Advanced topics in automata theory

Mikołaj Bojańczyk and Wojciech Czerwiński

Preface

These are lecture notes for a course on advanced automata theory, that we gave at the University of Warsaw in the years 2015-2018. The lectures were written down by the first author and the exercises by the second author, but we consulted each other extensively in the process of both teaching and writing.

Mikołaj Bojańczyk and Wojciech Czerwiński

Contents

1
Determinisation of ω-automata

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

$$a_1 a_2 a_3 \cdots$$

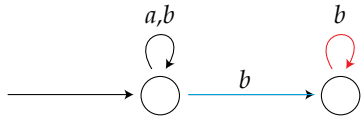
We write Σ^ω for the set of ω words over alphabet Σ. The topic of this chapter is McNaughton's Theorem, which shows that automata over ω-words can be determinised. A more in depth account of automata (and logic) for ω words can be found in [18].

1.1 Automata models for ω-words

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

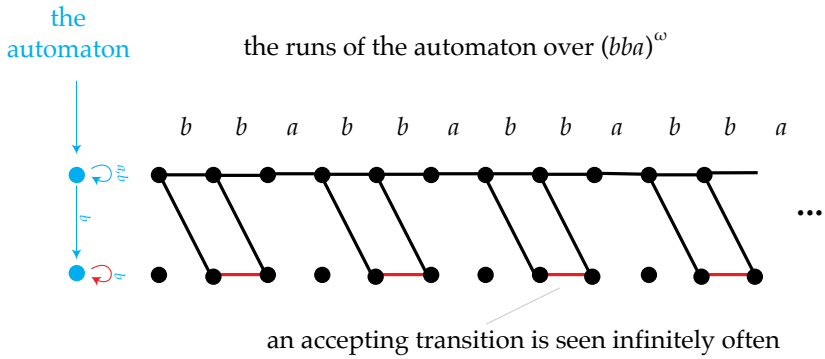
Example 1. Consider the set of words over alphabet $\{a, b\}$ where the letter a appears finitely often. This language is recognised by a nondeterministic Büchi

automaton like this (we adopt the convention that accepting transitions are red edges):



□

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



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

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

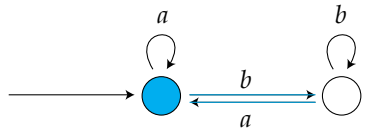
Proof. Take the automaton from Example 1. Suppose that there is a deterministic Büchi automaton that is equivalent, i.e. recognised the same language. Let us view the set of all possible inputs as an infinite tree, where the

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 determines automata for ω-words, we need something more powerful than the Büchi condition. One solution is called the Muller condition, and is described below. Later, we will see another, equivalent, solution, which is called the parity condition.

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

Example 2. Consider this automaton



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. If we set $\mathcal{F}$ to be all subsets which contain only transitions that enter the blue state, then the automaton will accept words which contain infinitely many a's and finitely many b's. □

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

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

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

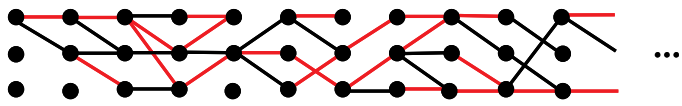
- nondeterministic Büchi automata
- nondeterministic Muller automata
- deterministic Muller automata

have the same expressive power, but deterministic Büchi automata are weaker. The theorem was first proved in [14]. The proof here is similar to one by Muller and Schupp [15]. An alternative proof method is the Safra Construction, see e.g. [18].
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. Then we show how the universal languages can be recognised by deterministic Muller automata.

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

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

Furthermore, every edge is either red or black, with red meaning "accepting". We assume that there are no parallel edges. Here is a picture of a width 3 dag:



In the pictures, we adopt the convention that the i-th column stands for the set of vertices $\{1,\dots,n\} \times \{i\}$. The top left corner of the picture, namely the vertex $(1,1)$, will be called the *initial vertex*.
As we will show, the essence of McNaughton's theorem is showing that for every n, there is a deterministic Muller automaton which inputs a width n dag and says if it contains a path that begins in the initial vertex and visits infinitely many red (accepting) edges. In order to write such an automaton, we need to encode as width n dag as an ω-word over some finite alphabet. This is done using an alphabet, which we denote by $[n]$, where the letters look like this:

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

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

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

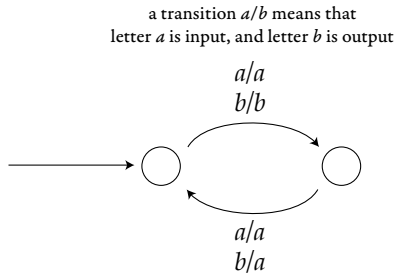
Proposition 1.3. *For every $n \in \mathbb{N}$ there is a deterministic Muller automaton recognizing the universal n state Büchi language.*

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

bimachine transducer or register transducer. If the input alphabet is Σ and the output alphabet is Γ, then a sequential transducer defines a function

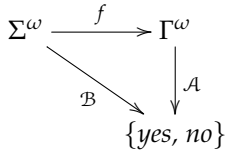
$$f : \Sigma^\omega \to \Gamma^\omega.$$

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



□

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



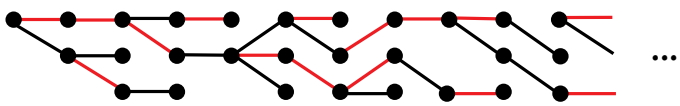
Proof. The states of automaton $\mathcal{B}$ are the product of the states for f and $\mathcal{A}$. (The assumption that the transducer is letter-to-letter is not necessary, but the construction becomes a bit more complicated.) ■

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

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

such that a word $w \in A^\omega$ is accepted by $\mathcal{A}$ if and only if $f(w)$ contains a path fro the initial vertex with infinitely many accepting edges. By composing the transducer with the automaton from the proposition, we get a deterministic Muller automaton equivalent to $\mathcal{A}$. It now remains to show the proposition, i.e. that the n state universal Büchi language can be recognised by a Muller automaton. The proof has two steps.

The first step is stated in Lemma 1.5 and says that a when dealing with the universal Büchi language, one can write a deterministic transducer which replaces an arbitrary dag by an equivalent tree. Here we use the name tree for a width n dag where, every non-isolated node other than (1,1) has exactly one incoming edge. Here is a picture of such a tree, with the isolated nodes not drawn:



Lemma 1.5. *There exists a sequential transducer*

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

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

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

Lemma 1.6. *There exists a deterministic Muller automaton recognising*

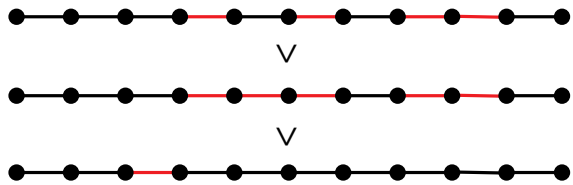
$$\{w \in [n]^\omega : w \text{ is a tree and contains a path with infinitely many accepting edges}\}.$$

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

1.2 *Reduction to trees*

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

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



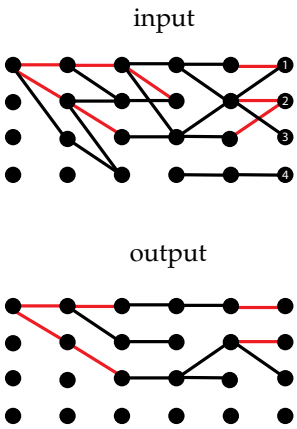
A finite path π in a width n dag is called *profile optimal* if it has lexicographically least profile among paths in w that have the same source and target.

Lemma 1.7. *There is a sequential transducer*

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

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

Proof. We use the following congruence property of optimal paths: if π and σ are optimal paths such that the target of π is equal to the source of σ, then also their composition $\pi\sigma$ is optimal. A corollary is that if we choose for every vertex in the input graph w an outgoing edge that participates in some optimal path, then the putting all of these edges together will yield a tree as in the statement of the claim. To produce such edges, after reading the first n letters, the automaton keeps in its memory the lexicographic ordering on the profiles of optimal paths lead from the root to nodes at depth n. Here is a picture of the construction:



The state of the tranducer is this information:

The reachable vertices are

❶ ❷ ❸

and the least profiles for reaching them are ordered as

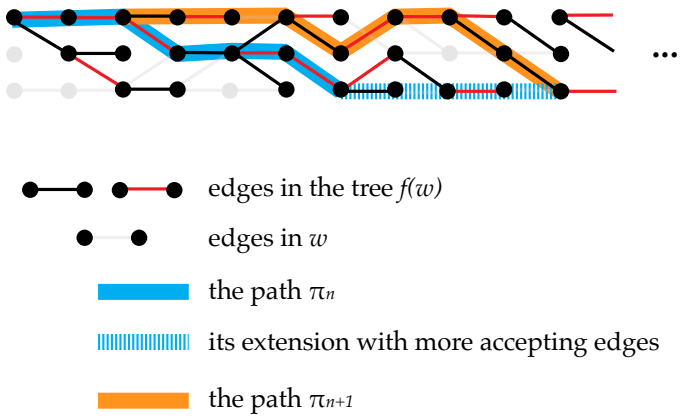
❶ = ❷ < ❸

■

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

Proof. Assume that the input w to f contains a path with infinitely many accepting edges. Define a sequence $\pi_0, \pi_1, \ldots$ of finite paths in $f(w)$ as follows by induction. In the definition, we preserve the invariant that each path in the sequence $\pi_0, \pi_1, \ldots$ can be extended to an an accepting path in the graph w. We begin with π_0 being the edgeless path that begins and ends in the root of the

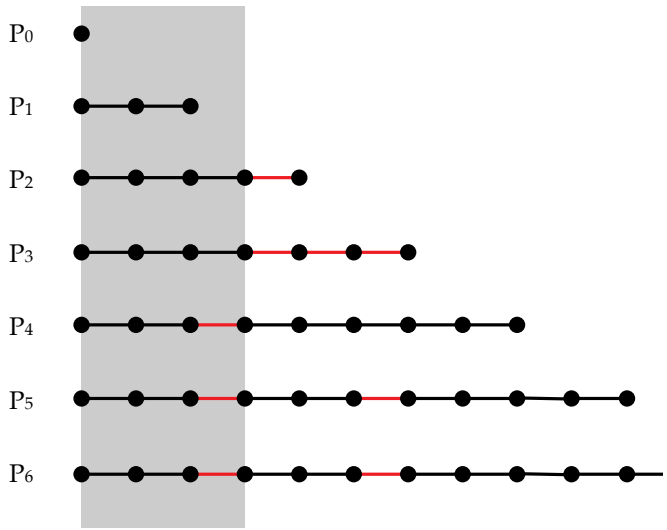
tree $f(w)$. This path π_0 satisfies the invariant, by the assumption that the input w contains a path with infinitely many accepting edges. Suppose that π_n has been defined. By the invariant, we can extend π_n to an infinite path in the graph w, and therefore we can extend π_n to a finite path in w that contains at least one more accepting edge. Define π_{n+1} to be the unique path in the tree $f(w)$ which has the same source and target as the new path that extends π_n with at at leat one accepting edge.



Define P_n to be the profile of the path π_n. We claim that the sequence of profile $P_0, P_1, P_2, \ldots$ has a well defined limit

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

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



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

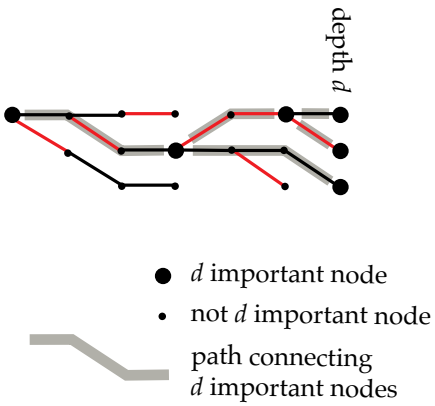
Proof. Toward a contradiction, suppose that P has the letter "accepting" finitely often, i.e. there is some i such that after i, only the letter "non-accepting" appears in P. Choose n so that $\pi_n, \pi_{n+1}, \ldots$ have profile consistent with P on the first i letters. By construction, the profile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

Consider the set of finite paths in the tree $f(w)$ which have profile that is a prefix of P. This set of paths forms a tree (because it is prefix-closed). This tree has bounded degree (assuming the parent of a path is obtained by removing the last edge) and it contain paths of arbitrary finite length (suitable prefixes of the paths $\pi_1, \pi_2, \ldots$). The Köning lemma says that every finitely branching tree

with arbitrarily long paths contains an infinite path. Applying the König lemma to the paths in $f(w)$ with profile P, we get an infinite path with profile P. By Claim 1.9 this path has infinitely many accepting edges. ■

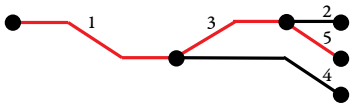
1.3 *Finding an accepting path in a tree graph*

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

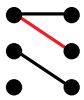


Definition of the Muller automaton. We now describe the Muller automaton for Lemma 1.6. After reading the first d letters of an input tree (i.e. after reading the input tree up to depth d), the automaton keeps in its state a tree, where the nodes correspond to nodes of the input tree that are important for depth d, and the edges corresponds to paths in the input tree that connect these nodes. This tree stored by the automaton is a tree with at most n leaves, and

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

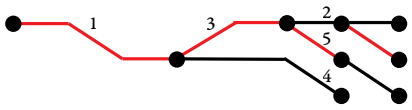


The big black dots correspond to important nodes for the current depth, red edges are accepting, black edges are non-accepting, while the numbers are the identifiers. All identifiers are distinct, i.e. different edges get different identifiers. It might be the case (which is not true for the picture 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:

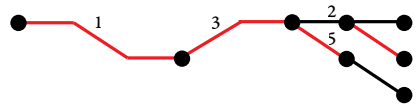


To define the new state, perform the following steps.

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



3. Eliminate unary nodes, thus joining several edges into a single edge. This means that a path which only passes through nodes of degree one gets collapsed to a single edge, the identifier for 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.

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

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

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

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

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

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

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

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

- some transition in X refreshes identifier i; and
- none of the transitions in X delete identifier i.

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

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

Problem 1. Are the following languages regular:

1. prefix of v belongs infinitely often to the fixed regular language of finite words $L \subseteq \Sigma^*$;

2. word v contains infinitely many infixes of the form $ab^p a$, where p is prime;

3. word v contains infinitely many infixes of the form $ab^p a$, where p is even;

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

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

Problem 2. Show that language of words "there exists a letter b" cannot be accepted by a nondeterministic automaton with Büchi acceptance condition, where all the states are accepting (but possibly transitions over some letters missing in some states).

