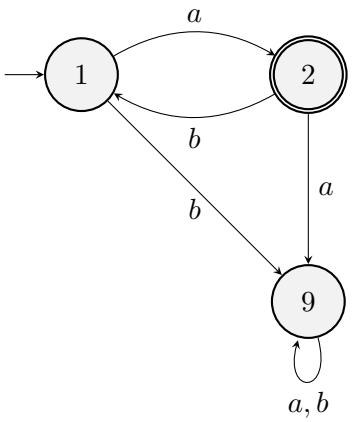
Regular Languages and Automata: Part 2

1 Language Equivalence

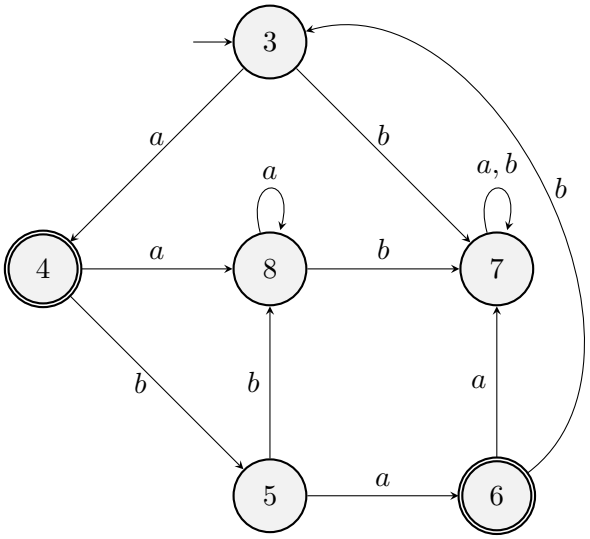
See also the video "Language equivalence" on Canvas.

It's not obvious how to test whether two regexps are language equivalent. But let's see how to test whether two DFAs are language equivalent.

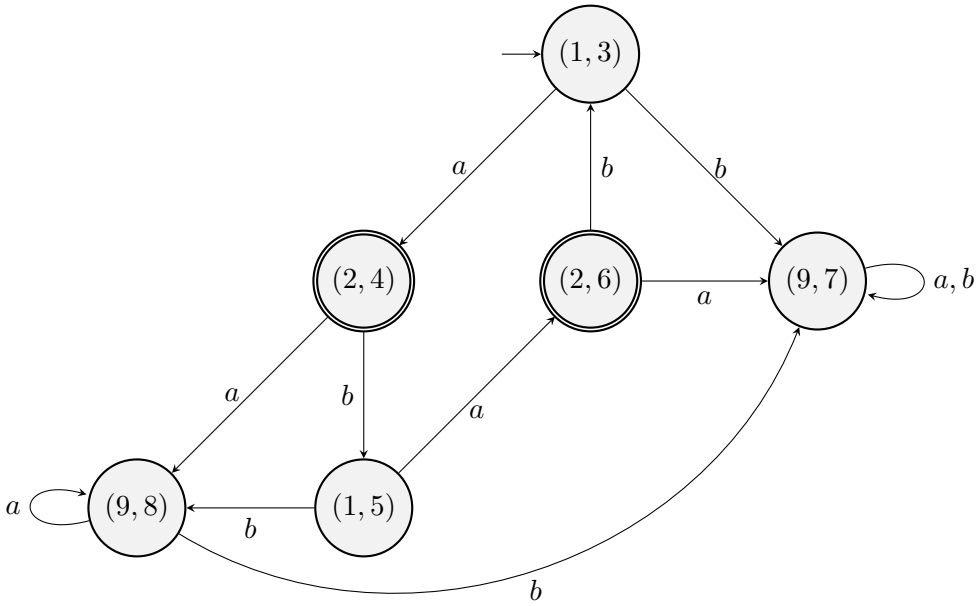
Here's a first suggestion. Start at the initial state of each automaton. If one is accepting and the other rejects, then the automata are not language equivalent (since one accepts ε and the other doesn't). If they both accept or both reject then see what pair of states we transition to by inputting `a`, and what pair by inputting `b`. And we carry on forever. For example, comparing the automata



and



leads to the following:



In this example, we see that each pair of states consists of either two accepting states or two rejecting states. So the two automata are language equivalent.[1]

In the case of two automata that are not language equivalent, then there's some word that one accepts and the other one doesn't. This procedure will eventually find it and tell us.

