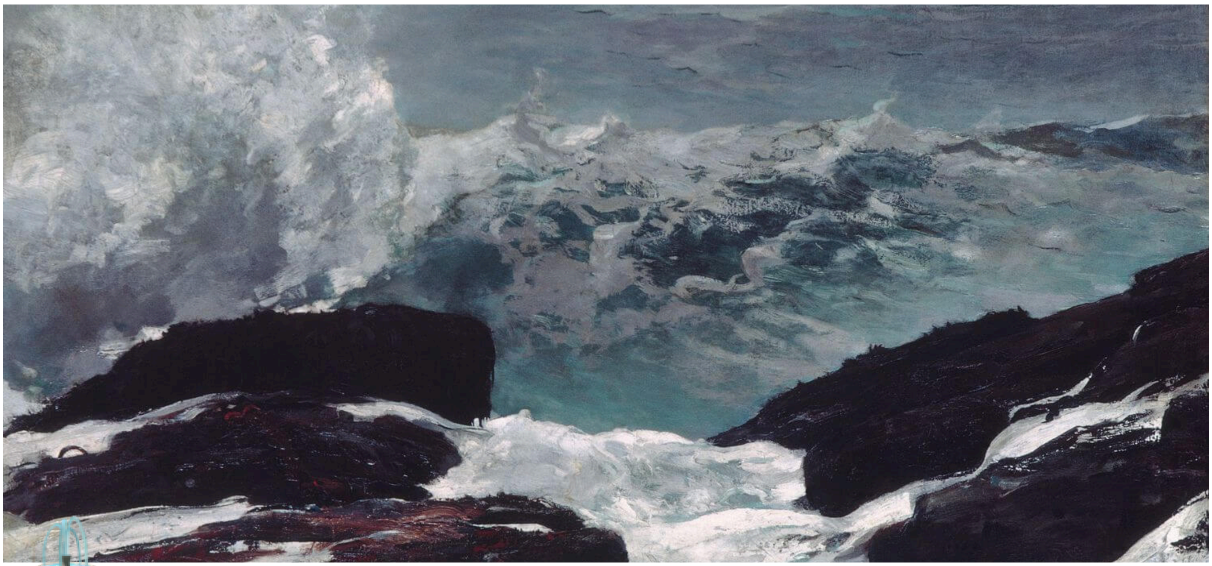


Regular expressions and automata

Regular expressions and automata

Look at these two problems:

1. A string is a “valid password” when
    a. It contains at least 8 characters and at least 2 digits.
    b. Given a string, say whether it is a valid password.
    c. ***This is an example of a matching problem.***
2. Given a string (e.g. a file)
    a. List all occurrences of email addresses within it.
    b. Each occurrence should be represented as a pair of numbers (*i, m*)
        a. where i is the start position
            a. e.g. 0 for an occurrence at the start of the file
        b. and *m* is the length.
        c. The second is a *finding problem*.

- Such problems and variations as this often arise in computing, so people have made tools to solve them efficiently.

- To use such a tool, you have to specify when a word is a valid password or an email address or whatever.
- Often, we do this by means of a *regular expression*.

1.2 Definitions

- The alphabet (set of characters) is called Σ.
- To keep things simple, let's suppose that it's {a, b, c}.
- In a practical situation, it might instead be the ASCII alphabet, which has 128 characters.
- Or it might be the Unicode alphabet, which has 137, 439 characters.
- In any case, we'll assume that Σ is a finite set and contains at least two characters.

1. We write $\sum^*$ for the set of all words.
2. A *language* is a set of words
   a. i.e. a subset of $\sum^*$.
3. For example:
   a. The set of valid passwords is a language, and so is the set of email addresses.
   b. A regular expression (regexp)
      i. such as $c(bb|ca)^*$
         1. represents a language
   c. just as an arithmetic expression
      a. such as 2 + (5 × 3)
         a. represents a number.

How regexps work:

- The regexp a matches only the word a.
- The regexp b matches only the word b.
- The regexp c matches only the word c.
- The regexp ε matches only the empty word ε.

- If *E* and *F* are regexps
  - then the regexp EF matches any word that's a concatenation of a word matched by *E* and a word matched by *F*.
- If *E* and *F* are regexps
  - then the regexp $E|F$ matches any word that either *E* or *F* matches.
- If *E* is a regexp, then the regexp E^* matches any word that's a concatenation of several (i.e. zero or more) words matched by *E*.
- The (rarely used) regexp $\emptyset$ doesn't match any word.