Problem 3. Show that language "finitely many occurrences of letter a" cannot be accepted by a deterministic automaton with Büchi acceptance condition.

Problem 4. Show that every language accepted by some nondeterministic automaton with Muller acceptance condition is also accepted by some nondeterministic automaton with Büchi acceptance condition.

Problem 5. Assume that we have changed the acceptance condition into such which investigates which sets of transitions are visited infinitely often. Does it affect the expressivity of automata? How it is for Büchi acceptance condition? And how for Muller acceptance condition?

Problem 6. Show that nonemptiness is decidable for automata with Muller acceptance condition.

2
Infinite duration games

In this chapter, we prove the Büchi-Landweber Theorem [6, Theorem 1], see also [18, Theorem 6.5], about 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 solving "Church's Problem"[7].

2.1 Games

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

Definition 2.1 (Game)**.** *Suppose that $W \subseteq \Sigma^\omega$ is a set of ω-words. Define a game with winning condition W to be:*

- *a directed graph, not necessarily finite, whose vertices are called positions of the game;*
- *a distinguished initial position;*

- *a partition of the positions into positions controlled by player 0 and positions controlled by player 1;*
- *a labelling function that maps each position to a label from* Σ.

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. By applying the labelling function to the path, we get a word in Σ^ω; if this word belongs to W then player 0 wins, otherwise player 1 wins.
To formalise the notions in the above paragraph, one uses the concept of a strategy. A *strategy* for player $i \in \{0,1\}$ is a function which inputs a history of the play so far (a path from the initial position to some position controlled by player i), and outputs the new position (consistent with the edge relation in the graph). Given strategies for both players, call these σ_0 and σ_1, a unique play is determined, which is either a finite path ending in a dead end, or an infinite path. This play is called winning for player 0 if it is finite and ends in a dead end controlled by the opposing player 1; or if it is infinite and satisfies the winning condition after applying the labelling function. Otherwise, the play is winning for player 1. A winning strategy for player i is defined to be a strategy σ_i such that for every possible strategy σ_{1-i} of the opponent, the resulting play is winning for player i.

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 come up with strange games (e.g. using the axiom of choice) which are not determined because none of the players has a winning strategy. The goal of this lecture is to show a theorem by Büchi and Landweber: if the winning condition of the game is recognised by an automaton, then the game is

determined, and furthermore the winning player has a finite memory winning strategy, in the following sense.

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

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

The two ingredients above define a strategy for player i in the following way: the next move chosen by player i in a position v is obtained by applying the function f_v to the state of the automaton after reading the history of the play, including v. We will apply this definition also to games with infinitely many positions, but we will only care about finite memory sets M (which leads us to consider finite state automata over infinite alphabets).

An important special case is when the set M has only one element, in which case the strategy is called *memoryless*. In this case, the new position chosen by the player only depends on the current position, and not on the history of the game before that.

Theorem 2.3 (Büchi-Landweber Theorem)**.** *For every ω-regular language W there exists a finite set M such that for every game with winning condition W, one of the players has a winning strategy that uses memory M.*

The proof of the above theorem has two parts. The first part is to identify a special case of games with ω-regular winning conditions, called parity conditions.

Definition 2.4 (Parity condition)**.** *Define the n-rank parity condition to be the set of ω-words over the alphabet $\{1, \ldots, n\}$ where the smallest number appearing infinitely often is even.*

A parity game is a game where the winning condition is the n-rank parity language for some n. Parity games are important because not only can they be won using finite memory strategies, but even memoryless strategies are enough.

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

The above theorem is proved in Section 2.2. The second step of the Büchi-Landweber theorem is the reduction to parity games. This essentially boils down to transforming deterministic Muller automata into called deterministic parity automata. In a parity automaton, there is a ranking function from states to numbers, and a run is considered accepting if the minimal rank appearing infinitely often is even. This is a special case of the Muller condition, but it turns out to be expressively complete in the following sense:

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

Proof. The lemma can be proved in two ways. One of them is to show that, by taking more care in the determinisation construction in McNaughton's Theorem, we can actually produce a parity automaton. Here we use an alternative construction, based on a data structure called the later appearance record [12]. The construction is presented in the following claim.

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

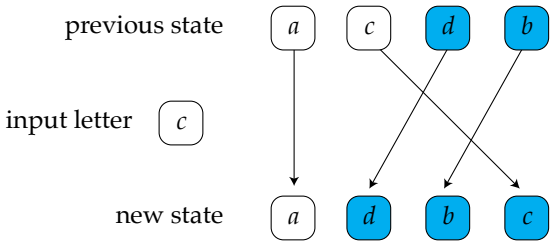
$$g : Q \to \mathcal{P}(\Sigma)$$

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

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

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

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



Consider a run of this automaton over some infinite input $w \in \Sigma^\omega$. Take some blue suffix of maximal size that appears infinitely often in the run. Then the letters in this suffix are exactly those that appear in w infinitely often. Therefore, to get the statement of the claim, we order Q first by the number of blue positions, and in case of the same number of blue positions, we use some arbitrary total ordering. The function g returns the set of blue positions. This completes the proof of the claim. ∎

The conversion of Muller to parity is a straightforward corollary of the above lemma: one applies the above lemma to the state space of the Muller automaton, and defines the ranks according to the Muller condition. This completes the proof of the lemma on conversion of Muller automata into parity automata. ∎

Let us now finish the proof of the Büchi-Landweber theorem. Consider a game with an ω-regular winning condition $L \subseteq \Sigma^\omega$. By Lemma 2.6, there is a

deterministic parity automaton which recognises the language L. 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). This is a parity game, with the ranks inherited from the automaton. In a position (v, q), the player controlling position v chooses an edge in the original game, and the state is updated deterministically according to the transition function of the automaton. 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;

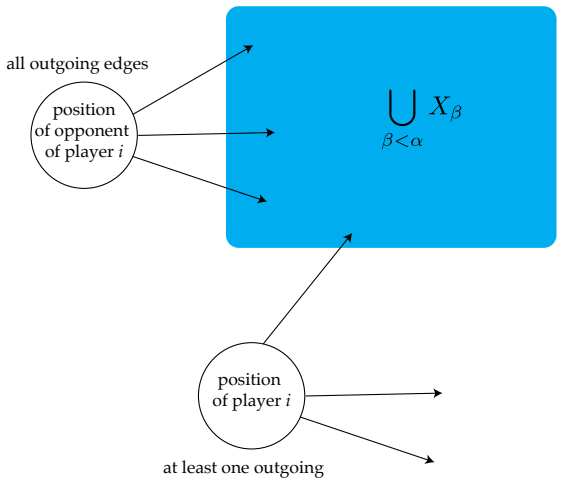
2. player i wins from position (v, q) in the product game, where q is the initial state of the deterministic parity automaton recognising L.

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, for every position v, condition 2 must hold for either player 0 or 1; furthermore, a positional strategy in the product game corresponds to a finite memory strategy in the original game, where the memory is the states of the deterministic parity automaton.
This completes the proof of the Büchi-Landweber Theorem. It remains to show memoryless determinacy of parity games, which is done below.

2.2 *Memoryless determinacy of parity games*

In this section, we prove Theorem 2.5 on memoryless determinacy of parity games. Recall that in a parity game, the positions are assigned ranks from a set $\{0, \ldots, n\}$, and the goal of player 0 is to ensure that for infinite plays, the minimal number appearing infinitely often is even. 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 from $\{0, \ldots, n\}$ that are used in the game.

Attractors. Consider a set of positions X in a parity game (actually the winning condition is irrelevant for the definition). For a player $i \in \{0,1\}$, we define below the i-attractor of X, which intuitively represents positions where player i can force a visit to the set X. The attractor is approximated using ordinal numbers. Define X_0 to be X, and for an ordinal number $\alpha > 0$, define X_α to be $X_{<\alpha}$, i.e. the union of X_β ranging over ordinals $\beta < \alpha$, plus the following two sets of positions: positions owned by player i where some outgoing edge leads to $X_{<\alpha}$; and positions owned by the opponent of i where all outgoing edges lead to $X_{<\alpha}$. Here is a picture:



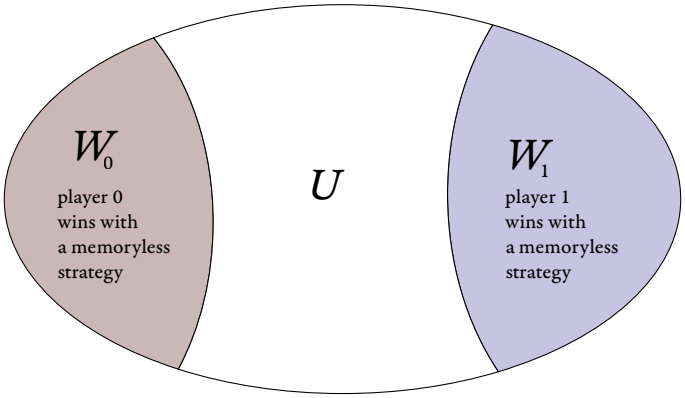
The set X_α grows as α grows, and therefore at some point it stabilises. This stable set is called the i-attractor of X. Over positions in the i-attractor, player i has a memoryless strategy which guarantees that after a finite number of steps, the game will end up in X or in a dead end owned by the opponent of player i. This strategy, called the attractor strategy, is to choose the neighbour that belongs to X_α with the smallest possible index α.

Induction base. Recall that we prove memoryless determinacy by induction on the number of ranks that are used in the game. The induction base is when only one rank is used. Let i be the parity of the only rank, without loss of generality we assume $i \in \{0, 1\}$. This means that every infinite play is won by player i. This does not necessarily mean that player i wins the game, because the game might end up in a dead end owned by player i. Let X be the $(1-i)$-attractor of the empty set (i.e. the attractor from the point of view of the opponent of player i). We claim that on positions from X, the opponent of player i has a memoryless winning strategy, and on positions outside X, player i has a memoryless winning strategy. For positions in X, the opponent of player i plays the attractor strategy, which guarantees reaching a dead end position owned by player i in a finite number of steps. For positions outside X, we make the following observation, which follows immediately from the definition of X:

- if player i owns a position outside X, then some outgoing edge leads to a position outside X; and
- if the opponent of player i owns a position outside X, then all outgoing edges lead to a position outside X.

It follows that if the play begins outside X, then player i has a memoryless strategy that guarantees avoiding X forever, in particular this strategy is winning.

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

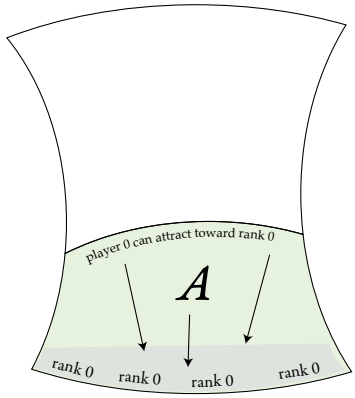


By definition, for every position in $w \in W_i$, player i has a memoryless winning strategy that wins when starting in position w. In principle, the memoryless strategy might depend on the choice of w, but the following lemma shows that this is not the case.

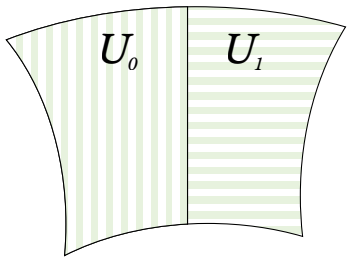
Lemma 2.8. *Let $i \in \{0,1\}$ be one of the players. There is a memoryless strategy σ_i for player i, such that if the game starts in W_i, then player i wins by playing σ_i.*

Proof. By definition, for every position $w \in W_i$ there is a memoryless winning strategy, which we call the strategy of w. We want to consolidate these strategies into a single one that does not depend on w. Choose some well-ordering of the vertices from W_i, i.e. a total ordering which is well-founded. 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). The consolidated strategy is defined as follows: when in position w, play according to the strategy of the companion of w. The key observation is that for every play using this consolidated strategy, the sequence of companions is non-decreasing in the well-ordering, and therefore it must stabilise at some companion v; and therefore the play must be winning for player i, since from some point on it is consistent with the strategy of v. ∎

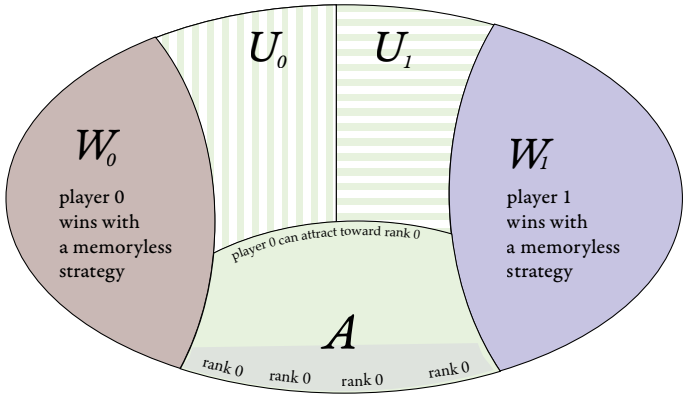
Consider the minimal rank that appears in the entire game, let it be n. By symmetry, we assume that this minimal rank is 0. (The symmetric case is when the minimal rank is 1.) Define A to be the 0-attractor, inside the game limited to U, of positions that are in U and have minimal rank 0. Here is the picture of the game restricted to U:



In the original game, if the play begins in a position from A and player 0 plays the attractor strategy on the set A, then the play is bound to either end up in a position in U that has rank 0, or in the set W_0. Let us consider the game restricted to set $U - A$. Since this game does not use rank 0, the induction assumption can be applied to get a partition of $U - A$ into two sets of positions U_0 and U_1, such that on each U_i player U_i has a memoryless winning strategy, assuming that the game is limited to $U - A$:



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



Lemma 2.9. *U_1 is empty.*

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

- in U_1, use the winning memoryless strategy inherited from the game restricted to $U - A$;

- in W_1, use the winning memoryless strategy from Lemma 2.8;
- in other positions do whatever.