Either way, the procedure will always stop, since there are only finitely many pairs of states.

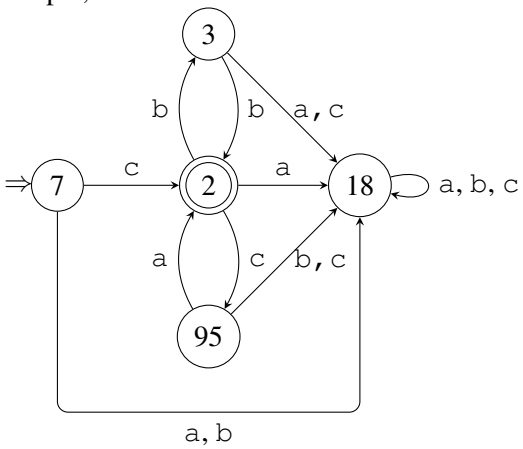
2 Minimal automata

See also the video "Minimal automata" on Canvas.

So far we have not worried about the size of an automaton. But in fact certain automata are *mininal*, which informally means that they can't be made any smaller. To be precise, a DFA is said to be minimal when it has the following two properties.

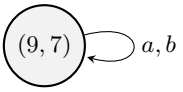
1. Each state x is *reachable*. This means that there's a path from the initial state to x.
2. Any two distinct states x, y are *inequivalent*. This means that there's a word that x accepts (i.e. that leads from x to an accepting state) but y rejects, or vice versa.

For example, look at this DFA.



We show that it's minimal, as follows.

- 7 is reachable via ε, and 2 via `c`, and 3 via `cb`, and 95 via `cc`, and 18 via `ca`.
- 2 is inequivalent to the other states, since 2 accepts ε and the other states reject it.

[1] Note that in the video the transitions  are missing.

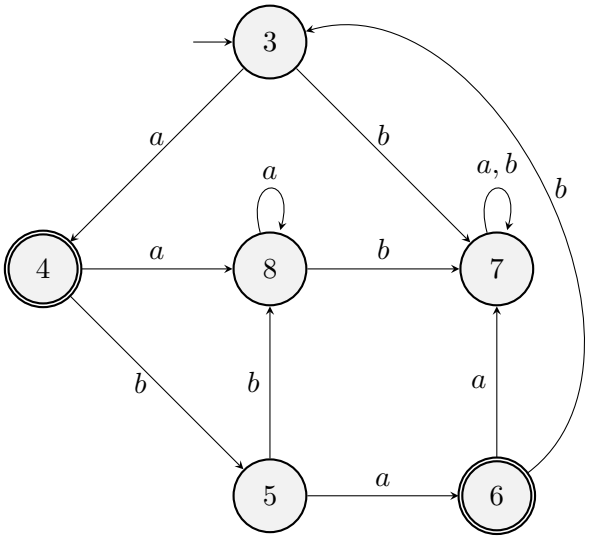
- 3 is inequivalent to the other states, since 3 accepts `b` and the other states reject it.
- 95 is inequivalent to the other states, since 95 accepts `a` and the other states reject it.
- 7 is inequivalent to the other states, since 7 accepts `c` and the other states reject it.

3 Minimizing an automaton

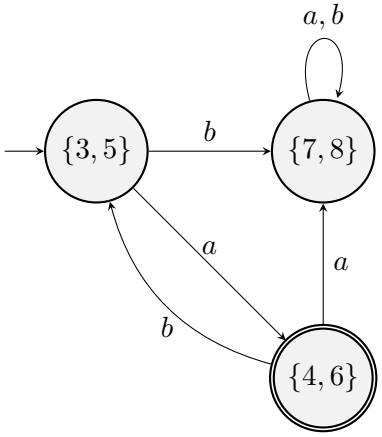
See also the video "Minimizing a DFA" on Canvas.