1.3 Precedence

- For arithmetic expressions, × has higher precedence than +.
- Knowing this enables us to parse the expression 3+4×2 as 3 + (4 × 2)
- For regexps, the precedence laws are as follows:
  - Juxtaposition
    - which means "putting things next to each other"
    - has higher precedence than |
    - has lower precedence than *.
  - Knowing this enables us to parse $c(bb|ca)^*$ as $c(((bb)|(ca))^*)$

More operators that you can use:

- $E^+$is short for EE^*. It matches any word that is a concatenation of one or more words matched by *E*.
- $E?$ is short for $\epsilon|E$

These have the same precedence as *

- Some tools provide additional operators, which make it possible to express fancier languages.
- Expressions using these additional operators may be called "regular expressions" in the tool documentation, but technically they are not regular.

Examples:

YES

NO

1. Does the regexp $c(bb|ca)^*$ *match* ccacabb*?*

2. Does the regexp $c(bb|ca)^*$*match* cbbcacac*?*

3. Does the regexp $(c(bb|ca)^*)^*$*match* cccacacbbcbbca*?*

4. Do $(a|b)c^*$*and* $ac^*|bc^*$ *represent the same language?*

5. Do $(a|b)c^*$*and* ac^**represent the same language?*

Some questions about regular expressions

Before we look into regexps in more detail, let's consider some questions. For some of these, the answers are far from obvious, but will emerge over the coming lectures.

Regular and Irregular Languages

Recall that a regexp represents a language. Any language $L \subseteq \sum^*$ that can be represented in this way is said to be *regular*. Questions:

1. Are there any languages that are not regular? Answer: Yes, and we'll see some examples.

2. Is the complement of a regular language always regular? (For example, is there a regexp for those words that are not matched by $c(bb|ca)^*$*?)* Answer: Yes.

3. Is the intersection of two regular languages always regular? (For example, is there a regexp for those words that are matched by both $cc(bb|ca)^*$ *and* $c(bbbb|cca)^*$*?)* Answer: Yes.

2.2 Decidability questions

- A *decision problem* is a problem that, for any given argument, has a Yes/No answer.
- For example, the finding problem above is not a decision problem, because the answer (for a given file) is a set of pairs of numbers.
- A decision problem is said to be *decidable* when there is some program that, given an argument, says whether the answer is Yes or No.
- We can ask the following questions about regexps:

  1. Is the matching problem for the regexp c(bb|ca)**decidable*?

In other words, is there some program that, when given a word *w* over our alphabet $\sum = a, c, c$, returns True if *w* matches $c(bb|ca)^*$, and False if it doesn't? (If *w* isn't a word over our alphabet, then it doesn't matter what happens.) Answer: Yes.

2. Is the matching problem for the regexp $(c(bb|ca)^*)^*$decidable? Answer: Yes.

3. Is it the case that, for *every* regexp *E*, the matching problem for *E* is decidable? Answer: Yes.

4. Is the matching problem for regexps decidable?

   In other words, is there some program that, when given a regexp *E* and word *w*, returns True if *w* matches *E*, and False if it doesn't? Answer: Yes.

5. Is language equality for regexps decidable?

   In other words, is there some program that, when given regexps *E* and *F*, returns True if they represent the same language and False otherwise? Answer: Yes.

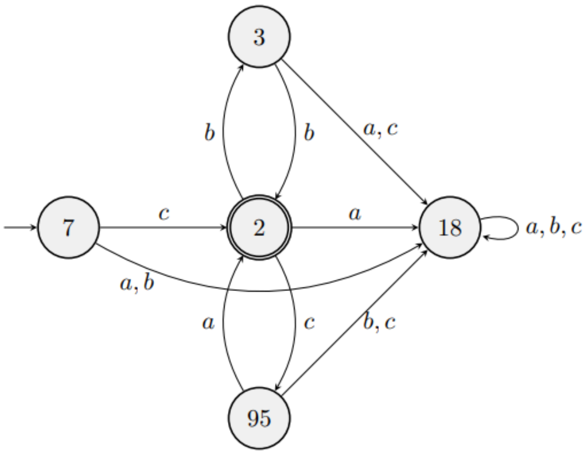
Efficiency questions

- In the previous section, we asked whether certain problems can be solved at all.
- Another question is: can they be solved efficiently?
  - After all, your customers aren't willing to wait a long time for an answer from your program.
  - This question isn't very precise, but it's important.
  - We'll see that for some of these problems, we can give a reasonably efficient solution.

