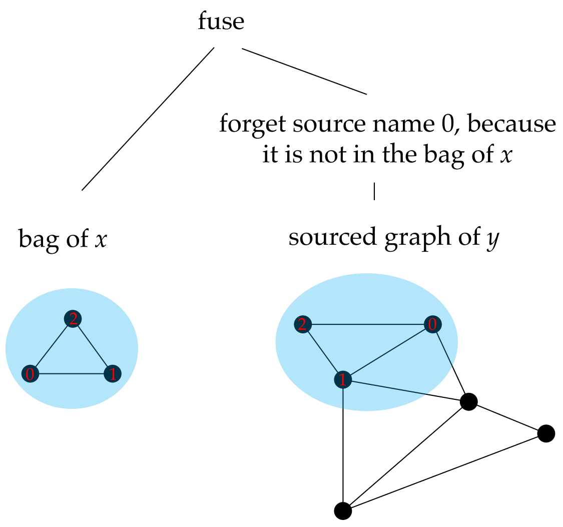
Fact 15.8. *A graph has treewidth k if and only if (when viewed as a sourced graph without any sources) it can be generated by a term in the algebra of width k sourced graphs, starting with constants that have at most $k+1$ vertices.*

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 15.8 can be viewed as a tree over a ranked alphabet Σ_k where:

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

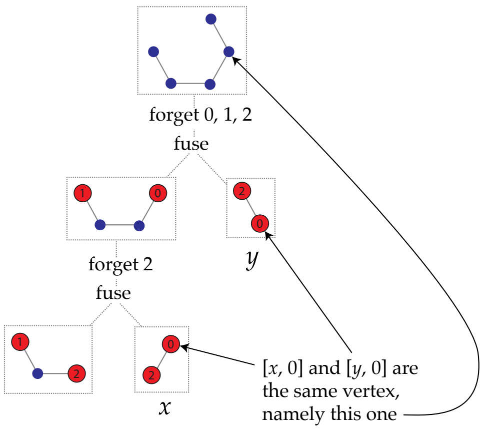
A width k tree decomposition can be converted into a corresponding tree over the above alphabet in linear time. Since the fusion operation as used in Σ_k has arity two, the conversion produces tree decompositions with binary branching (which can break properties, like no bag being used twice). By Theorem 14.3, for finite trees over alphabet Σ_k, MSO is equivalent to tree automata. Since tree automata can be evaluated in time linear in the size of the input tree (it is easier to use the bottom-up deterministic variant), it follows that MSO formulas on trees can also be evaluated in linear time. Therefore, Lemma 15.6 will follow once we prove the following lemma.

Lemma 15.9. *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*

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

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

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



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

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

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

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

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

Problem 101. Show that a graph has treewidth 1 iff it is a forest.

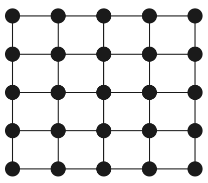
Problem 102. Compute the treewidth of the clique of n vertices.

Problem 103. Consider the following game on a graph G between k cops and one robber. The robber has a fast motorbike, cops have helicopters. In between moves everybody occupies one vertex. A round of the game is played as follows:

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

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

Problem 104. Let G_k be a grid $k \times k$ (with k^2 vertices). Show that G_k has treewidth which is either k or $k-1$. The actual answer is k, but showing this is a bit technical.


square grid of dimension 5

Problem 105. Determine the treewidth of the full bipartite graph with n vertices on the left and n vertices on the right.

Problem 106. Show that the vertex cover problem can be solved on a graph G in time $2^{\mathcal{O}(\text{tw}(G))} \cdot n^{\mathcal{O}(1)}$.

Problem 107. A graph G is called a *minor* of graph H, denoted $G \trianglelefteq H$, if G can be obtained from H by a sequence of operations of one of the following three types: 1) deleting a vertex, 2) deleting an edge, 3) contracting an edge, i.e. unifying two endpoints of this edge. Show that $G \trianglelefteq H$ implies $\text{tw}(G) \leq \text{tw}(H)$.

Problem 108. Show that there exists a function f such that if a graph G is connected then it has a walk (a path which is allowed to visit vertices multiple times) that visits all vertices and visits every edge at most $f(\text{tw}(G))$ times. Show that this is no longer true if we want to limit the number of visits to every vertex.

Problem 109. 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 110. Consider two representations of graphs as logical structures:

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

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

With edge representation, MSO can quantify over sets of vertices, while with incidence representation, MSO can quantify over sets of vertices and edges. When proving Lemma 15.6, we used edge representation. Show that the lemma and also Problem 109 remain true with the incidence representation.

Problem 111. Show that the language of connected graphs is definable in MSO on graphs (assume edge representation, as described in the Problem 110).

Problem 112. Show that the language of all forests is definable in MSO on graphs (assume edge representation, as described in the Problem 110).

Problem 113. Show that the language of grids is definable in MSO on graphs and find an appropriate formula (assume edge representation, as described in the Problem 110).

Problem 114. Recall the edge and incidence representations from Problem 110. Show a property of graphs that is definable using incidence representation but not using edge representation.

Problem 115. Recall the edge and incidence representations from Problem 110. Find a class of graphs $\mathcal{C}$ such that the following problem is decidable for the edge representation but not for incidence representation: given a formula of MSO, decide if it is true in some graph from $\mathcal{C}$.

Problem 116. Show that "has an Euler cycle" is a graph property that is not definable in MSO, even if one uses the incidence representation from Problem 110.

Problem 117. Consider the extension of MSO, called counting MSO, where one can write a formula "the size of set X is divisible by n" for every n. Show that having an Euler cycle is definable in counting MSO.

Problem 118. Show that Lemma 15.6 remains true when we use counting MSO (see Problem 117) and incidence representation.

Problem 119. The grid theorem [47, 16] says that if a class of graphs has unbounded treewidth, then it has square grids of arbitrarily large dimensions as minors. Using the grid theorem, show that if a class $\mathcal{C}$ of finite graphs has unbounded treewidth, then the following problem is undecidable: given an MSO formula φ, decide if it is true in some graph from $\mathcal{C}$.

Problem 120. Show that for every $k \in \mathbb{N}$ there exists $t \in \mathbb{N}$ such that if a graph has treewidth $\geq t$ then it has k vertex disjoint cycles. *Hint:* use the grid theorem.

Problem 121. Show that for a planar graph one can check in time $2^{\mathcal{O}(\sqrt{k}\log(k))} \cdot n^{\mathcal{O}(1)}$ whether it contains a simple path with at least k vertices. *Hint:* use the grid theorem for planar graphs in the following form: if a planar graph has treewidth $\geq 5k$ then it has the $k \times k$ grid as a minor.

Problem 122. Recall $\exists$SO from Problem 99. Let us model a graph as relational structure using the edge representation discussed in Problem 110 (for the incidence representation, the same result would be true). Show that a property of graphs is definable in $\exists$SO if and only if it is in the class NP (this is Fagin's theorem).

Problem 123. Show that the following problem is undecidable: the input is a formula of $\exists$SO that uses only equality (and the quantified relations); the question is if this formula is true in some finite structure (i.e. a finite universe equipped with equality only).

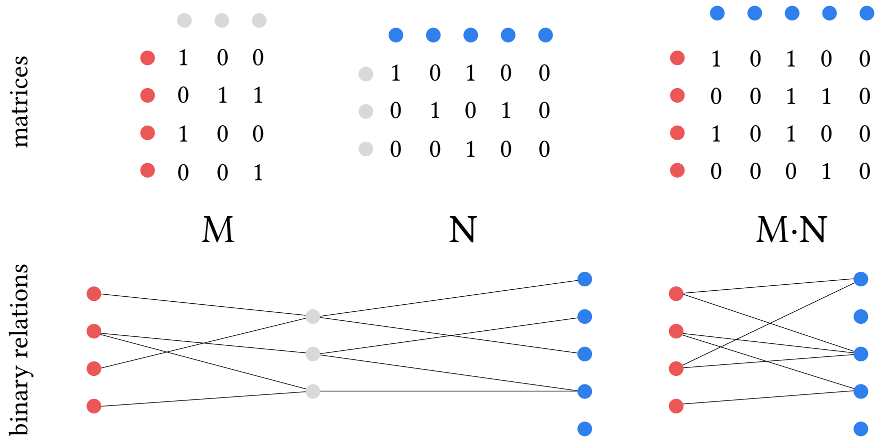
Problem 124. Show that there is a polynomial time algorithm deciding whether a given graph is planar. *Hint:* assume that there exists a polynomial algorithm deciding whether a given graph G is a minor of an input graph H.

Problem 125. Show that there exists a polynomial time algorithm deciding whether a given graph can be drawn on torus without crossing edges.

16

Parsing in matrix multiplication time

The classical dynamic CYK algorithm for parsing context-free grammars runs in cubic time (in terms of the input word). In this chapter we present a parsing algorithm of Valiant [62], which parses context-free languages in approximately the same time as matrix multiplication. The matrices are Boolean, which means that the entries are 0 or 1, addition is $\vee$ and multiplication is $\wedge$. For readers wary of matrices, an $n \times m$ Boolean matrix is the same as a binary relation between $\{1, \ldots, n\}$ and $\{1, \ldots, m\}$, and matrix multiplication is composition of relations, as in this picture:



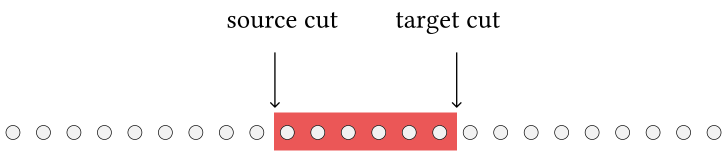
The naive algorithm for matrix multiplication runs in time n^3, but smarter algorithms run faster, e.g. the Strassen algorithm runs in time approximately $\mathcal{O}(n^{2.8704})$, and the record holder as of 2024 is $\mathcal{O}(n^{2.3715})$, see [63]. The exponent in the running time of matrix multiplication is denoted by ω. We know that this value is at least 2, because one needs to read the matrices, and currently it is known to be at most 2.3715. The purpose of this chapter is to explain an algorithm, due to Valiant, which employs matrix multiplication to parse context-free languages in sub-cubic time. The Valiant algorithm is not practical for parsing, because the constant factors are large in the fast matrix multiplication algorithms, but it is a milestone in the theory of algorithms.

Theorem 16.1. *Assume that multiplication of $n \times n$ Boolean matrices can be computed in time $\mathcal{O}(n^\omega)$ for some real number ω. Then membership in a context-free language can be decided in time at most*

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

For $\omega > 2$, the running time can be further improved to $\text{poly}(\mathcal{G}) \cdot n^\omega$ which we justify later in footnote 1.
For the rest of this chapter, fix ω and a context-free grammar $\mathcal{G}$. We assume that the grammar is in Chomsky Normal Form, i.e. every rule is of the form

$X \leftarrow YZ$ or $X \leftarrow a$, where X, Y, Z are nonterminals and a is a terminal. A grammar can be converted into Chomsky Normal Form in polynomial time, so this assumption can be made without loss of generality.
The main data structure that will be used in this algorithm is what we call a parse matrix for some input string. Define an *interval* in an input string to be a connected sequence of positions. We think of an interval as connecting two cuts (i.e. spaces between positions) in an input string, as in the following picture:



A parse matrix is a collection of facts of the form "the infix at interval I can be generated by a nonterminal X". We always want this collection to be *sound*, i.e. every fact in the parse matrix should actually be true, but it does not need to be *complete*, which means that the parse matrix contains all true facts. The information about nonterminal X in a parse matrix can be seen as a Boolean matrix M_X where the rows are source cuts, and the columns are target cuts. All nonzero entries will be strictly above the diagonal, since the source cut must be strictly before the target cut. (Since we do not have ϵ-productions, every nonterminal generates only nonempty strings). A parse matrix for a string of length n is called a *parse matrix of length n*. In this parse matrix, the underlying Boolean matrices for each nonterminal have dimension $(n+1) \times (n+1)$, since there are $n+1$ cuts.
As mentioned above, a parse matrix can be viewed as a collection $M = \{M_X\}_X$ of Boolean matrices, indexed by nonterminals. For parse matrices M, N of same length, define their product $M \cdot N$ by

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

where the union ranges over rules of the grammar and $M_Y \cdot M_Z$ is matrix multiplication. The product consists of facts of the form "interval I can be generated by X", which can be derived by taking rule $X \to YZ$ in the grammar, and a decomposition of I into two intervals J and K, such that the matrix M contains that fact "interval J can be generated by Y" and the matrix N contains that fact "interval K can be generated by Z". Since we have described product of parse matrices using matrix multiplication, we get the following observation.

Lemma 16.2. *For length n parse matrices, product can be computed in time $\mathcal{O}(n^\omega)$.*

We say that a parse matrix M is *closed* if it satisfies $M \cdot M \subseteq M$, and we say that it is closed on an interval I 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 16.3. *There is an algorithm which runs in time*

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

and which computes the closure of a length $2n$ parse matrix, assuming that it is closed on the intervals $\{1, \ldots, n\}$ and $\{n+1, \ldots, 2n\}$.

Before proving the proposition, we show how it implies the Theorem 16.1.

