Computer Science

Dr. Y. Liow (March 29, 2024)

Contents

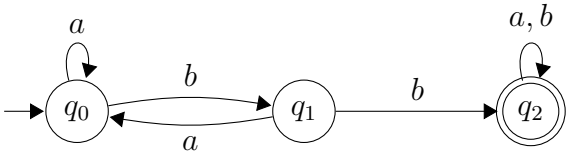
Chapter 12

Regular Languages

12.1 State Diagram debug: dfastatediag.tex

The following is the **state diagram** of a **deterministic finite automata DFA**:

state diagram
deterministic finite automata
DFA



A state diagram is also called a **transition diagram**.

transition diagram

A DFA is also called a **deterministic finite state machine DFSM**.

deterministic finite state machine
DFSM

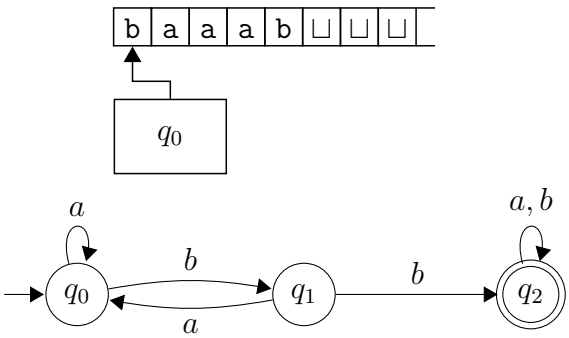
A directed edge in the state diagram is called a **transition**. A transition labeled with symbol a is called an a–**transition**. In this example, a and b are the symbols used to label the transitions. We say that the set $\{a, b\}$ is the **alphabet** of this DFA. Let's use Σ to denote the set $\{a, b\}$.

transition
a–transition
alphabet

You think of the above state diagram as the program of the DFA: it describes how this machine computes.

This is how you run the machine with a given input. The input will be a string of a's and b's. For instance the input can be *baaab*; it can even be ϵ the empty string. However *abbc* is *not* an input for this machine: c does not appear in the transitions! The input *baaab* is placed on an input tape for the machine read.

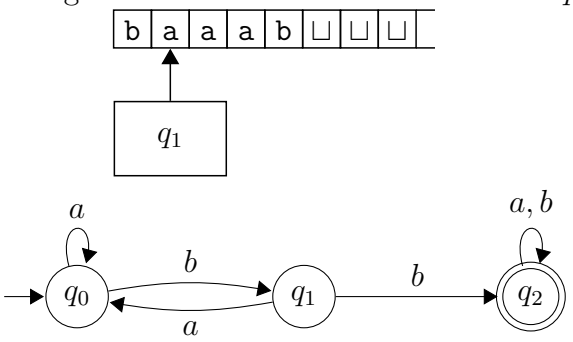
- Start at the **start state**. This is the node with an arrow pointing to it from nowhere. For the above example, the start state is q_0. Look for it now. Put your finger on it. (I'm using $\sqcup$ to denote a blank on the tape.)



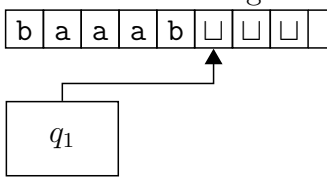
- Read the first character of the input string; you always read the string left to right. For the above DFA with input *baaab*, the first character is

b (the b on the left).

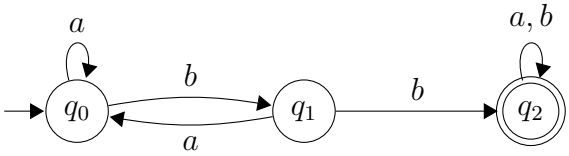
- Go from the node you're pointing at to the next node using the edge labeled with the character you've just read. For our DFA with input $baaab$, we move along the b–transition and land in state q_1.



- Repeat the above process, reading the remaining characters of the input left-to-right, until you have no more characters. The characters of the input are b, a, a, a, b and starting with q_0, we go through the states q_1, q_0, q_0, q_0, q_1. The last state after reading the whole input is q_1.



You can think of the above as follows. Given a DFA:



If you have a string $baaab$, then this string traces out a *path* in the DFA which can be described by this notation:

$$q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_0 \xrightarrow{a} q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_1$$

This is an informal notation. I'll show you two proper formal notations for "computation" in the next section.

- A **accept state** or **final state**. is a node denoted with a double-circle boundary. When you have finished scanning your string left to right, and if you are on a final state, then the string is **accepted** by the DFA, otherwise the string is not accepted.

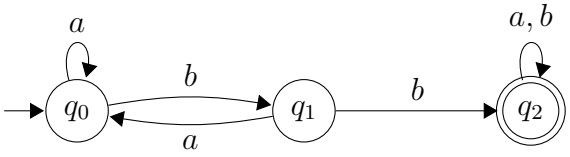
accept state

final state

accepted

You should think of a DFA as a string recognizer in the sense that when you feed it a string, it will tell you whether the string is accepted or not.

Example 12.1.1. Write down the list of states you pass through as you run the DFA with the string *ababb*.



What is the last state? Is it an accept state? Is the string accepted by the DFA? The formal way to write down the steps in the computation is as follows:

$$\begin{aligned}(q_0, ababb) &\vdash (q_0, babb)\\ &\vdash (q_1, abb)\\ &\vdash (q_0, bb)\\ &\vdash (q_1, b)\\ &\vdash (q_2, \epsilon)\end{aligned}$$

The (q, x), where q is a state and x is a word, is called an **instantaneous description**.

instantaneous description

Here's a way to think of the above: First of all do you see that instantaneous descriptions are elements of the set of product of set of states and set of words. If I write the set of states as Q, then the the instantaneous descriptions are the elements of

$$Q \times \Sigma^*$$

Next, do you see that $\vdash$ is a relation on $Q \times \Sigma^*$?

Definition 12.1.1. Let M be a DFA. The language **accepted** by M, denoted $L(M)$ is the set of **all** strings over Σ accepted by M.

accepted

Note that a directed graph is the state diagram of a DFA over Σ if:

initial state

- There is exactly one special node called the **start state** or **initial state**. This is drawn with a single-line boundary with an arrow pointing to it from nowhere. (Some authors use a wedge instead of an arrow.)
- There is 0 or more nodes called the **accept states** or **final states**. These are drawn with a double-line boundary.
- For each node v, there are exactly $|\Sigma|$ edges going **out** from v to other nodes. These edges are labeled with the elements in Σ. Note that every

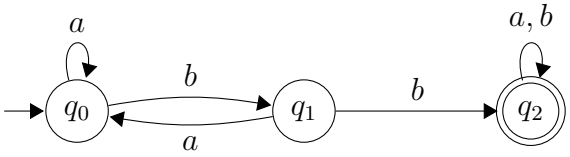
alphabet in Σ is used in labeling the edges of v.

- In a DFA state diagram, the nodes are called **states** and the edges are called **transitions**. If a transition is labeled with $a \in \Sigma$, we also call it an a–**transition**.

states
transitions
a–transition

Note that given the state diagram of a DFA, say M, an input can contain only the characters in Σ where Σ is the set of symbols appearing in the transitions of M.

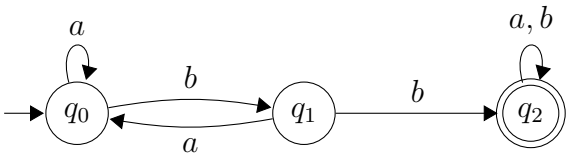
Exercise 12.1.1. Refer to the above DFA



1. What is the string of shortest length accepted by the above DFA?
2. Write down the computation of the string *baba*. Is this string accepted by the DFA?
3. Write down the computation of the string *abab*. Is this string accepted by the DFA?
4. How many strings of length 5 are accepted by the DFA?

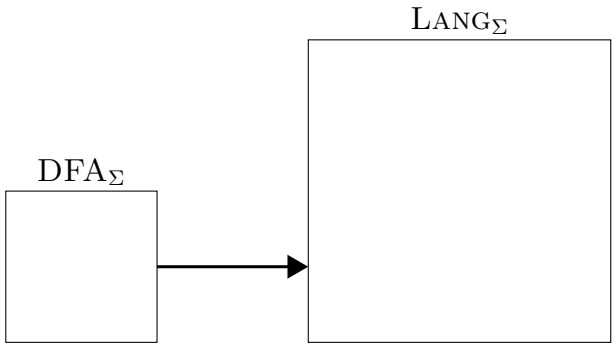
□

Exercise 12.1.2. What is the language accepted by the above DFA:



□

So far we have seen examples where you're given a DFA (or at least the state diagram), and you look at the string or the full language accepted by the DFA. In the following diagram, DFA_Σ is the collection (i.e., set) of all DFAs over Σ. Lang_Σ is the collection of languages over Σ. Each DFA gives us a language. (You'll see that indeed, the collection of languages that is, in some sense, much larger than the collection of DFAs.)



Now let's start with a language.

Exercise 12.1.3. Draw a state diagram that accepts the following language of strings over $\Sigma = \{a, b\}$

$$L = \{\}$$

Exercise 12.1.4. Draw a state diagram that accepts the following language of strings over $\Sigma = \{a, b\}$

$$L = \{a\}$$

Exercise 12.1.5. Draw a state diagram that accepts the following language of strings over $\Sigma = \{a, b\}$

$$L = \{\epsilon\}$$

□

Exercise 12.1.6. Draw a state diagram that accepts the following language of strings over $\Sigma = \{a, b\}$

$$L = \{\epsilon, a\}$$

□

Exercise 12.1.7. Draw a state diagram that accepts the following language of strings over $\Sigma = \{a, b\}$

$$L = \{a, b\}$$

□

Exercise 12.1.8. Draw a state diagram that accepts the following language of strings over $\Sigma = \{a, b\}$

$$L = \{a, a^2\}$$

□

Exercise 12.1.9. Draw a state diagram that accepts the following language of strings over $\Sigma = \{a, b\}$

$$L = \{ab, ba\}$$

□

Exercise 12.1.10. Draw a state diagram that accepts the language of strings over $\Sigma = \{a, b\}$ that contains aaa, i.e.

$$L = \{x \in \Sigma^* \mid x = ya^3z \text{ for some } y, z \in \Sigma^*\}$$

□

Exercise 12.1.11. Draw a state diagram that accepts the language of strings over $\Sigma = \{a, b\}$ that does not contain aaa. □

Exercise 12.1.12. Draw a state diagram that accepts the language of strings over $\Sigma = \{a, b\}$ that contains $abab$. □

Exercise 12.1.13. Draw a state diagram that accepts the language of strings over $\Sigma = \{a, b\}$ where the number of a's is exactly $\equiv 1 \pmod 4$. So the number of a's can be $1, 5, 9, \ldots$. □

Exercise 12.1.14. Draw a state diagram that accepts the following language

$$L = \{w \in \Sigma^* \mid |w|_a, |w|_b \text{ both even}\}$$

where $\Sigma = \{a, b\}$. Here, $|w|_a$ is the number of a's in w and $|w|_b$ is the number of b's in w. For example $|abaab|_a = 3$ and $|abaab|_b = 2$. Therefore $abaab \notin L$. However $abbaaa \in L$. □

In the study of languages a very important question to ask is what is the expressivity (or power) of the devices used in describing languages. So for a fixed Σ, can all languages over Σ be the language be accepted by some DFA defined over Σ?