Introducing automata

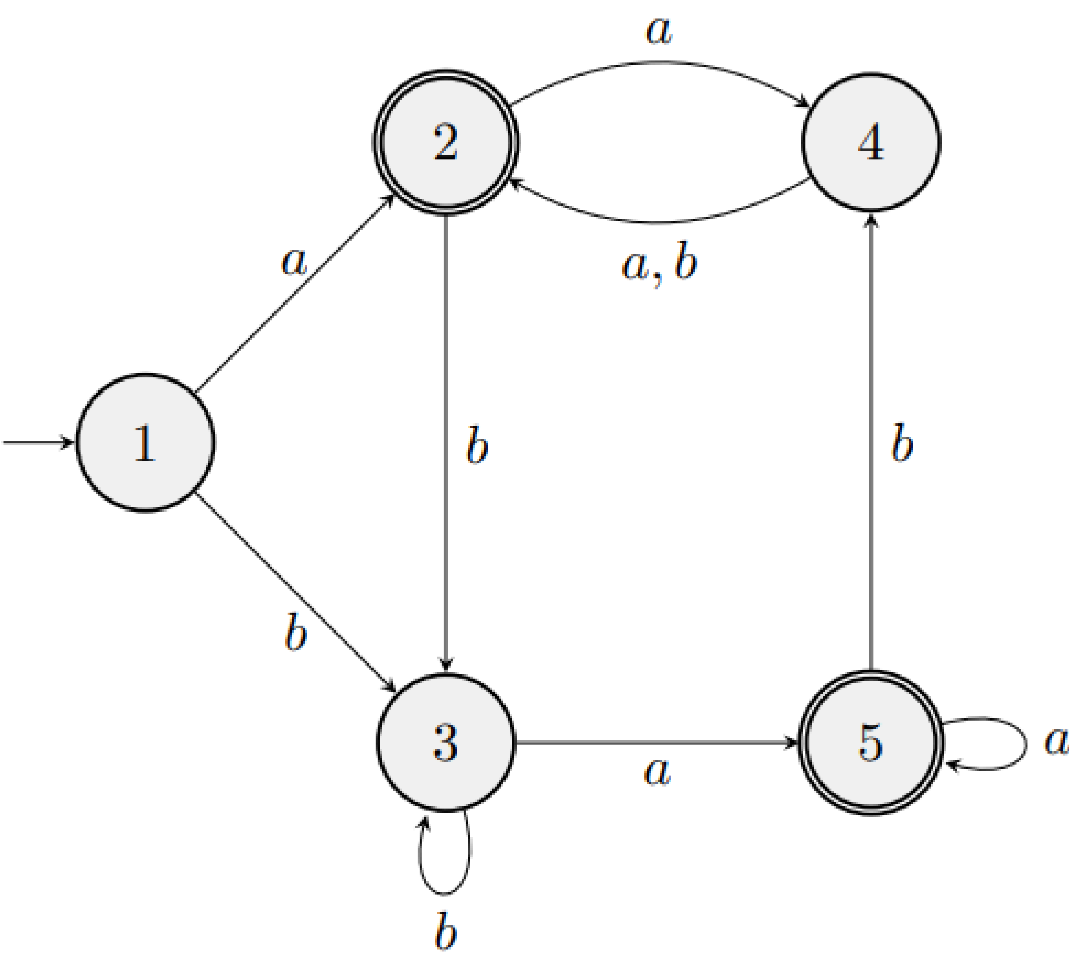
3.1 Deterministic automata

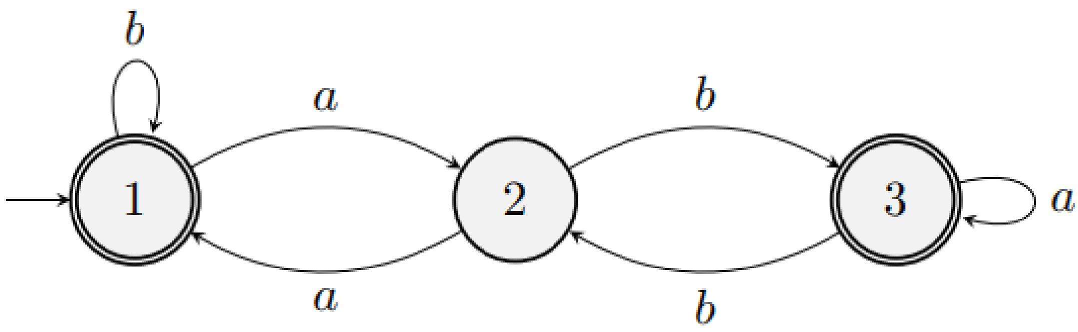
3.1 Deterministic automata



Recall that we wanted a program to solve the matching problem for $\mathtt{c(bb|ca)}^*$. This can be achieved by the automaton shown. There are five *states*, represented as circles. The automaton processes a word by starting at the *initial state* (indicated by $\rightarrow$) and performing a *transition* as it inputs each letter. When the whole word has been input, the automaton returns Yes if the current state is *accepting*, indicated by a double ring. It returns No if the current state is *rejecting*, indicated by a single ring.

This is a *total deterministic finite automaton* (total DFA). "Total" because the initial state and the result of a transition exist, "deterministic" because the initial state and the result of each transition are specified. "Finite" because the set of states is finite.