Proof of Theorem 16.1. Suppose that we want to know if the grammar $\mathcal{G}$ generates a word w of length n. Define M to be the length n parse matrix where M_X contains intervals $\{i\}$ such that nonterminal X generates the i-th letter of w, using a rule of the form $X \to a$. This parse matrix can be computed in time linear in n. The word w is generated by the grammar if and only if the closure M^* contains the full interval on the component corresponding to the starting nonterminal. It suffices therefore to compute the closure M^*. To make the computation easier, suppose that the length of the word is a power of two, i.e. $n = 2^k$. We do a divide an conquer approach: we compute the closures of the parse matrix for the first and second halves of w (using a recursive procedure), and then combine these using the algorithm from Proposition 16.3. The running time of this algorithm is at most

$$T(n) + 2T(\frac{n}{2}) + \cdots + 2^k T(\frac{n}{2^k}). \tag{16.1}$$

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

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

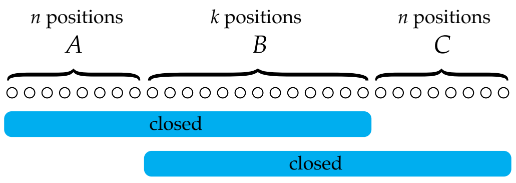
so the above sum is bounded by

$$T(n) + \frac{T(n)}{2} + \cdots + \frac{T(n)}{2^k} < 2T(n),$$

which shows that the running time (16.1) is at most two times slower than $T(n)$, thus proving the theorem, given the bounds on $T(n)$ from Proposition 16.3. ∎

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

Lemma 16.4. *Suppose that M is a length $k + 2n$ parse matrix that is closed on the intervals $A \cup B$ and $B \cup C$ as depicted below:*



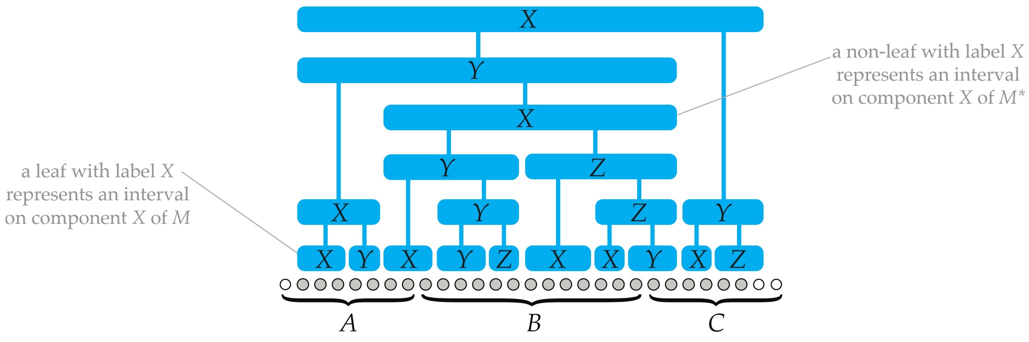
Then the closure M^* can be computed in time $\text{poly}(\mathcal{G}) \cdot n^\omega + T(n)$.

Proof. Define N to be $M \cup M \cdot M$ restricted to intervals that contain B or are disjoint with B. Here, the sum takes the union of all facts stored in the two parse matrices. The main observation in the lemma is the following claim.

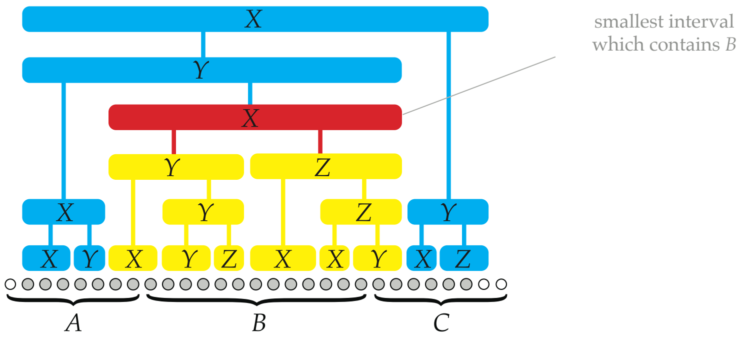
Claim 16.5. $M^* = M \cup N^*$.

Before proving the claim, 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 16.2, the parse matrix N can be computed in time $\text{poly}(\mathcal{G}) \cdot n^\omega$, using matrix multiplication for the product $M \cdot M$. Because the matrix M is closed over intervals A and C, it follows that N is also closed over these intervals. Since all entries of N contain B or are disjoint with B, it is essentially

a matrix of length $2n$ whose first and second halves are closed. It follows that N^* can be computed in time $T(n)$.
It remains to prove the claim. 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:

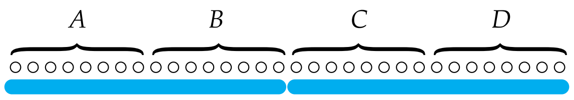

X
Y
X
Y
Z
X
Y
Z
Y
X
Y
X
Y
Z
X
X
Y
X
Z
A
B
C
a non-leaf with label X represents an interval on component X of M^*
a leaf with label X represents an interval on component X of M

In the parse tree, use red consider the smallest interval which contains B, and use yellow for the descendants of the red interval:



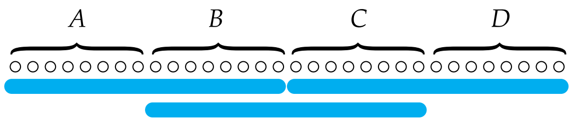
By minimality, each yellow interval is contained in either $A \cup B$ or $B \cup C$, and therefore belongs to M by the closure assumptions on M. Therefore, the red itself belongs to $M \cdot M$. The red interval contains B, and the blue intervals are disjoint with B, therefore the red and blue intervals are in N. It follows that the red and blue intervals form a parse tree corresponding to the matrix N^*. ■

Proof of Proposition 16.3. 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 assumption of the proposition. Let us write A, B, C, D for the intervals describing the four quarters of $2n$, as in the following picture:

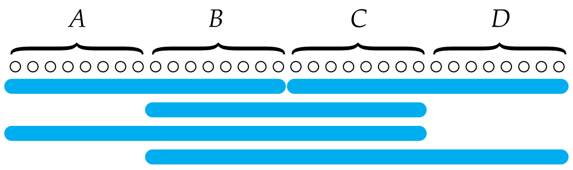


As in Lemma 16.4, the blue rectangles indicate the intervals which are closed.

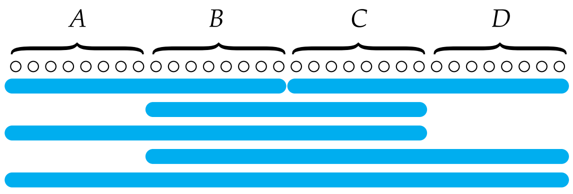
1. By induction, compute the closure of the interval $B \cup C$:



2. Using Lemma 16.4 twice, compute the closures of $A \cup B \cup C$ and $B \cup C \cup D$:



3. Using Lemma 16.4, 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 16.1, 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

$$4^i \cdot (\frac{n}{2^i})^\omega \leq n^\omega,$$

which gives the bound in the proposition. [1] ∎

Problem 126. Show that the operation $M \circ N$ is not associative.

Problem 127. Design an algorithm, which for an undirected graph G with n vertices answers whether there exists a subgraph of G, which is

1. a triangle, in time $\mathcal{O}(n^\omega)$;

2. a cycle with 4 vertices, in time $\mathcal{O}(n^\omega)$;

[1] Assuming that $\omega > 2$, we can get rid of the $\log(n)$ in the running time, because the sum $\sum_{i=0}^{k} 4^i (\frac{n}{2^i})^\omega$ is bounded by $n^\omega \sum_{i=0}^{\infty} (\frac{1}{2^{\omega-2}})^i$ which is $\mathcal{O}(n^\omega)$, because $\frac{1}{2^{\omega-2}} < 1$.

3. a cycle with k vertices, in time $\mathcal{O}(n^{\omega})$;
4. a clique with 4 vertices, in time $\mathcal{O}(n^{1+\omega})$;
5. a clique with 5 vertices, in time $\mathcal{O}(n^{2+\omega})$;
6. a clique with 6 vertices, in time $\mathcal{O}(n^{2\omega})$;
7. a clique with $3k$ vertices, in time $\mathcal{O}(n^{k\omega})$.

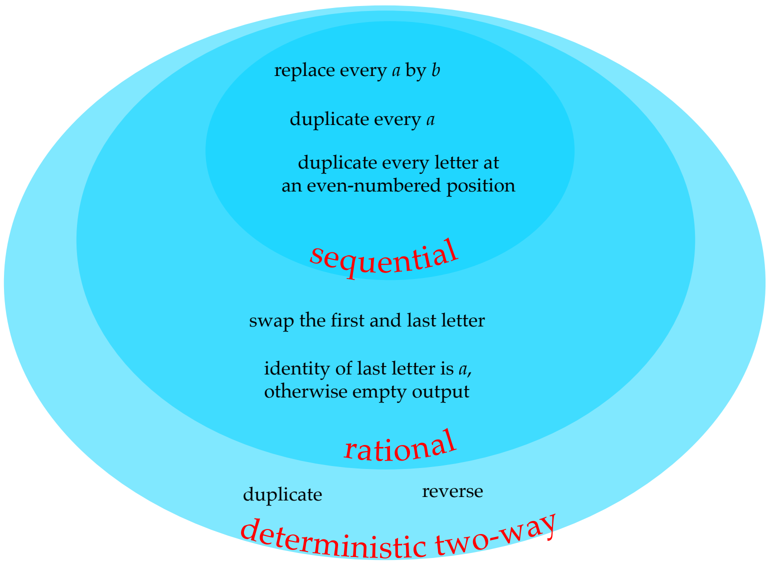
Problem 128. Design an algorithm, which for an undirected graph $G = (V, E)$ with $3n$ vertices answers whether there exists a subset $S \subseteq V$ with $|E(S, V - S)| \geq k$ in time $\mathcal{O}(2^{n\omega} \cdot \mathrm{poly}(n))$ (by $E(A, B)$ we denote the set of all the edges with one endpoint in A and another endpoint in B).

Problem 129. Let $U, V \subseteq \mathbb{N}^d$ be sets of d-dimensional vectors, each one with n vectors. Show that for $n \geq d$ one can check whether there are $u \in U$ and $v \in V$ such that $u \perp v$ in time $o(n^2 d)$.

Problem 130. Design an algorithm, which multiplies two matrices of size $n \times n$ in time $\mathcal{O}(n^{\log_2 7})$.

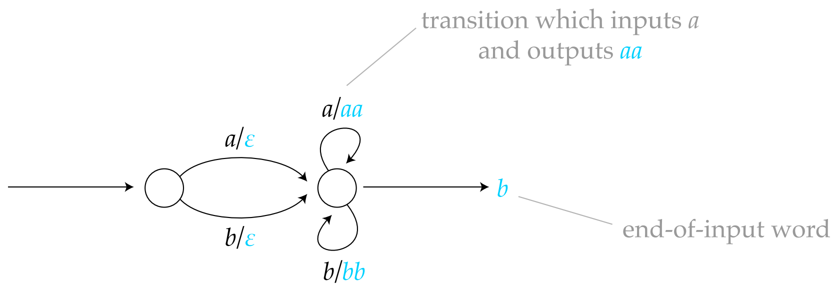
17
Two-way transducers

In this chapter, we talk about transducers, i.e. automata that input words and output words. We cover three families of transducers as shown below:


replace every a by b
duplicate every a
duplicate every letter at an even-numbered position
sequential
swap the first and last letter
identity of last letter is a, otherwise empty output
rational
duplicate
reverse
deterministic two-way

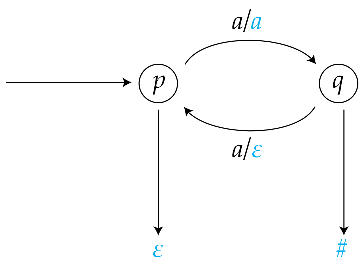
17.1 *Sequential functions*

Recall the definition of a *nondeterministic finite automaton with output* from Definition 4.11. This is an NFA where every transition is labelled by a (possibly empty) *output word* over a designated output alphabet, and every final state is labelled by a (possibly empty) *end-of-input word*, also over the output alphabet. Here is an example:



The output of a run is obtained by concatenating the output words of all transitions used, followed by the end-of-input word of the last state used. The semantics of the automaton is defined to be the function which maps an input word to the multiset of words over the output alphabet that are produced by accepting runs (if the same output is produced by n different accepting runs, then it appears n times in the output multiset).
The automaton in the picture above has the following outputs: if the input word is empty, then the output multiset is empty; if the input word is nonempty, then the automaton produces exactly one output (i.e. a multiset with one word) which is obtained from the input by deleting the first letter, doubling the other letters, and appending b to the end.
Define a DFA *with output* to be the special case of an NFA with output where: (a) the transition relation is a deterministic, i.e. for every state there is a unique outgoing transition for each input letter; and (b) all states are final. Under these assumptions, the automaton produces exactly one output for every input, and therefore its semantics can be viewed as a function from words over the input

alphabet to words over the output alphabet. Any function obtained this way is called a *(left-to-right) sequential function*[1]. Here is an example:



The transducer above erases a's at even-numbered positions, and appends # or nothing to the output, depending on the parity of the input length. Other examples of left-to-right sequential functions include: "erase all appearances of letter a" or "erase all appearances of letter a at even-numbered input positions".

Define a *right-to-left sequential function* symmetrically: the syntax is the same, except that in the semantics, the input letters are read from right to left, and the end-of-input word is produced after reading the leftmost position. The function "identity if the input ends with a, otherwise empty output" is a right-to-left sequential function but not a left-to-right sequential function.

[1]The name sequential is used for at least four transducer models in the literature, starting with the original transducer models described by Shannon [54, Section 8] and later developed by Moore [40] and Mealy [39]. Both the Moore and Mealy models – which are two non-equivalent models of letter-to-letter transducers – were called sequential by their authors. In those days, sequential seems to have been a synonym for "recognised by an automaton". Then, Ginsburg introduced a model, called submachines, that could produce words (and not just letters) in transitions [31]. Soon Ginsburg's model started to be called sequential, see e.g. [25, p. 298]. Then, Schützenberger extended submachines with end-of-input words [52]. Now it is Schützenberger's model – originally called subsequential – that is being called sequential, e.g. [29], and this is the convention that we adopt here.