We claim that the above memoryless strategy is winning for all positions from U_1, and therefore U_1 must be empty by assumption on W_1 being all positions where player 1 can win in a memoryless way. Suppose player 1 plays the above strategy, and the play begins in U_1. If the play never leaves $U - A$, then player 1 wins by assumption on the strategy. Suppose that the play does leave $U - A$. If it enters W_0 or A, this would have to be a choice of player 0, but positions with such a choice already belong to W_0 or A. Therefore, if the play leaves $U - A$, then it enters W_1, where player 1 wins as well. ■

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

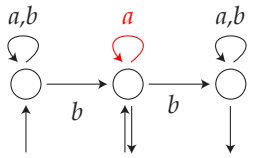
- in U_0, use the winning memoryless strategy inherited from the game restricted to $U - A$;
- in W_0, use the winning memoryless strategy from Lemma 2.8;
- in A use the attractor strategy to reach rank 0 inside U;
- on other positions, i.e. on W_1, do whatever.

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

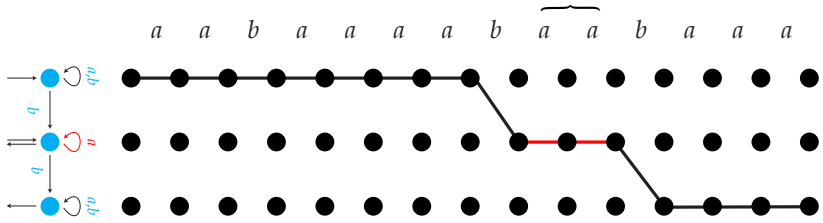
3
Distance automata

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 4. 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 [13].

Theorem 3.1 (Limitedness of distance automata)**.** *The following problem is decidable:*

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

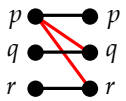
The problem in the above theorem was called limitedness in [13]. Our algorithm uses the Büchi-Landweber theorem from Chapter 2.

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

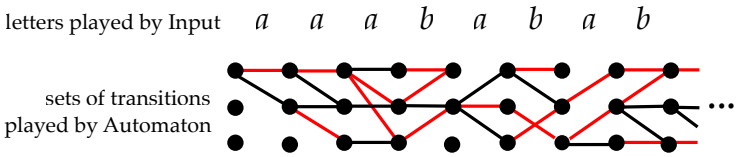
- player Input chooses a letter of the input alphabet;
- player Automaton responds with a set of transitions over this letter.

A move of player Automaton in a given round, which is a set of transitions, can be visualised as a bipartite graph, which says how the letter can take a state to a

new state, with costly transitions being red and non-costly transitions being black, like below:



For the definition of the game, it is important that player Automaton does not need to choose all possible transitions over the letter played by player Input, only a subset. After all rounds have been played, the situation looks like this:



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

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

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

If either of the condition above is not met, then player Input loses. The following lemma implies the decidability of the limitedness problem.

Lemma 3.2. *For a distance automaton, the following conditions are equivalent, and furthermore one can decide if they hold:*

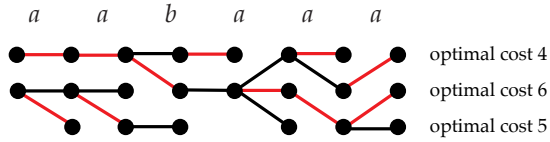
1. *the automaton is limited;*

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

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

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

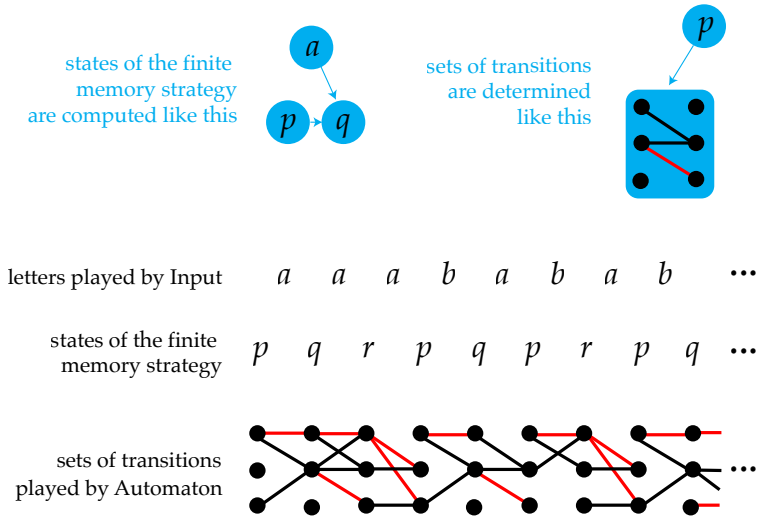
Implication from 1 to 2. We want to prove that if the automaton is limited, then player Automaton has a winning strategy for some finite m, which will turn out to be the same m as in the definition of limitedness. Here is the strategy of player automaton. Define a run ρ of the distance automaton over an input word w to be optimal if it has minimal cost among runs that have the same input word, same source state and same target state. The strategy of player Automaton is as follows: if the input word is $a_1 \cdots a_i$, then in the i-th round, player Automaton responds with sets of transitions Suppose that player Input has played a sequence of letters. Then the sets of transitions chosen by Automaton are so that the transitions form a forest, consisting only of optimal runs, where all reachable configurations (i.e. reachable by some run from an initial state) are covered, as in the following picture:



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

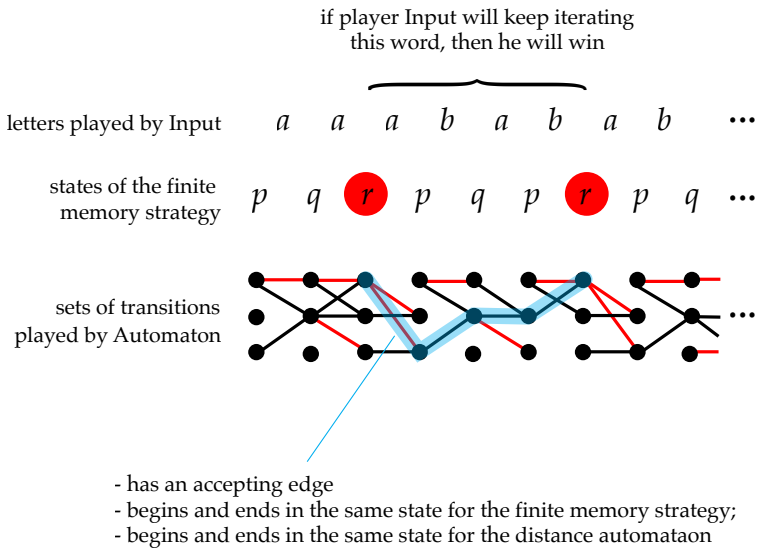
Implication from 3 to 2. Suppose that player Automaton wins the limitedness game with bound ω. We will prove that player Automaton can also win the limitedness game with a finite bound.

By the Büchi-Landweber theorem, if player Automaton can win the game with bound ω, then he can also win the game with a finite memory strategy. We will show that this finite memory strategy is actually winning for a finite bound. By unfolding the definition of a finite memory strategy in the limitedness game, there is a deterministic automaton, call it the strategy automaton, over the input alphabet of the original distance automaton, and a function f from the states of the strategy automaton to sets of transitions in the origial distance automaton, such that if in the first i rounds the letters produced by player Input were $a_1 \cdots a_i$, then the response of player Automaton in the i-th round is obtained by applying f to the state of the strategy automaton after reading $a_1 \cdots a_i$. Here is a picture of how the strategy works:



We claim that this same winning strategy produces runs where the cost is at most (number of states in the distance automaton) times (number of states in the automaton for the strategy), thus proving the implication from 3 to 2 in the lemma. To prove the claim, suppose that the strategy produces a run exceeding the above bound. Using a pumping argument we can find a loop that can be

exploited by player Input to force player Automaton into a path that has infinitely many costly edges, as in the following picture:



■

4

Tree automata and MSO

In this section we discuss the connection between automata and monadic second-order logic (MSO). The presentation here is largely based on [18]. The connection was originally discovered simultaneously by three authors [4, 10, 19], in their quest to answer a question by Tarski: "is the MSO theory of the natural numbers with successor decidable"? The combination of logic and automata is an important theme of logic in computer science, Vardi calls this combination "a match made in heaven" [20]. A crowning achievement is Rabin's Theorem [16], 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 main technical tools to prove Rabin's Theorem[1], namely McNaughton's Theorem on determinisation of ω-automata from Chapter 1, and memoryless determinacy of parity games from Chapter 2. It remains only to state the appropriate definitions and put the tools to work.

[1] Büchi says this in [5, 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."

4.1 Monadic second-order logic

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

Example 5. Suppose that we view an undirected graph as relational structure (i.e. a model as in logic), where the universe is the vertices and there is one binary relation $E(x, y)$ for the edges; this relation is symmetric. The following formula

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

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

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

The above formula illustrates all syntactic constructs in MSO: one can quantify over elements, over sets of elements, one can test membership of elements in sets, and one can use the relations available in the input model (in the case of graphs, only one binary relation).
Here is another example for graphs. The following MSO formula says that the input graph is three colourable:

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

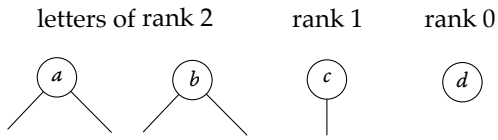
□

We say that a property of relational structures over some vocabulary (e.g. graphs as in the above example) is MSO definable if there is a formula of

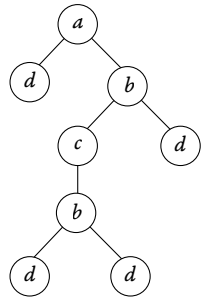
MSO which is true exactly in those structures which have the property. In this chapter, we use MSO to describe properties of trees (finite and infinite), in the next chapter, we talk about graphs.

4.2 *Finite trees*

We begin by considering finite trees. We show that, over finite trees, MSO has the same expressive power as finite state automata. 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 finite tree over a ranked alphabet Σ is defined to be a finite (rooted) tree where every node has a label from Σ. Furthermore, for each node the number of children is the same as the rank of the label, and we assume that these children are ordered, i.e. one can speak of the first child, second child, etc. Here is a picture of a finite tree over the alphabet from the example above:



We will consider infinite trees in Section 4.3.
A finite tree t over an alphabet Σ is viewed as a relational structure in the following way. The universe is the nodes. For every $i \in \{0, 1, \ldots\}$ there is a

binary predicate $child_i(x, y)$ which says that y is the i-th child of x, and for every $a \in \Sigma$ there is a predicate $a(x)$ which says that x has label a. Using MSO, one can define a descendant predicate: a node y is a descendant of x if and only if y belongs to every set X that contains x and is closed under children. We say that an MSO formula is true in a tree if it is true in the relational structure described above. This only makes sense for formulas that have no free variables (sentences), and which use the vocabulary (relation names) described above. We say that a set of finite trees L over a ranked alphabet Σ is MSO definable if there is an MSO formula φ such that

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

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

Example 6. Suppose that the ranked alphabet is

The set of trees with an even number of nodes is MSO definable, namely the formula is "false". This is because all trees over the above ranked alphabet have an odd number of nodes. More effort is required for "even number of leaves". Here the formula says that there exists a set X of nodes, which contains all leaves but not the root, and such that for every non-leaf node, it belongs to X if and only if it has an even number of children in X. □

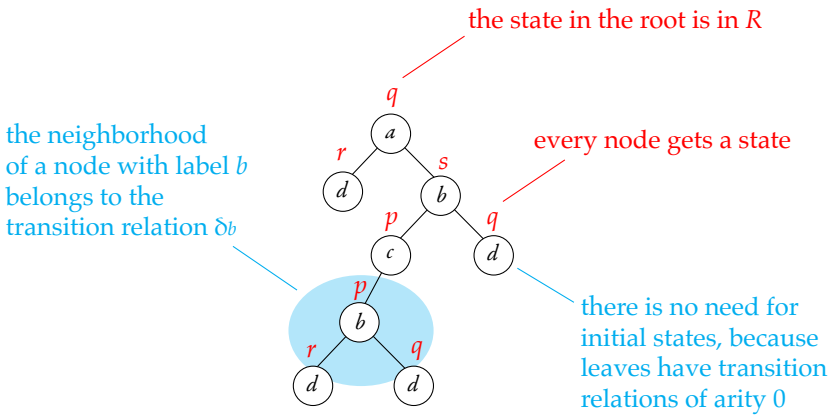
Tree automata. We introduce automata for finite trees, and show that they have the same expressive power as the logic MSO. The proof is not hard, but it explains an important phenomenon: (a) MSO can simulate automata because it

can formalise the existence of runs in a nondeterministic auotomaton; (b) automata can simulate MSO, because they have the appropriate closure properties.

Definition 4.1. *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* $\delta_a \subseteq Q^n \times Q$*.*

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

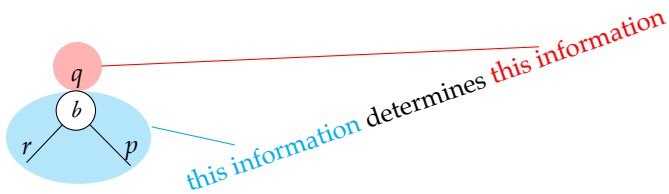


Lemma 4.2. *Languages recognised by nondeterministic tree automata are closed under union, intersection and complementation.*

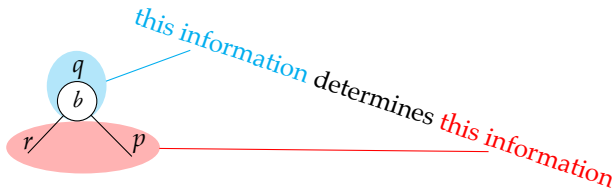
Proof. For union, we can simply take the disjoint union of two nondeterministic tree automata. Intersection can be done using a cartesian product, or by using union and complementation. For complementation, we use determinisation: the same proof as for automata on words – the subset construction – shows that

for every nondeterministic tree automata there is an equivalent one that is (bottom-up) deterministic. Since (bottom-up) deterministic automata can be complemented by complementing the root states, we get the lemma. ∎

Example 7. In the above lemma, we used bottom-up deterministic automata, and argued that they are equivalent to nondeterministic ones. This type of determinism can be illustrated as follows



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



We also require that there is only one root state. The only way that to-down deterministic automaton can reject a tree is by encountering an undefined result in the partial transition function. Top-down deterministic automata are strictly less expressive than nondeterministic ones (and therefore also bottom-up deterministic ones), see Exercise **??**. □

Theorem 4.3. *Let Σ be a finite ranked alphabet. The following conditions are equivalent for every set L of finite trees over Σ:*

1. *L is definable in* MSO*;*

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

Proof.