Key points in a DFA:

- *A finite set X of states.*
- *An initial state*

$p \in X$

- *A transition function*

$\delta : X * \sum - > X$

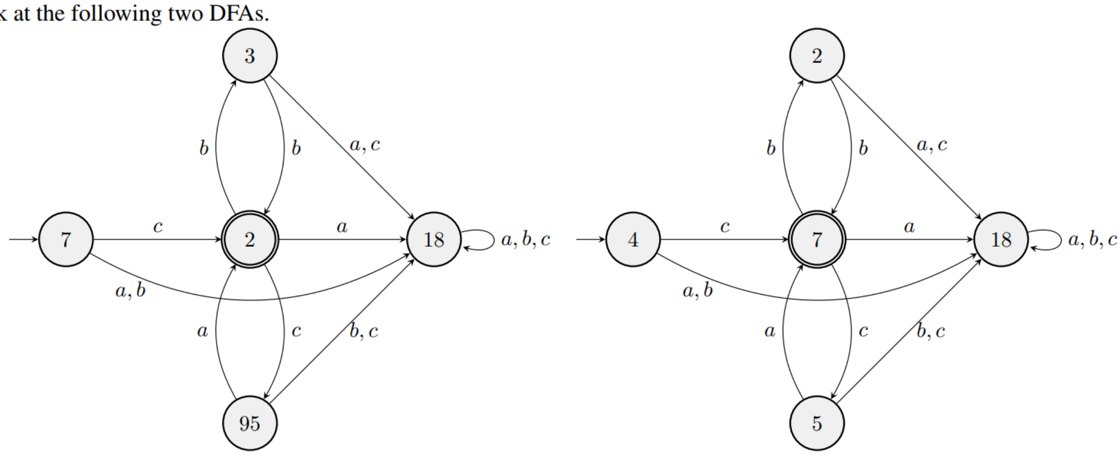
- *A set of accepting states*

$Acc \subseteq X$

$$
\begin{array}{rcl}
X & = & \{7, 2, 3, 95, 18\} \\
p & = & 7 \\
\delta & = & \{(7, \mathsf{a}) \mapsto 18, (7, \mathsf{b}) \mapsto 18, (7, \mathsf{c}) \mapsto 2, \\
 & & (2, \mathsf{a}) \mapsto 18, (2, \mathsf{b}) \mapsto 3, (2, \mathsf{c}) \mapsto 95, \\
 & & (3, \mathsf{a}) \mapsto 18, (3, \mathsf{b}) \mapsto 2, (3, \mathsf{c}) \mapsto 18, \\
 & & (95, \mathsf{a}) \mapsto 2, (95, \mathsf{b}) \mapsto 18, (95, \mathsf{c}) \mapsto 18, \\
 & & (18, \mathsf{a}) \mapsto 18, (18, \mathsf{b}) \mapsto 18, (18, \mathsf{c}) \mapsto 18\} \\
\mathsf{Acc} & = & \{2\}
\end{array}
$$

3.2 Isomorphisms

Look at the following two DFAs.