17.2 *Rational functions*

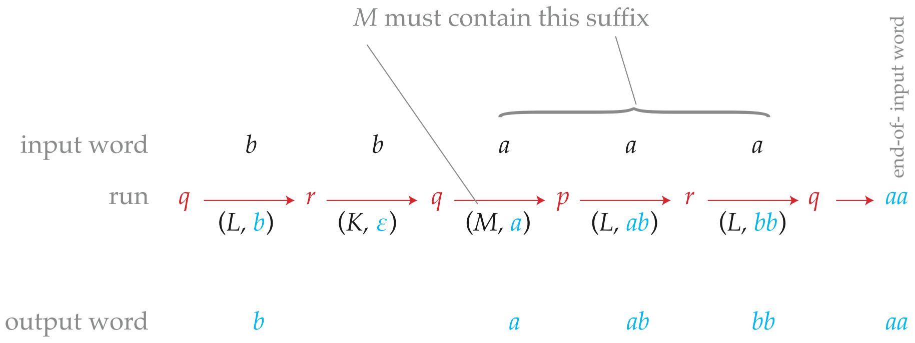
We now move to a richer class of functions from words to words, called the *rational functions*[2]. This class admits several equivalent definitions; we give five. Another advantage is that the class is symmetric, i.e. there is no need to define "right-to-left rational functions". We begin with two definitions that use NFA's with output.

Functional and unambiguous NFA's with output. We say that an NFA with output is *functional* if for every input word, the output multiset contains exactly one word, but possibly with multiplicities. In other words, there might be several accepting runs, but all accepting runs produce the same output word, and there is always at least one accepting run. We say that an NFA with output is *unambiguous* if for every input word, the output multiset contains exactly one word, used exactly one time. In other words, for every input there is exactly one accepting run. Functional, and therefore also unambiguous, NFA's with output can be viewed as recognising functions from words to words, by mapping an input word to the unique output word in the output multiset. Functional NFA's with output are essentially the same as the original definition of rational functions given by Eilenberg in [25, Chapter IX].
We will later show that – when viewed as recognisers of functions from words to words (without multiplicities of outputs) – functional and unambiguous automata have the same expressive power, i.e. nothing is gained by using functional but possibly ambiguous NFA's with output.

Lookahead DFA with output. A *lookahead* NFA *with output* is a model that extends an NFA with output as follows: instead of pairs (input letter, word over the output alphabet), the transitions are pairs (regular language over the input alphabet, word over the output alphabet). A transition labelled by a pair (L, w)

[2]The name rational comes from Eilenberg. Eilenberg introduced rational subsets of any monoid [25, Chapter VII], which covers the special case of rational relations [25, Chapter IX] defined as rational subsets of monoids of the form $\Sigma^* \times \Gamma^*$, which in turn covers the special case of *rational functions* which are functional rational relations.

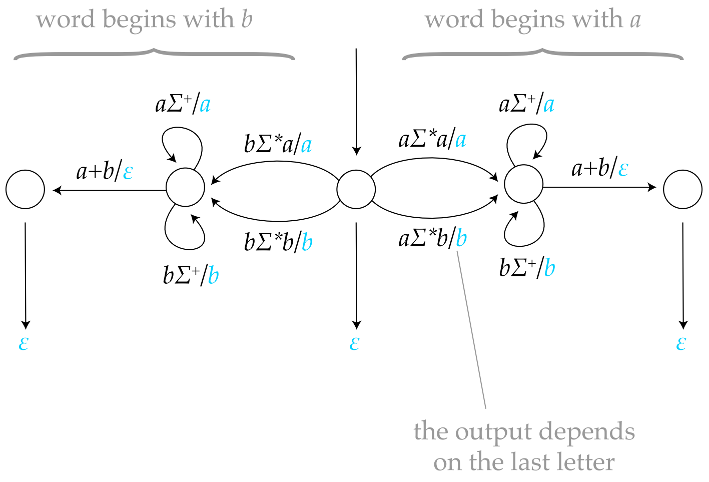
can be applied if the unread part of the input belongs to L; the effect of using such transition is that w gets added to the output and one input letter is consumed. Here is a picture of a run:



M must contain this suffix
end-of- input word
input word
b b a a a
run
q r q p r q aa
(L, b) (K, ε) (M, a) (L, ab) (L, bb)
output word
b a ab bb aa

A *lookahead* DFA *with output* is the special case where (a) for every state, the regular languages labelling outgoing transitions form a partition of all nonempty words; and (b) every state is final.

Example 43. The following lookahead DFA with output swaps the first and last letters:

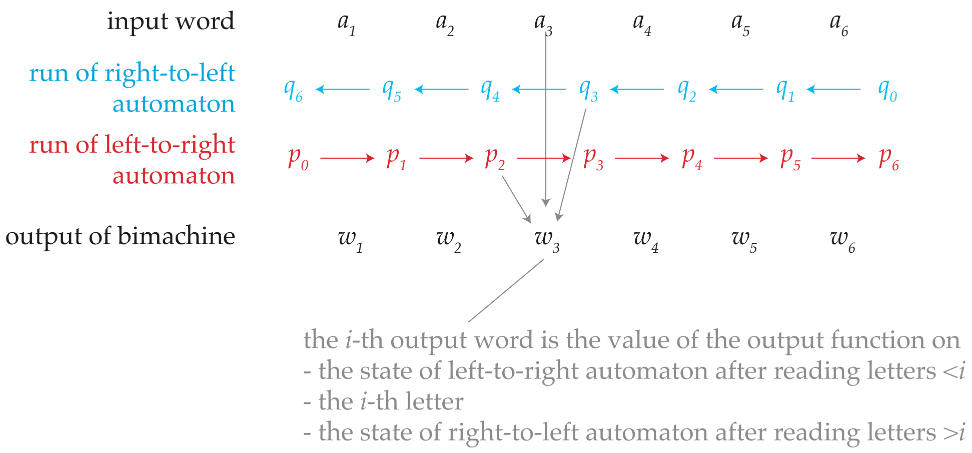


□

Eilenberg bimachine. We now present Eilenberg bimachines, which are essentially another syntax for lookahead DFA with output. An *Eilenberg bimachine* [25, Chapter XI.7] consists of two finite automata $\mathcal{A}, \mathcal{B}$ over the input alphabet – with $\mathcal{A}$ left-to-right deterministic and $\mathcal{B}$ right-to-left deterministic – as well as an output function of type

$$\text{states of } \mathcal{A} \times \text{input alphabet} \times \text{states of } \mathcal{B} \quad \to \quad (\text{output alphabet})^*.$$

In the automata $\mathcal{A}, \mathcal{B}$ the final states are irrelevant and can be omitted from the syntax. The semantics of the bimachine is defined as follows. Given a nonempty input word, define for each position in the input word an output word as described in the following picture:


input word
run of right-to-left automaton
run of left-to-right automaton
output of bimachine
a_1 a_2 a_3 a_4 a_5 a_6
q_6 q_5 q_4 q_3 q_2 q_1 q_0
p_0 p_1 p_2 p_3 p_4 p_5 p_6
w_1 w_2 w_3 w_4 w_5 w_6
the i-th output word is the value of the output function on
- the state of left-to-right automaton after reading letters $<i$
- the i-th letter
- the state of right-to-left automaton after reading letters $>i$

The output of the bimachine is defined to be the concatenation of the output words, in the order inherited from the input positions. To deal with empty inputs, an Eilenberg bimachine is equipped with an designated output word that is used for the empty input.

Equivalence of the models. The following theorem shows that all the models described above are equivalent. We use the name *rational function* for a word-to-word function that is defined by any one of the equivalent models in the theorem.

Theorem 17.1. *The following models are equivalent, in terms of the functions from words to words that they define:*

1. *functional* NFA *with output;*
2. *lookahead* DFA *with output;*
3. *unambiguous* NFA *with output.*
4. *Eilenberg bimachines.*
5. *compositions of right-to-left sequential functions with left-to-right sequential functions.*

Proof sketch.

$1 \subseteq 2$ Consider a functional NFA with output $\mathcal{A}$. We define an equivalent lookahead DFA as follows. The lookahead DFA computes some run of the functional NFA that can be extended to an accepting run. Each transition is chosen using the lookahead, to determine if it can be extended to an accepting run. If more than one transition can be chosen, some arbitrary tie-breaking mechanism is used.

$2 \subseteq 3$ Consider some lookahead DFA with output $\mathcal{A}$. We define an equivalent Eilenberg bimachine as follows. Let $\mathcal{B}$ be a right-to-left DFA (without output) that simultaneously recognises all the languages which are used in the transitions of $\mathcal{A}$, i.e. the lookahead languages. The simulating NFA with output guesses the runs of these two automata (the run for $\mathcal{B}$ is right-to-left, and the run for $\mathcal{A}$ is left-to-right, and depends on the run of $\mathcal{B}$). This guess is unambiguous, because the automata $\mathcal{A}$ and $\mathcal{B}$ are unambiguous.

$3 \subseteq 4$ Consider an unambiguous NFA with output $\mathcal{A}$. We define an equivalent Eilenberg bimachine as follows. The left-to-right automaton is a left-to-right powerset construction applied to states of $\mathcal{A}$, i.e. its states are sets of states in $\mathcal{A}$ and the transition function is defined by

$$P \cdot a = \{q : \text{the automaton } \mathcal{A} \text{ has a transition } p \overset{a/w}{\to} q \text{ for some } p \in P\ \}.$$

The right-to-left automaton is defined symmetrically, i.e. its transition function is defined by

$$a \cdot P = \{p : \text{the automaton } \mathcal{A} \text{ has a transition } p \overset{a/w}{\to} q \text{ for some } q \in P\ \}$$

The output function maps a triple (P, a, Q) to the unique output word w such that the automaton has a transition

$$p \overset{a/w}{\to} q \qquad p \in P, q \in Q.$$

This function is well defined by the assumption that $\mathcal{A}$ is unambiguous.

$4 \subseteq 1$ Consider an Eilenberg bimachine $\mathcal{A}$. We define an equivalent functional – in fact, unambiguous – NFA with output as follows. The states of the simulating automaton are pairs (state of the left-to-right automaton in $\mathcal{A}$, state of the right-to-left automaton $\mathcal{A}$). The transition relation is defined by

$$(q, ap) \overset{a/w}{\to} (qa, p)$$

w is the output word in the bimachine that is associated to the triple (q, a, p). This automaton is unambiguous by the determinism assumptions in the definition of a bimachine.

$2 \subseteq 5$ A right-to-left sequential function can label the input word with states of right-to-left automata recognising the lookahead, and a left-to-right sequential function can then simulate the DFA with lookahead.

$5 \subseteq 3$ Functional NFA with output are closed under compositions and generalise both left-to-right and right-to-left sequential functions.

■

17.3 *Deterministic two-way transducers*

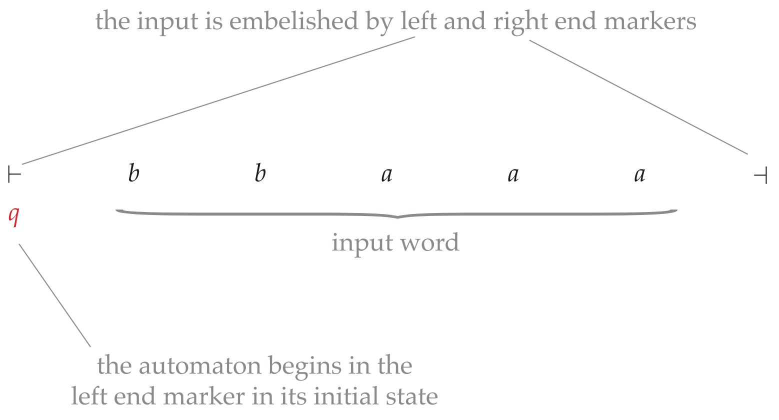
We now turn to the most powerful class of transducers discussed in this chapter, namely *deterministic two-way transducers*. In the next chapter, we will present an equivalent one-way model, which uses registers to store parts of the output.

Definition 17.2. *A* deterministic two-way transducer *consists of:*

- *finite* input and output alphabets Σ *and* Γ*;*
- *a finite set of* states Q *with a distinguished* initial *state;*
- *a transition function*

$$\delta : Q \times (\Sigma \cup \{\vdash, \dashv\}) \to \{accept\} \cup (Q \times \{left, stay, right\} \times \Gamma^*)$$

The semantics of the transducer are defined similarly to Turing machines. Actually, the model is equivalent to a Turing machine where there is one read-only input tape and one append-only output tape. The automaton begins in the following configuration:


the input is embelished by left and right end markers
⊢
b
b
a
a
a
⊣
q
input word
the automaton begins in the
left end marker in its initial state

(For two-way automata, the head is over a letter, as opposed to one-way automata, where the head is between letters.) At any given moment, the automaton applies its transition function to its current state and the symbol under the head. The result of the transition might be "accept", in which case the automaton ends its run, or a triple (state, direction, output word), in which case the new state is assumed, the head is moved in the direction, and the output word is appended to the output. The output letters are used in chronological order, i.e. those which are output at the beginning of the run are at the beginning of the output, regardless of the position of the head when executing the transition. The run of the automaton might fail, either by moving out of the word (i.e. moving left on the left marker or moving right on the right marker), or by entering an infinite computation that never sees a final state; such failing runs do not produce any output, and therefore the semantics of the automaton is a partial function from Σ^* to Γ^*.
Typical things that can be done using a two-way transducer are duplication or reversing the input. The main result of this chapter is that deterministic two-way automata are closed under composition.

Theorem 17.3 ([1, 19]). *Functions recognised by deterministic two-way transducers are closed under composition.*

For sequential and rational functions, closure under composition is done using a straightforward product construction. For two-way automata, the construction is much more challenging, since the automata begin composed might choose to move in different directions.
The rest of this chapter is devoted to proving Theorem 17.3. We do it in two steps. First, we show in Lemma 17.4 a weaker version – namely that deterministic two-way automata are closed under pre-composition with rational functions. Then we bootstrap the weaker version to get composition with deterministic two-way automata.