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

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

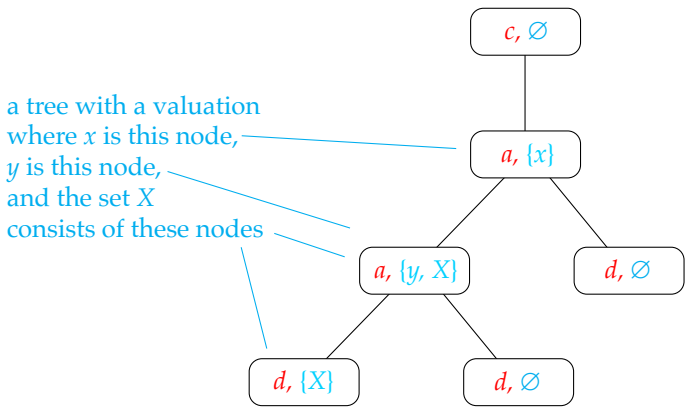
Formally speaking, $\mathrm{root}(x)$ is a shortcut for a formula which says that x is not a child of any node, and $\mathrm{child}_i(x) \in X_{q_i}$ is a shortcut for a formula which says that there exists a node that is the i-th child of x (because we have children as relations and not functions) and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

1 ⇒ 2 By induction on formula size, we show that every MSO formula can be converted into an automaton. The main issue is that we go to subformulas, free variables appear, and we need to say how an automaton deals with free variables. Consider a formula φ of MSO whose

set of free variables is $\mathcal{X}$ (some of these variables are first-order, some are second-order). We define the *language* of such a formula to be the set of trees over an extended alphabet $\Sigma \times \mathcal{P}(\mathcal{X})$ defined as follows. (In the extended alphabet, the ranks are inherited from Σ.) A tree t over this alphabet belongs to the language if

1. For every first-order variable $x \in \mathcal{X}$, there is exactly one node in the tree whose label contains x on the second coordinate;
2. The formula φ is satisfied in the tree under the valuation which maps a first-order variable x to the unique node with x in its label, and which maps a second-order variable X to the set of nodes which have X in their label.

Here is a picture:



By induction on the size of an MSO formula, we show that its language, as defined above, is recognised by a tree automaton. The proof uses the closure properties from Lemma 4.2.

■

4.3 *Infinite trees*

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

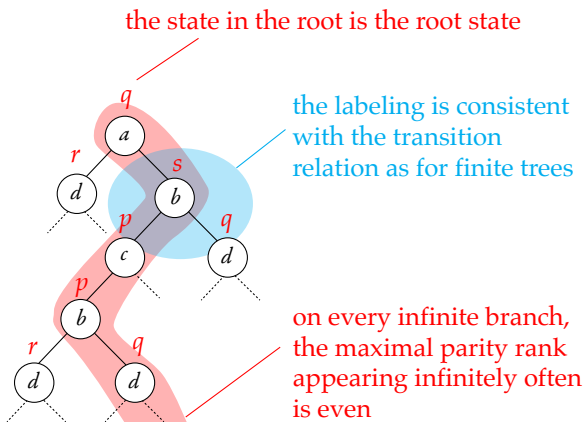
$$t : \underbrace{\{0,1\}^*}_{\text{nodes of the complete binary tree}} \to \Sigma.$$

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

Definition 4.4. *The syntax of a nondeterministic parity tree automaton consists of*

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

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



We now state Rabin's Theorem. Rabin's original proof did not use the parity acceptance condition, but something that is now called the *Rabin condition*, see [18].

Theorem 4.5 (Rabin's Theorem)**.** *Let Σ be a finite alphabet. The following conditions are equivalent for every set L of infinite trees over Σ:*

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

The proof has the same structure as in the case of finite trees. The only difference is that for infinite trees, closure under complementation is far from obvious. The difficulty is that we use only nondeterministic automata; in fact no deterministic model for infinite trees is known that would be equivalent to MSO. Therefore, the rest of this chapter is devoted to proving closure under complementation for languages recognised by nondeterministic parity tree automata.
A corollary of the above statement of Rabin's theorem is that the structure

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

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

Alternating parity tree automata. To show complementation of nondeterministic tree automata, we will pass through a more powerful model. The syntax of an *alternating parity tree automaton* is defined the same as in Definition 4.4 for nondeterministic automata, with the following differences: (1) the set of states is partitioned into two subsets Q_0 and Q_1; and (2) for each letter a, the transition relation has form

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

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

Theorem 4.6.

1. *For every nondeterministic parity tree automaton, one can compute an alternating one that recognises the same language.*

2. *Languages recognised by alternating parity tree automata are closed under complement.*

3. *For every alternating parity tree automaton, one can compute a nondeterministic one that recognises the same language.*

Before proving the above result, we show how it completes the proof of Theorem 4.5. Recall that the only missing ingredient in Theorem 4.5 was complementation of nondeterministic parity tree automata. This is achieved by using Theorem 4.6 as follows: make the automaton alternating, complement it, make it nondeterministic again.

Proof of Theorem 4.6. For item 1, let $\mathcal{A}$ be a nondeterministic parity tree automaton with states Q and transitions Δ. The simulating alternating automaton has states $Q + \Delta$. In a state from Q, player 0 chooses a transition applicable to the current configuration (and the head of the simulating automaton stays in the same node of the input tree). In a state from Δ, player 1

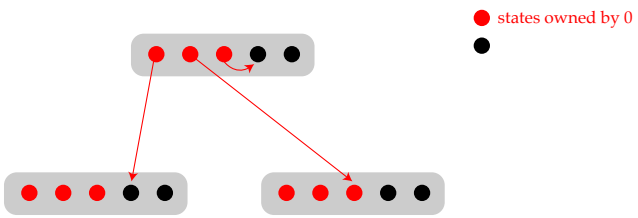
chooses either the left or right child, and updates the state accordingly. The parity condition for Q is inherited from the original nondeterministic automaton, and all states from Δ are assigned the least important rank.
For item 2, let $\mathcal{A}$ be an alternating parity tree automaton. Define $\overline{\mathcal{A}}$ to be the alternating parity tree automaton by swapping the roles of players 0 and 1, and incrementing the ranking function so that even ranks become odd and vice versa. To prove that $\overline{\mathcal{A}}$ is the complement of $\mathcal{A}$, we show below that the following conditions are equivalent for every input tree t:

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

The equivalences $1 \Leftrightarrow 2$ and $4 \Leftrightarrow 5$ are by definition of the language recognised by an alternating automaton. The equivalence $2 \Leftrightarrow 3$ is by construction of $\overline{\mathcal{A}}$. The equivalence $3 \Leftrightarrow 4$ is because $G_{\overline{\mathcal{A}}}(t)$ is a parity game, and it is therefore determined, i.e. one of the players has a winning strategy.
It remains to show the last item of the theorem, namely that alternating parity tree automata can be made nondeterministic. Suppose that $\mathcal{A}$ is an alternating parity tree automaton, with states Q and input alphabet Σ. By memoryless determinacy of parity games, it follows that a tree t is accepted if and only if player 0 has a memoryless winning strategy σ_0 in the game $G_{\mathcal{A}}(t)$. We will find a nondeterministic parity automaton on trees which checks this. A memoryless strategy σ_i for player 0 can be represented as a tree $[\sigma_i]$ over alphabet

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

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



We will show that the language

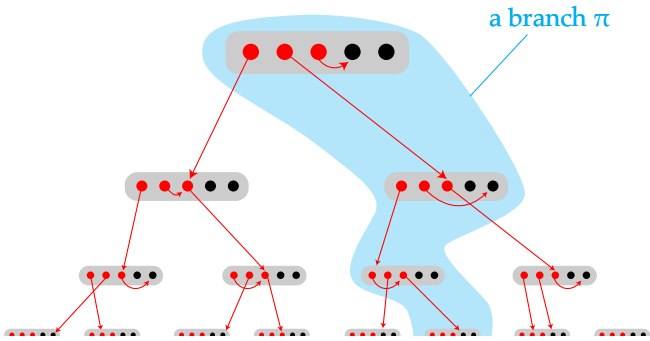
$$\{(t, [\sigma_0]) : \sigma_0 \text{ is a memoryless strategy for player 0 in } \mathcal{G}_{\mathcal{A}}(t)\} \tag{4.1}$$

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

Claim 4.7. *There is a* nondeterministic *parity automaton $\mathcal{B}$ over ω-words, such that the following conditions are equivalent for every tree t, branch $\pi \in 2^\omega$ and memoryless strategy σ_0 for player 0:*

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

2. *The automaton $\mathcal{B}$ accepts the ω-word over alphabet $\Sigma \times \Gamma_0 \times \{0,1\}$ that is obtained from (t, σ_0) by reading the branch $\pi \in 2^\omega$ (the i-th letter of the ω-word consists of the label of the i-th node in π as well as the turn that π takes after that node). Here is a picture:*



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

Apply the above claim, yielding a nondeterministic parity automaton. By McNaughton's Theorem, see Chapter 1, there exists an equivalent deterministic parity automaton, call it $\mathcal{D}$. It is not difficult to see that a memoryless strategy σ_0 wins in the game $G_{\mathcal{A}}(t)$ if and only if every branch in the tree $(t, [\sigma_0])$ is rejected by the automaton $\mathcal{D}$. This can be checked by a (deterministic top-down) parity automaton on trees, which runs the automaton $\mathcal{D}$ on every branch (and has the acceptance condition complemented). ∎

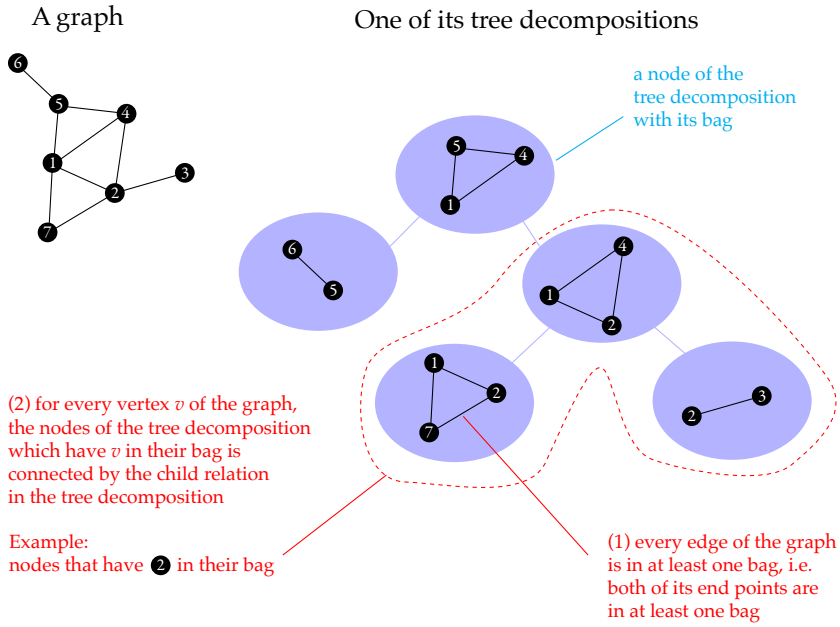
5
Treewidth

In this chapter, we present Courcelle's Theorem, which says that every formula of MSO can be evaluated in linear time on graphs that have bounded treewidth. (For MSO formulas defining properties of graphs, see Example 5.) Tree width is a graph parameter, i.e. every graph has a some treewidth, which is a natural number. The treewidth of a graph describes the smallest width of a tree decomposition that can produce the graph. The general idea is that small width tree decompositions can be obtained for graphs that are similar to trees, although there exists other inequivalent ways of quantifying similarity to a tree, e.g. clique width, see [11, Section 2.5].

5.1 Treewidth and how to compute it

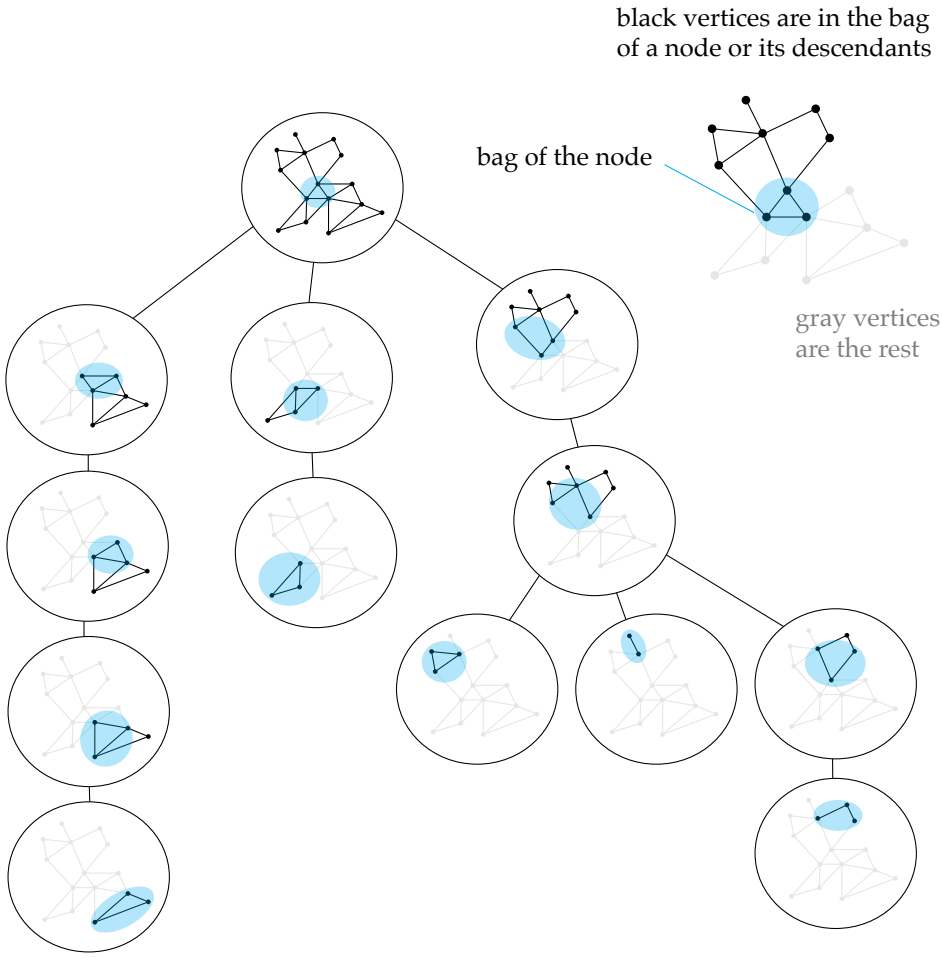
In this section, we define tree decompositions and treewidth.