If so, then we only need to study DFAs.

Unfortunately (or fortunately for those are masochistic or who love twisted plots) ... NO! In fact there are *many* languages *not* accepted by DFAs. Here's are a few:

- $\{a^n b^n \mid n \geq 0\}$
- The palindromic strings over Σ if $|\Sigma| > 1$.

12.2 Deterministic Finite State Machines debug: dfa.tex

We need to formalize DFAs.

Definition 12.2.1. A **deterministic finite automata** (**DFA**) or a **deterministic finite state machine** is defined by

deterministic finite automata
DFA
deterministic finite state machine

$$(\Sigma, Q, q_0, F, \delta)$$

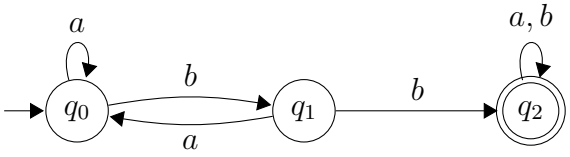
where

1. Σ is a finite set called the **alphabet** — alphabet
2. Q is a set of **states** — states
3. $q_0 \in Q$ is the **start state** or **initial state** — start state, initial state
4. $F \subseteq Q$ is the set of **accept states** or **final states** — accept states, final states
5. $\delta : Q \times \Sigma \to Q$ is the **transition function**. — transition function

In this case we will say that M is defined over Σ.

Note: In some books the ingredients for a DFA is listed in this order $(Q, \Sigma, \delta, q_0, F)$ instead where the set of states come first.

Exercise 12.2.1. Write down the formal definition of the DFA from the state diagram in the previous section.



□

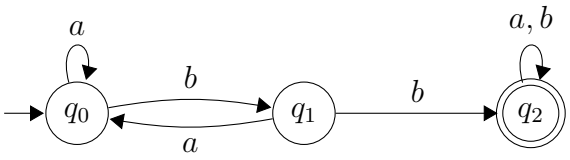
With the above formal definition of a DFA, we can now formally define what we mean when we say that a string is accepted by a DFA.

Definition 12.2.2. Let $M = (\Sigma, Q, q_0, F, \delta)$ be a DFA, $a \in \Sigma$ and $x \in \Sigma^*$.

- We write
$$(q, ax) \vdash (q', x)$$
if
$$\delta(q, a) = q'$$
i.e.,
$$(q, ax) \vdash (\delta(q, a), x)$$
This is a **computation** for string ax. The (q, ax) and (q', x) and in general the elements of $Q \times \Sigma^*$ are called **instantaneous descriptions** (ID). Therefore $\vdash$ is a relation on $Q \times \Sigma^*$. You can think of $Q \times \Sigma^*$ as an infinite directed graph where the edges are determined by $\vdash$.
instantaneous descriptions
- We write
$$(q, x) \vdash^* (q', x')$$
if either $(q, x) = (q', x')$ or there is a sequence of computations from (q, x) to (q', x'):
$$(q, x) \vdash \cdots \vdash (q', x')$$
In other words $\vdash^*$ is the **reflexive transitive closure** of $\vdash$.

Definition 12.2.3. Given $x \in \Sigma^*$, we say that x is **accepted** by M iff
accepted
$$(q_0, x) \vdash^* (q', \epsilon) \quad \text{where } q' \in F$$

Example 12.2.1. Consider this DFA again:



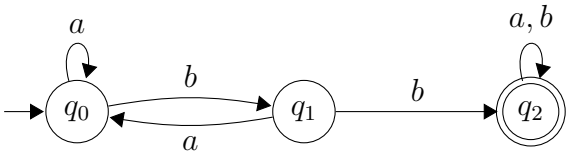
Write down the ID computation for aba. Is aba accepted by the DFA?

SOLUTION. The ID computation is

$$\begin{aligned}(q_0, aba) &\vdash (\delta(q_0, a), ba) = (q_0, ba) \\ &\vdash (\delta(q_0, b), a) = (q_1, a) \\ &\vdash (\delta(q_1, a), \epsilon) = (q_0, \epsilon)\end{aligned}$$

q_0 is not an accept state. Therefore aba is not accepted. □

Exercise 12.2.2. Draw part of $Q \times \Sigma^*$ relation as a directed graph using the $\vdash$ relation for the DFA



Of course $Q \times \Sigma^*$ is infinite so you can't draw the whole graph! Just include (q, x) for all x with length ≤ 4. After that draw the relation for $\vdash^*$. □

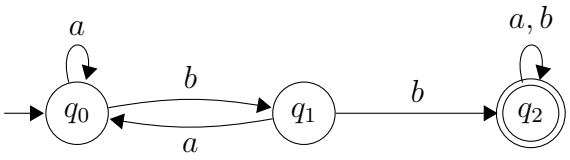
You can also define acceptance using an "extension" of δ. Note that the transition function basically defines one single edge (transition) in the state diagram because the second input is a character. There is an obvious extension to δ so that the second input is a string:

Definition 12.2.4. It is convenient to extend the transition function δ so that we allow strings for the second argument instead of just character. Define

$$\delta^* : Q \times \Sigma^* \to Q$$

(a) $\delta^*(q, \epsilon) = q$ for all $q \in Q$
(b) $\delta^*(q, ax) = \delta^*(\delta(a, q), x)$ for $a \in \Sigma$, $x \in \Sigma^*$, $q \in Q$.

Example 12.2.2. Consider this DFA again:



Write down the δ^* computation for aba. Is aba accepted by the DFA?

SOLUTION. Teh $\delta*$ computation is

$$\begin{aligned}\delta^*(q_0aba) &= (\delta(q_0, a), ba) = (q_0, ba)\\ &= (\delta(q_0, b), a) = (q_1, a)\\ &= (\delta(q_1, a), \epsilon) = (q_0, \epsilon)\end{aligned}$$

I stop at this point since the string now is ϵ. The last state is q_0 which is not an accept state. Therefore aba is not accepted by the DFA.

Comparing the above to

$$\begin{aligned}q_0aba &\vdash \delta(q_0, a), ba) = (q_0, ba)\\ &\vdash \delta(q_0, b), a) = (q_1, a)\\ &\vdash \delta(q_1, a), \epsilon) = (q_0, \epsilon)\end{aligned}$$

you see that using $\vdash$ notation to relate ID is mathematically the same as using δ. □

Now we can define the *whole* language accepted by M:

Definition 12.2.5. Let M be a DFA. Then the **language** accepted by M, denoted $L(M)$, is the language of strings over Σ accepted by M, i.e., language

$$L(M) = \{x \in \Sigma^* \mid (q_0, x) \vdash^* (q, \epsilon) \text{ where } q \in F\}$$

or

$$L(M) = \{x \in \Sigma^* \mid \delta^*(q_0, x) \in F\}$$

Exercise 12.2.3.

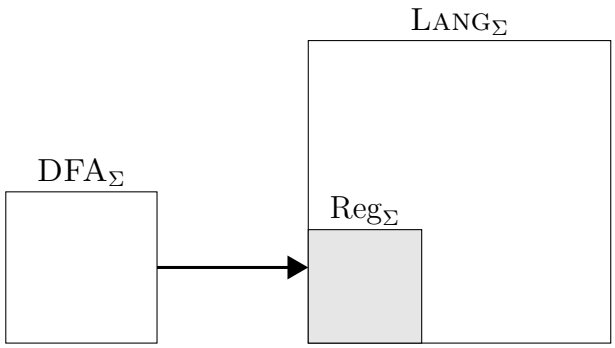
1. Draw DFA M over $\Sigma = \{0\}$ such that $L(M) = \{\}$.
2. Next, define this DFA formally.
3. Now, draw a graph where the nodes are (q, x) where q runs through all the states of your DFA and x runs through all the words Σ^* ... wait that's infinite ... you can't do that. OK ... just draw (q, x) where q runs over all states and x runs over all words of length <= 5. Draw directed edges in the graph so that there's an arrow from (q, x) to (q', x') if $(q, x) \vdash (q', x')$, i.e., $\delta(q, x) = \delta(q', x')$.
4. Finally graph the graph as the above except that there's an arrow from (q, x) to (q', x') if $(q, x) \vdash (q', x')$, i.e., $\delta^*(q, x) = (q', x')$.

□

Exercise 12.2.4. Do the exercise but this time for this language: The DFA accepts the language of strings over $\Sigma = \{a, b\}$ that contains $abab$. □

Languages over Σ of the above kind is important. So let's give them a name:

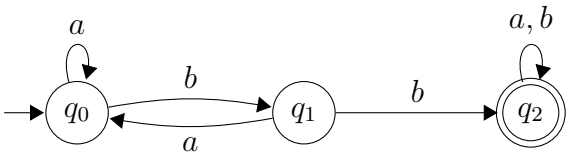
Definition 12.2.6. A language L over Σ is **regular** if $L = L(M)$ where M is a DFA over Σ. We will write Reg_Σ for the collection of all regular languages over Σ. regular



12.3 Implementing a single DFA in C++ debug:

implementing-a-single-dfa.tex

You can actually simulate the computation of a DFA using a programming language such as C++, Java, etc. Let's write a C++ program to simulate this DFA:



Here's the program:

```
#include <iostream>
#include <string>

int main()
{
    std::string x;
    std::cin >> x;

    int state = 0;
    std::cout << 'q' << state << ", ";
    for (int i = 0; i < x.length(); i++)
    {
        char c = x[i];
        switch (state)
        {
            case 0:
                if (c == 'a') state = 0;
                else if (c == 'b') state = 1;
                break;
            case 1:
                if (c == 'a') state = 0;
                else if (c == 'b') state = 2;
                break;
            case 2:
                break;
        }
        std::cout << 'q' << state << ", ";
    }

    std::cout << (state == 2 ? "" : "not ")
```

```
            << "accepted\n";
    return 0;
}
```

Study it carefully.

In the program, I'm using integer values to denote states.

For this program, the states of a computation is shown as well as whether is is accepted or not.

Exercise 12.3.1. Rewrite the above so that you have a δ function:

```
int delta(int state, char c)
{
    ...
}
```

(ANOTHER SECTIOH)

Of course the above program hardcoded lots of information. This means that it's impossible to run a different DFA, you would have to write another program. It's better to have a general library for creating DFAs.