Rational preprocessing. We begin by proving that deterministic two-way transducers can be pre-composed with rational functions. A different perspective on this result is that deterministic two-way transducers would not become more expressive if equipped with "regular lookaround", i.e. transitions that depend not only on the letter under the head, but also on some regular properties of the words to the left and right of the head.

Lemma 17.4. *Deterministic two-way transducers are closed under pre-composition with rational functions. In symbols,*

$$\underbrace{\mathsf{2Det}}_{\substack{\textit{functions recognised by}\\ \textit{deterministic two-way automata}}} \quad = \quad \mathsf{2Det} \quad \circ \quad \underbrace{\mathsf{Rat}}_{\textit{rational functions}}$$

Proof. The left-to-right inclusion is immediate, because the identity is a rational function. For the converse inclusion, recall the following characterisation

$$\mathsf{Rat} = \underbrace{\mathsf{Seq}^{\rightarrow}}_{\substack{\text{left-to-right}\\ \text{sequential functions}}} \circ \underbrace{\mathsf{Seq}^{\leftarrow}}_{\substack{\text{right-to-left}\\ \text{sequential functions}}}$$

from Theorem 17.1. By the above, to prove the theorem it is enough to show

$$\mathsf{2Det} \supseteq \mathsf{2Det} \circ \mathsf{Seq}^{\rightarrow} \qquad \mathsf{2Det} \supseteq \mathsf{2Det} \circ \mathsf{Seq}^{\leftarrow}.$$

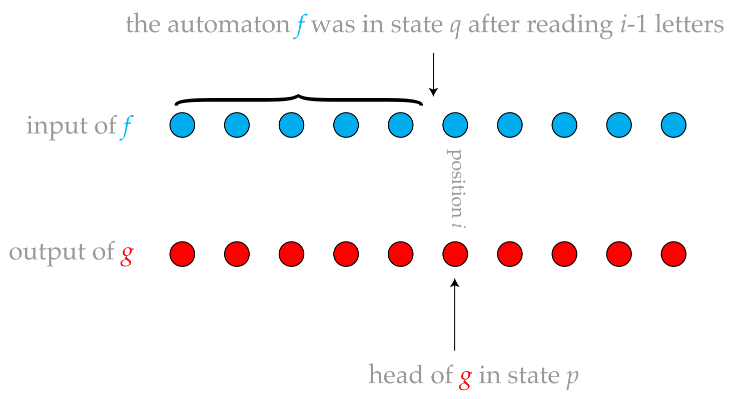
By symmetry of two-way automata, it is enough to prove the first inclusion. Summing up, it suffices to show that if f is left-to-right sequential and g is recognised by a deterministic two-way transducer, as in the following diagram,

$$\begin{tikzcd}
\Sigma^* \arrow[r, "\text{left-to-right sequential } f"] \arrow[dr, "f \circ g"'] & \Gamma^* \arrow[d, "\text{two-way } g"] \\
& \Delta^*
\end{tikzcd}$$

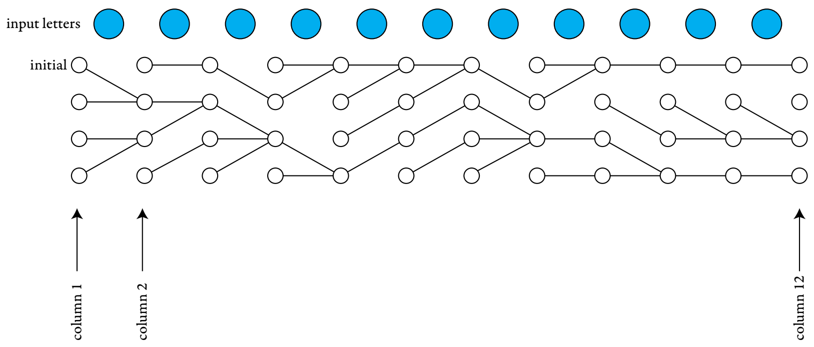
then the composition $f \circ g$ is also recognised by a deterministic two-way automaton. The difficulty is the machines for f and g have different types of movement.

The idea for the proof comes from Hopcroft and Ullman [61, Lemma 3]. To simplify notation, we assume that f is letter-to-letter, i.e. each transition of the underlying DFA with output produces exactly one output letter, and there are no end-of-input words. The proof for the general case – without the letter-to-letter assumption – can be easily inferred from the special case.

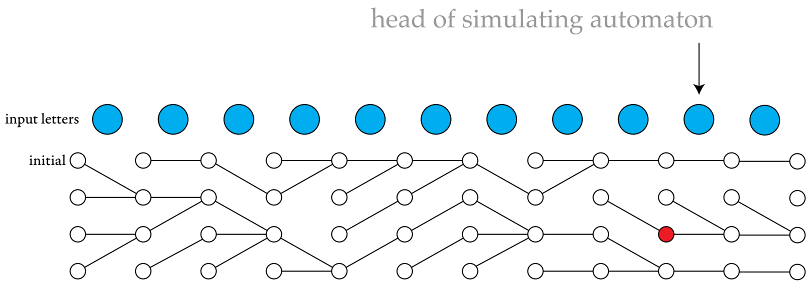
Suppose that the two-way automaton recognising g is in state p over the i-th position of its input (which is the output of f), like in the following picture:


the automaton f was in state q after reading i-1 letters
input of f
position i
output of g
head of g in state p

Then the simulating two-way automaton for the composition $g \circ f$ has its head over the i-th position of the input word (which is the input of f), and knows the states p and q described in the picture above. The question is how to maintain this information, especially when the simulated two-way automaton g wants to move its head to the left. The key insight is to consider the graph which describes the states of f and how they are updated by the transition function. This graph looks likes this:

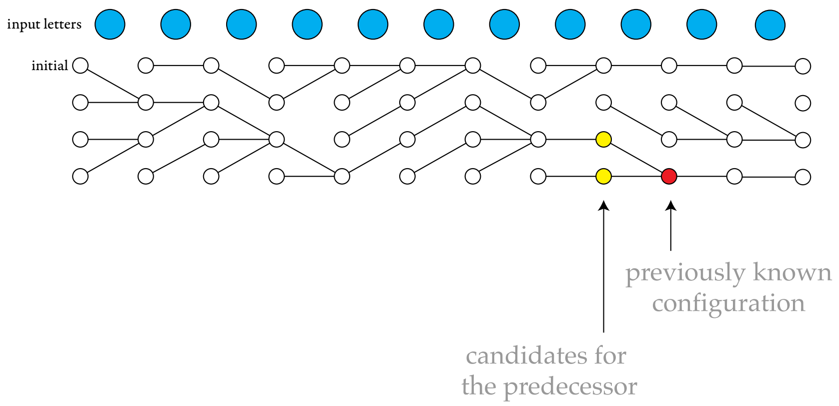


The vertices of the graph are configurations of f, i.e. pairs (state of f, column between positions in the word), and the edges correspond to transitions of the automaton. Each edge is labelled by an output letter. We number the columns beginning with 1. Because f is deterministic, the graph is a forest.
Define q_i to be the state of f in the i-th column, i.e. after reading the first $i - 1$ letters of the input word. The simulating two-way automaton uses the state q_i to get the i-th letter in the output $f(w)$. Suppose that the head of the simulating two-way automaton is over some position i in the input word, and the state q_i of the oracle is known, as indicated by a red circle in the following picture:



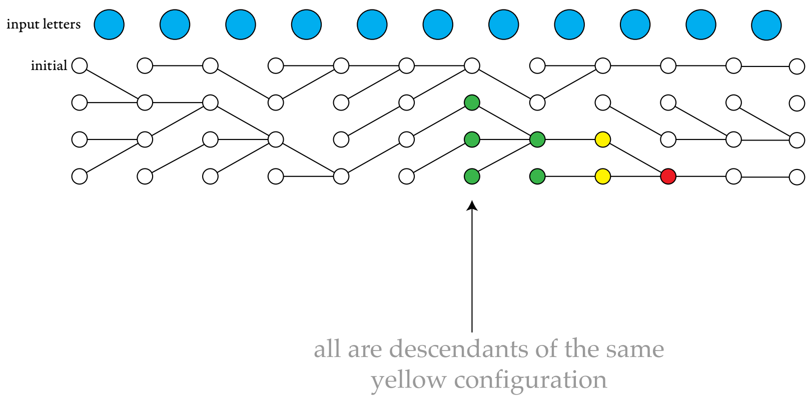
We show below how to maintain the state of f when simulating one transition of the two-way automaton g. If the transition of the two-way automaton g does

not move the head, or moves it to the right, there is no problem, since the transition function of f can be simply applied to the known state q_i.
The issue is when the simulated two-way automaton f wants to move the head to the left, and we need to compute the state q_{i-1}.
Here is the solution. In terms of the forest in the pictures above, we want to determine the unique child of the red node which has the initial configuration in its subtree. To find this unique child, we do the following. We start by moving the head one step to the left, which identifies all possible candidates for the predecessor configurations. Here is the picture, with the candidates being coloured yellow:


input letters
initial
previously known configuration
candidates for the predecessor

If there is only one yellow configuration, i.e. only one candidate for the predecessor, then we are done. The more interesting case is when there is more than one yellow configuration. In this case, we keep moving to the left, and use green to colour all descendants of the yellow configuration (and therefore of the red configuration as well). For each green configuration we remember which of the yellow configurations is its ancestor. Two cases may happen.

1. We might reach a column where all green configurations are descendants of the same yellow configuration, as in this picture:



In this case, the unique yellow configuration is the one that we want to compute. The question is how to return to this unique configuration? The solution is this: suppose that we stopped in column i, i.e. all green configurations in column i are descendants of the same yellow configuration, but this is not true for column $i+1$. We store in our memory the state of the unique yellow configuration that is the ancestor of all green configurations in column i. Then we start moving to the right, storing in each column that states reachable from the green configurations in column $i+1$. We stop when this set becomes a singleton – this happens exactly when we reach the column with the red node. Then we can move one step to the left and use our stored yellow state to determine the predecessor configuration of the red one.

2. The remaining case is when we reach the first column at the beginning of the input. Here we do the same trick to return to the red configuration, and we can keep in our state which branch of the subtree corresponds to the computation of the past oracle.

■

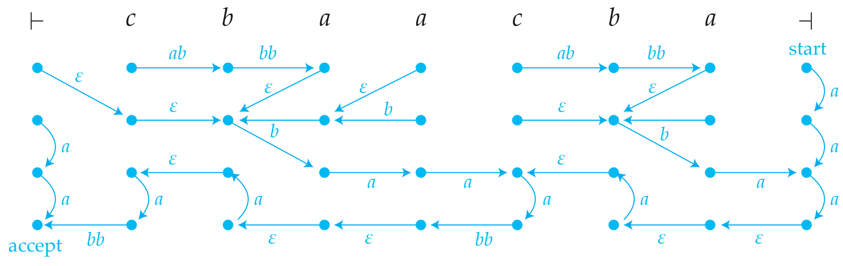
Closure under composition. Using Lemma 17.4 on pre-compositions with rational functions, we complete the proof of Theorem 17.3 on composition closure of deterministic two-way transducers. For our proof, it is more convenient to use a definition – clearly equivalent in terms of expressive power – of two-way transducers where the initial configuration is (initial state, end of input marker $\dashv$).

Fix two deterministic two-way transducers

$$\Sigma^* \xrightarrow{f} \Gamma^* \xrightarrow{g} \Delta^* .$$

We use the following colour coding. The first alphabet Σ is written in black. Blue is used for the states and output alphabet of f. Red is for the states and output alphabet of g. Our goal is to give a deterministic two-way transducer which recognises the composition $g \circ f$.

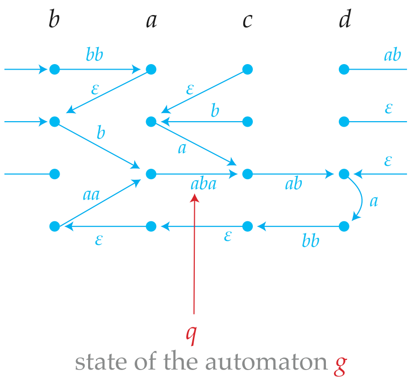
We begin with a naive construction that will not work. Take some input word $w \in \Sigma^*$, and consider the configuration graph of f on this input word, which looks like this:



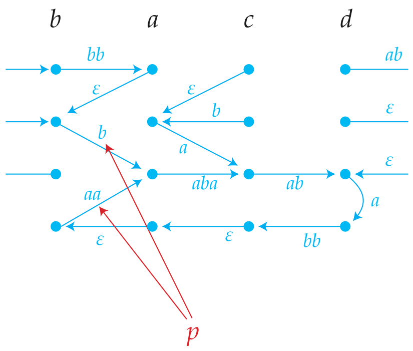
Vertices of the graph – the blue dots – are pairs (state, position in w extended with end markers), and the edges correspond to transitions. The transitions are labelled by output words from the intermediate alphabet Γ. We can represent the configuration graph as a labelling of the input word, with arrows stored in the positions where they originate, and the descriptions of the end markers stored in the adjacent input positions.