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



Define the *width* of a tree decomposition to be the maximal size of a bag minus one. In the example above, the width is 2, because the maximal bag size is 3. (If you don't like doing minus one, an alternative view is that the width is the maximal intersection of bags connected by the child relation, i.e. tree width is the maximal size of an *adhesion* in the tree decomposition.) The *treewidth* of a graph is the minimal width of a tree decomposition of it. Treewidth is a fundamental concept in graph theory, which plays a prominent role in the graph minor project of Robertson and Seymour.

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



Fact 5.1. *If a graph has treewidth k, then the number of edges is at most $k \cdot (k+1)$ times the number of vertices.*

Proof. A tree decomposition can always be modified so that the bag of a node contains at least one vertex that is not present in the bags of its descendants. Therefore, the number of nodes is at most the number of vertices. Each edge

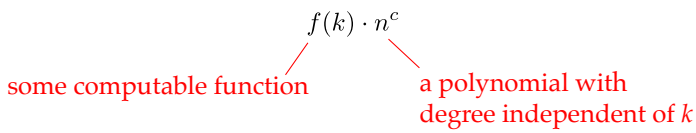
must be present in some node, and each node can have at most $k \cdot (k+1)$ edges. The bound is achieved by a clique over $k+1$ vertices. ∎

Computing a tree decomposition. We present an algorithm that computes tree decompositions of approximately optimal width (at most 4 times worse) and which runs in quadratic time when the treewidth is fixed. The algorithm is from Robertson and Seymour, see also [9, Theorem 7.18].

Theorem 5.2. *There is a function $f : \mathbb{N} \to \mathbb{N}$ and an algorithm which runs in time $f(k) \cdot n^2$ that approximates tree decompositions in the following sense:*

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

The theorem is not optimal: one can improve quadratic time to linear time, and one can compute tree decompositions of optimal width (i.e. have $< k$ instead of $< 4k$ in the output of the algorithm), see [3]. The function $f(k)$ is exponential, and there is little hope for improvement, because the following problem is NP-complete [1]: given k and a graph, decide if the graph has treewidth at most k. The theorem gives a (prototypical) example of a an algorithm that is *fixed parameter tractable*, i.e. the input has two parameters k, n and the running time is of the following form:

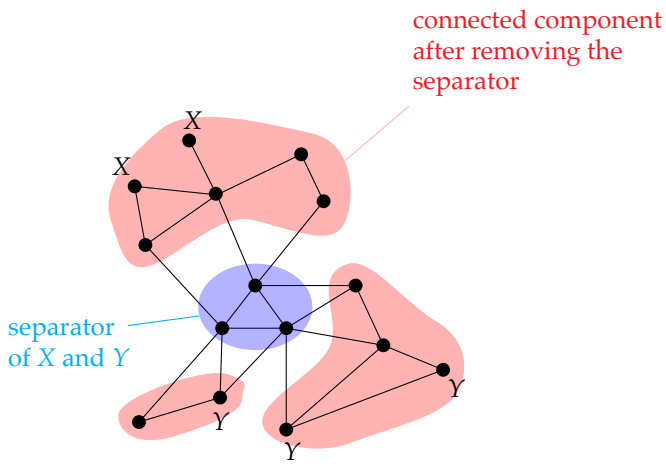


The algorithm uses the following lemma on computing separators.

Lemma 5.3. *Given a graph G and disjoint sets of vertices X, Y, one can compute a separator of minimal size in time*

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

Recall that a separator is a set of vertices S disjoint from $X \cup Y$ such that $G - S$ does not contain any path connecting X with Y, as in the following picture:

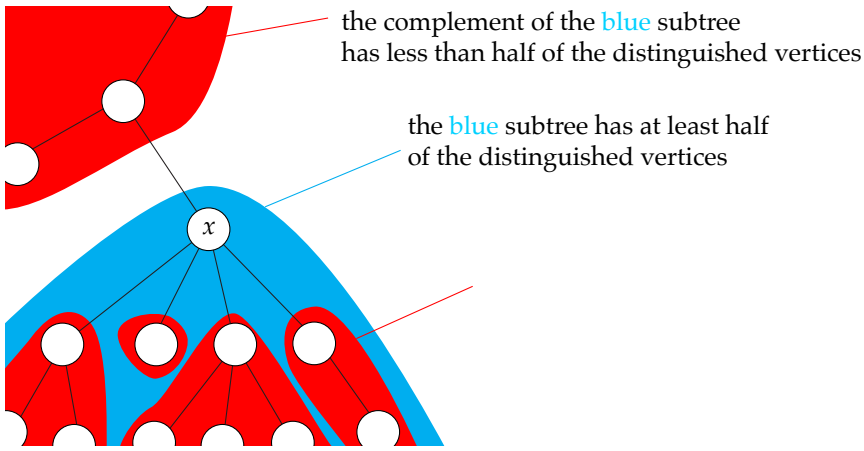


We do not prove the above lemma, it can be shown using the Ford-Fulkerson algorithm for computing maximum flow, see the discussion in [9, p. 198]. When the treewidth is fixed, the number of edges is linear in the number of vertices, and the size of the separator is bounded by a function of k (see the proof of Lemma 5.4), and therefore the running time of the algorithm is linear. The main step in proving Theorem 5.2 is the following lemma.

Lemma 5.4. *Let $k \in \mathbb{N}$. There is a linear time algorithm which does this:*

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

Proof. We begin with the algorithm, and then justify why it must succeed on graphs of treewidth $< k$. We enumerate all possible partitions of the distinguished vertices into three parts X, Y, Z, such that Z has size at most k, and each of X, Y has size at most $2k$. The idea is that Z is the distinguished vertices in the separator S. The number of such partitions is exponential in k, but is a constant if k is assumed to be fixed. For each such partition, compute a minimal size separator S of X and Y in the graph $G - Z$, and report success if the combined size of S and Z is at most k. This completes the algorithm. The running time is linear, because the size of the separator is fixed, and the number of edges is linear in the number of vertices by Fact 5.1.
We now justify that if G has treewidth $< k$ then the algorithm succeeds. If the graph has treewidth $< k$, then there is a tree decomposition where all bags have size at most k. Let t be this tree decomposition. Choose a node x of the tree decomposition so that half or more of the distinguished vertices of G appear in bags of x and its descendants, but this is no longer true for any of the children of x. Here is a picture:



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

Claim 5.5. *Let $n_1 \geq n_2 \geq \cdots \geq n_p$ be numbers in $\{1, \ldots, 2k\}$ with sum $\leq 3k$. Then*

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

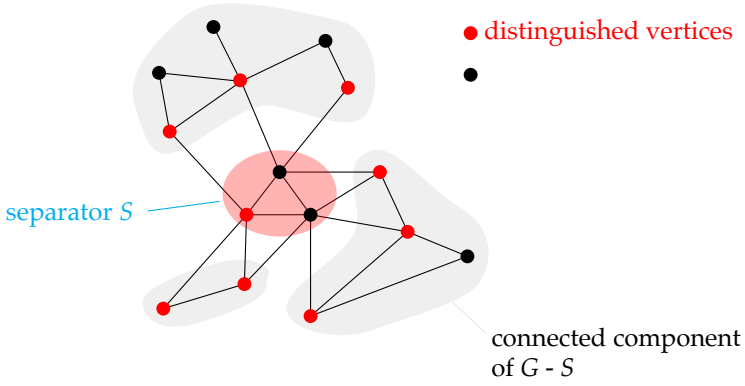
Proof. Take the first i such that the sum of the first i elements is $\geq k$. ∎

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

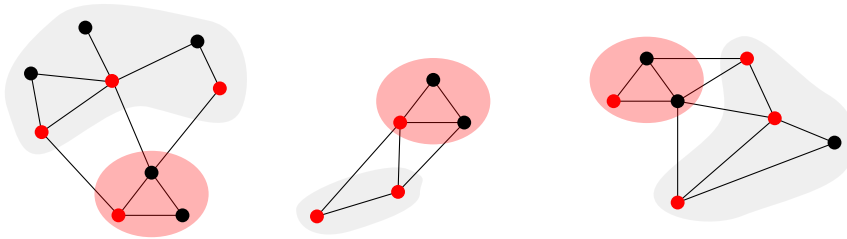
Proof of Theorem 5.2. We use a more detailed of statement of the algorithm, as described below.

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

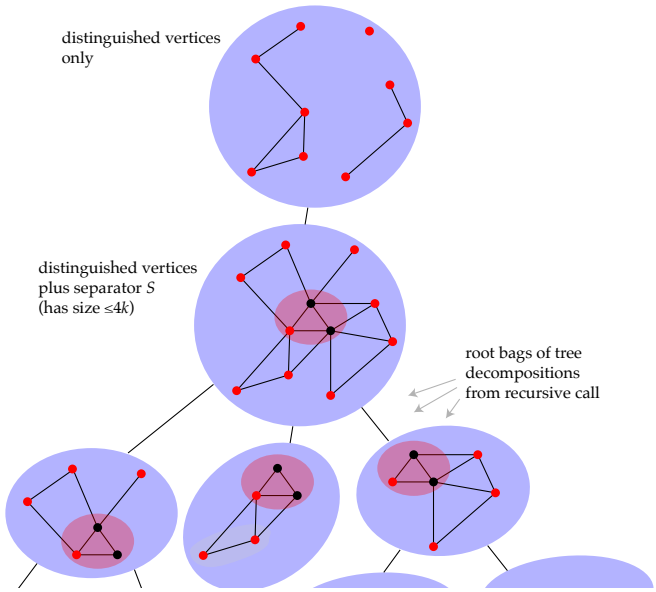
Suppose that G is the graph. If there are less than $3k$ distinguished vertices, we add some arbitrary vertices to make the set have size exactly $3k$. Apply Lemma 5.4, computing S, X and Y. If the input graph has treewidth $< k$ then the algorithm from the lemma must succeed. Find all connected components of the graph $G - S$. We know that each connected component has $\leq 2k$ distinguished vertices. Here is a picture:



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



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



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

5.2 *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 [8, Theorem 4.4].

Lemma 5.6. *For every $k \in \mathbb{N}$ and every formula of MSO φ on graphs, there is a linear time algorithm which does the following:*

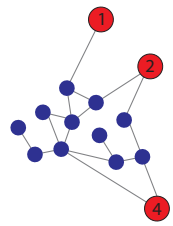
- **Input.** *A graph together with a tree decomposition of width $\leq k$;*
- **Question.** *Does the graph satisfy φ?*

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 [3]):

Theorem 5.7 (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 5.6. To this end, we present a more algebraic way of defining treewidth, so that tree decompositions can be viewed as trees over a finite ranked alphabet.

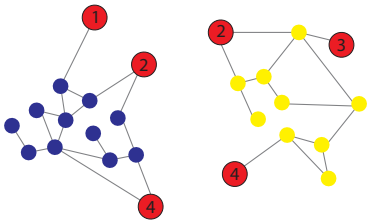
The algebra of tree decompositions. Define a *sourced graph* to be a graph with some but not necessarily all vertices being assigned natural numbers. The vertices with numbers are called the *sources* and the numbers are called the *source names*. We assume that the each source name is used for at most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



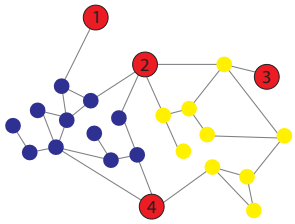
The purpose of sourced graphs is to fuse them. The operation inputs two sourced graphs, and it outputs their disjoint union with each pair of sources

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

two sourced graphs

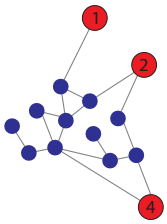


their fusion

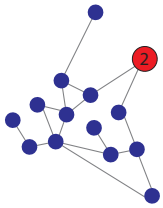


A second operation is forgetting a subset of source names, illustrated below:

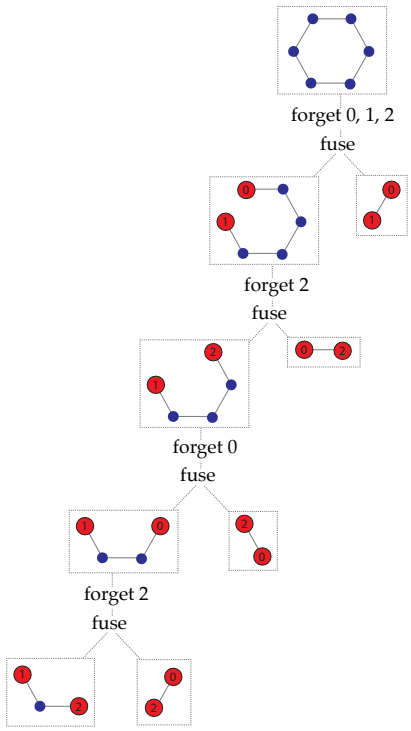
a sourced graph



after forgetting 1, 4.



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



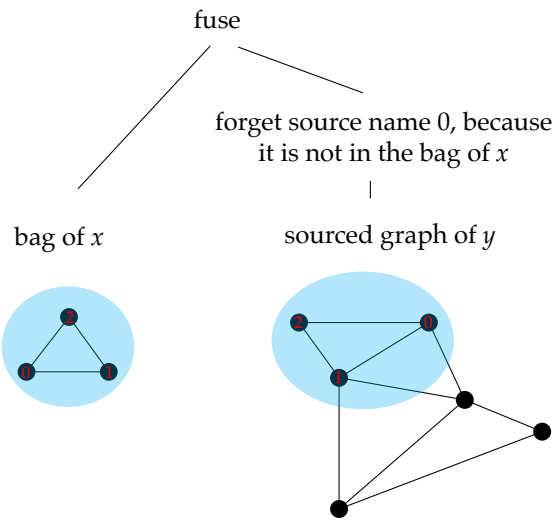
Fact 5.8. *A graph has treewidth k if and only if (when viewed as a sourced graph without any sources) it can be generated by a term in the algebra of width k sourced graphs, starting with constants that have at most $k+1$ vertices.*

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 5.8 can be viewed as a tree over a ranked alphabet Σ_k where:

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

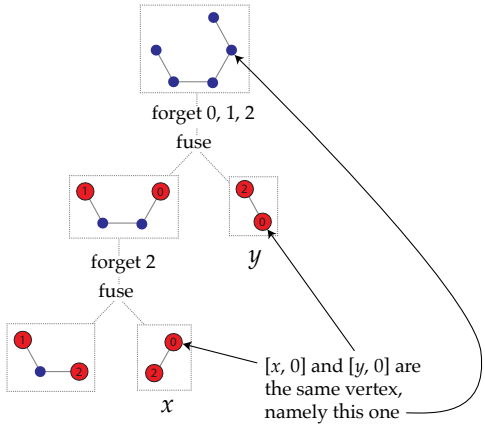
From a width k tree decomposition one compute a corresponding tree over the above alphabet in linear time. By Theorem 4.3, for finite trees over alphabet Σ_k,

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 5.6 will follow once we prove the following lemma.