To implement such as class, we have to look at the formal definition of a DFA. A DFA M is made up of

- Σ is a finite set
- Q is a finite set
- q_0 is an element in Q
- F is a subset of Q
- $\delta : Q \times \Sigma \to Q$ is a function

The definition makes it clear that we need to implement sets and functions.

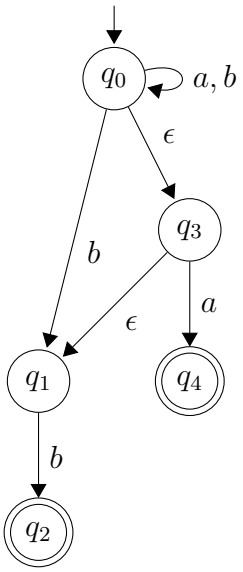
12.4 NFA State Diagram debug: nfastatediag.tex

The following is a **state diagram** or **transition diagram** of a **nondeterministic finite state automata NFA** Some people also called the device a **nondeterministic finite state machine NFSM**.

state diagram
transition diagram

nondeterministic finite state automata NFA

nondeterministic finite state machine NFSM



An edge labeled with symbol a is called an a–**transition**.

a–transition

Exercise 12.4.1. Visually, what are the differences between the state diagram of an NFA and a DFA? □

This is how you use the machine. It's very similar to the state diagram of a DFA except that you keep track of a **set** of states instead of just one.

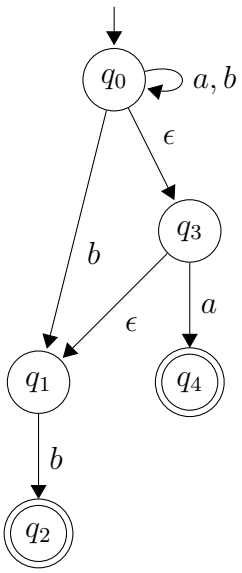
(1) Start at the **start state**. This is the same as DFA. Look for it now. Keep track of this state.
(2) Read the first character of the string; you always **start from the left** of the string and **move right**.
(3) Go from the node you're pointing at to the **nodes** (there might be more than one!) using the **edges** labeled with the character you've just read. Note that you might have more than one path!
(4) If there are ϵ-transition, you need to keep the states you can get to via these ϵ-transitions too.
(5) If a path cannot have a transition, that path dies.
(6) Repeat the above process (2)-(3) until you have no more characters.
(7) An **accept state** is a node with a double-circle boundary. It's also called

accept state

a **final state**. When you have finished scanning your string left to right, you will have a set of paths. If there is a path that ends in an accepting state, then the string is accepted.

final state

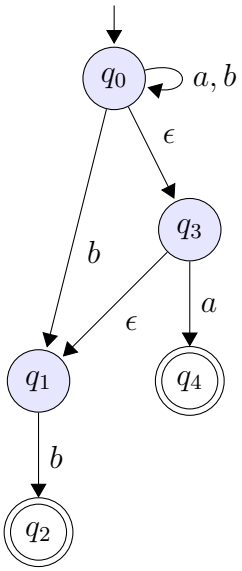
Now let's execute our NFA for several different words w. Let's call the above NFA N_1.



First let me run N_1 on $w = \epsilon$. We start N_1 at q_0.

- N_1 stops immediately at q_0 since there is no character to process.
- However N_1 is greedy and sees that it can also go to state q_3 *without* processing any character (from the input): there's a transition from q_0 to q_3 labeled ϵ. So it spawns a copy of itself, say N_2, at state q_3 while it stays at q_0. Note that N_2 is an exact copy of N_1 – it behaves the say way and it has the exact same input as N_1. The new machine N_2 is at state q_3 and stops there since there's nothing to process and N_2 cannot spawn a new copy of itself.
- However N_2 being just as greedy as N_1, sees that it can also go to q_1 without processing any character. So N_2 stays at q_3 while it spawns off another machine, say N_3, at state q_1 while N_2 stays at q_3. N_3 stops at state q_1.

At this point, all machines stops processing characters and they have spawn the maximum number of machines along ϵ-transitions. Everyone (everyit?) now reports their states to N_1 and disappears. N_1 looks at all the states that all of its descendents (and itself) have reached.



None of the 3 machine manage to reach an accepting state. So N_1 does not accept (i.e rejects) ϵ.

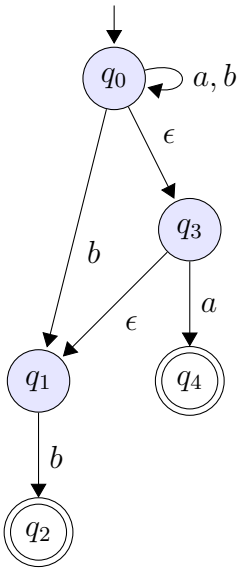
Now let me run N_1 on $w = a$. Again I start N_1 at q_0.

- N_1 begins at state q_0. It's about to process a, but before it does that ...
- N_1 is greedy and spawns another machine N_2 which is at state q_3 because there's an ϵ–transition from q_0 to q_3. N_2's input looks exactly like the input of N_1, i.e., N_2 is about to process a, but before it does that ...
- N_2 is also greedy and spawns another machine N_3 which is at state q_1 because there's an ϵ–transition from q_3 to q_1. N_3 is also about to process a.

At this point, we have 3 machines:

$$N_1 \text{ at } q_0, \quad N_2 \text{ at } q_3, \quad N_3 \text{ at } q_1$$

Each of them has an input of $w = a$ (i.e. none of them has processed any character yet). Here's the state diagram showing the states (shaded) where there are machines getting readying to process a:

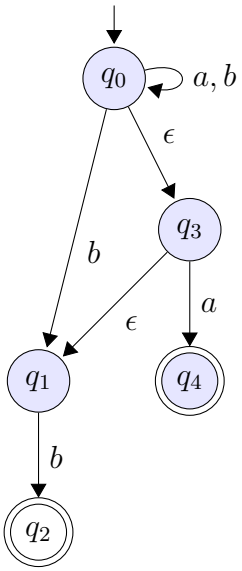


- Machine N_1 at state q_0 read a and arrives at state q_0. However N_1, at state q_0, can spawn new machines. N_1 spawns N_4 at state q_3. N_4 spawns another machine N_5 at state q_1.
- Machine N_2 at state q_3 processes character a and arrives at q_4. N_2 cannot spawn new machines since there are no ϵ-transitions from state q_4.
- Now N_3 at q_1 attempts to process a. But it fails!!! N_3 does not have an a-transition: it does not know what to do!!! Alas ... N_3 dies.

Altogether we now have the following machines:

$$N_1 \text{ at } q_0,\ N_2 \text{ at } q_4,\ N_4 \text{ at } q_3,\ N_5 \text{ at } q_1$$

(Don't forget that N_3 died.) Here's the state diagram again where shaded states mean at this point there's one at one machine at that state:



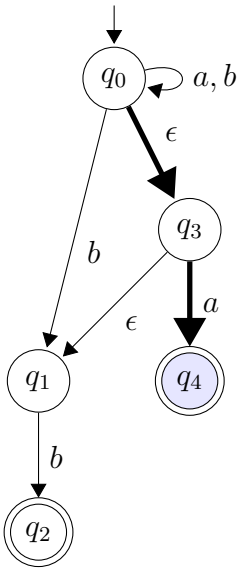
Now note that machine N_2 arrived at q_4 which is an accepting state. It does not matter that the other machines did not arrive at an accepting state. For the original machine N_1 to "win", it only requires one of its "descendents" arrives at an accepting state. Get it? So anyway, all the other machines disappear and only N_1 is left and N_1 says "YES" to the string a. Note that N_1 might have died during it's computation. It does not matter: in that case just think of N_1 as just sitting around and not doing any computation but simply waiting for all the descendents (including itself) to report back.

Sometimes you don't have to follow *all* the "descendents". You can see in this simple example that you if you go from q_0 to q_3 to q_4, the string processed is

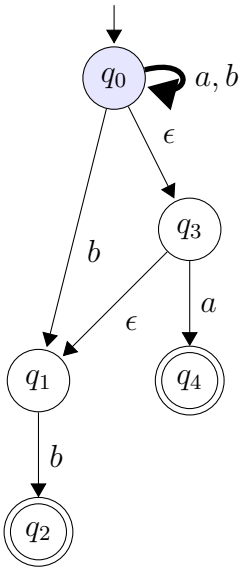
$$\epsilon a$$

which is of course

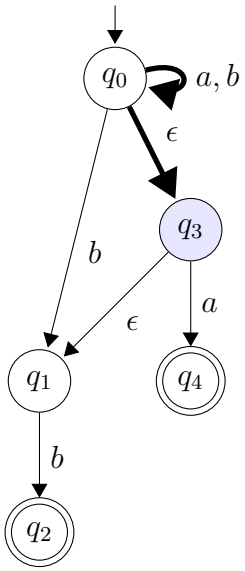
$$a$$



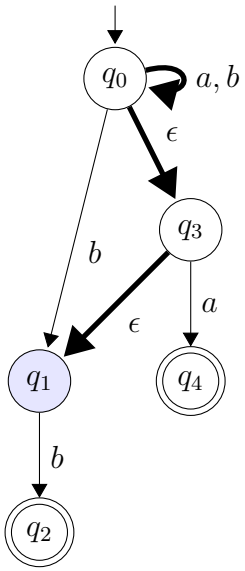
Here are the executions of all the possible machines. This machine processes a going from q_0 to q_0:



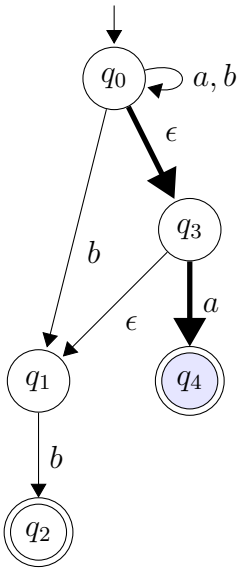
This machine processes a, ϵ going from q_0 to q_0 to q_3:



This machine processes a, ϵ, ϵ going from q_0 to q_0 to q_3, q_1:



And there's the one that I've already shown you:



However when the nondeterministic state diagram is complex and the string is long, you might not be so lucky: the "descendent" that accepts might be hard to find if you're not systematic.

Now let me run N_1 with input $w = b$. We start N_1 at q_0. Going along ϵ–transitions, we have the following:

- Machine N_1 is at q_0.
- N_1 spawns N_2 at q_3.
- N_2 spawns N_3 at q_1.

At this point, we have 3 machines:

$$N_1 \text{ at } q_0,\ N_2 \text{ at } q_3,\ N_3 \text{ at } q_1$$

Note that all 3 machines have the same input, i.e., their own copy of $w = b$ with no character read. I'm now done with the initialization.

Now these 3 machines are ready to process the first character of w (in fact the only character in w).