The natural construction for the composition $g \circ f$ would be to have an automaton which stores a state of g and a pointer to one of the letters from Γ

that are in the label of an edge in the configuration graph, as in the following picture:

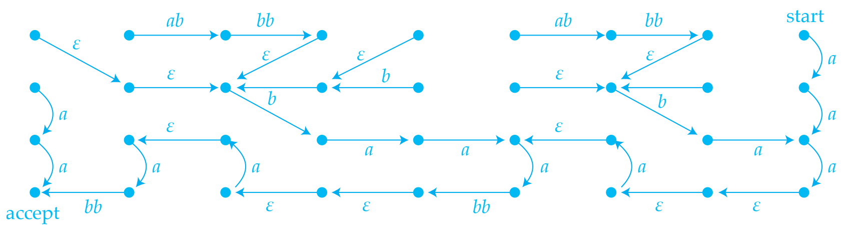


The problem with this construction is that a vertex in the configuration graph might have several incoming edges. For example, suppose that in the situation from the above picture, the automaton g decides to move its head to the left and change the state to p. Then the automaton for the composition $g \circ f$ would not know which of the following two choices should be made:

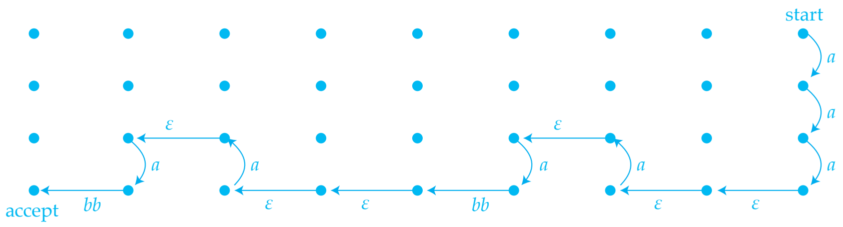


The solution – and also the reason why we use rational preprocessing from Lemma 17.4 – is to restrict the configuration graph of f to edges that are reachable from the initial configuration.

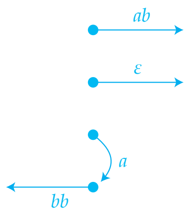
Lemma 17.5. *The following function is rational. The input is a configuration graph of f, like this:*



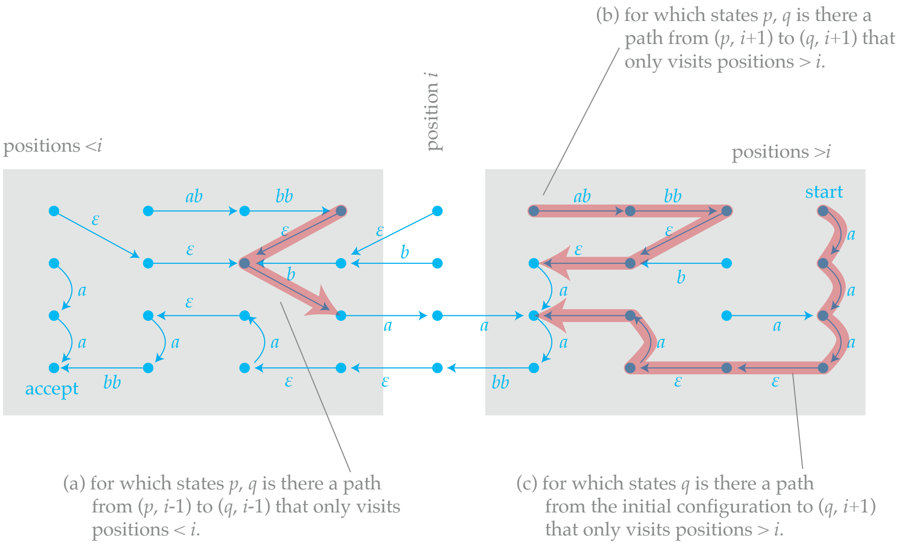
The output is the same graph, but only with those edges that are reachable from the initial configuration, like this:



Proof. We assume that a configuration graph is represented as word where each letter represents the outgoing transitions from one column (i.e. position in the input word with end markers). Here is a picture of a letter

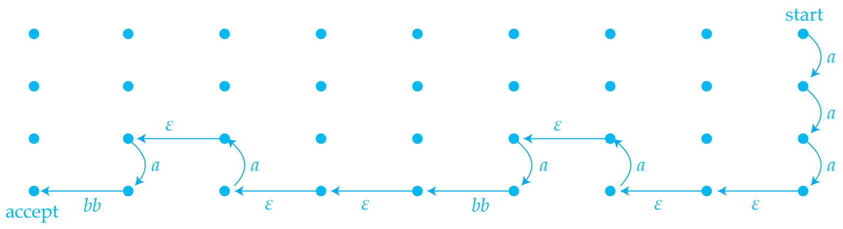


For this claim, it is convenient to use an Eilenberg bimachine as the representation of rational transducers. Given a position i in a configuration graph, the bimachine generates the following information:


(b) for which states p, q is there a path from (p, i+1) to (q, i+1) that only visits positions > i.
positions <i
position i
positions >i
start
accept
(a) for which states p, q is there a path from (p, i-1) to (q, i-1) that only visits positions < i.
(c) for which states q is there a path from the initial configuration to (q, i+1) that only visits positions > i.

The information can be generated by deterministic automata, as required by the definition of an Eilenberg bimachine, using the standard conversion of two-way automata (without output) to one-way automata. Based on this information and the label of the position i, one can determine which states in position i are reachable from the initial configuration. In its output, the bimachine only leaves edges that originate from reachable states. ■

By Lemma 17.4, deterministic two-way transducers are closed under rational preprocessing, and by Lemma 17.5 a rational function can restrict a configuration graph to reachable configurations. Therefore, in order to find a deterministic two-way transducer for the composition $g \circ f$, it suffices to give a deterministic two-way transducer which inputs configuration graph of f restricted to reachable configurations, like this:



and outputs the value of g on the labelling of the unique path from the starting configuration to the accepting configuration. Since the blue nodes have indegree at most one, this can be done using the naive construction described before Lemma 17.5.

Problem 131. Show that deterministic two-way automata (seen as acceptors of words) can be complemented with polynomial blowup.

Problem 132. Consider a sequential transducer, which defines a function $f : \Sigma^\omega \to \Gamma^\omega$. Show that this function is continuous with respect to the distance defined in Problem 62.

Problem 133. Show that the reverse function is not left-to-right sequential.

Problem 134. Which of the following functions over a unary alphabet are sequential?

1. $a^n \mapsto a^{n^2}$;
2. $a^n \mapsto a^{\lfloor \sqrt{n} \rfloor}$.

Problem 135. Show that the duplication function $w \mapsto ww$ is not rational.

Problem 136. Show that left-to-right sequential functions are closed under compositon, i.e.

$$\mathsf{Seq}^{\rightarrow} = \mathsf{Seq}^{\rightarrow} \circ \mathsf{Seq}^{\rightarrow}.$$

Problem 137. Show that rational functions are closed under compositon, i.e.

$$\mathsf{Rat} = \mathsf{Rat} \circ \mathsf{Rat}.$$

Problem 138. Show that if f is recognised by a deterministic two-way transducer and and g is rational (with suitable input and output alphabets), then $g \circ f$ is recognised by a deterministic two-way transducer.

Problem 139. Consider nondeterministic two-way automata with output. Show that for every nondeterministic two-way automaton with output $\mathcal{A}$ there is a deterministic two-way automaton with output $\mathcal{B}$ that uniformises it in the following sense: for every input word, $\mathcal{B}$ produces one of the outputs of $\mathcal{A}$. (If there is no output of $\mathcal{A}$, then also there is no output of $\mathcal{B}$.)

Problem 140. Show that the following problem is in polynomial time: given two letter-to-letter (i.e. each transition produces exactly one letter) left-to-right sequential functions with the same input alphabet, decide if for every input they produce the same output.

Problem 141. Show that the following problem is undecidable: given two left-to-right sequential functions with the same input alphabet, decide if for some input, they produce the same output.

18
Streaming string transducers

In this chapter we present a one-way automaton model that has the same expressive power as two-way transducers.
We begin by defining register transducers, which are automata that use registers to store parts of their output. We have already seen register transducers in Chapter 6 – in a more general setting, for arbitrary algebras – and we have even proved in Corollary **??** that their equivalence is decidable for the specific algebra of words with concatenation that we use in this chapter. To make this chapter self-contained, we give a stand-alone definition below.
Register transducers, as defined below, will turn out to be strictly more powerful than two-way transducers, but a model with the same expressive power as two-way transducers will be recovered by placing a certain copyless restriction on the register updates.

Definition 18.1. *A* register transducer *consists of:*

- *finite* input and output alphabets Σ *and* Γ*;*
- *a finite set of* states Q*;*
- *a finite set of* registers R*;*
- *an* initial configuration *in* $Q \times (R \to \Gamma^*)$*;*

- *a* transition function

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

- *an* output function

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

The automaton is run as follows. Define a *register valuation* to be any function from registers to words over the output alphabet Γ, and define a *register update* to be any function from registers to words over the alphabet $R + \Gamma$. There is an action of updates on valuations

$$(v \in \text{register valuations}, \tau \in \text{register update}) \quad \mapsto \quad v \cdot \tau \in \text{register valuations}$$

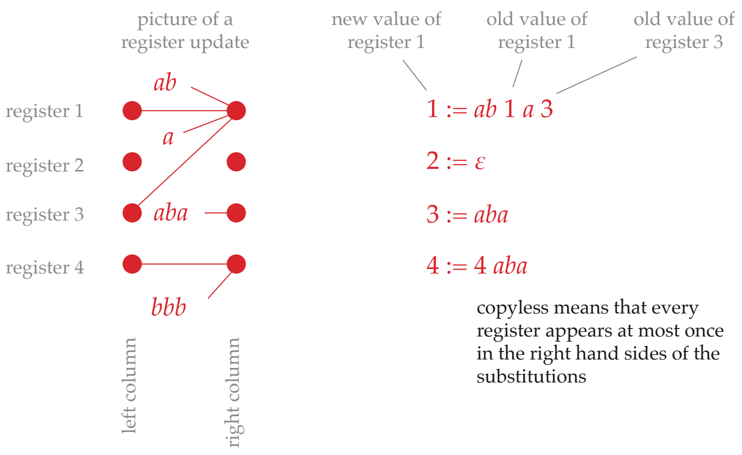
where $v \cdot \tau$ is obtained from τ by replacing each register name with its contents under τ. A *configuration* of the automaton is defined to be a pair (state, register valuation). The automaton begins in the initial configuration. When reading an input letter a, the automaton uses its transition function to determine its new state and the register update. More formally, the configuration is updated as follows:

$$(q, v) \cdot a \overset{\text{def}}{=} (p, v\tau) \qquad \text{where } \delta(q, a) = (p, \tau).$$

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

Example 44. Here is an automaton where the input and output alphabets are $\{a\}$, and the recognised function is $a^n \mapsto a^{5+3\cdot 2^n}$. The automaton has one register and one state. The initial configuration stores the word a in the unique register. When reading an input letter, the unique register r is updated by $r := rr$. The output function maps the unique state to $aaaaarrr$.
The function recognised by this register transducer is not recognised by any two-way transducer. There reason is that the function has exponential growth, while a two-way transducer has necessarily at most linear blowup, because a position in the input word can be visited at most once for each state. □

Copyless restriction. As argued in Example 44, register transducers can have exponential growth, and therefore are not in general equivalent to deterministic two-way transducers. To recover equivalence with two-way transducers, we use the *copyless restriction* (also known as the *single use restriction*) described in the following picture:



In other words, a register update is copyless if every vertex in the left column has outdegree at most one. The intuition is that the register contents are physical objects and can only be moved around and not duplicated.

Definition 18.2. *A* streaming string transducer[1] *is a register transducer where the transition function produces only copyless register updates.*

The output function need not be copyless. Requiring it to be copyless would not weaken the model, though, because the output function is applied only once. For example, if the output function uses each register at most k times, then by taking k disjoint copies of the registers we can make the output function copyless.

[1]The model and name of streaming string transducers comes from [3], although similar and essentially equivalent models have been known before in the literature on attributed grammars, e.g. attributed tree transducers from [30].

The goal of this chapter is to prove that streaming string transducers are equivalent to deterministic two-way transducers.

Theorem 18.3. *Streaming string transducers recognise the same word-to-word functions as deterministic two-way transducers*

The above theorem was proved by Alur and Cerny in [3]. A similar result (using a model of streaming string transducers with lookahead) can also be recovered from earlier work of Bloem, Engelfriet and Hogeboom: (a) MSO transductions are equivalent to deterministic two-way transducers [27]; and (b) MSO transductions are equivalent (even over trees) to a certain kind of attribute transducers [7].
We begin by describing the proof strategy. Our goal is to prove the equality

$$\underbrace{\mathsf{SST}}_{\substack{\text{functions recognised by}\\ \text{streaming string transducers}}} = \underbrace{\mathsf{2Det}.}_{\substack{\text{functions recognised by}\\ \text{deterministic two-way transducers}}} \tag{18.1}$$

As in the proof of Theorem 17.4, we write Rat for the class of rational functions. In Section 18.1, we prove the following inclusions

$$\mathsf{SST} \overset{\text{Lemma 18.4}}{\subseteq} \mathsf{2Det} \circ \mathsf{Rat} \quad \text{and} \quad \mathsf{SST} \circ \mathsf{Rat} \overset{\text{Lemma 18.5}}{\supseteq} \mathsf{2Det}.$$

In other words, every streaming string transducer can be recognised by a deterministic two-way automaton with preprocessing by a rational function, and likewise in the opposite direction. Rational functions are easily seen to be closed under composition, using a straightforward product construction, see Exercise 137. Combining the inclusions from Lemmas 18.4 and 18.5, and using closure of rational functions under composition, we get

$$\mathsf{SST} \circ \mathsf{Rat} = \mathsf{2Det} \circ \mathsf{Rat}. \tag{18.2}$$

To finish the proof of Theorem 18.3, it suffices to show that both streaming string transducers and deterministic two-way transducers are closed under preprocessing with rational functions. For deterministic two-way transducers, this was shown in Theorem 17.4 from Chapter 17. For streaming string transducers, this will be done in Lemma 18.7, which is the most challenging construction in this chapter. Combining these results, we get

$$\mathsf{SST} \overset{\text{Lemma 18.7}}{=} \mathsf{SST} \circ \mathsf{Rat} \overset{(18.2)}{=} \mathsf{2Det} \circ \mathsf{Rat} \overset{\text{Theorem 17.4}}{=} \mathsf{2Det}$$

which completes the proof of Theorem 18.3. It remains to prove Lemmas 18.4, 18.5 and 18.7.