Lemma 5.9. *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*

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

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



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

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

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

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

The professional terminology for the construction described above is "the graph generated by t can be produced from t using an MSO transduction", see [11, Section 1.7]. ∎

6
Weighted automata over a field

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

Definition 6.1 (Weighted automaton over $\mathbb{Q}$)**.** *A weighted automaton consists of the following ingredients:*

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

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

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

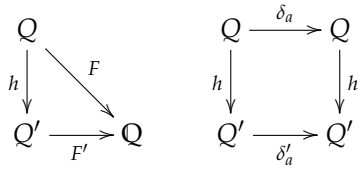
letter a, it applies the transition function δ_a to the current state, yielding a new state. Finally, it applies the output function F to the state at the end, yielding an element of the underlying field.

Example 8. A normal deterministic finite automaton with n states can be viewed as a special case of a weighted automaton. The set of stats will be $\mathbb{Q}^n$, and the reachable ones will only be vectors which have zero on all coordinates except the coordinate corresponding to the current state. □

6.1 *Minimisation of weighted automata*

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

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



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

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

Theorem 6.2. *Let $f : \Sigma^* \to \mathbb{Q}$ be a function computed by a weighted automaton. There exists a weighted automaton, called the syntactic automaton of f, which computes f and such that every reachable weighted automaton computing f admits a homomorphism into the syntactic automaton.*

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

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

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

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

is simply a permutation of coordinates:

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

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

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

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

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

6.2 *Algorithms for equivalence and minimisation*

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

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

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

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

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

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

This way we get a growing chain of linear subsets

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

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

Theorem 6.3. *The following problem is in polynomial time:*

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

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

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

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

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

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

$$qw = pw$$

holds for all input words of length at most n, where qw if the output of the automaton after reading word w assuming that the initial state was changed to q. This equivalence relation can be seen as a subset of

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

By linearity of the automaton, the subset is linear, i.e. it is closed under $+$ and multiplying by scalars. Therefore we have a sequence of subsets

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

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

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

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

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

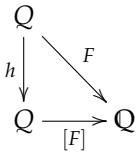
be the function which maps a state to its equivalence class. Because E_* is an equivalence relation and a linear set, the set $[Q]$ is a vector space, and h is a linear function. What is the dimension of $[Q]$ and how do we represent it and the function h, assuming that $Q = \mathbb{Q}^n$ for some n?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

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

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

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

diagram commute



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

6.3 *Undecidable emptiness*

In Theorem 6.3, we showed that equivalence of weighted automata is decidable, in fact in polynomial time. A corollary is that one can decide if a weighted automaton maps all inputs to zero. We now show that a dual problem, namely mapping some word to zero, is undecidable.

Theorem 6.4. *The following problem is undecidable:*

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

Section 6.3 is devoted to proving the above undecidability result theorem. There are two basic ingredients in the proof: hashing words as numbers, and composing weighted automata with sequential transducers. For the latter, it is convenient to use an equivalent version of weighted automata, which allows control states from a finite set. These ingredients are described below.

Hashing. The archetypical function that can be computed by a weighted automaton is mapping a string of digits to its interpretation as a fraction stored in binary (or ternary, etc) notation. This construction is described in the following lemma.

Lemma 6.5. *For every alphabet* Σ *there is a finite weighted automaton which computes an injective function to the strictly positive rational numbers*

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

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

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

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

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

The output function is

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

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

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

$$\delta_a : Q \to Q$$

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

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

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

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

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

to the vector v.

Lemma 6.6. *Weighted automata and weighted automata with states recognise the same functions.*

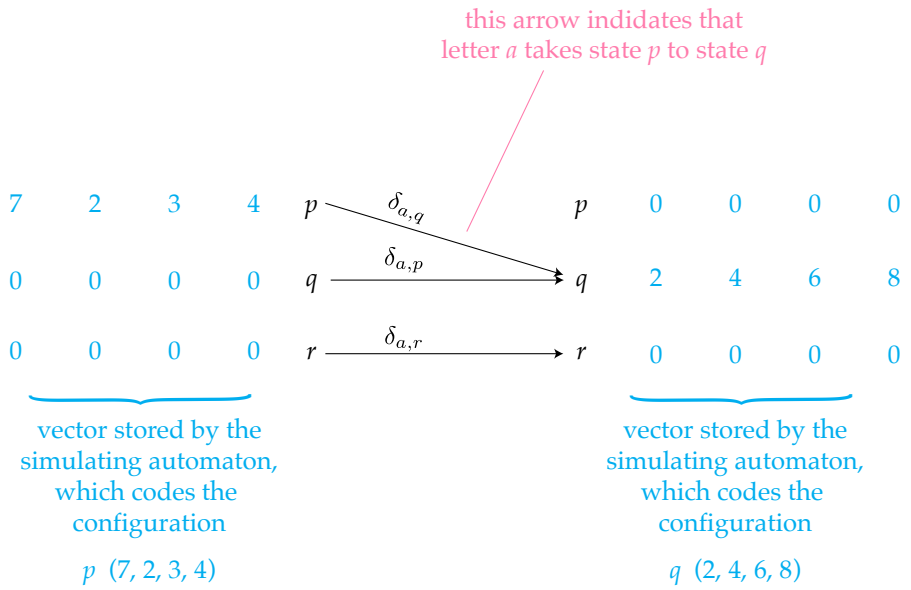
Proof. We need to show how a weighted automaton with states is converted into one without states. Consider an weighted automaton with states $\mathcal{A}$ which has control states Q and dimension n. We will simulate $\mathcal{A}$ by a weighted automaton $\mathcal{B}$ without states, which will have dimension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

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

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

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

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

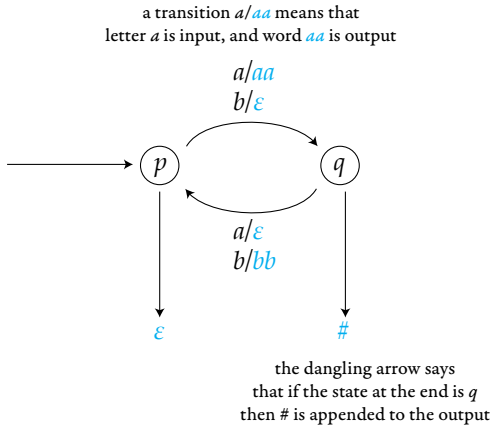


■

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

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

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



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

Lemma 6.7. *If $f, g : \Gamma^* \to \mathbb{Q}$ are recognised by finite weighted automata, then also the following functions are also recognised by finite weighted automata:*

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

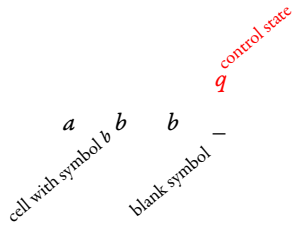
Proof. The weighted sum is immediate. For items 2 and 3, it is convenient to use an extended model, The closure properties in items 2 and 3 of the lemma are easy to do for extended weighted automata, and therefore the lemma follows from the above claim. ∎

Equipped with Lemmas 6.5 and 6.7, we can now prove the undecidability result from Theorem 6.4.

Proof of Theorem 6.4. A reduction from the halting problem. For a Turing Machine M, we will define a weighted automaton $\mathcal{A}$ such that $\mathcal{A}$ can output zero if and only if M has some halting computation. Suppose that Σ is the work alphabet of the machine M. We encode a configurations of the machine as a word over the alphabet

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

in the natural way, here is a picture:



To make the encoding unique, we assume that the first letter and the last letter are not just blank symbols, i.e. each one contains either the head, or a non-blank tape symbol, or both. It is not difficult to see that there is a sequential transducer $s : \Delta^* \to \Delta^*$ such that if w encodes a configuration, then $s(w)$ encodes the same configuration after one computation step. Define L to be the set of words of the form

$$w_1 \# w_2 \# \cdots \# w_{n-1} \# w_n \tag{6.1}$$

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

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

to be functions that map a word as in (6.1) to numbers that represent (according to Lemma 6.5) the words

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

respectively (it is not important what the functions do for inputs outside L). The first function applies the successor function to the first $n-1$ configurations (the red separator is used to not copy the last configuration), while the second functions copies the last $n-1$ configurations. The functions f_1, f_2 are recognised by weighted automata thanks to Lemmas 6.5 and 6.7. From the construction, it follows that the Turing machine has a halting computation if and only if

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

This problem can be reduced to checking if some weighted automaton can produce zero, thanks to the closure properties in Lemma 6.7. ∎

7
Polynomial grammars

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

Polymoial grammars. In the definitions below, it will be convenient to use polynomial functions from vectors to vectors. Define a *polynomial function* to be a function of the form

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

which is given by k polynomials representing the coordinates of the output vector, each one with n variables representing the coordinates of the input vector.
A polynomial grammar is a variant of a context free grammar. It generates rational numbers (as opposed to words) and uses polynomial functions in the rules (as opposed to concatenation). Also, nonterminals are allowed to generate tuples of rational numbers, although we require the starting nonterminal to generate only numbers (this restriction is not important).

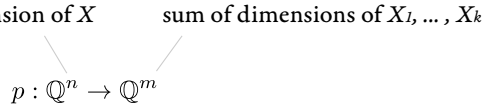
Definition 7.1 (Polynomial grammar)**.** *A* polynomial grammar *consists of*

- *a set $\mathcal{X}$ of nonterminals, each one with a* dimension *in* $\{1, 2, \ldots\}$*;*

- *a designated starting nonterminal of dimension 1;*
- *a finite set of productions of the form*

$$X \leftarrow p(X_1, \ldots, X_k)$$

where $k \in \{0, 1, \ldots\}$, $X_1, \ldots, X_k, X$ are nonterminals, and



is a polynomial function.

If a nonterminal has dimension n, then it generates a subset of $\mathbb{Q}^n$, which is defined as follows by induction. (The language generated by the grammar is defined to be the subset generated by its starting nonterminal.) Suppose that

$$X \leftarrow p(X_1, \ldots, X_k)$$

is a production and we already know that vectors $v_1, \ldots, v_k$ are generated by the terminals $X_1, \ldots, X_k$ respectively. Then the vector $p(v_1, \ldots, v_k)$ is generated by nonterminal X. The induction base is the special case of $k = 0$, where the polynomial p is a constant.

Example 9. This grammar (there is only the starting nonterminal, which has dimension one) generates all odd natural numbers

$$X \leftarrow 1 \qquad X \leftarrow X + 2.$$

If we replace $X + 2$ by $X \times 2$, then we get the powers of two. The following grammar generates numbers of the form 2^{2^n}:

$$X \leftarrow 2 \qquad X \leftarrow X^2.$$

We can also generate factorials. Apart from the starting nonterminal X, we have a nonterminal Y of dimension two which generates pairs of the form $(n, n!)$. The crucial rule is this:

$$Y \leftarrow p(Y) \qquad \text{where } p \text{ is defined by } (a, b) \mapsto (a + 1, (a + 1) \cdot b).$$

The remaining rules are $Y \leftarrow (1, 1)$ and $X \leftarrow \pi_1(Y)$ where π_1 is defined by $(a, b) \mapsto a$. $\square$

As usual, one can adopt an alternative fix-point view on grammars. We present this view since it will be used in the proof of Theorem 7.2. Define a *solution* to a grammar to be a function η which associates to each nonterminal X a set of vectors of rational numbers of same dimension as X, and which satisfies all of the productions in the sense that

$$\underbrace{X \leftarrow p(X_1, \ldots, X_k)}_{\text{is a production}} \qquad \text{implies} \qquad \eta(X) \supseteq p(\eta(X_1) \times \cdots \times (\eta(X_k))$$

It is not difficult to see that the function which maps a nonterminal to the set of vectors generated by it is a solution, and it is the least solution with respect to coordinate-wise inclusion.

This chapter shows the following theorem.

Theorem 7.2. *One can decide if the language of a polynomial grammar is contained in* $\{0\}$.

The nonzeroness problem is clearly semi-decidable: one can enumerate all derivations of the grammar, and stop when a derivation is found that generates a nonzero number. The crucial ingredient is having a finite witness that the generated language is contained in $\{0\}$. For this, we use the Hilbert Basis Theorem.

