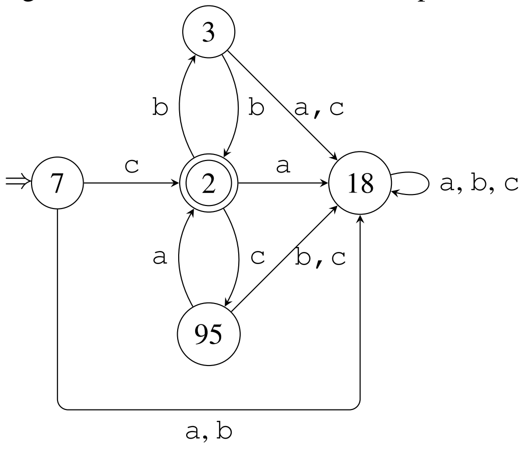
Applications of Kleene's Theorem

1 Regular language

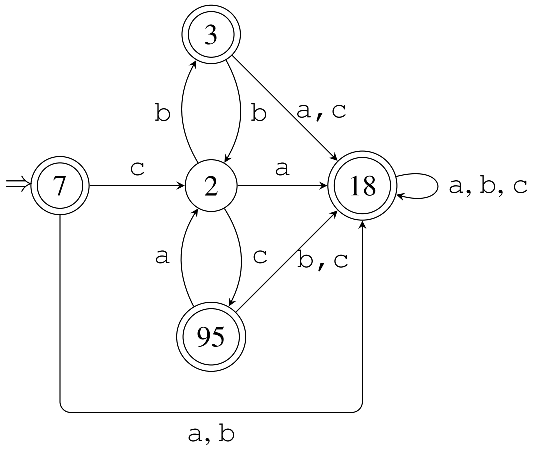
Recall that a *language* L is a set of words, i.e. a subset of Σ^*. We say that it's *regular* when it's the language of a regex. By Kleene's theorem, this is equivalent to being the language of a DFA.

2 Complementation

The complement of L, written $\overline{L}$ is the set of all words that are not in L. How can we show that the complement of a regular language is regular? This isn't obvious if we think about regexes. But using (total) DFAs, it is clear: just replace accepting states by non-accepting ones and vice versa. For example, for the alphabet $\{\texttt{a}, \texttt{b}, \texttt{c}\}$, we know that $\texttt{c}(\texttt{bb}|\texttt{ca})^*$ is recognized by



so its complement is recognized by



We know that complementation satisfies some laws:

$$\begin{aligned}
\overline{L \cap M} &= \overline{L} \cup \overline{M} \\
\overline{L \cup M} &= \overline{L} \cap \overline{M} \\
\overline{\overline{L}} &= L
\end{aligned}$$

The first two laws are called *de Morgan's laws*, and the last one says that complementation is *involutive*. It follows that we can express intersection in terms of complementation and union:

$$L \cap M = \overline{\overline{L \cap M}} = \overline{\overline{L} \cup \overline{M}}$$

Therefore, if L and M are regular, so is $L \cap M$ (since the union of regular languages is regular).

(It's also easy to directly construct a DFA for $L \cap M$ out of DFAs for L and M, by taking ordered pairs of states.)

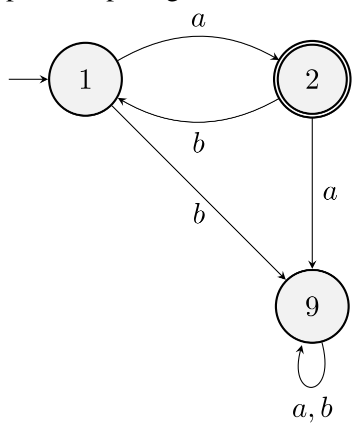
For example, think of the password question on the first exercise sheet. You can make a DFA that determines whether a word has at least 3 characters, and another that determines whether it has a letter, and another that determines whether it has a digit. Then we obtain a DFA for the intersection of these languages. Kleene's theorem tells us that there's a corresponding regex.

