Deterministic Finite Automata

Bakhadyr Khoussainov

Computer Science Department, The University of Auckland, New Zealand
bmk@cs.auckland.ac.nz

In this lecture we introduce deterministic finite automata, one of the foundational concepts in computing sciences. Finite automata are the simplest mathematical model of computers. Informally, a finite automaton is a system that consists of states and transitions. Each state represents a finite amount of information gathered from the start of the system to the present moment. Transitions represent state changes described by the system rules. Practical applications of finite automata include digital circuits, language design and implementations, image processing, modeling and building reliable software, and theoretical computing.

1 Strings and languages

Recall that an alphabet is a finite set Σ of symbols. In most cases, our alphabets contain symbols $a, b, c, d, \ldots$. If Σ contains k letters then we say that Σ is a k-letter alphabet. A 1-letter alphabet is called a **unary alphabet**, and a 2-letter alphabet is a **binary alphabet**. We will often deal with the binary alphabet $\{a, b\}$.

Let Σ be an alphabet. The elements of the alphabet are called **input symbols**. A finite sequence of symbols from Σ is called a **string** of the alphabet. Sometimes strings are also called **words**. Thus, each string is of the form $\sigma_1\sigma_2\ldots\sigma_n$, where each σ_i is a symbol of the alphabet. The **length** of a given string is the number of symbols it has. Thus, the length of strings *aaab*, *bababa*, and *bb* are 4, 6, and 2, respectively. There is a special string whose length is 0 called the **empty string**. We denote this string by λ. Using mathematical induction, it is easy to show that the number of strings of length n of a k-letter alphabet Σ is equal to k^n.

Let Σ be an alphabet. The set of all strings over the alphabet Σ is denoted by $\Sigma^\star$. Thus,

$$\Sigma^\star = \{\sigma_1\sigma_2\ldots\sigma_m \mid \sigma_1, \sigma_2, \ldots, \sigma_m \in \Sigma,\ m \in N\}.$$

Note that when $m = 0$, we have the empty string λ and therefore $\lambda \in \Sigma^\star$. We denote the strings of the alphabet by the letters $u, v, w, \ldots$.

Let u and v be strings. Then the **concatenation** of these two strings, denoted by $u \cdot v$, is obtained by writing down u followed by v. For example, $aab \cdot bba$ produces the string *aabbba*. It is clear that the concatenation operation on words satisfies the equality $u \cdot (v \cdot w) = (u \cdot v) \cdot w$ for all words u, v, w. Note that $\lambda \cdot u = u \cdot \lambda$ for any string u. Often, instead of writing $u \cdot v$ we may omit the dot $\cdot$ sign and write uv instead.

Let u be a string. The notation u^n represents the string obtained by writing down u exactly n times. Thus, u^n is the string obtained by concatenating u to itself n times. For example, $(ab)^3 = ababab$. When $n = 0$ then u^n is the empty string λ for any string u.

Let u and w be strings. We say that w is a **substring** of u if w occurs in u. More formally, w is a substring of u if $u = u_1 w u_2$ for some strings u_1 and u_2. For example, ab is a substring of $aaabbbbaa$ while aba is not. Clearly, every string u is a substring of itself.

We say that w is a **prefix** of u if u can be written as wu_1. For example, the prefixes of the string $abbab$ are λ, a, ab, abb, $abba$, and $abbab$.

Now we give an important definition:

Definition 1. *A* **language** *is a subset of $\Sigma^\star$, where Σ is an alphabet.*

When we use the term language we always assume, implicitly or explicitly, that we are given an alphabet. Here are some examples of languages: $\emptyset$, $\Sigma^\star$, $\{\lambda, a, aa, aaa, aaaa, aaaaa, \ldots\}$, $\{ab, ba\}$, $\{w \mid aba \text{ is a substring of } w\}$. We denote languages by capital letters U, V, W, L, etc. We now define several operations on languages.

Given two languages U and V, their **union** is $U \cup V$; their **intersection** is $U \cap V$; and the **complement** of U is $\Sigma^\star \setminus U$. Below we discuss two additional operations on languages.

The concatenation operation. Let U and W be languages. The **concatenation** of U and W is the language:

$$\{u \cdot w \mid u \in U, w \in W\}.$$

We denote this language by $U \cdot W$. For example, if $U = \{ab, ba\}$ and $W = \{aa, bb\}$ then $U \cdot W = \{abaa, abbb, baaa, babb\}$. Here is another example. Let $U = \{a\}$. Then for any language L, $U \cdot L = \{au \mid u \in L\}$.

The star operation. This operation is also often called the Kleene's star operator. Let U be the language. Consider the following sequence of languages:

$$U^0, U^1, U^2, U^3, U^4, \ldots$$

where U^n with $n \in N$ is defined recursively as follows: $U^0 = \{\lambda\}$, $U^1 = U$, $U^2 = U \cdot U$, and $U^n = U^{n-1} \cdot U$. We can take the union of all these languages and denote the resulting language by $U^\star$. Thus,

$$U^\star = U^0 \cup U^1 \cup U^2 \cup U^3 \cup \ldots.$$

The language $U^\star$ is called **the star of the language** U. More informally, $U^\star$ is the set of strings obtained by finite number of concatenations applied to strings from the language U. For example, if $U = \{a\}$ then $U^\star = \{\lambda, a, aa, aaa, aaaa, \ldots\}$. Note that the star of every language contains λ. The language $U^\star$ is always an infinite language apart from the cases $U = \emptyset$ and $U = \{\lambda\}$.

2 Deterministic finite automata

Let Σ be an alphabet and U be a language of Σ. A typical problem that we want to solve is the following. Design an algorithm that, given a string w, determines whether or not w is in U.