- N_1 notices that it can go to either q_0 or q_1!!! N_1 is greedy and wants *both* options. So N_1 goes to q_0, but before that it spawns N_4 and let N_4 go to q_1. Now N_1 sees that it can spawn off a new machine N_5 at state q_3 and N_5 can spawn a new machine N_6 at q_1. N_4 (at q_1) does not have any ϵ–transitions so N_4 does not produce new machines.
- N_2 at q_3 begin to process b ... but it does not know what to do with b!!!

N_2 dies!!!

- N_3 at q_1 reads b and go to state q_2. N_3 cannot spawn off a new machine since there are no ϵ–transition out of q_2.

Altogether, we now have the following machines:

$$N_1 \text{ at } q_0, \quad N_3 \text{ at } q_2, \quad N_4 \text{ at } q_1, \quad N_5 \text{ at } q_3, \quad N_6 \text{ at } q_1$$

Remember that we only care that *some* machine accepts at the end of processing all the characters of the input. If you look at the above, you'll see two machines at q_1. I'll ignore N_6 – obviously N_6 will behave like N_4 from this point on. So I focus on these machines:

$$N_1 \text{ at } q_0, \quad N_3 \text{ at } q_2, \quad N_4 \text{ at } q_1, \quad N_5 \text{ at } q_3$$

Note they have no more characters to process – end-of-input is reached. Is there at least one "winner"? Yes. N_3 manage to arrive at q_2 which is an accepting state. So the original machine N_1 says "YES" and accept b.

At this point we know that the language accepted by N_1, i.e., $L(N_1)$ does not contain ϵ, but contains both a and b.

Get the idea? You "bootup" the machine at the start state. Before the machine process any character, it greedily spawns of greedy descendents along ϵ-transitions. This ends the initialization. You have a collection of machines.

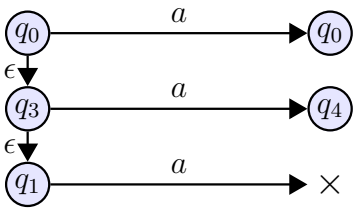
Next, each machine you have, let it process the next character from the input string. Remember that there might be multiple transtions for this character. from the input and along as many transitions labeled x as they can –there might be more than one transition labeled with x. Note that there might be a machine at a state where it does not know what to do with x. In that case, the machines crashes and dies. After reading that character x, they all greedily spawns off greedy descendents along ϵ–transitions.

Repeat the above until all characters in the input are processed.

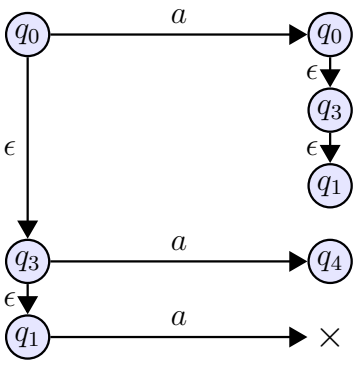
The original machine will accept the word if (at least) one of its descendents accept. (Remember: even if the original machine "dies" in the execution, it will be revived.)

Suggestion: To be organized, I suggest you draw your computation this way. Whenever you spawns machine, line them up vertically like this. Using the above NFA as example, the initialization will result in this diagram:

In other words, after initialization, there are 3 machines. When they all read a character, say a, draw horizontal arrows:



Draw an $\times$ when a machine crashes. Then you let the machine at q_0 and q_4 spawn off new machines along *epsilon*–transitions, drawing the ϵ-arrows vertically again:



Repeat. You will find this more organized and harder to miss things.

Definition 12.4.1. If the NFA is denoted N, then $L(N)$ is the language accepted by N, in other words $L(N)$ is the set of **all** strings over Σ accepted by N.

Exercise 12.4.2. Is b accepted by the above NFA? Is ba accepted? Is bab accepted? What is the language accepted by the NFA with the above transition diagram. □

Exercise 12.4.3. Draw an NFA diagram that accepts the following language of strings over $\Sigma = \{a, b\}$

$$L = \{\}$$

Compare with a DFA. □

Exercise 12.4.4. Draw an NFA diagram that accepts the following language of strings over $\Sigma = \{a, b\}$

$$L = \{a\}$$

Compare with a DFA. □

Exercise 12.4.5. Draw an NFA diagram that accepts the following language of strings over $\Sigma = \{a, b\}$

$$L = \{\epsilon\}$$

Compare with a DFA. □

Exercise 12.4.6. Draw an NFA diagram that accepts the following language of strings over $\Sigma = \{a, b\}$

$$L = \{\epsilon, a\}$$

Compare with a DFA. □

Exercise 12.4.7. Draw an NFA diagram that accepts the following language of strings over $\Sigma = \{a, b\}$

$$L = \{a, b\}$$

Compare with a DFA. □

Exercise 12.4.8. Draw an NFA diagram that accepts the following language of strings over $\Sigma = \{a, b\}$

$$L = \{a, a^2\}$$

Compare with a DFA. □

Exercise 12.4.9. Draw an NFA diagram that accepts the following language of strings over $\Sigma = \{a, b\}$

$$L = \{ab, ba\}$$

Compare with a DFA. □

Exercise 12.4.10. Draw a state diagram of an NFA that accepts the language of strings over $\Sigma = \{a, b\}$ that contains aaa, i.e.,

$$L = \{ x \in \Sigma^* \mid x = ya^3z \quad \text{for some} \quad y, z \in \Sigma^* \}$$

Compare with a DFA. □

Exercise 12.4.11. Draw a transition diagram of an NFA that accepts the language of strings over Σ that does not contain aaa. □

Exercise 12.4.12. Draw a transition diagram of an NFA that accepts the language of strings over $\Sigma = \{a, b\}$ that contains $abab$. Compare with a DFA. □

Exercise 12.4.13. Draw a transition diagram of an NFA that accepts the language of strings over $\Sigma = \{a, b\}$ where the number of a is exactly $\equiv 1 \pmod 4$. So the number of as can be $1, 5, 9, \ldots$. Compare with a DFA. □

Exercise 12.4.14. Let M be an DFA. Is it clear that the diagram of M is also a diagram for an NFA, say call it N? If is clear that $L(N) = L(M)$? This is all very informal. That's because we don't have a formal definition of an NFA yet other than a diagram. (Later you'll see that the converse is true.)

12.5 Nondeterministic Finite Automata debug: nfa.tex

Now I'll give you the formal definition of an NFA. Read the following definition *very* carefully and make sure the definition models the above examples of NFA state diagrams.

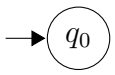
nondeterministic finite automata NFA

Definition 12.5.1. N is a **nondeterministic finite automata NFA** if $N = (\Sigma, Q, q_0, F, \delta)$ where

1. Σ is a finite set. This is called the **alphabets**. — alphabets
2. Q is a finite set. The elements are called **states**. — states
3. $q_0 \in Q$ is called the **initial state** or **start state**. — initial state, start state
4. $F \subseteq Q$ and elements of F are called **accept states** or **final states**. — accept states, final states
5. $\delta : Q \times \Sigma_\epsilon \to P(Q)$. This is called the **transition function** of this N. — transition function

In the above, $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $P(Q)$ is the powerset of Q.

Exercise 12.5.1. Here's an NFA state diagram for $\emptyset$ for $\Sigma = \{a, b\}$:



Write down the formal definition of this NFA.

SOLUTION.

The formal definition of this NFA is $(\Sigma, Q, q_0, \delta, F)$ where

- $\Sigma = \{a, b\}$
- $Q = \{q_0\}$
- δ is the function

$$\delta : Q \times \Sigma_\epsilon \to P(Q)$$

given by

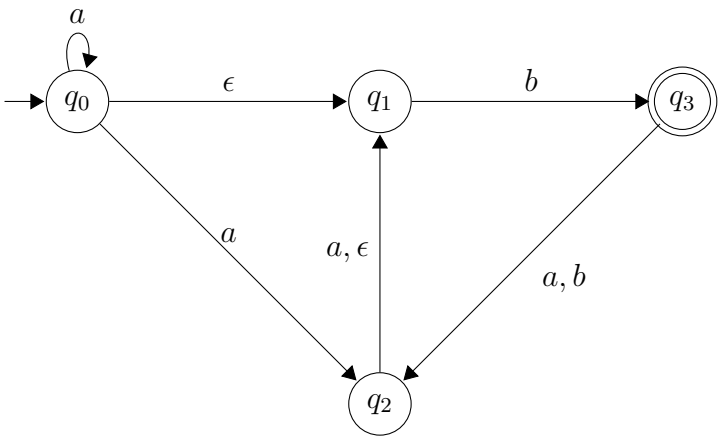
$$\delta(q_0, \epsilon) = \{\}$$
$$\delta(q_0, a) = \{\}$$
$$\delta(q_0, b) = \{\}$$

Exercise 12.5.2. Draw the (obvious) NFA N for the words of $\{a, b\}^*$ containing aba. Write down the formal definition of N. □

Exercise 12.5.3. Draw the (obvious) NFA N for the words of $\{a, b\}^*$ containing *abab* or *baba*. Write down the formal definition of N.

□

Exercise 12.5.4. Write down the formal definition of this NFA:



Compute all the states reached by the following words:

(a) ϵ
(b) a
(c) b
(d) aa
(e) ab
(f) ab
(g) ba
(h) bb
(i) Can you find a "simpler" NFA that accepts the same words as the above NFA? "Simpler" means "fewer states".

Definition 12.5.2. Let $S \subseteq Q$ and $a \in \Sigma_\epsilon$. Note that we can define

$$\delta(S, a) := \bigcup_{q \in S} \delta(q, a)$$

The following is a computation: Let $a \in \Sigma_\epsilon$.

$$(S, ax) \vdash (\overline{\delta(S, a)}, x)$$

where the notation $\overline{\quad}$ means the following: If S is a set of states, then we want to include those states reachable from S through an ϵ–transition, i.e.,

1. $q \in S \implies q \in \overline{S}$
2. $q \in \overline{S} \implies \delta(q, \epsilon) \subseteq \overline{S}$

Informally, you think of it this way:

1. Let S_0 be the set of all state S.
2. Let S_1 be the set of states in S_0 as well as the states $q' \in \delta(q, \epsilon)$ where $q \in S_0$.
3. In general, let S_i be the set of states in S_{i-1} as well as the states $q' \in \delta(q, \epsilon)$ where $q \in S_{i-1}$.

At some point S_i will stop changing, say S_n. Then S_n is the set of all states that are **reachable** from S, i.e., viewing the state diagram as a directed graph, S_n are the states q' such that there is a path from some $q \in S$ to q'. In other words the computation of $\overline{S}$ is a **reachability problem** in directed graph. If you remove from the graph where the transitions are not labeled ϵ (only ϵ–transitions are retained), then you are also computing the **transitive closure** of S. (This is from discrete 1.)

reachable

reachability problem

transitive closure

Easy, right? This is sometimes called the **ϵ–closure** of the set S.