3 Language equivalence

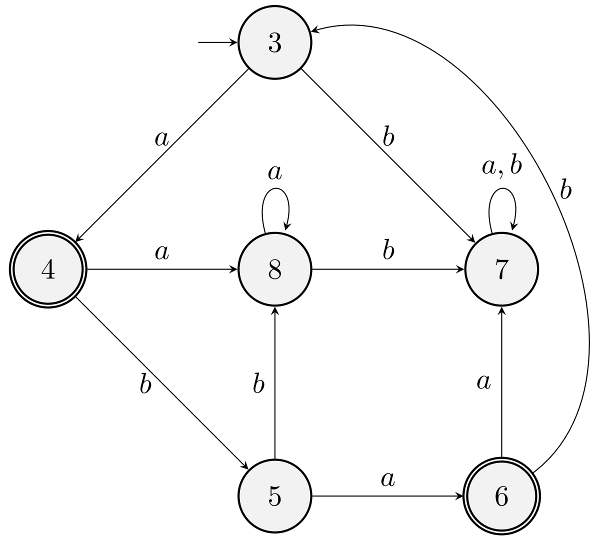
It's not obvious how to test whether two regexes are language equivalent. But let's see how to test whether two DFAs are language equivalent. **Note that this doesn't work for partial DFAs, so you should add an error state if needed.**

Here's a suggestion. Let's build an automaton-like diagram consisting of pairs (x, y) where x is a state of the first automaton and y one of the second.

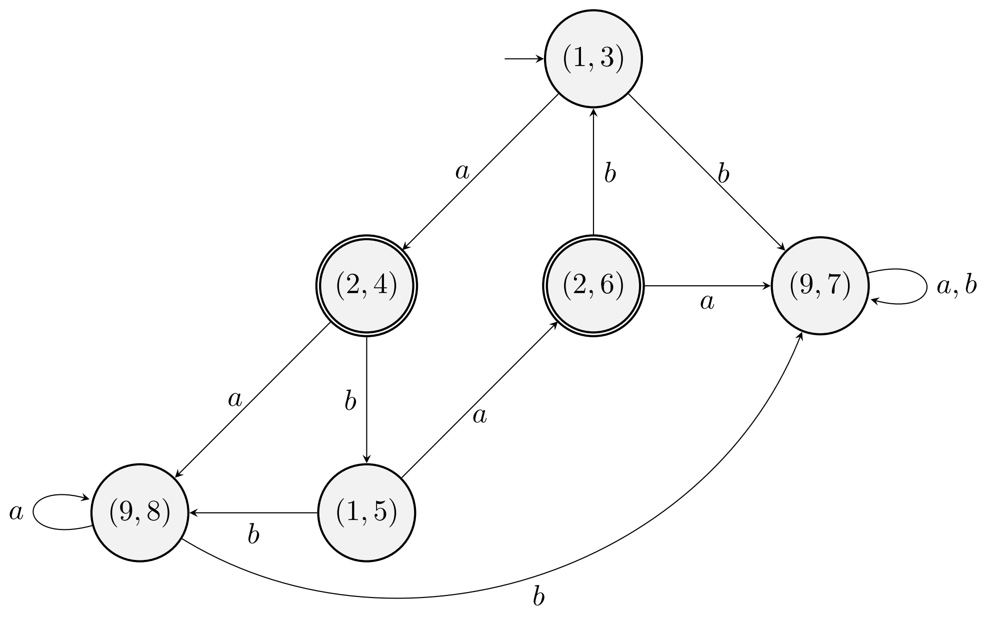
Start at the initial state of each automaton. If one is accepting and the other rejects, then the automata are not language equivalent (since one accepts ε and the other doesn't). If they both accept or both reject then see what pair of states we transition to by inputting a, and what pair by inputting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



to Automaton B:

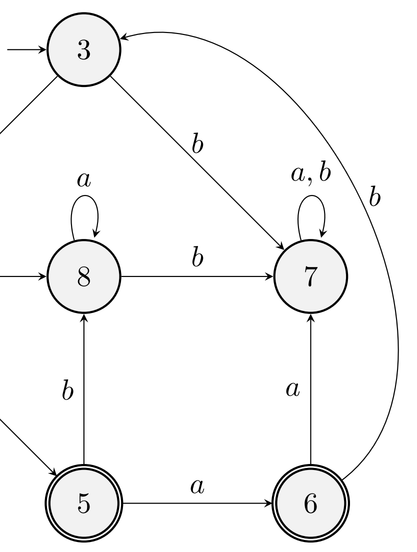


leads to the following:



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.

Now let's compare Automaton A to Automaton C:



In this case, when we hit the pair $(1, 5)$, we see that state 1 is rejecting in Automaton A, but state 5 is accepting in Automaton C. So these DFAs are inequivalent, and we read off the word ab, which Automaton A rejects and Automaton C accepts.

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