Here is an example. Let U be the language consisting of all strings w over the alphabet $\{a, b\}$ such that w contains the substring *aba*. We want to design an algorithm that, given a string w, determines whether or not w contains a substring *aba*. Here is the *Find-aba*(w) algorithm. Let w be the string $w = \sigma_1 \dots \sigma_n$.

1. Initialize variables i and *state* of integer type as $i = 1$ and $state = 0$
2. *While* $i \leq n$ do
   (a) If $state = 0$ and $\sigma_i = b$ then set $state = 0$.
   (b) If $state = 0$ and $\sigma_i = a$ then set $state = 1$.
   (c) If $state = 1$ and $\sigma_i = a$ then set $state = 1$.
   (d) If $state = 1$ and $\sigma_i = b$ then set $state = 2$.
   (e) If $state = 2$ and $\sigma_i = a$ then set $state = 3$.
   (f) If $state = 2$ and $\sigma_i = b$ then set $state = 0$.
   (g) If $state = 3$ and $\sigma_i \in \{a, b\}$ then $state = 3$.
   (h) Increment i by one.
3. If $state = 3$ then output *accept*. Otherwise output *reject*.

The important factor in this algorithm is the values of the variable *state*. These values are are 0, 1, 2, and 3. Call these the states of the program. The program makes its transitions from one state to another depending on the input σ_i it reads. Thus, we have a transition function associated with the program. The table of the transition function is presented in Table 1.

	a	b
0	1	0
1	1	2
2	3	0
3	3	3

Table 1. Transition function for the *Find-aba*(w)

The state 0 is the *initial state* of the algorithm. The state 3 is the *accepting state* as w contains *aba* when $state = 3$. Finally, if the algorithm outputs *reject* then the *state* variable has value 0, 1, or 2. Thus, we have given a finite state analysis of the *Find-aba*(w) algorithm. Now we abstract from this example and give the following definition:

Definition 2. *A* **deterministic finite automaton** *is a* 5*-tuple* (S, q_0, T, F, Σ)*, where*

1. S *is the set of* **states**.
2. q_0 *is the* **initial state** *and* $q_0 \in S$.
3. T *is the* **transition function** $T : S \times \Sigma \to S$.
4. F *is a subset of* S *called the set of* **accepting states**.
5. Σ *is an alphabet.*

We abbreviate deterministic finite automaton as DFA. We often write (S, q_0, TF) *instead of* (S, q_0, T, F, Σ). *We use letters* $\mathcal{A}$, $\mathcal{B}$, ... *to denote finite automata.*

Thus, the state analysis of the $Find\text{-}aba(w)$ algorithm above gives us the automaton $(S, 0, T, F)$, where: $S = \{0, 1, 2, 3\}$, 0 is the initial state, T is presented in Table 1 above, and $F = \{3\}$.

As for transition functions from the previous lecture, we can visualize finite automata as labeled graphs. Let (S, q_0, T, F) be a DFA. The states of the DFA are represented as vertices of the graph. We put an edge from state s to state q if there is an input signal σ such that $T(s, \sigma) = q$. The edges labeled with σ are called **σ-transitions**. The initial state is presented as a vertex with an ingoing arrow without a source. The final states are vertices that are doubly circled. We call this visual presentation a **transition diagram** of the automaton. For example, the transition diagram of the automaton for the $Find\text{-}aba(w)$ algorithm is given in Figure 1.

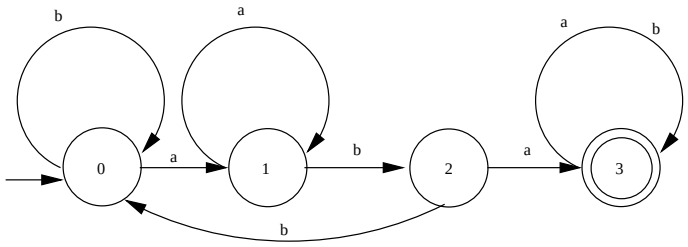

Fig. 1. Transition diagram for a DFA for the $Find\text{-}aba(w)$ algorithm

Another example of a DFA is presented in Figure 2.

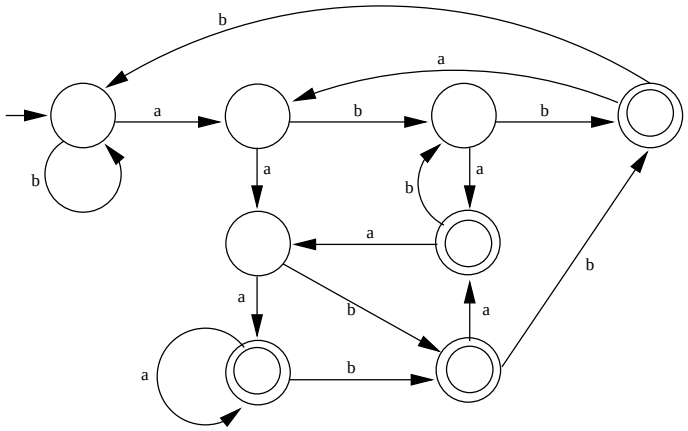

Fig. 2. An example of a transition diagram

Let $\mathcal{M} = (S, q_0, T, F, \Sigma)$ be a DFA. Take any string $w = \sigma_1 \dots \sigma_n$ of the alphabet Σ. The **run** of the automaton on this string is the sequence of states

$$s_1, s_2, \dots, s_n, s_{n+1}$$

such that $T(s_i, \sigma_i) = s_{i+1}$ for all $i = 1, \dots, n$ and that s_1 is the initial state. Thus, the run of $\mathcal{M}$ on string $w = \sigma_1 \dots \sigma_n$ can be thought of as the execution the following algorithm $Run(\mathcal{M}, w)$:

1. Initialize $s = q_0$ and $i = 0$.
2. Print s.
3. *While* $i \leq n$ do
   (a) Set $\sigma = \sigma_i$.
   (b) Set $s = T(s, \sigma)$.
   (c) Print s.
   (d) Increment i

This algorithm outputs the states of the DFA $\mathcal{M}$ as it reads through the string w. First, the algorithm prints the initial state. It then reads the first input σ_1, makes a transition from the initial state to state $T(q_0, \sigma_1)$ and outputs $T(q_0, \sigma_1)$, reads the next symbol σ_2, makes a transition, and so on. Note that every run of $\mathcal{M}$ is a path starting from the initial state q_0. One can also visualize the run as a path labeled by the string w and starting with q_0. For example, the path $0, 0, 0, 1, 1, 2, 3, 3$ is the run of the automaton in Figure 1 on the string *bbaabab*. On the same input string the automaton on Figure 1 produces the run 0, 0, 0, 1, 4, 7, 5, 2.