Each of them solves the matching problem for $\mathtt{c}(\mathtt{bb}|\mathtt{ca})^*$. They are almost the same, but not quite. We see the following correspondence:

State of the left DFA	State of the right DFA
~~3~~	~~2~~
7	4
2	7
18	18
95	5

1. This is called an *isomorphism*.
   - It is a bijection (one to one correspondence) between the sets of states of the left DFA and the set of states of the right DFA with the following properties.
     - The initial state in the left DFA corresponds to the initial state in the right DFA.
     - For each state x in the left DFA there is a value corresponding to it in the right DFA
     - For each character c, the result of starting at x and reading c in the left DFA corresponds to the result of starting at x' and reading c in the right DFA.
     - For each state x in the left DFA corresponding to x'in the right DFA, they're either both accepting or both rejecting.
2. To make the isomorphism obvious, the two diagrams were drawn the same way.
3. Because isomorphic automata have the same language (i.e. they accept the same words), we can leave the circles blank when drawing an automaton.

a. You might like to imagine that each circle is filled with its coordinates on the page.

b. However, if we want to refer to specific circles, it is helpful to number them in some way.

Vending machines

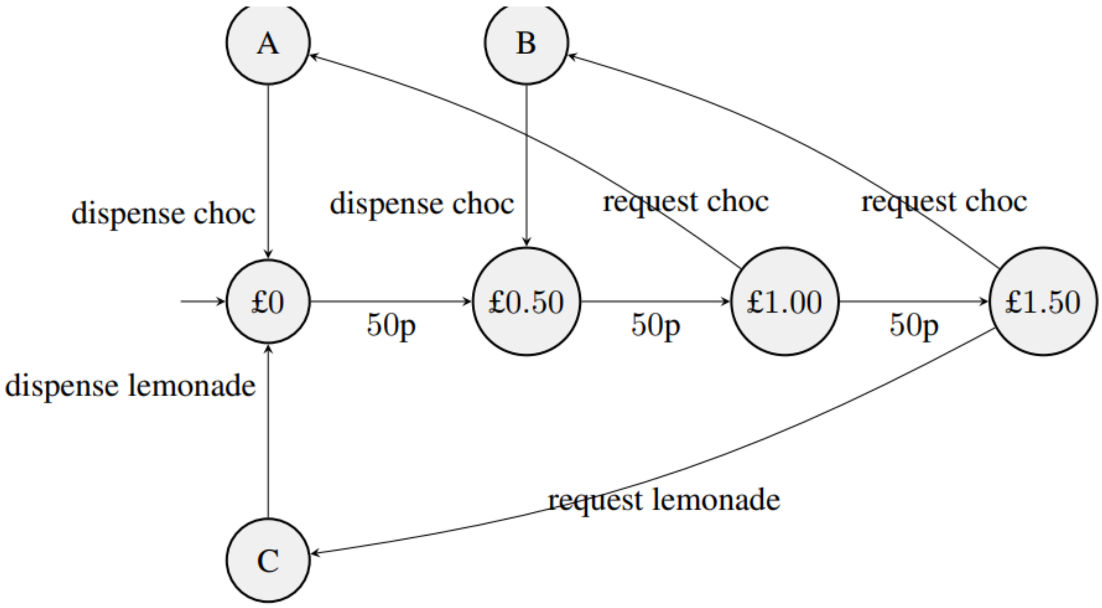
The idea of a DFA was invented for a specific purpose: solving a language's matching problem.

All a DFA can do is input letters and say whether the word that's been read is accepted or not.

But for other purposes, there are other kinds of automaton.

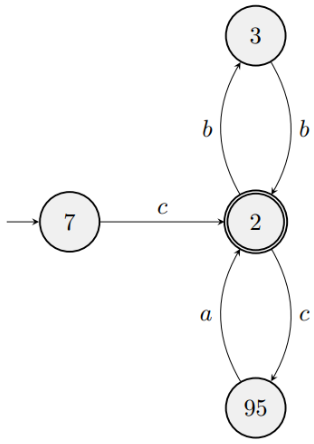
In the lobby there's a vending machine that can receive 50p coins, up to a maximum of £1.50. Chocolate costs £1.00 and lemonade costs £1.50.