Hilbert's Basis Theorem. Let X be a set of variables. Let us write $\mathbb{Q}[X]$ for the set polynomials with variables in X and coefficients in $\mathbb{Q}$. Define an *ideal* to be a set $I \subseteq \mathbb{Q}[X]$ with the following closure properties:

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

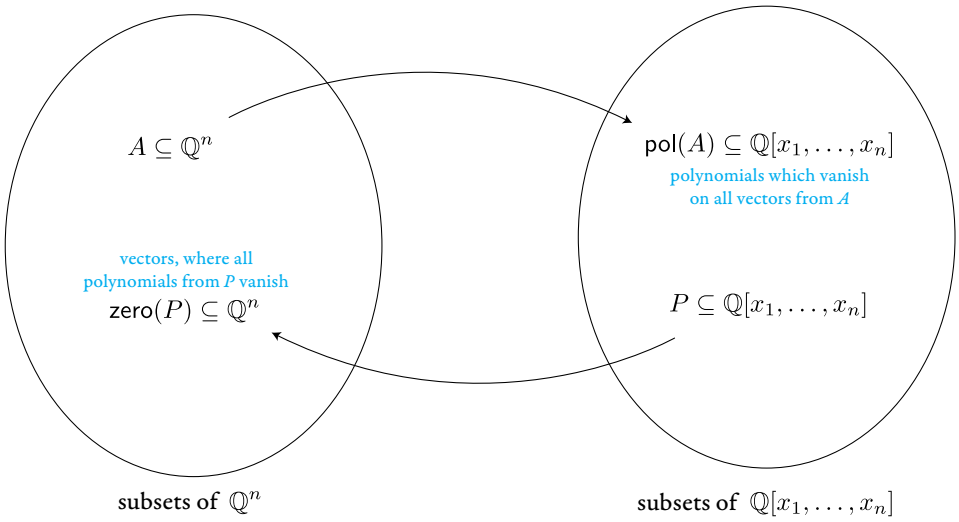
If $P \subseteq \mathbb{Q}[X]$ is a set of polynomials, then the ideal generated by P, denoted by $\langle P \rangle$, is the set of polynomials of the form

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

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

Theorem 7.3. *If X is a finite set of variables, then every ideal in $\mathbb{Q}[X]$ is finitely generated, i.e. of the form $\langle P \rangle$ for some finite set of polynomials. Furthermore, given a polynomial p and a finite set of polynomials P, one can decide if p belongs to the ideal $\langle P \rangle$.*

Algebraic closure. We say that a polynomial function $p : \mathbb{Q}^n \to \mathbb{Q}$ *vanishes* on a set $X \subseteq \mathbb{Q}^n$ if p is the constant zero over this set. For a fixed dimension n, consider the following two operations pol and zero which go from sets of vectors to sets of polynomials, and back again:



The picture above is a special case of what is known as a *Galois connection*. Note that the operation pol produces only ideals. For a set of vectors $A \subseteq \mathbb{Q}^n$, define its *closure* as follows:

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

Example 10. Suppose that the dimension is $n = 1$. Then $\mathsf{pol}(A)$ contains only the constant zero polynomial whenever A is infinite. This means that the algebraic closure of any infinite set $A \subseteq \mathbb{Q}$ is the whole space $\mathbb{Q}$. On the other hand, when A is finite, then there is a polynomial which vanishes exactly on the points from A, and therefore $\overline{A} = A$. For higher dimensions, an example of a closed set is the unit circle, because in this case the ideal $\mathsf{pol}(A)$ is generated by the polynomial $x^2 + y^2 = 1$. $\square$

The closure operation defined above is easily seen to be a closure operator, in the sense that it can only add elements to the set, it is monotone with respect to inclusion, and applying the closure a second time adds nothing. By the Hilbert Basis Theorem, the algebraic closure can be represented by giving a finite basis for the ideal $\mathsf{pol}(A)$.

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

Lemma 7.4. *Let $\mathcal{G}$ be a polynomial grammar with nonterminals $\mathcal{X}$, and let η be a solution (not necessarily the least solution). Then $\overline{\eta}$, defined by $X \in \mathcal{X} \mapsto \overline{\eta(X)}$ is also a solution.*

Proof. We first prove two inclusions, (7.1) and (7.2), which show how closure interacts with polynomial images and Cartesian products. The first inclusion is about polynomial images:

$$p(\overline{A}) \subseteq \overline{p(A)} \qquad \text{for every } A \subseteq \mathbb{Q}^n \text{ and polynomial } p : \mathbb{Q}^n \to \mathbb{Q}^m. \qquad (7.1)$$

To prove the above inclusion, we need to show that if a polynomial vanishes on $p(A)$, then it also vanishes on $p(\overline{A})$. Suppose that a polynomial q vanishes on $p(A)$. This means that the polynomial $q \circ p$ vanishes on A, which means that $q \circ p$ also vanishes on $\overline{A}$, by definition of closure. Therefore q, vanishes on $p(\overline{A})$.

The second inclusion is about Cartesian products:

$$\overline{A} \times \overline{B} \subseteq \overline{A \times B} \qquad \text{for every } A \subseteq \mathbb{Q}^n \text{ and } B \subseteq \mathbb{Q}^m. \tag{7.2}$$

We need to show that if a polynomial vanishes on $A \times B$, then it also vanishes on $\overline{A} \times \overline{B}$. Suppose that q vanishes on $A \times B$. Take some $b \in B$. The polynomial $q(\_, b)$ vanishes on A, and therefore it vanishes on $\overline{A}$ by definition of closure. Therefore, q vanishes on $\overline{A} \times B$. Applying the same reasoning again, we get that q vanishes on $\overline{A} \times \overline{B}$, proving the inclusion (7.2).
We are now ready to prove the lemma. Let η be some solution to the grammar. Take some rule $X \leftarrow p(X_1, \ldots, X_n)$ in the grammar $\mathcal{G}$. The following shows that $\overline{\eta}$ is compatible with the rule, and therefore $\overline{\eta}$ is a solution to the grammar by arbitrary choice of the rule.

$$\begin{aligned}
p(\overline{\eta}(X_1), \ldots, \overline{\eta}(X_n)) &= \text{by definition of } \overline{\eta} \\
p(\overline{\eta(X_1)}, \ldots, \overline{\eta(X_n)}) &\subseteq \text{repeated application of (7.2)} \\
p(\overline{\eta(X_1) \times \cdots \times \eta(X_n)}) &\subseteq \text{by (7.1)} \\
\overline{p(\eta(X_1) \times \cdots \times \eta(X_n))} &\subseteq \text{because } \eta \text{ is solution and closure is monotone} \\
\overline{\eta(X)} &= \text{by definition of } \overline{\eta} \\
\overline{\eta}(X)
\end{aligned}$$

This completes the proof of the lemma. Note that the proof is not very specific to polynomials and rational numbers, and it would work for more abstract notions of algebra, as will be defined in Section 7.1. ∎

We now complete the proof of Theorem 7.2. Recall that by enumerating derivations, we have an algorithm that terminates if and only if the grammar generates some nonzero vector. We now give an algorithm that terminates if and only if the grammar generates a language contained in $\{0\}$. The algorithm simply enumerates through all *closed solutions* to the grammar, i.e. solutions which map each nonterminal to a closed set. By Lemma 7.4, the generated language is contained in $\{0\}$ if and only if there is a an assigment η which maps nonterminals to closed sets such that:

1. η is a solution to the grammar; and

2. η maps the starting nonterminal to $\{0\}$ or the empty set.

By Hilbert's Basis Theorem, we can enumerate candidates for η, by using finite sets of polynomials to represent closed sets. It remains show that, given η, one can check if conditions 1 and 2 above are satisfied. For this, we use the following lemma, which follows from decidability of membership $p \in \langle P \rangle$.

Lemma 7.5. *Assume that a closed set A is represented by a finite basis of the ideal $\mathsf{pol}(A)$. If A, B are closed sets, then the following are also closed, and their representations can be computed:*

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

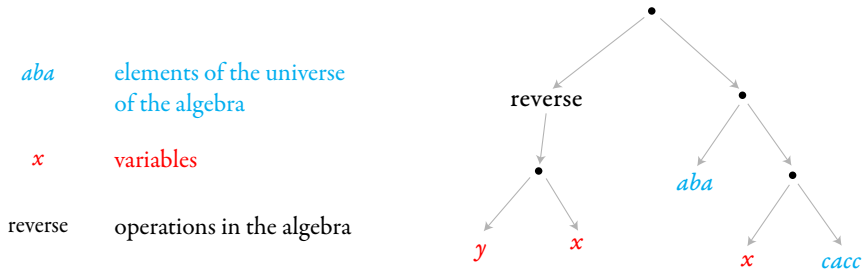
Furthermore, one can decide equality on closed sets.

7.1 Application to equivalence of register automata

We now use Theorem 7.2 to decide equivalence for automata which use registers to manipulate values in certain kinds of algebras. In this section, we use the notion of algebra from universal algebra, i.e. an *algebra* is defined to be a set equipped with some operations. Examples include

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

We adopt the convention that boldface letters like $\mathbf{A}$ and $\mathbf{B}$ range over algebras, and the universe (the underlying set) of an algebra $\mathbf{A}$ is denoted by A. Define a polynomial with variables X in an algebra $\mathbf{A}$ to be a term built out of operations from the algebra, variables from X and elements of the universe. Here is a picture of a polynomial with variables $\{x, y\}$ in an algebra where the universe is $\{a, b, c\}^*$ and the operations are concatenation (binary) and reverse (unary):



We write $\mathbf{A}[X]$ for the set of polynomials over variables X in the algebra $\mathbf{A}$. Such a polynomial represents a function of type $A^X \to A$ in the natural way. We extend this notion to function of type $A^n \to A^m$ by using m-tuples of polynomials with n variables.

Example 11. If the algebra is $(\mathbb{Q}, +, \times)$, then the polynomials are the polynomials in the usual sense, e.g. a binary polynomial is

$$x^2 + 3x^3y^4 + 7.$$

If we choose the algebra to be $\mathbb{Q}$ with the operations being addition and the family of scalar multiplications $\{x \mapsto ax\}_{a \in \mathbb{Q}}$, then the polynomials are exactly the affine functions. □

Definition 7.6 (Register automaton)**.** *Let* $\mathbf{A}$ *be an algebra. The syntax of a* register automaton over $\mathbf{A}$ *consists of:*

- *a finite input alphabet* Σ*;*
- *a finite set* R *of registers;*
- *a finite set* Q *of states;*
- *an initial configuration in* $Q \times A^R$*;*
- *a transition function* $\delta : Q \times \Sigma \to Q \times (\mathbf{A}[R])^R$*;*
- *an output dimension* n *and an* output function $F : Q \to (\mathbf{A}[R])^n$.

The semantics of the automaton is a function of type $\Sigma^* \to A^n$ defined as follows. The automaton begins in the initial configuration. After reading each letter, the configuration is updated according to

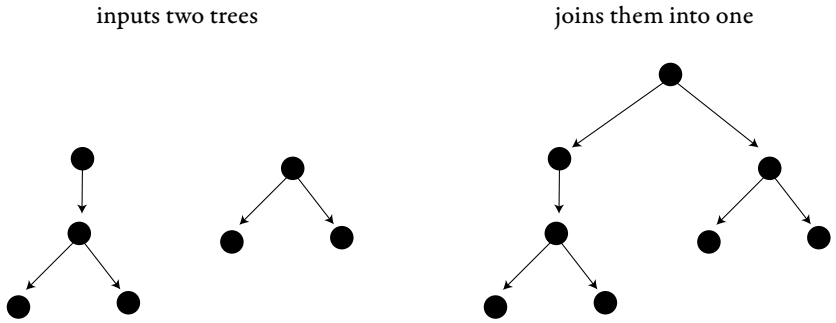
$$(q, v) \overset{a}{\mapsto} (p, f(v)) \qquad \text{where } \delta(q, a) = (p, f).$$

If the configuration after reading the entire input is (q, v), then the output of the automaton is obtained by applying $F(q)$ to v.

Example 12. A language $L \subseteq \Sigma^*$ is regular if and only if its characteristic function $\Sigma^* \to \{0, 1\}$ is recognised by a register automaton with no registers over the algebra with universe $\{0, 1\}$ and no operations. □

Example 13. Let Σ be a finite alphabet. Here is a register automaton over the algebra $(\Sigma^*, \cdot)$ which implements the reverse function $\Sigma^* \to \Sigma^*$. The automaton has one register, call it 1 and one state. When it reads a letter a, it executes the register update given by the polynomial $a \cdot 1$. The output function is the identity. □

Example 14. Consider an algebra $\mathbf{A}$ where the universe is the set of trees (viewed as directed graphs, with edges directed away from the root), and which has one binary operation depicted as follows:



It is not difficult to write a register automaton over the algebra $\mathbf{A}$, with input alphabet $\{a\}$, which maps a word a^n to a balanced binary tree of depth n. □

The following result is a direct corollary of Theorem 7.2. The result was first shown in [2, Theorem 4], were the proof was also based on the Hilbert Basis Theorem.

Theorem 7.7. *The following problem is decidable:*

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

The above theorem generalises Theorem 6.3, because weighted automata can be viewed as a special case of register automata over $(\mathbb{Q}, +, \times)$ where only linear maps are allowed as the register updates, instead of arbitrary polynomials, as allowed in Theorem 7.7. The generalisation of Theorem 6.3 is only in terms of decidability, since the running time of the algorithm for Theorem 7.7 is not estimated in any way, not to mention polynomial time. In fact, a lower bound of Ackermann time is given in [2, Theorem 1].

Proof. Suppose that the input functions are f, g. By doing a natural product construction, we can compute a register automaton that recognises the difference function $f - g$. Therefore, the problem boils down to deciding if a function h, e.g. the difference, is constant equal to zero. For this we use a grammar. Suppose that h is recognised by an register automaton with states Q and n registers. Define a grammar where the nonterminals are

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

The starting nonterminal is 1. By copying the transitions of the automaton, we can write the rules of the grammar so that nonterminal q generates exactly those tuples $\overline{a} \in \mathbb{Q}^n$ such that configuration $(q, \overline{a})$ can be reached over some input. By using the output function of the automaton, we can ensure that the starting nonterminal produces exactly the outputs of the automaton. Therefore, the language defined by the grammar is contained in $\{0\}$ if and only if the automaton can only produce 0, and the former is decidable by Theorem 7.2. ■

Note that the above theorem, with the same proof, would also work for tree register automata, i.e. a generalisation of register automata for inputs that are trees.

Other algebras. In Theorem 7.7, we showed that equivalence is decidable for register automata over the algebra $(\mathbb{Q}, +, \times)$. From this we can infer decidability for some other algebras, by coding them into rational numbers according to the following definition.

Definition 7.8. *Let* $\mathbf{A}$ *and* $\mathbf{B}$ *be algebras. We say that* $\mathbf{A}$ *can be simulated by polynomials of* $\mathbf{B}$ *(no relation to polynomial time computation) if there is some dimension* n *and an injective function*

$$\alpha : A \to B^n$$

with the following property. For every operation $f : A^m \to A$ *in the algebra* $\mathbf{A}$*, there is a polynomial* g *of* $\mathbf{B}$ *which makes the following diagram commute*

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

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

Corollary 7.9. *For every finite alphabet* Σ*, the equivalence problem is decidable for register automata over the algebra* $(\Sigma^*, \cdot)$*.*

Proof. By Theorem 7.7, it suffices to show that $(\Sigma^*, \cdot)$ can be simulated by polynomials of $(\mathbb{Q}, +, \times)$. Assume without loss of generality that Σ is $\{0, \ldots, n-1\}$. We use the coding:

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

The unique operation of the algebra $(\Sigma^*, \cdot)$, namely string concatenation, is encoded by the polynomial

$$((a, b), (a', b')) \quad \mapsto \quad (a \times a', b \times a' + b')$$

By the remarks after Theorem 7.7, the decidability result would extend to tree automata over the algebra $(\Sigma^*, \cdot)$. This yields the result that equivalence is decidable for tree-to-string transducers as considered in [17].