Definition 3. *Let $\mathcal{M} = (S, q_0, T, F)$ be a DFA. We say that $\mathcal{M}$* **accepts** *the string $w = \sigma_1 \dots \sigma_n$ if the run of M*

$$s_1, \dots, s_n, s_{n+1}$$

on w is such that the last state s_{n+1} is in F. We call such a run an **accepting run**.

Note that the run of $\mathcal{M}$ on w is always unique. Therefore, the run is either accepting or rejecting but not both. For example, the automaton in Figure 1 accepts those strings that contain *aba* as a substring and rejects all other strings. The automaton in Figure 2 accepts all the strings that have a in the third position from the last and rejects all other strings. Thus, we now define the following important concept:

Definition 4. *Let $\mathcal{M} = (S, q_0, T, F, \Sigma)$ be a DFA. The* **language accepted by** *$\mathcal{M}$, denoted by $L(\mathcal{M})$, is the following language:*

$$\{w \mid \text{ the automaton } \mathcal{M} \text{ accepts } w\}.$$

A language $L \subseteq \Sigma^\star$ is **DFA recognizable** *if there exists a DFA $\mathcal{M}$ such that $L = L(\mathcal{M})$.*

Now we give several simple examples:

1. Consider a DFA with exactly one state. If the state is the accepting state then the automaton accepts the language $\Sigma^\star$. Otherwise, if the state is not accepting, the automaton accepts the empty language $\emptyset$.

2. Consider the language $\{w\}$ consisting of one word $w = \sigma_1 \dots \sigma_n$. This language is DFA recognizable. The automaton recognizing this language just remembers the entire string. Indeed, the following DFA $(S, 0, T, F)$ recognizes the language:
   (a) $S = \{0, 1, 2, 3, 4, \dots, n+1\}$.
   (b) 0 is the initial state.
   (c) For all $i \leq n-1$, $T(i, \sigma_{i+1}) = i+1$. In all other cases $T(s, \sigma) = n+1$.
   (d) The accepting state is n.

   The transition diagram of this automaton when $w = abbab$ is in Figure 3.

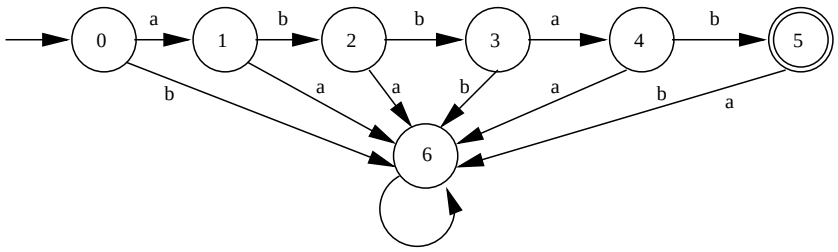

Fig. 3. A DFA recognizing the language $\{abbab\}$.

3. The language $L = \{aw \mid w \in \{a, b\}^\star\}$ is DFA recognizable. The transition diagram of a DFA that recognizes the language is in Figure 4. Formally, here is the automaton $\mathcal{M} = (S, q_0, T, F)$ recognizing L:
   (a) $S = \{0, 1, 2\}$
   (b) The initial state is 0.
   (c) The transition table is defined as follows: $T(s, \sigma) = 1$ for $s = 0$ and $\sigma = a$, $T(s, \sigma) = 1$ for $s = 1$ and $\sigma \in \{a, b\}$, and $T(s, \sigma) = 2$ in all other cases.
   (d) 1 is the accepting state.

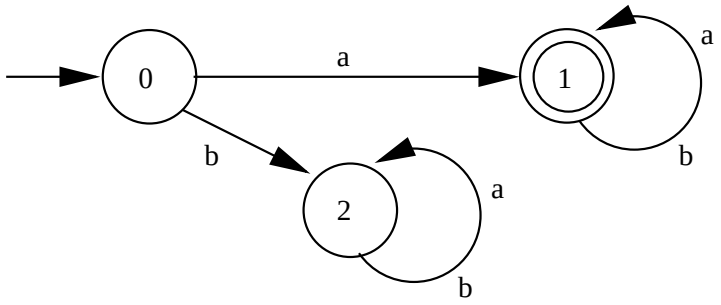

Fig. 4. A DFA recognizing the language $\{aw \mid w \in \{a, b\}^\star\}$.

3 Constructing finite automata

Suppose that we are given a language L and are asked to design a DFA that recognizes the language L or argue that such a DFA does not exist. A very helpful idea for solving this type of problem is to pretend that you are the machine and see how you would go about solving the problem. We explain this in the following two examples.