Here is an automaton for the machine:



- Note that this automaton can *input* money and requests, and also *output* chocolate and lemonade.
- There are no accepting states, since recognizing words is not the purpose of this machine.

Partial deterministic automata



It's more efficient than the one at the start of Section 3.1. It is a *partial DFA*, meaning that δ is merely a partial function, i.e. it can sometimes be undefined.[1] As soon as a character cannot be input, the word is rejected. For example, the word `cbccabbabcababccabcc` is rejected after just three characters. (A partial DFA can also have no initial state, but then every word is rejected straight away, so this isn't very useful.)

We can easily turn a partial DFA into a total DFA

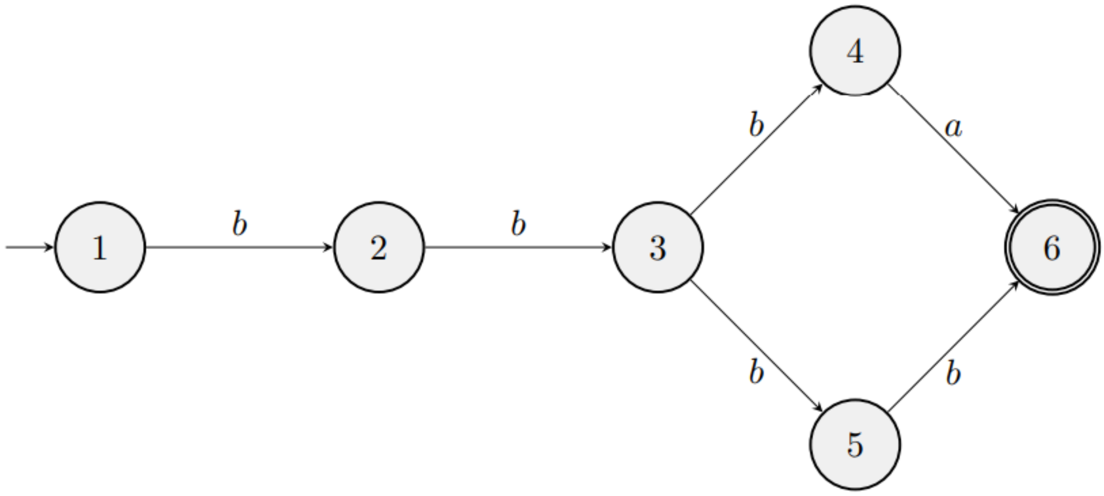
- just add an extra non-accepting state, called the *error state* (18 in the example).
- Transitions that are undefined in the partial DFA go to the error state.
- And every transition from the error state goes to the error state.
  - If the partial DFA has no initial state, the error state will be initial

Please note that a total DFA is also a partial DFA.

Nondeterministic automata

Sometimes it is difficult to obtain a DFA for a regexp, but we can more easily obtain a *nondeterministic* finite automaton (NFA).

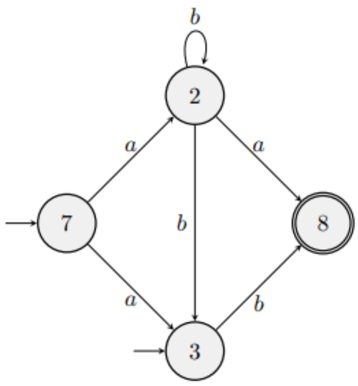
Here's an example, for the language bb(ba|bb).



- A nondeterministic finite automaton (NFA) differs from a DFA in two respects.
- Firstly an NFA can have several initial states.
- Secondly, from a given state, when a (or any other character) is input, there can be several possible nextstates.
- Thus δ is a *relation* but not a function.
- The automaton chooses its initial state, and chooses what state to moveto as it inputs a character.
- A word *w* is *acceptable* when there is **some** path from an initial state to an accepting state that goes through the characters of *w*.

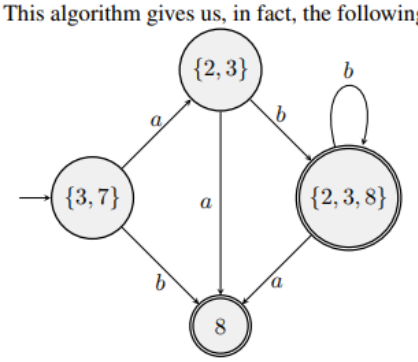
5.2 Determinizing an NFA: transforming an NFA into a DFA

- An NFA is useless in practice: we want a program that always says Yes to a good word and No to a bad word, and an NFA doesn't do that.
- But we can *determinize* it, i.e. turn it into a DFA that recognizes the same language.
- To see how this works, look at the following example.