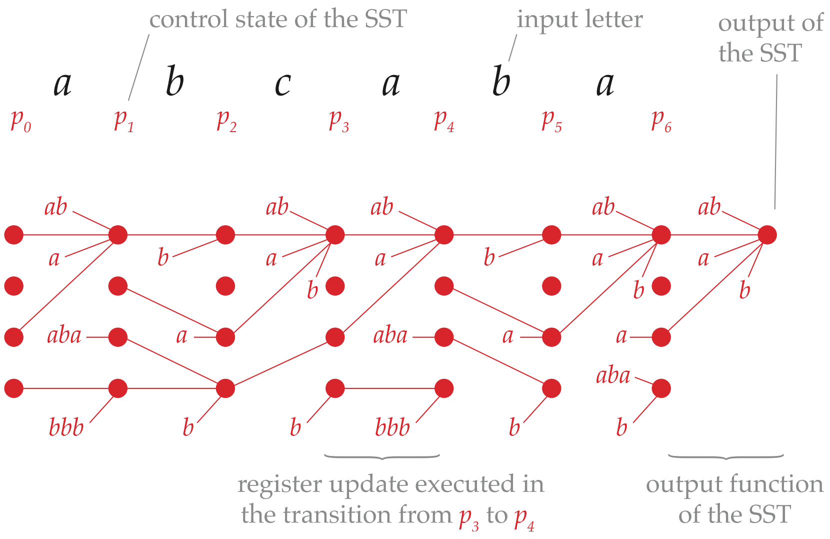
18.1 Equivalence after rational preprocessing

In this section, we prove that streaming string transducers and deterministic two-way transducers are equivalent if we allow rational preprocessing

Lemma 18.4. *Every streaming string transducer can be decomposed as a rational function followed by a deterministic two-way transducer. In other words*

$$\mathsf{SST} \subseteq \mathsf{2Det} \circ \mathsf{Rat}.$$

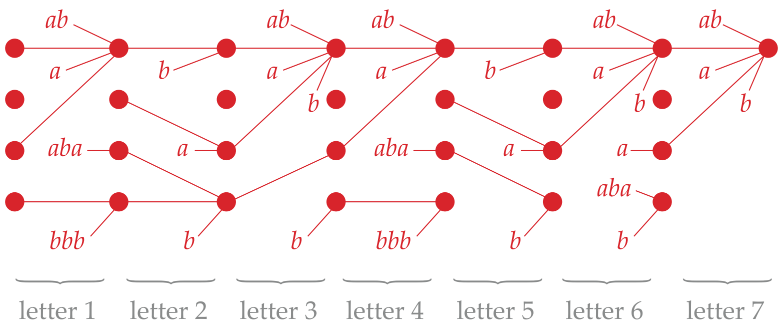
Proof. Fix a streaming string transducer. A run of the transducer looks like this:


control state of the SST
input letter
output of the SST
a b c a b a
p_0 p_1 p_2 p_3 p_4 p_5 p_6
ab a aba bbb b ab a b a b aba bbb b a b ab a b aba b
register update executed in the transition from p_3 to p_4
output function of the SST

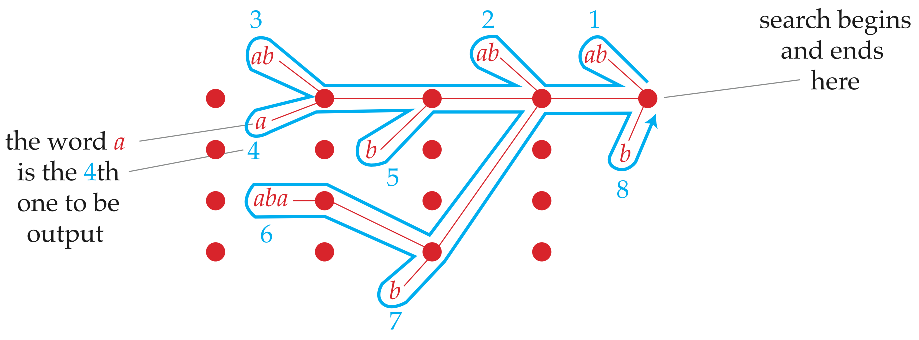
It is not hard to see that there is a rational – in fact left-to-right sequential – transducer which transforms an input word

$$a \qquad b \qquad c \qquad a \qquad b \qquad a$$

to a word describing the corresponding sequence of register updates:



By using the above rational transducer as a preprocessor, to prove the lemma it is enough to find a deterministic two-way transducer which inputs a tree that describes the register updates, and outputs the final value. To do this, we use a depth-first search through the tree as explained in the following picture

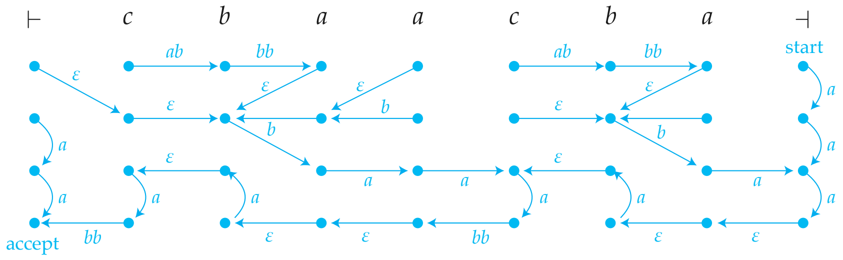

3
2
1
search begins and ends here
ab
ab
ab
a
4
b
5
b
8
the word a is the 4th one to be output
aba
6
b
7

It is easy to implement a depth-first search using a deterministic two-way automaton. One simply has to remember the current register and the direction from which it came. ∎

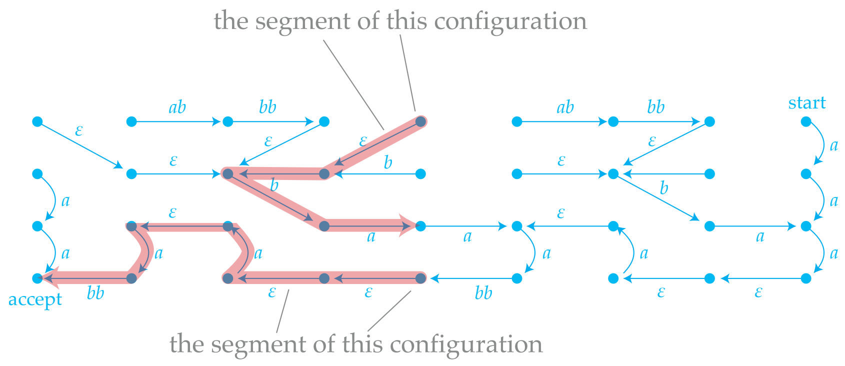
Lemma 18.5. *Every deterministic two-way transducer can be decomposed as a rational function followed by a streaming string transducer. In other words*

$$\mathsf{2Det} \subseteq \mathsf{SST} \circ \mathsf{Rat}.$$

Proof. As in the proof of Theorem 17.3, it is more convenient to use a definition of two-way transducers where the initial configuration is (initial state, end of input marker ⊣). Consider the configuration graph of the two-way automaton over a given input word, as in the following picture:

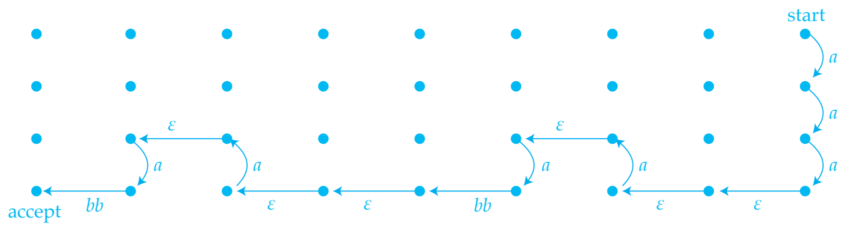


We begin with a naive idea, which will not work because of the copyless restriction. For a vertex in the configuration graph, define its *segment* to be the (unique, by determinism) path that begins in the configuration, and is cut off at the first visit to the same column as the source configuration, as in the following picture:


the segment of this configuration
the segment of this configuration
start
accept
ab
bb
ε
b
a

The segment might accept/reject/loop without returning to the column of the source configuration. The naive idea would be to store for each state q the output word that is found by reading the labels on the segment of the configuration that has state q in last read position. The problem with this construction is that it violates the copyless restriction, because configurations can have more than one incoming edge, and therefore the labels of one segment can be shared by several longer segments.

Like in the proof of Theorem 17.3, the solution is to restrict the configuration graph to edges that are reachable from the initial configuration. As shown in Lemma 17.5, a rational function can be used to restrict the configuration graph to reachable configurations, so that the result looks like this:



When only reachable edges are used, the indegree is at most one, because otherwise the automaton would loop, which cannot happen by the assumption that it defines a total function. Using the naive idea, one can write a streaming string transducer which inputs a configuration graph with only reachable edges

– represented as a word over a finite alphabet in any natural way – and outputs the label of the segment corresponding to the initial configuration. ■

18.2 *Lookahead removal*

In this section we show that functions recognised by register transducers and streaming string transducers are closed under pre-composition with rational functions.

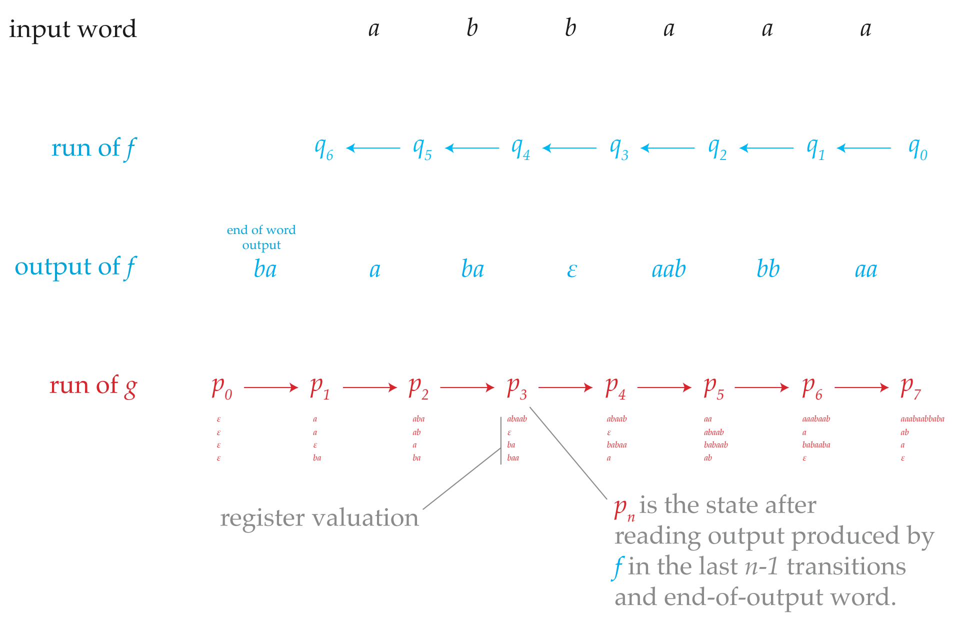
A different perspective on this result is that register transducers and streaming string transducers would not become more expressive if equipped with an oracle that gives regular information about the input word to the left and right of the head. Since the information about the word to the left of the head can be stored in the state, the interesting part of the oracle is the one that talks about the word to the right of the head. In other words, in this section we show that lookahead can be eliminated from the transducers without affecting expressive power.

Lemma 18.6. *Functions recognised by register transducers are closed under pre-composition with rational functions.*

Proof. Consider functions

$$\begin{tikzcd}\Sigma^* \arrow[r, "f"] & \Gamma^* \arrow[r, "g"] & \Delta^*\end{tikzcd}$$

such that f is rational and g is recognised by a register automaton. We use the following colour coding. The first alphabet Σ is written in black. Blue is used for the states and output alphabet of f. Red is for the states and output alphabet of g. A run of the composition $g \circ f$ looks like this:



The register transducer for the composition $f \circ g$ stores a function

$$\text{states of lookahead } f \qquad \to \qquad \text{configurations of } g$$

which maps a state q of f to the configuration that would be used by g assuming that q is the state of the lookahead f after reading the unread part of the input (in a right-to-left pass). Such a function can be represented by using

$$(\text{number of states in lookahead } f) \times (\text{number of registers in } g)$$

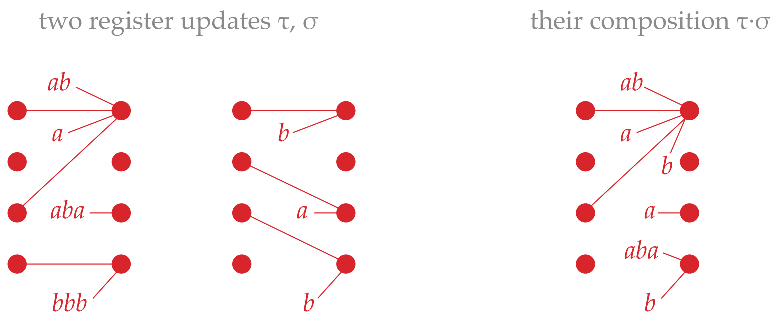
registers; and the representation can be updated in the transition function. After reading the entire word, the transducer for the composition looks at the value of the function under the initial state of f, and then applies the output function of g. ■

The construction in the above lemma cannot be used for streaming string transducers because it violates the copyless restriction. The violation comes from merging states in the right-to-left sequential function f. For example, suppose that the state transformation of f over some input letter $a \in \Sigma$ looks like this:

$$\begin{tikzcd}
q_0 & q_0 \arrow[dl, "\varepsilon"'] \\
q_1 & q_1 \arrow[l, "aaa"'] \\
q_2 & q_2 \arrow[l, "ba"'] \\
q_3 & q_3 \arrow[l, "\varepsilon"']
\end{tikzcd}$$

Then the register transducer described in the proof of Lemma 18.6 would duplicate the information stored for state q_1, using it for both q_0 and q_1. To eliminate lookahead for streaming string transducers, we use a data structure, called a transformation forest, which stores register updates organised in a forest structure so that composition can be done without copying. We describe this data structure below.