ϵ–closure

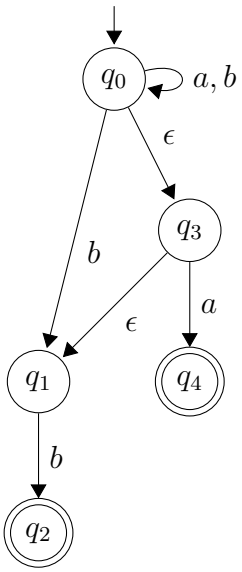
As an algorithm, here's how you compute the ϵ–closure of S:

```
ALGORITHM: EPSILON-CLOSURE
INPUT: S - a set of states
       delta - a transition function

let DONE be an empty set
put all the states in S into BOUNDARY
let NEW be an empty set
while 1:
    for q in BOUNDARY:
        put states in delta(q, epsilon) which are not in DONE
```

```
        and not in BOUNDARY into NEW
    put all the states in BOUNDARY into DONE
    if NEW is empty:
        break the loop
    else:
        put all the states in NEW into BOUNDARY

return DONE
```

For states not in S, each of them is placed in NEW and then in BOUNDARY and then DONE. Assume that adding and removing from NEW, BOUNDARY, DONE can be done in $O(1)$.

Exercise 12.5.5. Refer to our first NFA again:



- What is $\overline{\{\}}$?
- What about $\overline{\{q_0\}}$?
- What about $\overline{\{q_1\}}$?
- What about $\overline{\{q_2\}}$?
- What about $\overline{\{q_4\}}$?
- What about $\overline{\{q_0, q_1\}}$?
- Write down $\overline{S}$ for all subsets S of $\{q_0, ..., q_4\}$.

□

Definition 12.5.3. Let N be an NFA. The definition of $\vdash^*$ from $\vdash$ is just like the case of the DFA. A string $x \in \Sigma^*$ is accepted by the above N if

$$(\overline{\{q_0\}}, x) \vdash^* (S, \epsilon)$$

and $S \subseteq Q$ contains at least one accepting state, i.e., $S \cap F \neq \emptyset$.

Here's an example to prove the power of the NFA ...

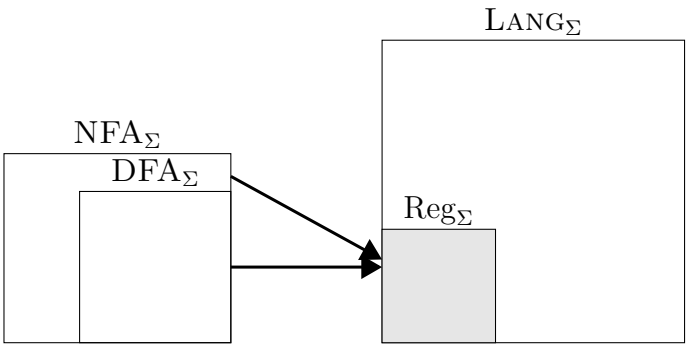
Exercise 12.5.6. Design an NFA N that accepts words in $\{a, b\}^*$ which ends in either aba or aba. Write down the formal definition of N. □

12.6 DFAs are as Powerful as NFAs debug: dfa-as-powerful-as-nfa.tex

Theorem 12.6.1.
(a) If M is a DFA*, then there is an* NFA *say N such that $L(M) = L(N)$.*
(b) If N is an NFA*, then there is a* DFA *say D such that $L(N) = L(D)$.*

I will call this the "NFA = DFA" theorem. (I made this name up myself, it's not standard).

So although the NFA is more flexible than the definition of a DFA (so that every DFA is an NFA almost by definition), it turns out the collection of languages described by DFAs and NFAs are actually the same – you can't describe a larger collection of languages. Here a picture:



Recall that a language is regular if it's the language accepted by a DFA. So Reg_Σ is the set of hits of DFA_Σ in the above diagram. The set of machines, NFA_Σ, hits the same set of languages as Reg_Σ – you don't get more.

The proof of part (a) of the NFA = DFA is easy and I'll leave it as an exercise. Make sure you do it on your own.

Now to prove part (b) of NFA = DFA. Believe it or not, the proof is easy too. Furthermore, the prove is constructive. The point is to look very carefully at the definitions above especially of the computation and to massage the objects in the definition.

Let $N = (\Sigma, Q, q_0, F, \delta)$ be an NFA. Don't forget that the transition function δ has the following form:

$$\delta : Q \times \Sigma_\epsilon \to P(Q)$$

where $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $P(Q)$ is the powerset of Q. Furthermore, I'm extending the δ to accept (S, x) where S is a *set* of states. This is done in the following

(obvious!) way:

$$\delta(S, c) = \bigcup_{q \in S} \delta(q, c)$$

where $S \subset Q$ and $c \in \Sigma_\epsilon$.

Suppose it's possible to construct a DFA

$$M = (\Sigma, Q^{\text{DFA}}, q_0^{\text{DFA}}, F^{\text{DFA}}, \delta^{\text{DFA}})$$

so that $L(M) = L(N)$. What would M look like?

Take a look at the definition of computation:

$$(S, cx) \vdash (\overline{\delta(S, c)}, x)$$

where $S \subseteq Q$ and $c \in \Sigma_\epsilon$. Now look that definition of the computation of a DFA that we would like to have:

$$(q, cx) \vdash (\delta^{\text{DFA}}(q, c), x)$$

Doesn't that tell you that the DFA should somehow have **sets** as states?!?! In particular, the sets are **subsets** of Q. See it? Just to repeat, Q^{DFA} is a set of set of states. Therefore $Q^{\text{DFA}} \subseteq P(Q)$. Maybe we need $Q^{\text{DFA}} = P(Q)$??? We'll see.

What about δ^{DFA} for our guess? When you compare the above computations symbolically, doesn't that say that for if $S \in Q^{\text{DFA}}$ and $a \in \Sigma$, then

$$\delta^{\text{DFA}}(S, c) \stackrel{\text{def}}{=} \overline{\delta(S, c)} = \overline{\bigcup_{q \in S} \delta(q, c)}$$

In that case,

$$(S, ax) \vdash (\overline{\delta(S, c)}, x)$$

does become:

$$(S, ax) \vdash (\delta^{\text{DFA}}(S, c), x)$$

Viola!!!

So far we have Q^{DFA} and δ^{DFA}. Think about it. We start our machine with q_0. But our Q^{DFA} is a set of set of states. So the initial state is not q_0 but $\overline{\{q_0\}}$. Therefore we define the initial state of our DFA as $q_0^{\text{DFA}} \stackrel{\text{def}}{=} \overline{\{q_0\}}$. And of course $q_0^{\text{DFA}} \in Q^{\text{DFA}}$, at least if we let $Q^{\text{DFA}} = P(Q)$.

What about the accepting states F^{DFA}? You already know that a string is accepted by the NFA if you have at least one computation leading to one accepting state. Formally our above definition says that x is accepted if

$$(\overline{\{q_0\}}, x) \vdash^* (S, \epsilon), \quad S \cap F \neq \emptyset$$

Well ... obviously we define F^{DFA} to be those sets of state that contain at least one accepting state in F of the NFA:

$$F^{\text{DFA}} \stackrel{\text{def}}{=} \{S \in Q^{\text{DFA}} \mid S \cap F \neq \emptyset\}$$

I mean, what else can it be?!?

As an example of a specific state of our new DFA, note that since the states of the new DFA are the subset of Q (the states of the NFA) and the emptyset $\emptyset$ is an element of $P(Q)$, our DFA has a state labeled as $\{\}$. Note that by our definition above

$$\delta^{\text{DFA}}(\emptyset, c) = \emptyset$$

(Right?)

So let's begin the formal proof of NFA = DFA part (b).

Proof. Let $N = (\Sigma, Q, q_0, F, \delta)$ be an NFA. Given a subset S of Q, $\overline{S}$ denotes the ϵ-closure of S. Define $M = (\Sigma, Q^{\text{DFA}}, q_0^{\text{DFA}}, F^{\text{DFA}}, \delta^{\text{DFA}})$ as follows:

- $Q^{\text{DFA}} = P(Q)$
- $q_0^{\text{DFA}} = \overline{\{q_0\}}$
- $F^{\text{DFA}} = \{S \in Q^{\text{DFA}} \mid S \cap F \neq \emptyset\}$
- $\delta^{\text{DFA}} : Q \times \Sigma \to Q$ is defined as follows: Let $S \in Q^{\text{DFA}}$ and $c \in \Sigma$. Then

$$\delta^{\text{DFA}}(S, c) = \overline{\bigcup_{q \in S} \delta(q, c)}$$

Then

$$L(M) = L(N)$$

(Details are left to the reader.) □

The above NFA-to-DFA construction is sometimes called the **subset construction** or the **powerset construction**.

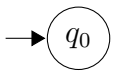
subset construction
powerset construction

I hope you recall that $|P(Q)| = 2^{|Q|}$. That means that the resulting DFA is big when compared against the original NFA. Here size is measured in terms

of the number of states. We'll see later that there is an algorithm to simplify DFAs.

Of course if there is a state that cannot be reached from the initial node, you can just remove it. So we usually don't draw *all* the subsets of Q (of the NFA) for the DFA. We usually start with the start state of the DFA (i.e., the ϵ–closure of the start state of the NFA) and keep adding DFA states until the DFA is completed. So in your NFA has 5 states. The complete subset construction gives you a DFA with $2^5 = 32$ states. However if you only include states reachable for the start state of the DFA, the DFA might be smaller; the other states cannot be reached the start state of the DFA and hence does not take part in the computation of a string. (They can be included of course.)

Exercise 12.6.1. Here's an NFA for $\emptyset$ for $\Sigma = \{a, b\}$:



Convert it to a DFA using the subset construction.

Here's the solution. Let δ denote the transition function of N. Note that

$$\begin{aligned}\delta(q_0, \epsilon) &= \{\}\\ \delta(q_0, a) &= \{\}\\ \delta(q_0, b) &= \{\}\end{aligned}$$

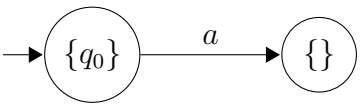
First of all the states are labeled as all the subsets of $\{q_0\}$.