Let's think how to find out whether the word `abb` is acceptable. We can do it by trial and eror, but there's an algorithmic way: *keep track of the current set of possible states*. Initially the set of possible states is $\{3,7\}$. After inputting a, the set of possible states is $\{2,3\}$. After inputting b, the set of possible states is $\{2,3,8\}$. After inputting b again, the set of possible states is $\{2,3,8\}$. We have reached the end of the word, and we note that one of the currently possible states, viz. 8, is accepting. Therefore the word `abb` is acceptable.

This algorithm gives us, in fact, the following partial DFA:



The "states" of the DFA are *sets* of states of the NFA. The initial "state" is the set of all the initial states of the NFA. A "state" is accepting when it contains an accepting state of the NFA. From a given "state", when we input a character, we collect all the possible next states. This process is called *determinization*.

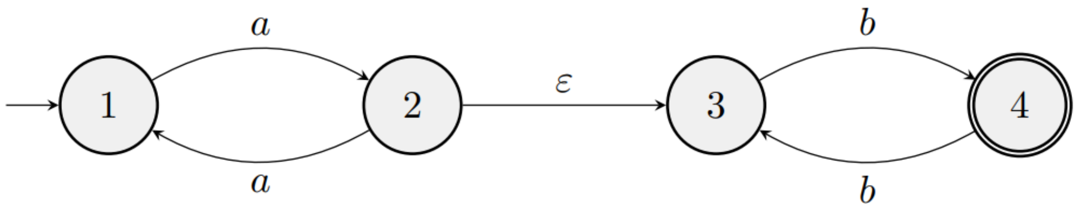
We see that a word w is accepted by the DFA iff it's acceptable to the NFA. Therefore they represent the same language.

6 ε-transitions

Sometimes even an NFA is difficult to obtain, but we can obtain an automaton that spends some time thinking.

As it thinks, it moves from one state to another **without inputting any character**.

Here's an example, for the regexp $a(aa)^*b(bb)^*$.



- We call this a *nondeterministic automaton with ε-transitions* or εNFA for short.
- A word w is acceptable when there is some path from an initial state to an accepting state that goes through the characters of w, padded with ε transitions.

Please note that an NFA is also an εNFA.

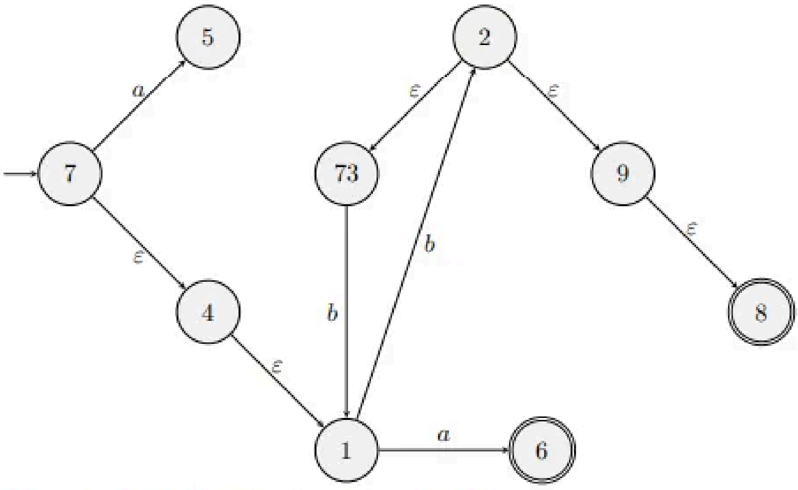
Removing ε-transitions

Happily, it's possible to remove the ε-transitions from a εNFA

i.e. convert it to an NFA that recognizes the same language.

Let's call the algorithm "Prior epsilon absorption".

To get the idea, let's look at the following εNFA.