Composing register updates. We begin with defining a composition operation on register updates. Here is the picture:

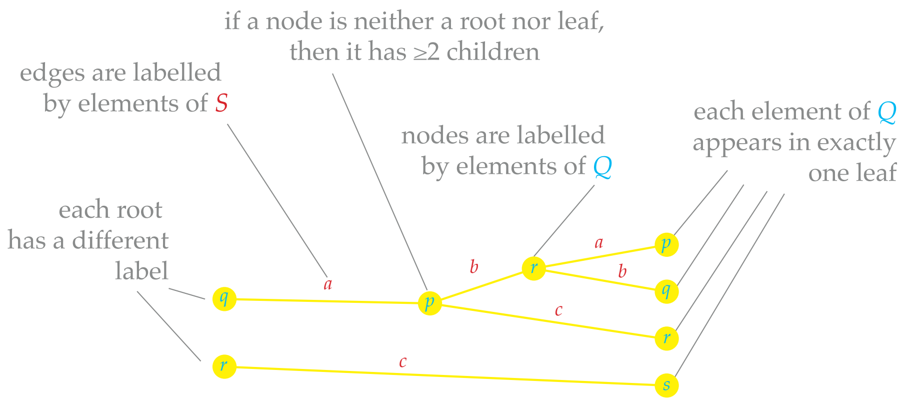


The composition operation is defined so that if τ, σ are two register updates and v is a register valuation, then

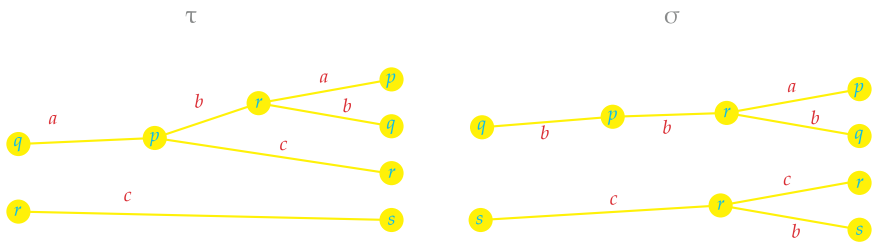
$$v \cdot (\tau \cdot \sigma) = (v \cdot \tau) \cdot \sigma.$$

Using the above composition, we can view the set of register updates – for a fixed set of register names and output alphabet – as a monoid.

Transformation forests. Suppose that M is a monoid and Q is a finite set. (Our intended application is that S is the monoid of register updates for some streaming string transducer, but the abstract definition requires less notation.) Define a *transformation forest* (over M and Q) to be any labelled forest of the following form:

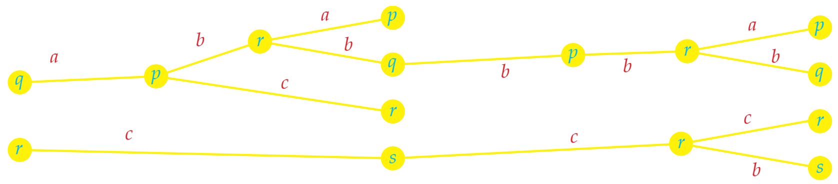

if a node is neither a root nor leaf, then it has ≥2 children
edges are labelled by elements of S
nodes are labelled by elements of Q
each element of Q appears in exactly one leaf
each root has a different label
q
a
p
b
r
a
p
b
q
c
r
r
c
s

We now describe how transformation forests can be composed. Suppose that we have two transformation forests τ and σ, as illustrated below:

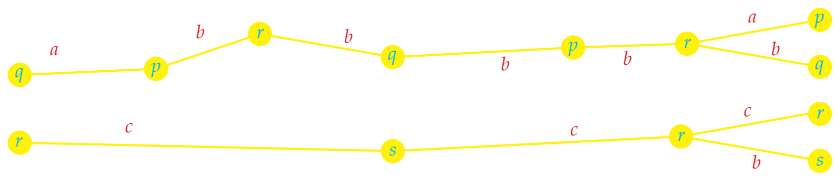


Their composition $\tau\sigma$ is obtained by doing the following steps.

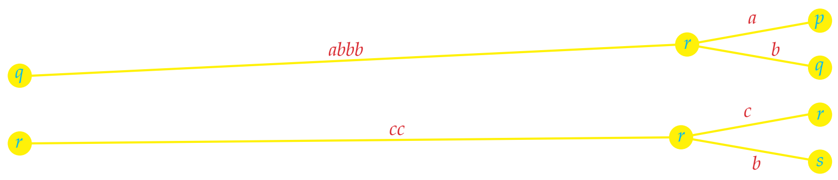
1. To each root of σ we can associate a unique leaf of τ with the same label, because roots of σ have different labels and all labels appear in leaves of τ. Merge each root of σ with the associated leaf of τ:



2. Eliminate nodes that do not reach any node leaf of σ:



3. Contract into a single edge every path that uses only nodes with unary branching (except the source and target):



   The label of a contracted path is the product, in the semigroup S, of the labels of edges on the path before the contraction.

It is not hard to see that this operation is associative, i.e.

$$\tau(\sigma\rho) = (\tau\sigma)\rho.$$

Also, there is a neutral element, namely the transformation forest where each leaf is a root (and there are no edges). Therefore, the set of transformation forests is a monoid, which we denote by $M^{[Q]}$. The reader might recognise transformation forests from Lemma 10.6 from Chapter 10. In that lemma, the monoid M had two elements "accepting" and "non-accepting". In this chapter, M will be the infinite monoid of copyless register updates.

Lookahead elimination for streaming string transducers. Equipped with the data structure of transformation forests, we are ready to prove the copyless variant of Lemma 18.6.

Lemma 18.7. *Functions recognised by streaming string transducers are closed under pre-composition with rational functions. In other words*

$$\mathsf{SST} = \mathsf{SST} \circ \mathsf{Rat}.$$

Proof. The left-to-right inclusion is immediate, since the identity is a rational function. For the converse inclusion, recall the following equality

$$\mathsf{Rat} \quad = \quad \underbrace{\mathsf{Seq}^{\rightarrow}}_{\substack{\text{left-to-right}\\ \text{sequential functions}}} \quad \circ \quad \underbrace{\mathsf{Seq}^{\leftarrow}}_{\substack{\text{right-to-left}\\ \text{sequential functions}}}$$

from Theorem 17.1. Since both streaming string transducers and left-to-right sequential functions are instances of left-to-right automata, a straightforward product construction can be used to yield the inclusion

$$\mathsf{SST} \supseteq \mathsf{SST} \circ \mathsf{Seq}^{\rightarrow}$$

Therefore, in order to prove the lemma it suffices to show

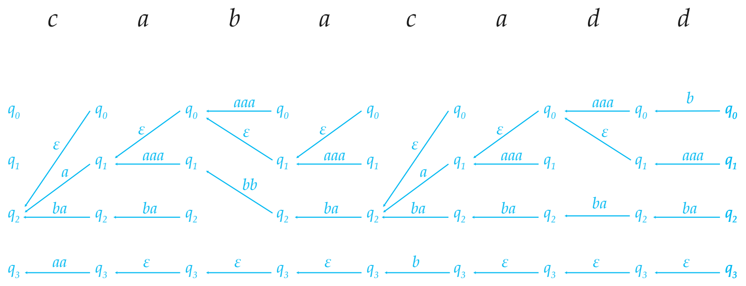
$$\mathsf{SST} = \mathsf{SST} \circ \mathsf{Seq}^{\leftarrow}.$$

Here we cannot use a simple product construction, because we compose automata that move in different directions. The rest of the proof is devoted to proving the above inclusion. We use the same notation and colour convention as in the proof of Lemma 18.6. Let

$$\begin{tikzcd}\Sigma^* \arrow[r, "f"] & \Gamma^* \arrow[r, "g"] & \Delta^*\end{tikzcd}$$

be functions such that f is right-to-left sequential and g is a streaming string transducer. Our goal is to design a streaming string transducer that recognises the composition $g \circ f$. To make notation lighter, we assume that f has empty end-of-input words. This assumption can be lifted without greater conceptual difficulty.

Overview of the construction. The idea is that instead of storing register valuations, the streaming string transducer for $g \circ f$ will store register updates, organised in a transformation forest. To illustrate this idea, consider the configuration graph of the right-to-left sequential function f over an input word $w \in \Sigma^*$, as shown in the following picture:

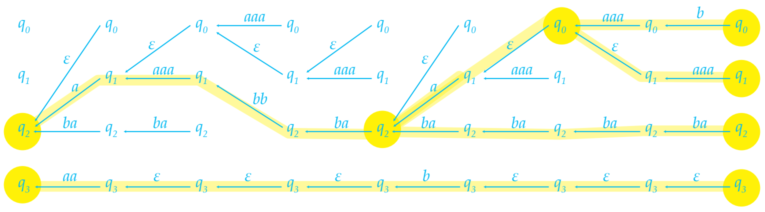


Nodes of the configuration graph are labelled by states of f and edges are labelled by output words of f. Because the f is right-to-left deterministic, the configuration graph is a forest, with the roots in the first column. The output of f is obtained by reading from left to right the labels on the path that goes from the unique leaf with the initial state of f to the unique root that is its ancestor. (We use the assumption that the end-of-input words are empty; otherwise we would need to add one more column at the left end of the picture.)

The automaton recognising the composition $g \circ f$ will store in its configuration a transformation forest

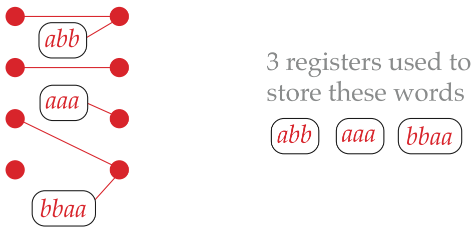
$$t \in \underbrace{(\text{register updates of } g)}_{\substack{\text{monoid of copyless register} \\ \text{updates for registers and} \\ \text{output alphabet of } g}}{}^{[\text{states of } f]}.$$

The nodes of this transformation forest will correspond to the leaves of the configuration graph, their closest common ancestors, and the roots that are reachable from leaves, as represented by the big yellow circles below:



For a path connecting two adjacent yellow nodes, the transformation forest t will store the register update done by g on that path. To describe the automaton in more detail, we begin by discussing how copyless register updates, and therefore also transformation forests over the monoid of copyless register updates, can be stored in the configuration of a streaming string transducer.

Storing register updates. Recall the graphical representation of register updates that was used when defining the copyless restriction. A copyless register update can be stored by a streaming string transducer like this:



In general, to store a copyless register update we need a bounded number of bits to store the tree structure of the update plus

$$2 \cdot (\text{number of registers in } g)$$

registers to store the output words used in the update. To store a transformation forest

$$t \in (\text{register updates of } g)^{[\text{states of } f]}.$$

we use a bounded number of bits to store the structure of the forest and its labelling by states of f, plus

$$\underbrace{2 \cdot (\text{number of registers in } g)}_{\substack{\text{registers to store}\\ \text{a register update}}} \cdot \underbrace{2 \cdot (\text{states in } f)}_{\substack{\text{number of edges in}\\ \text{a transformation forest}}}$$

registers to store the register updates. The following claim says that transformation forests can be updated in a copyless way.

Claim 18.8. *Fix a transformation forest*

$$s \in (\textit{register updates of } g)^{[\textit{states of } f]}.$$

Then the function

$$t \in (\textit{register updates of } g)^{[\textit{states of } f]} \quad \mapsto \quad ts \in (\textit{register updates of } g)^{[\textit{states of } f]}$$

can be done using a copyless register update.

Proof. Almost by definition, copyless register updates can be composed using a copyless register update. The same is true when composing transformation forests ts, because each label from t and each label from s is used at most once in the composition. In fact, copyless register updates can be seen as a special case of transformation forests, see Exercise 146. ■

The automaton. Before describing the automaton, let us introduce some notation that will be used in its definition and correctness proof. Let q be a state of f and let p be a state of g. Define f_q to be the right-to-left sequential function obtained from f by changing the initial state to q and define $[p, w, q]$ to be the run of g – viewed as a sequence of transitions – which begins in state p and reads the word $f_q(w)$. We have the following equality, which is obtained by unravelling the definitions:

$$[p, wa, q] = [p, w, aq] \cdot [p(f_q(a)), a, q] \qquad \text{for every } w \in \Sigma^* \text{ and } a \in \Sigma. \tag{18.3}$$

In the above, we write $\_q$ and $p\_$ for the state transformations of the automata underlying f and g.

Equipped with the above notation, we are ready to define the streaming string transducer recognising the composition $g \circ f$. After reading an input word $w \in \Sigma^*$, the transducer will store a transformation forest

$$t_w \in (\text{register updates of } g)^{[\text{states of } f]}$$

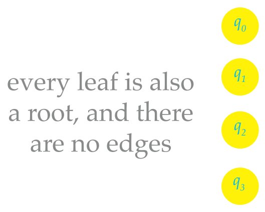
whose intuitive meaning was described at the beginning of the proof. The transformation forest t_w is stored as described before Claim 18.8, and it satisfies the following invariant:

(*) Let q be a state of f and let π be the unique root-to-leaf path in t_w that ends in a leaf with label q. Then the composition of register updates labelling π is the same as the register update done by the run $[\text{initial state of } g, w, q]$.