Assume that we want to construct a DFA that recognizes the language

$$L = \{w \mid w \in \{ab\}^\star \text{ and } w \text{ contains an odd number of } a\text{'s and an even number of } b\text{'s}\}.$$

Let's pretend that we are the machine. We start reading from left to right the input string w, and see symbols a or b. Do we need to remember the *entire* string we have seen so far in order to tell whether we have passed through an odd number of a's and an even number of b's? Our answer is "no" because of the following observation. We keep two coins both having two colored sides *blue* and *red*. We associate the first coin with the symbol a and the second with the symbol b. When we start, both coins are on their *blue* sides. Reading w, when we see a we flip the first coin and when we see b we flip the second coin. Thus, if the first coin is on its *blue* side than we have seen an even number of a's, and if the coin is on its *red* side then we have seen an odd number of a's. The same holds true for the symbol b. Therefore, at at any given time, our state is determined by the current colors of the coins. There are four possible states only:

1. The first coin is *blue* and the coin is *blue*. Code this state as 0.
2. The first coin is *red* and the coin is *blue*. Code this state as 1.
3. The first coin is *blue* and the coin is *red*. Code this state as 2.
4. The first coin is *red* and the coin is *red*. Code this state as 3.

When we finish reading the whole input string w we accept the string if the first coin is on its *red* side and the second coin is in its *blue* side. In all other cases we reject the string. Thus, all this reasoning helps us to build a DFA that recognizes L. The Figure 5 is the transition diagram of the automaton we have built.

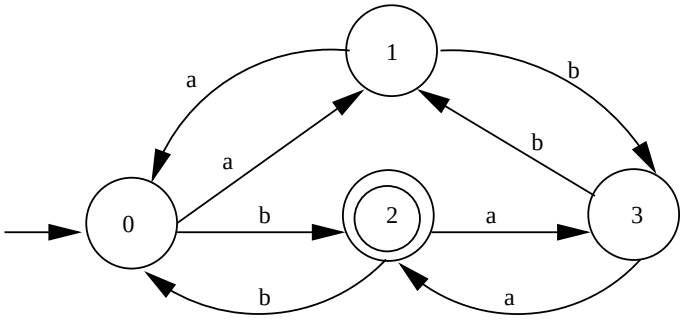

Fig. 5. A DFA recognizing the language $\{w \mid w$ has an odd number of a's and an even number of b's$\}$.

Consider the following language $L = \{a^n b^n \mid n \in N\}$. Let's pretend to be a machine that recognizes L. Given an input string w, we have the following constraints as dictated by the definition of run of DFA. We must read the string w from left to right. We are not allowed to come back to any position in w which we have passed. Assume w starts with b. We then reject w. Assume that w starts with an a. We read a and remember that we have read one a. Informally, this tells us that we have to create a state, let's denote it by s_1, that remembers that w starts with an a. If b is the next and the last symbol then we accept w. Otherwise, we reject w. If the second symbol is a, then we remember that we have read two a's. Informally, this tells us that we have to create a state, let's denote it by s_2, that remembers that w starts with aa. We must keep s_1 and s_2 separate. Indeed, if we declare $s_1 = s_2$ then we have to accept the string aab which is clearly not desirable. Lets continue this on. Assume that w is of the form $a^n u$, where $n \geq 1$, and we have created states s_1, s_2, ..., s_n. State s_i detects that the input string starts with a^i. Consider the first symbol of u. If it is a b then we can use our states s_n, s_{n-1}, ..., s_1 and control the next n symbols of w. If all the symbols of u are b and there are exactly n of them, we accept w and otherwise reject. However, if the first symbol of u is a, then we need to remember that we have read exactly $n+1$ symbols of a. This tells us that we have to create a state, let's denote it by s_{n+1}, that remembers that w starts with a^{n+1}. We must keep s_{n+1} separate from all the states s_1, s_2, ..., s_n we have built so far. Thus, we see that in order to recognize L, our informal analysis tells us that we need to have infinitely many states. This intuitive reasoning based on pretending to be a machine for recognizing L gives us a reason to believe that no DFA recognizes L. Indeed, our intuition here is correct as will be shown in Lecture 13.

Constructing automata for the union operation. Here is a design problem we want to solve. Assume we are given two DFA $\mathcal{M}_1 = (S_1, q_0^{(1)}, T_1, F_1)$ and $\mathcal{M}_2 = (S_2, q_0^{(2)}, T_2, F_2)$. These two DFA recognize languages $L_1 = L(\mathcal{M}_1)$ and $L_2 = L(\mathcal{M}_2)$. We want to design a DFA $\mathcal{M} = (S, q_0, T, F)$ that recognizes the union $L_1 \cup L_2$.

We use the method of pretending to be a machine that recognizes $L_1 \cup L_2$. The first attempt for building the desired $\mathcal{M}$ is this. For an input string w, we simulate the first machine $\mathcal{M}_1$. If $\mathcal{M}_1$ accepts w then $w \in L_1 \cup L_2$. If $\mathcal{M}_1$ rejects w then we run the second machine $\mathcal{M}_2$ on w. If $\mathcal{M}_2$ accepts w then $w \in L_1 \cup L_2$. Otherwise, we reject w. The problem with this idea is that we can *not* read the input w twice. If we are *not* allowed to run $\mathcal{M}_1$ and $\mathcal{M}_2$ on w turn by turn then why don't we run $\mathcal{M}_1$ and $\mathcal{M}_2$ on w *in parallel*?