The start state is the ϵ-closure of $\{q_0\}$. However in N, there are no ϵ–transitions out of q_0. So the ϵ-closure of $\{q_0\}$ is in fact $\{q_0\}$, i.e. $\overline{\{q_0\}} = \{q_0\}$ The DFA is now this:

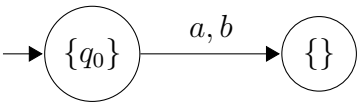
Now I will compute the a–transition of the state $\{q_0\}$. Let δ^{DFA} denote the transition function of the DFA that we're building. Then

$$\begin{aligned}\delta(\{q_0, a\}) &= \overline{\bigcup_{q \in \{q_0\}} \delta(q, a)}\\ &= \overline{\delta(q_0, a)}\\ &= \overline{\emptyset}\\ &= \emptyset\end{aligned}$$

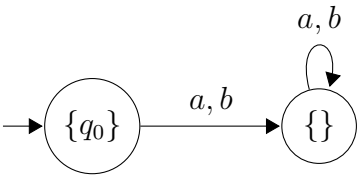
The (incomplete) DFA now looks like this:



Using the same reasoning we have



It's easy to see that in the DFA, the a– and b–transitions from the state $\{\}$ goes back to itself. Therefore the completed DFA is this:



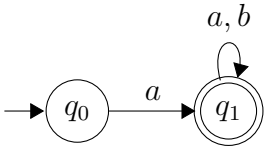
□

Note that the above DFA does accept the same language as the NFA we started with, i.e., $\{\}$.

Note that the subset construction does not necessarily give the simplest DFA. Clearly the simplest DFA that accepts $\{\}$ is

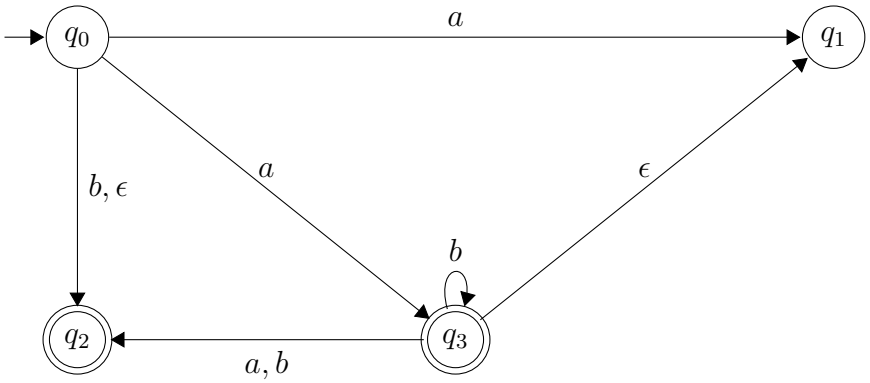
"Simplest" meaning "fewest states".

Exercise 12.6.2. Here's the NFA for the words in $\{a, b\}^*$ beginning with a:



Convert it to a DFA using the powerset construction. □

Exercise 12.6.3. Using the powerset construction, find a DFA M such that $L(M) = L(N)$ where N is the following NFA:



□

Exercise 12.6.4. Here's the formal definition of an NFA $N = (\Sigma, Q, q_0, F, \delta)$ where $\Sigma = \{a, b\}$, $Q = \{q_0, q_1, q_2, q_3, q_4\}$, $F = \{q_4\}$ and δ is given by the following table (the cases where the δ–value is an empty set are omitted):

$q \in Q$	$x \in \Sigma_\epsilon$	$\delta(q, x)$
q_0	ϵ	$\{q_0, q_1\}$
q_0	b	$\{q_2\}$
q_1	ϵ	$\{q_3\}$
q_1	ϵ	$\{q_2\}$
q_1	a	$\{q_0\}$
q_1	b	$\{q_2, q_4\}$
q_2	ϵ	$\{q_4\}$
q_2	b	$\{q_0, q_4\}$
q_3	a	$\{q_4\}$
q_4	ϵ	$\{q_1, q_2, q_3\}$

Construct a DFA M such that $L(M) = L(N)$. □

Exercise 12.6.5. Design a DFA that accepts words containing *abab* or *baba*. [Hint: It's easier to first design two NFAs, one accepting words containing *abab* and another accepting words containing *baba*. Next design an NFA accepting words containing *abab* or *baba*. Finally convert this NFA to a DFA using the subset construction. I'll come back and talk about closure operators that will give you a general method for constructing DFAs and NFAs whenever the describe of the language contains the word "or".] □

12.7 Closure Rules debug: closure.tex

Now suppose if you want to show L is regular, you can try to construct a DFA for it. That would mean that you're always starting from scratch. In this section we'll adopt a more sophisticated approach. In particular, we see how to show a language is regular by looking at other language(s).

We fix a alphabet Σ. Every language over Σ is a subset of Σ^*. Therefore the set of *all* languages over Σ is the powerset of Σ^*, i.e., $P(\Sigma^*)$. In this collection of languages is the set of regular languages over Σ. Let's call this set Reg_Σ. Therefore $\text{Reg}_\Sigma \subsetneq P(\Sigma^*)$.

Reg_Σ

Here's why the above are called **closure rules**. Consider all languages over Σ. This is $P(\Sigma^*)$. Right? Remember powerset? You can think of $\cup$ (union) as a binary operation on $P(\Sigma^*)$. In other words let $L, L' \in P(\Sigma^*)$, then $L \cup L' \in P(\Sigma^*)$. This is obvious.

closure rules

Now suppose $L, L' \in \text{Reg}_\Sigma$. We want to know if $L \cup L' \in \text{Reg}_\Sigma$. This is true, but not immediate. If this is the case, then we say that $\cup$ is a **closed binary operation** on Reg_Σ. This is similar to saying $+$ is a closed binary operator on $\mathbb{Z}$: adding two integers will give you an integer. However, note that the binary operator / division on $\mathbb{Z}$ is not closed: $5/2 = 2.5$ which is *not* in $\mathbb{Z}$. Obviously working with Reg_Σ is harder than with $\mathbb{Z}$!

closed binary operation

Why are closure rules useful? Such rules will allow us to re-use previously known regular languages. For instance if $\cup$ is a closed operator on regular languages and you want to study

$$L'' = \{x \in \{a,b\}^* \mid \text{ number of } a \text{ in } x \text{ is } \equiv 1 \pmod 5 \text{ or } x \text{ contains } a^3\}$$

then, assuming you see the decomposition

$$L'' = L \cup L'$$

where

$$\begin{aligned} L &= \{x \in \{a,b\}^* \mid \text{number of } a \text{ in } x \text{ is } \equiv 1 \pmod 5\} \\ L' &= \{x \in \{a,b\}^* \mid x \text{ contains } a^3\} \end{aligned}$$

are both regular, then you know immediately that L is also regular. You can than apply all tools and techniques in regular languages to study L. In fact you can also study L'' by studying the components of its decomposition: L and L'.

The above is a general philosophy is used throughout any form of study, computer science, math, sciences, etc. You decompose a problem and you ask which "universe" you are in so that you don't use the wrong formulas or theorems.

We will look at union, intersection, set difference, etc. Note that if L is a language over Σ, complementation of L will be with respect to Σ^*, i.e., $\overline{L} = \Sigma^* - L$. Here are some more operators:

Definition 12.7.1. Let L and L' be languages over Σ.

- The **concatenation** of L and L' is defined as follows: concatenation

$$LL' = \{z \in \Sigma^* \mid z = xx' \text{ for some } x \in L \text{ and } x' \in L'\}$$

- We can define L^n for $n \geq 0$ as follows:

$$L^n = \begin{cases} \{\epsilon\} & \text{if } n = 0 \\ L^{n-1}L & \text{if } n > 0 \end{cases}$$

- The **Kleene star** of L, written L^* is defined by Kleene star

$$L^* = \bigcup_{n \geq 0} L^n = \{x_1 \cdots x_n \mid x_i \in L \text{ for } i = 1, \ldots, n\}$$

You can easily make up your own operators. For instance here's one that you have seen before: If L is a language L^R is the reversal of L, i.e., the words in L^R are the reversed of L. For instance if $abaa \in L$, then $aaba \in L^R$. You can then ask this question: if L is regular, is L^R also regular? Note that $-^R$ is a unary operator. I'll give you another one: Let L be a language. Define $\frac{1}{2}L$ be the language of half-words of L where a half-word is just the *first half* of the word. For instance if $w = aaba$, then $\frac{1}{2}w$ is aa. For the case when w has odd length, $\frac{1}{2}w$ is the first $\lfloor |w|/2 \rfloor$ characters. For instance $\frac{1}{2}aabbbba = aab$. You can ask this question: If L is regular, is $\frac{1}{2}L$ also regular?

For binary language operator, here's an example: Suppose L and L' are two languages, define $\text{Zip}(L, L')$ to be the language of words w such that

$$w = x_1 y_1 x_2 y_2 x_3 y_3 \cdots$$

where $x = x_1 x_2 x_3 \cdots$ is a word in L and $y = y_1 y_2 y_3 \cdots$ is a word in L'. (If x and y has different lengths, you just add ϵ's so that they have the same length.) You can then ask if $\text{Zip}(L, L')$ is regular if L, L' are regular.

You can create your oan unary and binary operators on languages are ask if your operator is closed in the collection of regular languages.

We will prove the following closure rules. In the proofs you will see the power of NFAs.

Theorem 12.7.1. *Let L, L' be languages.*

(a) L, L' *regular* $\implies$ $L \cup L'$ *regular*
(b) L, L' *regular* $\implies$ $L \cap L'$ *regular*
(c) L *regular* $\implies$ $\overline{L}$ *regular*
(d) L, L' *regular* $\implies$ $L - L'$ *regular*
(e) L, L' *regular* $\implies$ LL' *regular*
(f) L *regular* $\implies$ L^* *regular*

In other words, regular languages are closed under union, intersection, complement, set difference, concatenation, and Kleene star. □

12.8 Closure: Intersection

Let's prove closure of $\cap$ on Reg_Σ.

Since we assumed that L and L' are regular, by definition, there must be DFAs M and M' such that $L = L(M)$ and $L' = L(M')$. To show that $L \cap L'$ is regular, we must construct a DFA M'' such that $L(M'') = L \cap L'$. Right? This is all by definition.

Of course M'' must be constructed from M and M'. What do you want M'' to do? Suppose you're given a string $x \in \Sigma^*$. By definition a string x is in L'' if and only if x is in both L and L'. This means that M'' must be a machine that will accept x if and only if x is accepted by both M and M'. This means that M'' must be able to "run" M and M' separately and must "know" if M and M' accepted the x or not. So in other words suppose that when you run M and M' with x, you get the following sequence of states:

$$\begin{aligned} M &: q_0 \to s_1 \to s_2 \to \cdots \to s_n \\ M' &: q'_0 \to s'_1 \to s'_2 \to \cdots \to s'_n \end{aligned}$$

where s_i are states in M and s'_i are states in M'. Of course M'' is a single DFA and so as you run M'' you should only see a single sequence of states, not two. How would you create a single sequence of states from the above? What if we do this:

$$M'' \ (???) : \ q_0, q'_0 \to s_1, s'_1 \to s_2, s'_2 \to \cdots \to s_n, s'_n$$

So we want "q_0, q'_0" to be a single "thing". AHA!!! Tuples!!!

$$M'' : \ (q_0, q'_0) \to (s_1, s'_1) \to (s_2, s'_2) \to \cdots \to (s_n, s'_n)$$

Note in particular that (s_i, s'_i) is not the same as (s'_i, s_i). This is the first hint that the states of M'' must be

$$Q'' = Q \times Q' = \{(q, q') \mid q \in Q, q' \in Q'\}$$

i.e., the product of Q and Q', the set of states of M and M'.