■

Problem 7. 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.

8
Parsing in matrix multiplication time

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

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

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

The naive algorithm for the above problem runs in time n^3, but smarter algorithms run faster, e.g. the Strassen algorithm runs in time approximately $\mathcal{O}(n^{2.8704})$, and the record holder as of 2017 is $\mathcal{O}(n^{2.3729})$.

Theorem 8.1. *Assume that multiplication of $n \times n$ Boolean matrices can be computed in time $\mathcal{O}(n^\omega)$ for $\omega \in \mathbb{Q}$. Then membership in a context-free language can be decided in time at most*

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

For the rest of this chapter, fix ω and a context-free grammar $\mathcal{G}$. We assume that the grammar is in Chomsky Normal Form, i.e. every rule is of the form

$X \leftarrow YZ$ or $X \leftarrow a$ (where a is a nonterminal). A grammar can be converted into Chomsky Normal Form in polynomial time, so this assumption can be made without loss of generality. Define an length n *parse matrix* to be a family

$$M = \{M_X\}_{X \text{ is a nonterminal}}$$

such that each M_X is a family of intervals in $\{1, \ldots, n\}$. The intuition is that $I \in M_X$ means that the nonterminal X generates the infix corresponding to I. For parse matrices M, N of same dimension, define their product $M \circ N$ by

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

where the union ranges over rules of the grammar and $M_Y \circ M_Z$ is the family of intervals that can be decomposed as a disjoint union of an interval from M_Y followed by an interval from M_Z.

Lemma 8.2. *For length n parse matrices, product can be computed in time $\mathcal{O}(n^\omega)$.*

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

We say that a parse matrix M is *closed* if it satisfies $M \circ M \subseteq M$, and we say that it is closed on an interval $I \subseteq \{1, \ldots, n\}$ if it is closed when restricted to intervals contained in I. For a parse matrix M, define its *closure* M^* to be the least (with respect to inclusion) parse matrix that contains M and is closed.

Proposition 8.3. *There is an algorithm which runs in time $T(n)$ of order at most*

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

and computes the closure of a length $2n$ parse matrix, assuming that it is closed on the intervals $\{1, \ldots, n\}$ and $\{n+1, \ldots, 2n\}$.

Before proving the proposition, we show how it implies the Theorem 8.1.

Proof of Theorem 8.1. Suppose that we want to know if the grammar $\mathcal{G}$ generates a word w of length n. Define M to be the length n parse matrix where M_X contains intervals $\{i\}$ such that nonterminal X generates the i-th letter of w. This parse matrix can be computed in time linear in n. It is easy to see that w is generated by the grammar if and only if the closure M^* contains $\{1, \ldots, n\}$ on the component corresponding to the starting nonterminal. It suffices therefore to compute M^*. To make the computation easier, suppose that the length of the word is a power of two, i.e. $n = 2^k$. We do a divide an conquer approach: we compute the closures of the parse matrixs for the first and second halves of w (using a recursive procedure), and then combine these using the algorithm from Proposition 8.3. The running time of this algorithm is at most

$$T(n) + 2T(\frac{n}{2}) + \cdots + 2^k T(\frac{n}{2^k}). \tag{8.1}$$

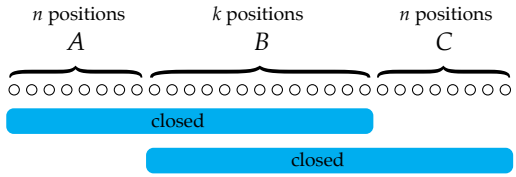
Because $T(n)$ is at least linear, it follows that

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

which shows that the running time (8.1) is at most $\log n$ times slower then $T(n)$, thus proving the theorem, given the bounds on $T(n)$ from Proposition 8.3. ∎

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

Lemma 8.4. *Suppose that M is a length $k + 2n$ parsing matrix that is closed on the intervals $A \cup B$ and $B \cup C$ as depicted below:*



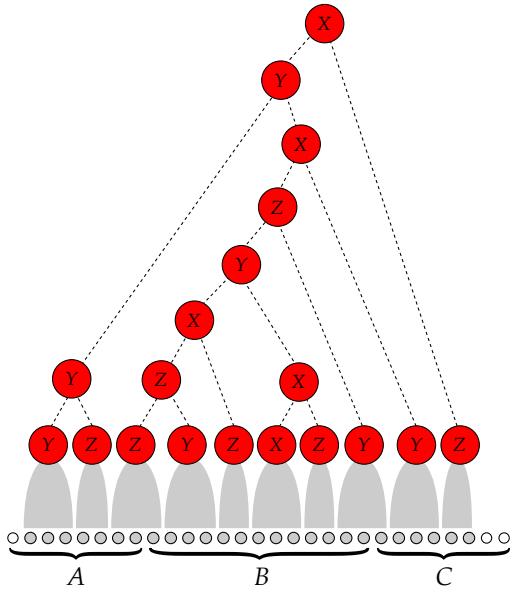
Then the closure M^* can be computed in time $\text{poly}(\mathcal{G}) \cdot n^\omega + T(n)$.

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

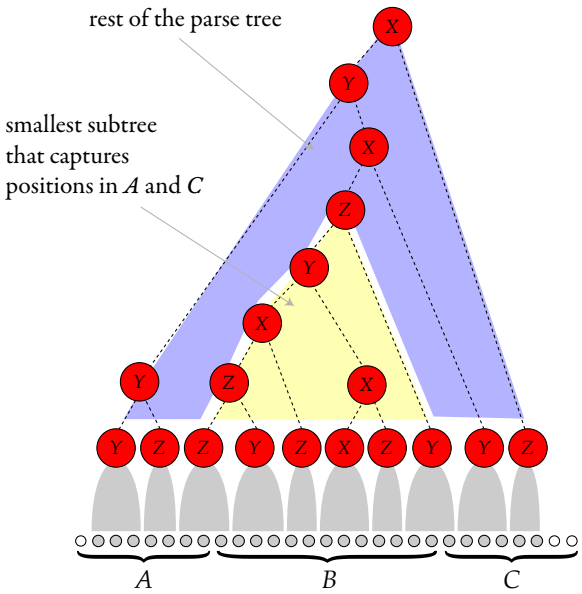
$$M^* = M \cup N^*. \tag{8.2}$$

Before proving the above equality, we note that the right side of the above equality can be computed in time as in the statement of the lemma, thus proving the lemma. By Lemma 8.2, N can be computed in time $\text{poly}(\mathcal{G}) \cdot n^\omega$. Because the matrix M was closed over intervals A and C, it follows N is also closed over these intervals. Since all entries of N contain B, it is essentially a matrix of length $2n$ whose first and second halves are closed. It follows that N^* can be computed in time $T(n)$.

It remains to prove the equality (8.2). The inclusion $\supseteq$ is immediate, it remains to justify the inclusion $\subseteq$. We need to show that if M^* contains interval I on nonterminal X, then this is true for $M \cup N^*$. If I is contained in $A \cup B$ or $B \cup C$, then this implication holds by the closure assumptions on M. The remaining case is when I contains B. The reason for M^* containing I on nonterminal X is a parse tree as described in the following picture:

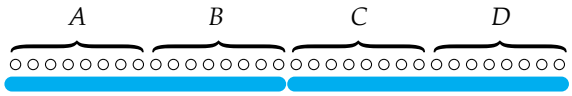


The leaves in the picture correspond to intervals from M, and the inner nodes correspond to intervals that appear when computing the closure M^*. Consider the smallest subtree of the parse tree which captures positions from both B and C, decomposing the parse tree into two parts (yellow and blue):



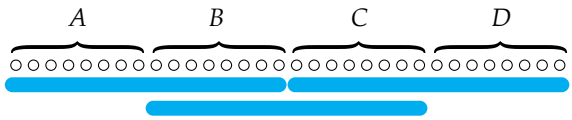
By minimality, the left subtree of the yellow subtree captures positions only in $A \cup B$, and the right subtree of the yellow subtree captures only positions in $B \cup C$. Therefore, the root of the yellow subtree corresponds to the matrix $M \circ M$. It follows that all leaves of the blue part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

Proof of Proposition 8.3. Here is the algorithm. Suppose that M is a length $2n$ parse matrix which is closed on its first and second halves, as in the statement of the proposition. Let us write A, B, C, D for the intervals describing the four quarters of $2n$, as in the following picture:

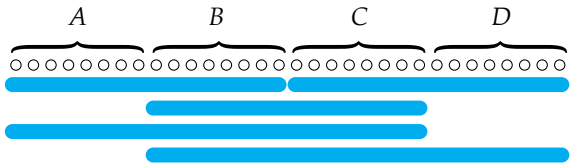


As in Lemma 8.4, the blue rectangles indicate the intervals which are closed.

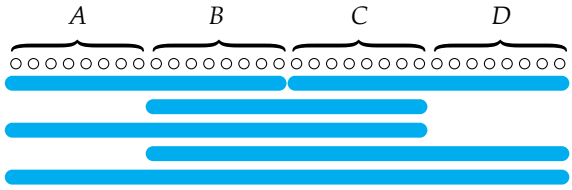
1. By induction, compute the closure of the interval $B \cup C$:



2. Using Lemma 8.4 twice, compute the closures of $A \cup B \cup C$ and $B \cup C \cup D$:



3. Using Lemma 8.4, 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 8.1, we get

$$T(n) = 3c \cdot n^{\omega} + 4 \cdot 3c \cdot (\frac{n}{2})^{\omega} + \cdots + 4^{k} \cdot 3c \cdot (\frac{n}{2^{k}})^{\omega}.$$

Because n^{ω} is at least quadratic (an algorithm for matrix multiplication must at least read two $n \times n$ matrices), it follows that

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

which gives the bound in the proposition. ∎

Problem 8. Show that the operation $M \circ N$ on parse matrices is not associative.

Exercise solutions

1
Determinisation of ω-automata

Problem 1. *Are the following languages regular:*

1. *prefix of v belongs infinitely often to the fixed regular language of finite words $L \subseteq \Sigma^*$;*
2. *word v contains infinitely many infixes of the form $ab^p a$, where p is prime;*
3. *word v contains infinitely many infixes of the form $ab^p a$, where p is even;*
4. *word v contains arbitrary long infixes in the fixed regular language of finite words L;*
5. *prefix of v belongs infinitely often to the fixed language of finite words $L \subseteq \Sigma^*$ (not necessarily regular).*

Solution. Solutions of the points:

1. YES. Let $\mathcal{A}$ be an automaton for L. We make the same automaton with the same final states and Büchi acceptance condition.
2. NO. Assume that yes and $\mathcal{A}$ is an automaton for this language. Let $\mathcal{A}$ has n states and let $p > n$ be some prime number. The word $(b^p a)^\omega$ belongs to L, so $\mathcal{A}$ has an accepting run on it. Note that in every block b^p some two states are the same. We pump the part b^k between them p times, so

that the block has now the length b^{p+kp}, which is not prime. The now word is accepted by $\mathcal{A}$, because it has an accepting run (which came out from the pumping of an old run), but no block has prime length.

3. YES. It suffices to count length of the block b^k modulo 2 and go into an accepting state on a which finishes such a block.

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

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

■

Problem 2. *Show that language of words "there exists a letter b" cannot be accepted by a nondeterministic automaton with Büchi acceptance condition, where all the states are accepting (but possibly transitions over some letters missing in some states).*

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

Problem 3. *Show that language "finitely many occurrences of letter a" cannot be accepted by a deterministic automaton with Büchi acceptance condition.*

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

Problem 4. *Show that every language accepted by some nondeterministic automaton with Muller acceptance condition is also accepted by some nondeterministic automaton with Büchi acceptance condition.*

Solution. We have an automaton $\mathcal{A}$ with Muller condition, we will be trying to make an automaton $\mathcal{A}'$ with Büchi condition such that $L(\mathcal{A}') = L(\mathcal{A})$. For every $S \in \mathcal{F}$ we will do a separate gadget in automaton $\mathcal{A}'$ such that acceptance in this gadget is if and only if $\inf(\rho) = S$. At the beginning we just make a copy of $\mathcal{A}$, but such that no states are accepting. Beside that for every $S \in \mathcal{F}$ we add a gadget $\mathcal{A}_S$. The idea is that automaton $\mathcal{A}'$ will jump into the gadget $\mathcal{A}_S$ if it wants to choose that $\inf(\rho) = S$ and now is exactly this moment from which on only states from S will occur. Let $S = \{q_1, \ldots, q_k\}$.
Observe now that if $\inf(\rho) = S$ and there are no states outside of S in ρ then states occurring in ρ have also an infinite subsequence of the form $(q_1 q_2 \cdots q_k)^\omega$. Thus we can just investigate whether there exist such a subsequence. Gadget A_S will be the following. It contains $|S|$ copies of $\mathcal{A}$: $\mathcal{A}_{S,1}, \ldots, \mathcal{A}_{S,|S|}$. The copy $\mathcal{A}_{S,i}$ has only one accepting state: q_i. In the copy $\mathcal{A}_{S,i}$ transitions are like in $\mathcal{A}$ with the only exception that from state q_i we go to the next copy: $\mathcal{A}_{S,(i+1) \mod |S|}$. Now we can easily observe that ρ visits infinitely many times accepting state iff it infinitely many times changes a copy. Therefore it has a subsequence of states of the form $(q_1 q_2 \cdots q_k)^\omega$, so indeed $\inf(\rho) = S$. ∎

Problem 5. *Assume that we have changed the acceptance condition into such which investigates which sets of transitions are visited infinitely often. Does it affect the expressivity of automata? How it is for Büchi acceptance condition? And how for Muller acceptance condition?*

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

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

states.

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

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

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

■

Problem 6. *Show that nonemptiness is decidable for automata with Muller acceptance condition.*

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

2
Infinite duration games

3
Distance automata

4
Tree automata and MSO

5
Treewidth

6
Weighted automata over a field

7
Polynomial grammars

Problem 7. *Show that the following problem is decidable: given a polynomial grammar and a finite set $X \subseteq \mathbb{Q}$, decide if the language generated by the grammar is equal to X.*

Solution. ∎

8
Parsing in matrix multiplication time

Problem 8. *Show that the operation $M \circ N$ on parse matrices is not associative.*

Solution. ■

Bibliography

[1] 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.

[2] Michael Benedikt, Timothy Duff, Aditya Sharad, and James Worrell 0001. Polynomial automata - Zeroness and applications. *LICS*, pages 1–12, 2017.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[17] 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.

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

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

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