Now let's look at how to minimize a DFA, i.e. make it as small as possible. There are two steps. Firstly, remove any unreachable states. Secondly, find states that are equivalent, so that we can unify them. To begin with, unify all the accepting states and all the rejecting states. Then try to draw the `a` transitions and `b` transitions—you might find you need to split these blocks. Keep going until you don't need to split anything more.

To illustrate the second step, let's take the automaton



To begin with, we have an accepting state $\{4, 6\}$ and a rejecting state $\{3, 5, 7, 8\}$. The `a` transition from $\{4, 6\}$ takes us to $\{3, 5, 7, 8\}$, and so does the `b` transition. But when we try to draw an `a` transition from $\{3, 5, 7, 8\}$, we can't, because the `a` transition from 3 and 5 goes into $\{4, 6\}$ but the one from 7 and 8 goes into $\{3, 5, 7, 8\}$. So we have to split $\{3, 5, 7, 8\}$ into two parts, viz. $\{3, 5\}$ and $\{7, 8\}$. Now we have three states, and we can complete the diagram without any problem:



As an extra check, you should show this is minimal.

4 Non-regular languages

See also the video "Non-regular languages" on Canvas.

We know that some languages are not regular, because there are only countably many regexps and uncountably many languages. But are there any *useful* non-regular languages? The answer is Yes. Here is an example.

Suppose a word is built up from open brackets, written `a`, and closed brackets, written `b`, and we want to know whether it's well-bracketed. This is a commonly arising problem; for example, when you write a Java program on Eclipse, Eclipse

checks whether your brackets match correctly. This can be done using a stack. When you read a, you push a pebble onto the stack, and when you read b you pop the pebble off the stack. If you reach the end of the word, and the stack is empty, then you know the word is well-bracketed. But if it's not empty, or you read b when the stack is empty, the word is not well-bracketed.

Let's think for a moment about your computer. It has a finite memory, and therefore only finitely many states. It can try to run the above program, allocating part of its memory to represent the stack. But if the word begins with a very large number of a's, the stack will overflow.

Can your computer run a program that works for *all* words? The program should read in a word, letter by letter, and then announce whether or not the word is well-bracketed.

One solution is to use an external stack. That way, even though your computer has only finite memory, there's no limit on the amount of memory that the stack can occupy. But what if you don't have access to external memory? Can you install such a program on your computer? The answer is No. We shall now prove this.

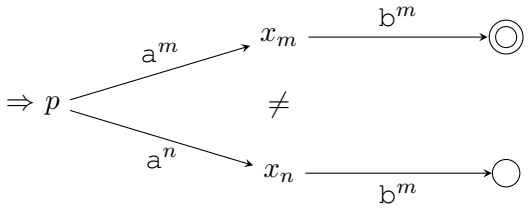
To put this more technically, we shall prove that the set L of well-bracketed words is not regular.

Suppose L is regular; then there's a DFA $(X, p, \delta, \mathsf{Acc})$ that recognizes it. Let's say that

- x_0 is the initial state p
- x_1 is the state that we reach when we start at p and read a
- x_2 is the state that we reach when we start at p and read aa
- x_3 is the state that we reach when we start at p and read aaa
- etc.

In summary, x_n is the state that we reach when we start at p and read $\mathtt{a}^n$. Now we're going to show that these states are all distinct, which implies that there are infinitely many states, a contradiction.

Suppose m and n are natural numbers with $m < n$. We want to show that $x_m \neq x_n$. If we start at x_m and read $\mathtt{b}^m$ we reach an accepting state, because $\mathtt{a}^m\mathtt{b}^m \in L$, but if we start at x_n and read $\mathtt{b}^m$ we reach a non-accepting state, because $\mathtt{a}^n\mathtt{b}^m \notin L$. So x_m and x_n can't be the same.



This is a general method to prove that a language is not regular. For a different language, you'll need to adjust the definition of x_n, and adjust the way you show $x_m \neq x_n$ for $m < n$.

Example: let the alphabet be $\{\mathtt{a}, \mathtt{b}\}$. Prove that the set of words in which a occurs more times than b is not regular.

Let's repeat the main point: a computer with finitely many states, and no access to external memory, cannot solve the matching problem for any non-regular language. For example, it can't determine whether a word (read in letter by letter) is well-bracketed.