How should we initialize our M'' engine? Obviously $q''_0 = (q_0, q'_0)$ where q_0 and q'_0 are the initial states of M and M' respectively.

When is x accepted? Obviously if the final state after running x through the M'' is (q, q'), then

$$x \in L'' \iff x \in L \text{ and } x \in L'$$

forces us to say that (q, q') is an accepting state exactly when $q \in F$ and $q' \in F'$ where F, F' are sets of accepting states of M,M' respectively:

$$(q, q') \in F'' \iff q \in F, \;\; q' \in F'$$

This means that we must have

$$F'' = F \times F'$$

Furthermore it's now clear how to define δ'' the transition function of M'' in terms of δ, δ' the transition functions of M, M' respectively. If the symbol is $a \in \Sigma$ and δ and δ' have the following effect:

$$q \mapsto r$$
$$q' \mapsto r'$$

for an a–transition, then δ'', the transition function of M'' (if M'' exists) must have the following effect

$$(q, q') \mapsto (r, r')$$

In other words

$$\delta''\bigg((q, q'), a\bigg) = \bigg(\delta(q, a), \;\; \delta'(q', a)\bigg)$$

The above gives the intuition on how to construct M''. Now to present the formal proof:

Proof. Let $M = (\Sigma, Q, q_0, F, \delta)$ and $M' = (\Sigma, Q', q_0', F', \delta')$ such that $L = L(M)$ and $L' = L(M')$.

Define $M'' = (\Sigma, Q'', q_0'', F'', \delta'')$ as follows:

- $Q'' = Q \times Q'$
- $q_0'' = (q_0, q_0')$. Note that $q_0'' \in Q''$.
- $F'' = F \times F'$. Note that $F'' \subseteq Q''$. [Prove this yourself.]
- $\delta'' : Q'' \times \Sigma \to Q''$ by

$$\delta''((q, q'), a) = (\delta(q, a), \delta'(q', a))$$

where $a \in \Sigma$ and $(q, q') \in Q''$. Note that δ'' is well-defined since $(\delta(q, a), \delta'(q', a)) \in Q''$. Recall that δ'' induces the function $\delta''^* : \Sigma^* \times Q'' \to Q''$

By definition, M'' is a DFA.

We now want to show that $L(M'') = L(M) \cap L(M')$. Let $x \in \Sigma^*$. Then

$$\begin{aligned} x \in L(M'') &\iff \delta''^*(q_0'', x) \in F'' \\ &\iff \delta''^*((q_0, q_0'), x) \in F \times F' \\ &\iff (\delta^*(q_0, x), \delta'^*(q', x)) \in F \times F' \\ &\iff \delta^*(q_0, x) \in F, \ \ \delta'^*(q_0', x) \in F' \\ &\iff x \in L(M), \ \ x \in L(M') \\ &\iff x \in L(M) \cap L(M') \end{aligned}$$

Hence $L(M'') = L(M) \cap L(M')$. □

So the proof is complete, right? Are you sure?

Lemma 12.8.1. *Assume the notation in the above proof. Then*

$$\delta''^*((q, q'), x) = (\delta^*(q, x), \delta^*(q', x))$$

for any $x \in \Sigma^*$, $q \in Q$ and $q' \in Q'$.

Proof. We will prove this by Mathematical Induction on $|x|$. In other words let $P(n)$ be the statement that for any string $x \in \Sigma^*$ with $|x| = n$, we have

$$\delta''^*((q, q'), x) = (\delta^*(q, x), \delta^*(q', x))$$

When $|x| = 0$, i.e., $x = \epsilon$, we have, by definition

$$\begin{aligned} \delta''^*((q, q'), x) &= (q, q') \\ \delta^*(q, x) &= q \\ \delta^*(q', x) &= q' \end{aligned}$$

Hence $\delta^*((q, q'), x) = (\delta^*(q, x), \delta^*(q', x))$. Therefore $P(0)$ holds.

Now we want to prove that for $n \geq 0$, if $P(n)$ is true, then $P(n+1)$ is also true. Now assume that

$$\delta^*((q, q'), x) = (\delta^*(q, x), \delta^*(q', x))$$

for any string $x \in \Sigma^*$ with $|x| = n \geq 0$. Consider a string $y \in \Sigma^*$ of length $n+1$. Then $y = ax$ where $a \in \Sigma$ and $x \in \Sigma^*$ with $|x| = n$. Then by the recursive definition of δ''^*

$$\delta''^*((q, q'), ax) = \delta''^*\Big(\delta''((q, q'), a),\ \ x\Big)$$

For clarity, since $\delta''((q, q'), a) \in Q'' = Q \times Q'$, we can write $\delta''((q, q'), a) = (\delta(q, a), \delta(q', a)) = (r, r')$ for some $r \in Q$ and $r' \in Q'$. Continuing the above,

$$\delta''^*((q, q'), ax) = \delta''^*((r, r'), x) = (\delta^*(r, x),\ \ \delta'^*(r', x))$$

using the induction hypothesis since $|x| = n$. Therefore substituting back the expression for r and r':

$$\begin{aligned}
\delta''^*((q, q'), y) &= \Big(\delta^*(r, x),\ \ \delta'^*(r', x)\Big) \\
&= \Big(\delta^*(\delta(q, a), x),\ \ \delta'^*(\delta(q', a), x)\Big) \\
&= (\delta^*(q, ax), \delta'^*(q', ax)) \\
&= (\delta^*(q, y), \delta(q', y))
\end{aligned}$$

Therefore the statement holds for any string y of length $n+1$.

By Mathematical Induction, the statement holds for any string. □

product construction

intersection construction

I will call this either the **product construction** (because we're using cross product to form the new states) or the **intersection construction**. Note that the above theorem does not just tell you that if L and L' are regular, then $L \cap L'$ is regular. The theorem actually includes the recipe for constructing the DFA for $L \cap L'$ from the DFAs of L and L'. In other words the proof is constructive. So the proof is very handy.

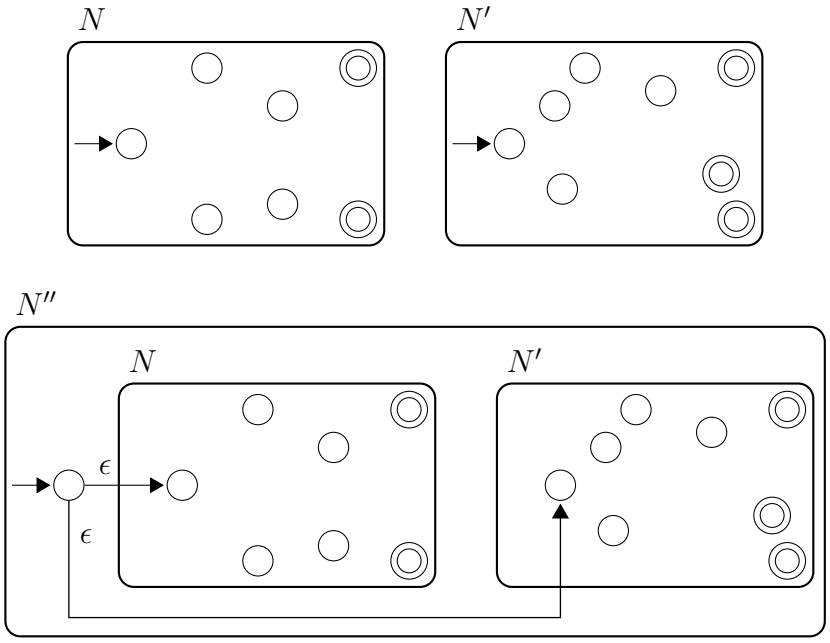
Exercise 12.8.1. Design a DFA that accepts strings which contains

- at least three 0's and
- contains 010.

□

12.9 Closure: Union

In this section, we will prove the closure rule for union. The idea is actually very simple. Suppose you have two DFAs N and N' (or DFAs):



Let $L = L(N)$ and $L' = L(N')$ where $N = (\Sigma, Q, q_0, F, \delta)$ and $N' = (\Sigma, Q', q_0', F', \delta')$.

Define $N'' = (\Sigma, Q'', q_0'', F'', \delta'')$ as follows:

- $Q'' = Q \cup Q' \cup \{q_0''\}$. Note that technically speaking Q and Q' are sets in different universes. We need to assume that we can put them both into the same universe in such a way that if $q \in Q$ and $q' \in Q'$, then $q \neq q'$ in this new universe. Furthermore q_0'' must obviously be an element in this universe distinct from any element in Q and Q'.
- q_0'' is already explained above
- $F'' = F \cup F'$
- Define $\delta'' : \Sigma \times Q'' \to Q''$ as follows: Let $q \in Q''$ and $a \in \Sigma$. Note that since $Q'' = Q \cup Q'$, either $q \in Q$ or $q \in Q'$. Furthermore since $Q \cap Q' =$, q cannot be in both Q and Q'. We define

$$\delta''(q, a) = \begin{cases} \delta(q, a) & \text{if } q \in Q \\ \delta'(q, a) & \text{if } q \in Q' \end{cases}$$

Note that by definition, N'' is an NFA.

Now prove that $L(N'') = L \cup L'$. (Exercise).

I'll call this the **union construction**.

union construction

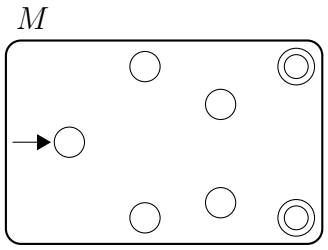
Exercise 12.9.1. Construct a DFA that accepts strings containing 0110 or 1001.

12.10 Closure: Complement

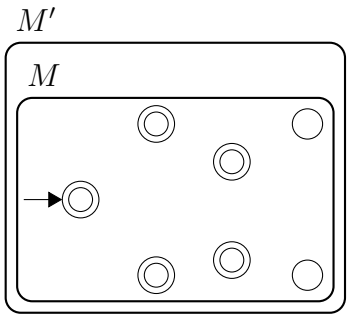
Given a DFA M, you can easily construct another DFA $\overline{M}$ such that

$$L(\overline{M}) = \overline{L(M)}$$

Here's what you do: you simply change the accept state to non–accept state and non–accept state to accept state. Diagrammatically, if this is M:



then M' is



WARNING: This method does not work when the construction is applied to an NFA.