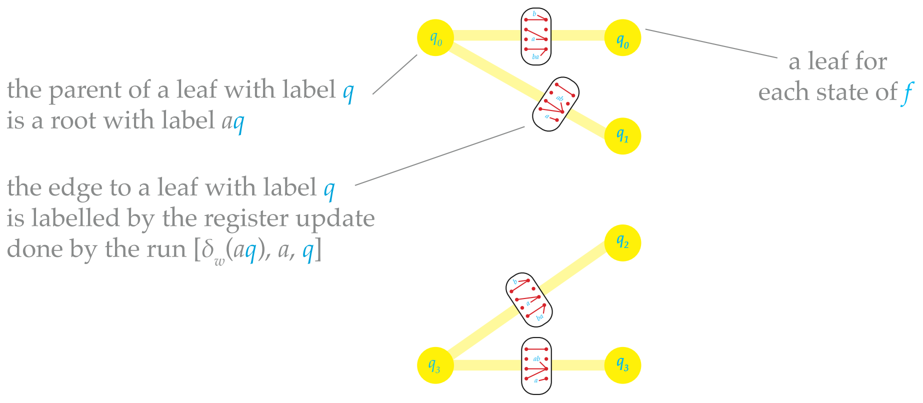
To update its configuration, the transducer will also store in its finite state space the function δ_w defined by

$$q \in \text{states of } f \quad \mapsto \quad \text{target state of the run } [\text{initial state of } g, w, q]$$

Using (18.3), it is not hard to see how δ_{wa} can be computed from δ_w and an input letter a. It remains to show how to update the transformation forest t_w. Initially, t_ε is a forest with no edges and one leaf per state of f, like this



and therefore the invariant (*) is satisfied because π is the empty path which yields an identity register update. When reading a letter a, the transformation forest is updated as follows. The new transformation forest t_{wa} is defined to be the composition – in the monoid of transformation forests – of t_w with the following transformation forest:


the parent of a leaf with label q is a root with label aq
the edge to a leaf with label q is labelled by the register update done by the run $[\delta_w(aq), a, q]$
a leaf for each state of f
q_0
q_0
q_1
q_2
q_3
q_3

Using the equality (18.3), it is not hard to check that t_{wa} satisfies the invariant. Furthermore, the update can be done while preserving the copyless discipline, by Claim 18.8.

It remains to define the output function so that the automaton recognises the composition $g \circ f$. By the invariant, once the automaton has finished processing an input w, by looking at the transformation forest t_w we can recover the register update τ that is done by the run of g on $f(w)$, i.e. the run

$$[\text{initial state of } g,\ w,\ \text{initial state of } f].$$

To get the output of $g \circ f$ on w, it remains to apply τ to the empty register valuation, and finally apply the output function of g to the resulting register valuation. All of this can be done using the register representation of the transformation forest t_w. ■

Problem 142. Show that the class SST of functions recognised by streaming string transducers has the following closure properties:

1. if f, g are in SST, then so is $w \mapsto f(w)g(w)$.

2. if f is in SST, then so is $w \mapsto$ reverse of $f(w)$.

Problem 143. Show that the class of regular languages is closed under inverse images of streaming string transducers, but not under forward images.

Problem 144. Show that a language $L \subseteq \Sigma^*$ is regular if and only if there is a streaming string transducer with input alphabet Σ and output alphabet $\{0, 1\}$ which recognises the characteristic function of L.

Problem 145. Define a *nondeterministic* streaming string transducer by (a) allowing several applicable transitions in each state; (b) distinguishing accepting states, so that only runs that end in an accepting state count. A *functional* streaming string transducer is a nondeterministic one where every accepting run produces the same output. Show that functional streaming string transducers recognise the same functions as deterministic ones.

Problem 146. Consider the least class of monoids that contains Γ^*, and is closed under:

- reversing the monoid operation, i.e. $m \cdot n$ becomes $n \cdot m$;

- submonoids;

- homomorphic images;

- if M is the class, then so is $M^{[Q]}$ for every finite set Q.

Show that this class contains, for any finite set R of registers, the monoid of copyless register updates with alphabet Γ and registers R.

Problem 147. A streaming string transducer is called *monotone* if its registers can be totally ordered as $r_1, \ldots, r_n$ so that every register update τ preserves the order in the following sense: after concatenating the words $\tau(r_1), \ldots, \tau(r_n)$ and keeping only the register names, we get a subsequence of $r_1, \ldots, r_n$. Show that every streaming string transducer can be decomposed as $g \circ f$ where f is a rational function and g is a monotone streaming string transducer.

Problem 148. Show that the following problem is PSPACE-hard (it is also in PSPACE, but this is more challenging to prove): given a streaming string transducer, decide if it produces the empty word for every input.

19
Learning automata

This chapter is about learning regular languages of finite words. All automata here are deterministic finite automata. The setup is that there are two parties: Learner and Teacher. Teacher knows a regular language. Learner wants to learn this language, and pursues this goal by asking two types of queries to the Teacher:

- *Membership.* In a membership query, Learner gives a word, and the Teacher says whether or not Teacher's language contains that word.
- *Equivalence.* In an equivalence query, Learner gives regular language, represented by an automaton, and Teacher replies whether or not the Teacher's and Learner's languages are equal. If yes, the protocol is finished. If no, Teacher gives a counterexample, i.e. a word where the Teacher's and Learner's languages disagree.

Membership queries on their own can never be enough to identify the language, since there are infinitely many regular languages that match any finite set of membership queries. Given enough time, equivalence queries alone are sufficient: Learner can enumerate all regular languages, and ask equivalence queries until the correct language is reached, without ever using membership queries. The lecture is about a more practical solution, which was found by Dana Angluin [4]. Angluin's algorithm is a protocol where Learner learns Teacher's language in a number of queries that is polynomial in:

- the minimal automaton of Teacher's language;
- the size of Teacher's counterexamples.

If Teacher provides counterexamples of minimal size, then the second parameter above is superfluous, i.e. the number of queries will be polynomial in the minimal automaton of Teacher's language. As mentioned above, we only talk about deterministic automata, and therefore the minimal automaton refers to the minimal deterministic automaton.

State words and test words. Suppose that Teacher's language is $L \subseteq \Sigma^*$. We assume that the alphabet is known to both parties, but the language is only known to Teacher. At each step of the algorithm, Learner will store an approximation of the minimal automaton of L, described by two sets of words:

- a set $Q \subseteq \Sigma^*$ of state words, closed under prefixes;
- a set $T \subseteq \Sigma^*$ of test words, closed under suffixes.

The idea is that the state words are all distinct with respect to Myhill-Nerode equivalence for Teacher's language, and the test words prove this. This idea is formalised in the following definitions.

Correctness and completeness. If T is a set of test words, we say that words $v, w \in \Sigma^*$ are T-equivalent if

$$wu \in L \quad \text{iff} \quad vu \in L \qquad \text{for every } u \in T$$

This is an equivalence relation, which is coarser or equal to the Myhill-Nerode equivalence relation of Teacher's language. In terms of T-equivalence we define the following properties of sets $Q, T \subseteq \Sigma^*$ that will be used in the algorithm:

- *Correctness.* All words in Q are pairwise T-non-equivalent;

- *Completeness.* For every $q \in Q$ and $a \in \Sigma$, there is some $p \in Q$ that is T-equivalent to qa.

If (Q, T) is correct and complete, then we can define an automaton as follows. The states are Q, the initial state being the empty word. When the automaton is in state $q \in Q$ and reads a letter a, it goes to the state p described in the completeness property; this state is unique by the correctness property. The accepting states are those states that are in Teacher's language.

Lemma 19.1. *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 19.1, 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 19.2. *After step 6, (Q, T) is no longer complete.*

Proof. Let (Q, T) be the pair in step 4, and let $a_1 \cdots a_n$ be the counterexample, which witnesses that the automaton for (Q, T) does not recognise Teacher's language. Define T' to be T plus all suffixes of the counterexample, and suppose toward a contradiction that (Q, T') is complete. If (Q, T') is complete, then the automata for (Q, T) and (Q, T') are the same. Define q_i to be the state of either of these automata after reading $a_1 \cdots a_i$. By construction, the state q_i is a word which is T'-equivalent to $q_{i-1}a_i$, and since $a_{i+1} \cdots a_n \in T'$, it follows that

$$q_{i-1}a_i \cdots a_n \in L \qquad \text{iff} \qquad q_i a_{i+1} \cdots a_n \in L.$$

Since q_0 is the empty word, the above and induction imply that

$$a_1 \cdots a_n \in L \qquad \text{iff} \qquad q_n \in L$$

which means that the automaton gives the correct answer to the counterexample, a contradiction. ∎

Problem 149. Show that one can design an algorithm for learning DFA without membership queries and counterexamples, which finds a correct DFA in exponential time. Show that one cannot do better.

Problem 150. Show that there is no algorithm, which asks only membership queries and guesses a correct DFA at the first time it asks an equivalence query. Show that the same holds for a fixed number of mistaken equivalence queries allowed.

Problem 151. Show that there is no algorithm running in polynomial time, which learns a correct DFA in the following setting: both membership and equivalence queries are allowed, but in the case when answer for an equivalence query is ”NO” Teacher delivers no counterexample.

Bibliography

[1] Alfred V Aho and Jeffrey D Ullman. A Characterization of Two-Way Deterministic Classes of Languages. *J. Comput. Syst. Sci.*, 4(6):523–538, 1970.

[2] M H Albert and J Lawrence. A proof of Ehrenfeucht's Conjecture. *Theoretical Computer Science*, 41:121–123, 1985.

[3] Rajeev Alur and Pavol Cerný. Expressiveness of streaming string transducers. *FSTTCS*, 2010.

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

[5] Stefan Arnborg, Derek G Corneil, and Andrzej Proskurowski. Complexity of Finding Embeddings in a k-Tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, April 1987.

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

[7] Roderick Bloem and Joost Engelfriet. A Comparison of Tree Transductions Defined by Monadic Second Order Logic and by Attribute Grammars. *J. Comput. Syst. Sci.*, 61(1):1–50, 2000.

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

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

[10] Mikołaj Bojańczyk. Languages recognised by finite semigroups, and their generalisations to objects such as trees and graphs, with an emphasis on definability in monadic second-order logic. Lecture notes available at `mimuw.edu.pl/~bojan/paper/book-on-languages-recognised-by-finite-semigroups`, 2020.

[11] Mikołaj Bojańczyk. Slightly infinite sets. Lecture notes available at mimuw.edu.pl/~bojan/paper/atom-book. Accessed: July 3, 2025, 2025.

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

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

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

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

[16] Chandra Chekuri and Julia Chuzhoy. Polynomial Bounds for the Grid-Minor Theorem. *J. ACM*, 2016.

[17] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

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

[19] Michal Chytil and Vojtech Jákl. Serial Composition of 2-Way Finite-State Transducers and Simple Programs on Strings. *ICALP*, 52(Chapter 11):135–147, 1977.

[20] Thomas Colcombet and Damian Niwinski. On the positional determinacy of edge-labeled games. *Theor. Comput. Sci.*, 352(1-3):190–196, 2006.

[21] Thomas Colcombet and Daniela Petrisan. Automata and minimization. *SIGLOG News*, 2017.

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

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

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

[25] Samuel Eilenberg. *Automata, Languages, and Machines, Volume A*. Pure and Applied Mathematics, a series of monographs and textbooks: vol. 59-A. Academic Press, 1974.

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

[27] Joost Engelfriet and Hendrik Jan Hoogeboom. MSO definable string transductions and two-way finite-state transducers. *ACM Transactions on Computational Logic*, 2(2):216–254, April 2001.

[28] P ERDdS and A R&wi. On random graphs i. *Publ. math. debrecen*, 6(290-297):18, 1959.

[29] Filiot, Emmanuel and Reynier, Pierre-Alain. Transducers, logic and algebra for functions of finite words. *SIGLOG News*, 2016.

[30] Zoltán Fülöp. On attributed tree transducers. *Acta Cybern.*, 1981.

[31] Seymour Ginsburg. Some remarks on abstract machines. *Transactions of the American Mathematical Society*, 96(3):400–400, March 1960.

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

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

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

[35] Michael Kaminski and Nissim Francez. Finite-Memory Automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994.

[36] Daniel Kirsten. Distance desert automata and the star height problem. *RAIRO - Theoretical Informatics and Applications*, 39(3):455–509, July 2005.

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

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

[39] George H Mealy. A method for synthesizing sequential circuits. *Bell Syst. Tech. J.*, 34(5):1045–1079, 1955.

[40] Edward F Moore. Gedanken-Experiments on Sequential Machines. In *Automata Studies. (AM-34)*. Princeton University Press, Princeton.

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

[42] Jaroslav Nešetřil and Patrice Ossona de Mendez. *Sparsity*, volume 28 of *Algorithms and Combinatorics*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[43] Joël Ouaknine and James Worrell 0001. On linear recurrence sequences and loop termination. *SIGLOG News*, 2015.

[44] Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*, volume 57 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2013.

[45] Mojzesz Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Congrès Des Mathématiciens Des Pays Slaves, Warszawa*, volume 129, pages 92–101, 1929.

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

[47] Neil Robertson and P D Seymour. Graph minors. V. Excluding a planar graph. *Journal of Combinatorial Theory, Series B*, 41(1):92–114, August 1986.

[48] Julia Robinson. Definability and decision problems in arithmetic. *Journal of Symbolic Logic*, 14(2):98–114, 1949.

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

[50] 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.

[51] M P Schützenberger. On the definition of a family of automata. *Information and Control*, 4(2-3):245–270, September 1961.

[52] M P Schützenberger. Sur une variante des fonctions sequentielles. *Theoretical Computer Science*, 4(1):47–57, February 1977.

[53] 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.

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

[55] Imre Simon. Factorization forests of finite height. *Theoretical Computer Science*, 72(1):65–94, 1990.

[56] Alfred Tarski. A Decision Method for Elementary Algebra and Geometry, 1951.

[57] J W Thatcher and J B Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1):57–81, March 1968.

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

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

[60] Alan Mathison Turing et al. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[61] J D Ullman and J E Hopcroft. An Approach to a Unified Theory of Automata. *Bell System Technical Journal*, 46(8):1793–1829, October 1967.

[62] Leslie G Valiant. General Context-Free Recognition in Less than Cubic Time. *J. Comput. Syst. Sci.*, 10(2):308–315, 1975.

[63] Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. New bounds for matrix multiplication: from alpha to omega. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3792–3835. SIAM, 2024.

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