We take the following approach. On input string w, we run $\mathcal{M}_1$ and $\mathcal{M}_2$ simultaneously as follows. Initially, we remember the initial states $q_0^{(1)}$ and $q_0^{(2)}$. We read the first input symbol σ_1 of w, make simultaneous transitions on $\mathcal{M}_1$ and $\mathcal{M}_2$, and remember the states $s_1^{(1)} = T_1(q_0^{(1)}, \sigma_1)$ and $s_1^{(2)} = T_2(q_0^{(2)}, \sigma_1)$. We then repeat this process by making simultaneous transitions from $s_1^{(1)}$ to $s_2^{(1)} = T_1(s_1^{(1)}, \sigma_2)$ on $\mathcal{M}_1$, and from $s_1^{(2)}$ to $s_2^{(2)} = T_2(s_1^{(2)}, \sigma_2)$ on $\mathcal{M}_2$, where σ_2 is the second letter of w. We continue this on. Once we finish the simultaneous runs of $\mathcal{M}_1$ and $\mathcal{M}_2$ on w, we look at the resulting end states of these two runs. If one of these states is accepting then we accept w and otherwise we reject w. Thus, at any given stage of the two runs we remember a pair (p, q), where $p \in S_1$ and $q \in S_2$. Our transition on this pair on input σ is simply the simultaneous transitions from p to $T_1(p, \sigma)$ of $\mathcal{M}_1$ and from q to $T_2(q, \sigma)$ of $\mathcal{M}_2$. We can be in at most $|S_1| \cdot |S_2|$ states.

Now we formally define the DFA $\mathcal{M} = (S, q_0, T, F)$ that recognizes the language $L_1 \cup L_2$:

1. The set S of states is $S_1 \times S_2$.
2. The initial state is the pair $(q_0^{(1)}, q_0^{(2)})$.
3. The transition function T is the product of the transition functions T_1 and T_2, that is:

$$T((p, q), \sigma) = (T_1(p, \sigma), T_2(q, \sigma)),$$

   where $p \in S_1$, $q \in S_2$, and $\sigma \in \Sigma$.
4. The set F of final states consists of all pairs (p, q) such that either $p \in F_1$ *or* $q \in F_2$.

The proof that $\mathcal{M}$ is a DFA recognizing the union $L_1 \cup L_2$ is informally explained as follows. We first show that, nn the one hand, if w is in $L_1 \cup L_2$ then the automaton $\mathcal{M}$ accepts w. Indeed, if $w \in L_1 \cup L_2$ then either $\mathcal{M}_1$ accepts w or $\mathcal{M}_2$ accepts w. In either case, since $\mathcal{M}$ simulates both $\mathcal{M}_1$ and $\mathcal{M}_2$, the string w must be accepted by $\mathcal{M}$. On the other hand, if w is accepted by $\mathcal{M}$ then the run of $\mathcal{M}$ on w can be split into two runs: one is the run of $\mathcal{M}_1$ on w and the other is the run of $\mathcal{M}_2$ on w. Since $\mathcal{M}$ accepts w, it must be the case that one of the runs is accepting.

The notation for the automaton built is this: $\mathcal{M}_1 \oplus \mathcal{M}_2$.

Constructing automata for the intersection operation. Assume we are given two DFA $\mathcal{M}_1 = (S_1, q_0^{(1)}, T_1, F_1)$ and $\mathcal{M}_2 = (S_2, q_0^{(2)}, T_2, F_2)$. These two DFA recognize languages $L_1 = L(\mathcal{M}_1)$ and $L_2 = L(\mathcal{M}_2)$. We want to design a DFA $\mathcal{M} = (S, q_0, T, F)$ that recognizes the intersection $L_1 \cap L_2$.

We use the idea of constructing the DFA for the union of languages. Given an input w, we run $\mathcal{M}_1$ and $\mathcal{M}_2$ on w simultaneously as we explained for the union operation. Once we finish the runs of $\mathcal{M}_1$ and of $\mathcal{M}_2$ on w, we look at the resulting end states of these two runs. If both end states are accepting then we accept w and otherwise we reject w. Formally, we define the DFA $\mathcal{M} = (S, q_0, T, F)$ that recognizes the language $L_1 \cap L_2$ as follows:

1. The set S of states is $S_1 \times S_2$.
2. The initial state is the pair $(q_0^{(1)}, q_0^{(2)})$.
3. The transition function T is the product of the transition functions T_1 and T_2, that is:

$$T((p, q), \sigma) = (T_1(p, \sigma), T_2(q, \sigma)),$$

   where $p \in S_1$, $q \in S_2$, and $\sigma \in \Sigma$.
4. The set F of final states consists of all pairs (p, q) such that $p \in F_1$ *and* $q \in F_2$.

It is not difficult to prove that $\mathcal{M}$ constructed recognizes $L_1 \cap L_2$.

The notation for the automaton built is this: $\mathcal{M}_1 \otimes \mathcal{M}_2$.

Constructing automata for the complementation operation. Given a DFA $\mathcal{M} = (S, q_0, T, F)$ we want to design a DFA that recognizes the complement of $L(\mathcal{M})$. This is indeed a very simple problem as for this we just keep the same states, the initial state, and the transition

function T. The only change we make is that we swap the accepting states with the non-accepting states. Thus, the automaton that recognizes the language $\Sigma^\star \setminus L(\mathcal{M})$ is $\mathcal{M}^{(c)} = (S, q_0, T, S \setminus F)$. The most important part in this construction is the following observation. On *every* input w, the DFA $\mathcal{M}$ (and hence $\mathcal{M}^{(c)}$) has a unique run. Therefore, $\mathcal{M}$ accepts w if and only if $\mathcal{M}^{(c)}$ rejects w.