The word bbb is acceptable, because of the following path:

$$7 \xrightarrow{\varepsilon} 4 \xrightarrow{\varepsilon} 1 \xrightarrow{b} 2 \xrightarrow{\varepsilon} 73 \xrightarrow{b} 1 \xrightarrow{b} 2 \xrightarrow{\varepsilon} 9 \xrightarrow{\varepsilon} 8$$

This path consists of the following pieces:

$7 \xrightarrow{\varepsilon} 4 \xrightarrow{\varepsilon} 1 \xrightarrow{b} 2$ is a *slow* b-transition from 7 to 2;

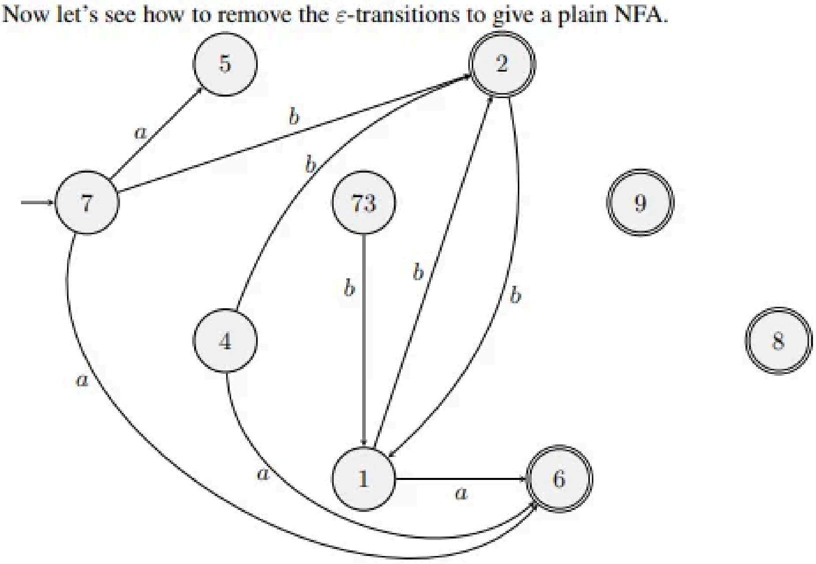
$2 \xrightarrow{\varepsilon} 73 \xrightarrow{b} 1$ is a *slow* b-transition from 2 to 1;

$1 \xrightarrow{b} 2$ is a *slow* b-transition from 1 to 2;

$2 \xrightarrow{\varepsilon} 9 \xrightarrow{\varepsilon} 8$ is *slowly accepting*.

You can see that this path consists of three *slow* b-*transitions* followed by *slow acceptance*. A slow b-transition consists of several (zero or more) ε-transitions, culminating in a b-transition. Slow acceptance consists of several ε-transitions, culminating in an accepting state.

Now let's see how to remove the ε-transitions to give a plain NFA.



1. **Slow b-transition**:
   - *A path starting with any number of ε-transitions, ending with a b-transition.*
   - *Example:*

$$7 \overset{e}{-} > 4 \overset{e}{-} > 1 \overset{b}{-} > 2$$

(Two ε-steps followed by a b-transition).

- *Slow transitions (e.g., slow b - transitions) include ε-transitions followed by one symbol transition (e.g., b).*

2. **Slow acceptance**:
   - *A path starting with ε-transitions, ending in an accepting state.*
   - *Slow acceptance is defined as a sequence of zero or more ε-transitions followed by reaching an accepting state*
   - *Therefore slow acceptance must only include ε-transitions (no other symbol transitions). Here's the distinction*
   - *Example:*

     $2 \overset{e}{-} > 9 \overset{e}{-} > 8$

     (Two ε-steps leading to the accepting state 8).
   - *Here, only ε-transitions are used to reach the accepting state 8.*
     - *No actual input symbols (like b) are consumed during slow acceptance.*

   Thus, slow acceptance is strictly ε-based, while slow b-transitions mix ε-moves with a final symbol transition.

- *These "slow" paths allow the automaton to traverse multiple states via ε before consuming an input symbol (b) or reaching acceptance*

- The automaton looks similar to before: the states are the same and the initial state is the same.
- The difference is that the transitions you see here are the slow transitions, and the accepting states you see here are the slowly accepting states.
- So in general, the *Prior Epsilon Absorption* algorithm is as follows
  - we take the same states and the same initial states

    - but include all the slow-transitions and all the slowly accepting states.
- Clearly we can remove the unreachable states 8, 9 and 4
- It’s always acceptable to remove unreachable states, because this doesn’t change the language of the automaton
- (i.e. the set of acceptable words).