4 Minimization algorithm

Suppose that we are given a DFA $\mathcal{A}$. The number of states of $\mathcal{A}$ measures the complexity of $\mathcal{A}$. If $\mathcal{A}$ has many states then this could be due to two reasons. One is that the language recognized by $\mathcal{A}$ can be complex. Therefore there could be no way to reduce the number of states of $\mathcal{A}$ without changing the language recognized. The second is that $\mathcal{A}$ could be implemented inefficiently and therefore can have many redundant states. In this case we would like to remove those redundant states and make the automaton $\mathcal{A}$ smaller. To make all these things more precise we give the following definition:

Definition 5. *We say that a DFA $\mathcal{A}$ is* **equivalent to a DFA** *$\mathcal{B}$ if $L(\mathcal{A}) = L(\mathcal{B})$.*

Thus, if $\mathcal{A}$ and $\mathcal{B}$ are equivalent DFA, then for any string w, $\mathcal{A}$ accepts w if and only if $\mathcal{B}$ accepts w. Consider the two automata, $\mathcal{A}_1$ and $\mathcal{A}_2$, in Figure 6: $\mathcal{A}_1$ is the automaton with three states and $\mathcal{A}_2$ with two states. Both $\mathcal{A}_1$ and $\mathcal{A}_2$ are equivalent and recognize the language

$$L = \{w \mid w \in \{a, b\}^\star \text{ and each } b \text{ in } w \text{ is followed by an } a \text{ later on after the } b\}.$$

There does not exist a one state automaton recognizing L. Therefore the two state automaton $\mathcal{A}_2$ has the fewest possible states recognizing L.

Fig. 6. Two equivalent automata

Our goal is now to provide a method that reduces the redundant states from a given automaton. We give the following definition.

Definition 6. *We say that a DFA $\mathcal{A}$ is* **minimal** *if no DFA $\mathcal{B}$ equivalent to $\mathcal{A}$ has fewer states than $\mathcal{A}$ itself.*

For example, the two state automaton $\mathcal{A}_2$ in Figure 6 is minimal. Note that, by dfeinition, for every DFA $\mathcal{A}$ there exists a minimal automaton equivalent to $\mathcal{A}$.

Let $\mathcal{A} = (S, q_0, T, F)$ be a DFA. Take a state $s \in S$ and a string w. Starting at state s, run the automaton on string w. We denote the last state of this run by $T(s, w)$. Formally, the state $T(s, w)$ is defined inductively as follows: $T(s, \lambda) = s$, $T(s, w\sigma) = T(T(s, w), \sigma)$, where $w \in \Sigma^\star$. We now give the following important definition that is used in constructing minimal automata.

Definition 7. *Let p and q be two states of the DFA $\mathcal{A}$. We say that p and q are* **indistinguishable** *if for all strings $w \in \Sigma^\star$, $T(p, w) \in F$ if and only if $T(q, w) \in F$. Otherwise, we say that the states p and q are* **distinguishable**

For example, consider the automaton $\mathcal{A}_1$ with three states in Figure 6. The two final states are indistinguishable.

If states p and q are distinguishable then this is equivalent to saying the following. There exists a string w such that either $T(p, w) \in F$ and $T(q, w) \notin F$ or $T(p, w) \notin F$ and $T(q, w) \in F$. We call any string w that satisfies one of these two properties a **witness** that distinguishes p and q. As an example, any accepting state p is distinguishable from a non-accepting state q. The string λ is a witness that distinguishes p and q because $T(p, \lambda) \in F$ and $T(q, \lambda) \notin F$.

Here is now the problem we would like to solve. Design an algorithm, that given a DFA $\mathcal{A} = (S, q_0, T, F, \Sigma)$, produces a minimal DFA equivalent to $\mathcal{A}$. In order to solve this problem, let's analyze the DFA $\mathcal{A}$ presented in Figure 7.

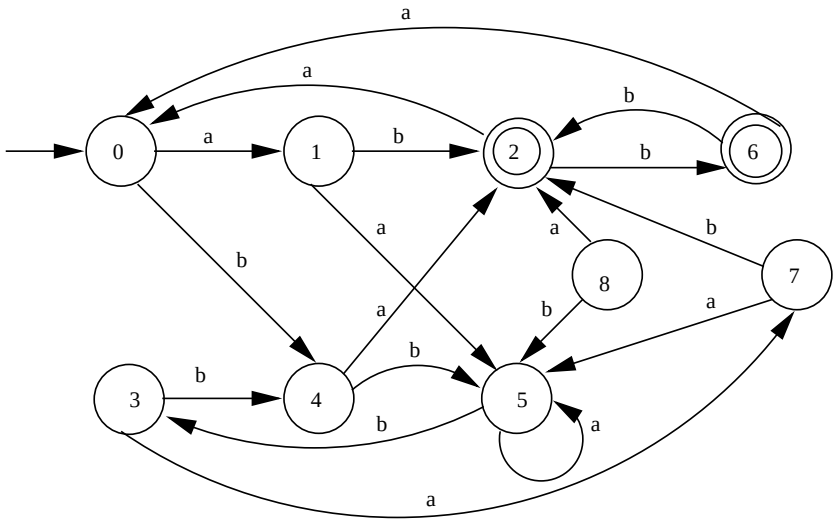

Fig. 7. A DFA for minimiaztion

The automaton $\mathcal{A}$ has 9 states. We want to reduce the number of states of this automaton and build a smaller automaton equivalent to it. First of all, we remove all the states that are

not reachable from the initial state. There is only one such state which is 8. So we take the state 8 out and the resulting automaton is still equivalent to the given one. From now on we assume that state 8 does not exist. Secondly, the idea is this. For each state i we want to collapse all states j indistinguishable from i into one state denoted by $[i]$. Thus, the state $[i]$ is the set $\{j \mid j$ is indistinguishable from $i\}$. Clearly, every state i belongs to $[i]$. Therefore $[i] \neq \emptyset$. This idea is implemented as follows. We form a 8×8 table whose rows and columns are named by the states. The cell (i, j) in the table represents the cell on row i and column j. Initially we have the Table 2.

	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								
5								
6								
7								

Table 2.

Our goal is to put X's into those cells (i, j) such that states i and j are distinguishable. We know that if one of the states i, j is accepting and the other is not then i and j are distinguishable. Thus, we put X into all cells (i, j) such that either i or j is in F but not both. This produces Table 3.

	0	1	2	3	4	5	6	7
0			X				X	
1			X				X	
2	X	X		X	X	X		X
3			X				X	
4			X				X	
5			X				X	
6	X	X		X	X	X		X
7			X				X	

Table 3.

We process Table 3 as follows. Take any cell (p, q) that is not marked with an X. Put an X into cell (p, q) if there is a symbol $\sigma \in \{a, b\}$ such that the cell (r, s) is marked with an X, where $r = T(p, \sigma)$ and $s = T(q, \sigma)$. This process produces Table 4.

We then process Table 4 as follows in the same way as processed the previous table. Namely, take any cell (p, q) without an X. Put an X into the cell (p, q) if there is a symbol $\sigma \in \{a, b\}$ such that the cell (r, s) is marked with an X, where $r = T(p, \sigma)$ and $s = T(q, \sigma)$. This process produces Table 5. Finally, we process Table 5 in the same way as above, and produce the final Table 6. We stop when there exists no cell such that the cell is without an X and can still get an X with the method described.

	0	1	2	3	4	5	6	7
0		X	X		X		X	X
1	X		X	X	X	X		
2	X	X		X	X	X		X
3		X	X		X		X	X
4	X	X	X	X		X	X	X
5		X	X		X		X	X
6	X			X	X	X		
7	X		X	X	X	X	X	

Table 4.

	0	1	2	3	4	5	6	7
0		X	X		X	X	X	X
1	X		X	X	X	X		
2	X	X		X	X	X		X
3		X	X		X	X	X	X
4	X	X	X	X		X	X	X
5	X	X	X	X	X		X	X
6	X			X	X	X		X
7	X		X	X	X	X	X	

Table 5.

Consider the final Table 6. Take any state i of the automaton, go through the ith row of the table, collapse all the states j such that the cell (i, j) is not marked with an X into one state denoted by $[i]$. Thus, we have $[0] = \{0, 3\}$, $[1] = \{1, 7\}$, $[2] = \{2, 6\}$, $[4] = \{4\}$, and $[5] = \{5\}$. Thus we have built a new automaton $\mathcal{A}_{new}$ with states [0], [1], [2], [4], and [5] presented in Figure below.

Keeping the above example in mind, we now present the minimization algorithm and prove its correctness. Let $\mathcal{A} = (S, q_0, T, F)$ be a DFA. Here is the $Minimization(\mathcal{A})$ algorithm:

	0	1	2	3	4	5	6	7
0		X	X		X	X	X	X
1	X		X	X	X	X	X	
2	X	X		X	X	X		X
3		X	X		X	X	X	X
4	X	X	X	X		X	X	X
5	X	X	X	X	X		X	X
6	X	X		X	X	X		X
7	X		X	X	X	X	X	

Table 6.

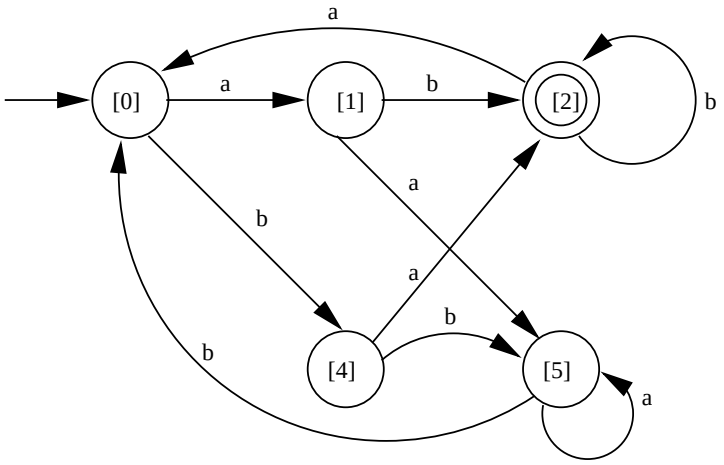

Fig. 8. The DFA $\mathcal{A}_{new}$

1. Remove states that are not reachable from the initial state of $\mathcal{A}$. (Comment: From now on we assume that *all* the states of $\mathcal{A}$ are reachable from the initial state).
2. Let 0, 1, ..., $n-1$ be all the states of $\mathcal{A}$, where 0 is the initial state. Make an $n \times n$ table whose rows and columns are labeled with states.
3. Initialize the table by marking with an X all the cells (i, j) such that one of the states i, j is an accepting state and the other is not.
4. Repeat the following until no cell (i, j) can be marked with an X:
   (a) Mark a cell (p, q) with an X if (p, q) does not have a mark, and there is a $\sigma \in \Sigma$ such that the cell $(T(p, \sigma), T(q, \sigma)$ has a mark.
5. For each state i define $[i]$ to be the set $\{j \mid$ the cell (i, j) is not marked$\}$, where $i = 0, \ldots, n-1$.
6. Construct the following new DFA:
   (a) Declare each $[i]$ to be a state of the new DFA.
   (b) The initial state is [0].
   (c) Put transitions from all $[i]$ to $[T(i, \sigma)]$ and label the transition by σ.

(d) Declare $[i]$ to be an accepting state if i is an accepting state.

The first step of the algorithm produces the automaton that is equivalent to the given one. Note that Step 6 of the algorithm defines states, the initial state, transition, and accepting states of the new automaton. Our goal is to show that the new automaton is correctly defined and is minimal. To do this we now analyze the algorithm. There are n^2 cells in the initial table defined at Step 2, the number of marked cells in the table increases with each iteration in Step 4, and hence the algorithm must terminate.

Fact 1*If a cell (p, q) is marked with an X then p and q are distinguishable.*

If (p, q) is marked at Step 2 of the algorithm then p and q are clearly distinguishable. The witness that distinguishes these states is λ. Consider the loop of the algorithm in Step 4. Assume (p, q) is marked with an X because there exists a $\sigma \in \Sigma$ such that $(T(p, \sigma), T(q, \sigma))$ has already been marked. By inductive assumption the states $p' = T(p, \sigma)$ and $q' = T(q, \sigma)$ are distinguishable. Let w be the string distinguishing p' and q'. Then σw distinguishes p and q. Indeed, in this case we have $T(p, \sigma w) = T(T(p, \sigma), w) = T(p', w)$ and $T(q, \sigma w) = T(T(q, \sigma), w) = T(q', w)$. This explains the fact.

Fact 2.*If a cell (p, q) does not have an X then p and q are indistinguishable.*

Assume a string $w = \sigma_1 \dots \sigma_k$ distinguishes p and q. Consider the following two sequences

$$p_0 = p, p_1 = T(p_0, \sigma_1), p_2 = T(p_1, \sigma_2), \dots, p_n = T(p_{n-1}, \sigma_n)$$

and

$$q_0 = q, q_1 = T(q_0, \sigma_1), q_2 = T(q_1, \sigma_2), \dots, q_n = T(q_{n-1}, \sigma_n).$$

The pair (p_n, q_n) gets an X by Step 2 of the algorithm since w distinguishes p and q. Therefore the pair (p_{n_1}, q_{n-1}) must get a mark. Continue this on we see that the pair (p_0, q_0) also gets a mark X. This contradicts the assumption of the lemma. This explains the fact.

For each state $i \in S$, the algorithm considers the set $[i] = \{j \mid \text{the cell } (i, j) \text{ is not marked}\}$ and declares $[i]$ to be a state of the new DFA. These states satisfy the following properties (the reader, check all these properties!):

Property 1. Every state i of the original automaton is in $[i]$.
Property 2. For any two states $[i]$ and $[j]$ either $[i] = [j]$ or $[i] \cap [j] = \emptyset$.
Property 3. If i and j are indistinguishable then $T(i, \sigma)$ and $T(j, \sigma)$ are also indistinguishable.

In the $Minimization(\mathcal{A})$ algorithm Step 5 constructs the deterministic finite automaton $\mathcal{A}_{new} = (S_{new}, q_{new}, T_{new}, F_{new})$ as follows:

1. The set S_{new} consists of all the states $[i]$.
2. The initial state q_{new} is $[0]$.
3. $T_{neq}([i], \sigma) = [T(i, \sigma)]$ for all $[i] \in S_{new}$ and $\sigma \in \Sigma$.
4. $F_{new} = \{[i] \mid i \in F\}$.

By properties 1 and 2, above the number of states in $\mathcal{A}$ is not smaller than the number of states in S_{new}. Property 3 above tells us that this definition of T_{new} is correct. In other words, T_{new} is a function from $S_{new} \times \Sigma$ to S_{new}.

Fact 3 *The DFA automata $\mathcal{A}_{new}$ and $\mathcal{A}$ are equivalent.*

The proof of this fact follows from how we defined $\mathcal{A}_{new}$ and the previous facts. Indeed, assume that w is accepted by $\mathcal{A}$. This means that $T(0, w) \in F$. By the definition of T_{new} we then must have $T_{new}([0], w) = [T(0, w)]$. Since $T(0, w) \in F$ it must be the case that $[T(0, w)] \in F_{new}$. Thus, w is accepted by $\mathcal{A}_{new}$. Now assume that $\mathcal{A}_{new}$ accepts w. Then $T_{new}([0], w) \in F_{new}$ which means $[T(0, w)] \in F_{new}$. Therefore, $T(0, w) \in F$ and hence $\mathcal{A}$ accepts w.

Now we want to show that the DFA $\mathcal{A}_{new}$ is a minimal DFA equivalent to the given DFA $\mathcal{A}$. For this we need to show that any minimal DFA equivalent to $\mathcal{A}$ does not have fewer states than $\mathcal{A}_{new}$. Thus, let $\mathcal{A}' = (S', q_0', T', F')$ be a minimal automaton equivalent to $\mathcal{A}$. We want to show that $\mathcal{A}_{new}$ and $\mathcal{A}'$ have the same number of states. We reason recursively that defines a function f from states of $\mathcal{A}'$ into the states of $\mathcal{A}_{new}$.

Basis: In this case we map q_0' to $[0]$. Thus we have a partial function $f : S' \rightarrow S_{new}$ such that no two states in S' are mapped into the same state in S_{new}.

Inductive step. Our hypothesis is the following. We have built a partial function $f : S' \rightarrow S_{new}$ such that no two states in S' are mapped into the same state in S_{new}. Moreover, for any $s' \in \mathit{Domain}(f)$ the following property is satisfied. The states s and $f(s)$ are indistinguishable in the sense that for all $w \in \Sigma^\star$, $T(s', w) \in F'$ if and only if $T_{new}(f(s), w) \in F_{new}$. Assume that f is not total.

Take a $p' \in \mathit{Domain}(f)$ and $\sigma \in \Sigma$ such that $T(p', \sigma) = q'$ and $q' \notin \mathit{Domain}(f)$. Such p' and σ must exist as f is not total. Define the value of f on q' as follows. Declare the value of f on q' to be $T_{new}(f(p'), \sigma)$. This value $T_{new}(f(p'), \sigma)$ of f can not be equal to any other value $s \in S_{new}$ from the range of f defined in the previous steps of the construction of f. Otherwise, s and $T_{new}(f(p'), \sigma)$ would be indistinguishable.

Continuing this construction of f, when the construction terminates we have a total bijective function from S' to S_{new}. This shows that $\mathcal{A}_{new}$ has the same number of states as the minimal the minimal automaton $\mathcal{A}'$. Thus, we have proved the algorithm $\mathit{Minimization}(\mathcal{A})$ is correct. In other words, given a DFA $\mathcal{A}$ the algorithm produces a minimal DFA equivalent to $\mathcal{A}$.