Exercise 12.10.1. Construct a DFA that does not accept strings containing 0110.

12.11 Closure: Difference

This will be an exercise in the future.

Let L, L' be languages. Then

$$L - L' = L \cap \overline{L'}$$

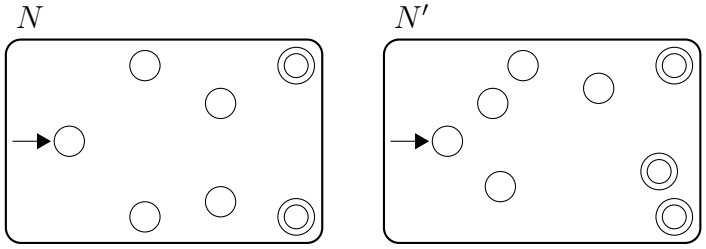
In other words, you use the complement construction and the product construction. That's it.

Exercise 12.11.1. Let $\Sigma = \{a, b\}$. Design a DFA that accepts the following languages:

$$L = \{w \in \Sigma^* \mid w \text{ contains 4 } a\text{'s and does not contain } a^3\}$$

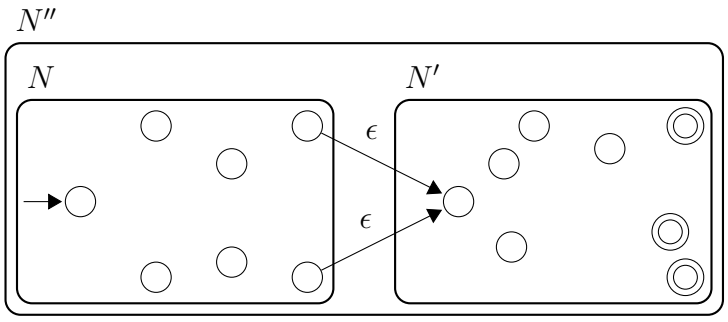
12.12 Closure: Concatenation

Let $L = L(N)$ and $L' = L(N')$ where N, N' are NFAs (or DFAs). I'm going to construct asn NFA N'' that accepts LL'. (The construction also works when N and N' are DFAs.)



The intuition is as follows. For a string $x \in \Sigma^*$, $x \in LL'$ if and only if x is accepted by N and after that if you run it on N', it is again accepted.

But how do you do "... after that if you run it on N'..."? Easy. Join up the two machines using ϵ-transitions. Just add ϵ-transitions from the accepting states of N to the starting state of N'. Of course the accept states of N are *not* accept states in N''. Where do you start? At the starting state of N. Therefore the start state of N'' is the start state of N. When do you finally accept? When the string reaches the an accept state of N'. Therfore the accept states of N'' are the accept states of N'. Here's N'':



It's clear that N'' constructed above is an NFA. Furthermore, it's clear that $L(N'') = LL'$. Since we know that any language accepted by an NFA is also accepted by a DFA, we're done.

Don't you think NFAs are nice?

I'll call this the **concatenation construction**.

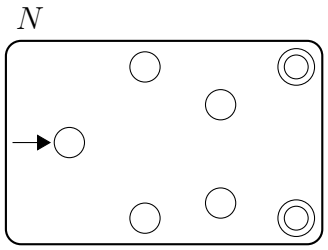
concatenation construction

Exercise 12.12.1. Construct a DFA that does not accept strings $w = w_1w_2$ where w_1 contains aba and w_2 begins with $bbaa$.

12.13 Closure: Kleene Star

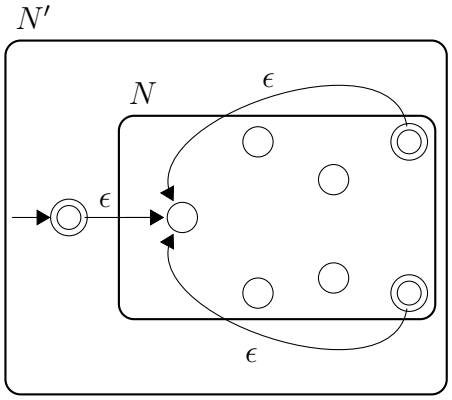
The construction of a DFA of L^* given a DFA of L is simple once you have seen the construction of the machine for concatenation of languages.

Given an NFA N (or DFA)



we will be constructing a new NFA N' such will accept $L(N)^*$.

First N' contains N. Next you add a new state to N' and make it the initial state of the new machine and you make this new state an accepting state. Why? Because, then N' will accept ϵ. Of course the initial state of N is now not an initial state in N'. Now join the new initial state to the old initial state of N with an ϵ–transition. Next you join the accept states of N, if there are any, to the old initial state of N with ϵ-transitions. Why? Because we are actually imitating the concatenation construction: if a string is accepted by N, then the string is still accepted by N'. But if the string is made up of xx' where x and x' are both accepted by N, then the ϵ–transitions we just added will allow N' to test acceptance on x' starting from the beginning of N. Right?



That's it. I'll call this the **Kleene star construction**.

Kleene star construction

Exercise 12.13.1. State precisely the Kleene star construction: Let $N =$

$(\Sigma, Q, q_0 F, \delta)$. let $N* = (\Sigma, Q', q_0', F', \delta')$ be the Kleene star construction using N. Of course q_0' is a new state that is not in Q, i.e., $q_0' \notin Q$. State Q', F', δ' precisely. □

Theorem 12.13.1. *Let N be an NFA and N' be the NFA obtained by the Kleene star construction. Then*

$$L(N') = L(N)^*$$

Exercise 12.13.2. Construct an NFA that accepts $\{a, aba, abba\}^*$. □

Exercise 12.13.3. Is it possible to create another Kleene star construction that has only one accept state?

Exercise 12.13.4. Let $\Sigma = \{a, b\}$. Construct an NFA that accepts

$$(\{w \in \Sigma^* \mid w \text{ ends with } bb\} \cdot \{w \in \Sigma^* \mid w \text{ starts with } aa\})^*$$

□

Exercise 12.13.5. Let $L = \{aba, bab\}$. Construct a DFA that accepts L^*.
□

12.14 Closure: Miscellaneous

There are a lot more closure rules/operations. Here one:

Exercise 12.14.1. Prove or disprove the following: L regular $\implies$ L^R regular.

12.15 Closure: Homomorphism

WATCHOUT! The following is not in the textbook!

Definition 12.15.1. Let $f : \Sigma \to \Sigma'$ be a map. Note that f induces a map $f^* : \Sigma^* \to \Sigma'^*$ which is defined as follows: First of all for $x \in \Sigma$,

$$f^*(x) = \begin{cases} f(x') \cdot f(x'') & \text{if } x = x'x'' \neq \epsilon \text{ where } x' \in \Sigma, x'' \in \Sigma^* \\ \epsilon & \text{if } x = \epsilon \end{cases}$$

(For the math experts: f^* is a semigroup homomorphism. You can also extend it to be group homomorphism on corresponding free group. The $(\cdot)^*$ is a function from sets to semigroups or to groups).

[CHECK]

Exercise 12.15.1. Let $f : \{a, b, c\} \to \{0, 1\}$, $f(a) = f(b) = 0$, $f(c) = 1$.

- Let L be the set of strings with 3 a's. What is $f^*(L)$?
- Let L' be the set of strings with 3 1's. What is $f^{*-1}(L')$?

Exercise 12.15.2. Let $f : \Sigma \to \Sigma'$ as above.

- $L \subseteq \Sigma^*$ regular $\implies$ $f^*(L)$ regular
- $L' \subseteq \Sigma'^*$ regular $\implies$ $(f^*)^{-1}(L')$ regular

12.16 Regular Expressions debug: regex.tex

Note that both DFA and NFA description of a finite state automata uses sets and functions.

Let

$$L = \{ab^n \mid n \geq 0\}$$

This is a regular language, i.e., it is accepted by a DFA. Describing a language by specifying a DFA is tedious (whether using the formal method or by using a DFA diagram.) A **regular expression** is a concise **textual** description of a regular language. For the above example the regex is

$$ab^*$$

You will see later that regular expressions are equal in power to DFA and NFA. Historically, regex actually appears first: Regex was first studied as a mathematical tool for describing how our brain thinks.

Application: This is the method used in most software systems. In particular, regular expressions are absolutely crucial to web application where there are lots of check for validity of input strings.

First let me define what regular expressions are:

regular expression

Definition 12.16.1. Let Σ be a finite set. A **regular expression** (regex) over Σ is a string made up of characters from Σ and the symbols $\emptyset$, *, $\cup$, (, and). It is defined recursively as follows:

- (RE1) $\emptyset$ is a regular expression
- (RE2) ϵ is a regular expression
- (RE3) c is a regular expression for every $c \in \Sigma$
- (RE4) If r is a regular expression, then so is r^*
- (RE5) If r,r' are regular expressions, then so is $r \cup r'$
- (RE6) If r,r' are regular expressions, then so is rr' (Sometimes rr' is written $r \cdot r'$ where $\cdot$ should be treated as a symbol.)
- (RE7) If r is a regular expression, then so is (r).

You should think of the regex as a string. So the $\cup$ is just a special character; later we'll see that this character corresponds to set union when we compute the language corresponding to a regex. Likewise $a \in \Sigma$ is a regex should be thought of as a character in a regex. In fact some authors will use a different symbol to denote $\emptyset$, a (of Σ), $\cup$, *, $\cdot$ in describing a regex.

Example 12.16.1. Let $\Sigma = \{a, b\}$. The $aa^* \cup ba^*b$ is a regular expression since:

1. a is a regex by (RE3)
2. a^* is a regex by (RE4)
3. aa^* is a regex by
4. b is a regex by
5. ba^* is a regex by
6. ba^*b is a regex by
7. $aa^* \cup ba^*b$ is a regex by

Make sure you know which rule is used. □

Exercise 12.16.1. Let $\Sigma = \{0, 1\}$. Show that the following are regular expressions.

1. $01 \cup (10 \cup 11)$
2. $1^* \cup (\cup(10 \cup 11)$

□

Exercise 12.16.2. Which of the following are not regex over $\{a, b\}$?

1. $\emptyset\emptyset$
2. $\emptyset^*$
3. ${}^*\emptyset$
4. $a^* \cap b$
5. a^*bc
6. $a \cup cupa$
7. $P(a \cup b)$

□

For those authors who prefer to invent new symbols to use in a regex, they usually use bold. For instance instead of writing the regex $aa^* \cup ba^*b$, they would write $\mathbf{a}\mathbf{a}^* \cup \mathbf{b}\mathbf{a}^*\mathbf{b}$. (Sorry, I can't bold the union and Kleene star).

The above defines regex. It doesn't tell you anything about languages. The following relates a regex to its language:

Definition 12.16.2. Let r be a regex defined over Σ. $L(r)$, the language generated by r, is defined recursively as follows:

1. $L(\emptyset) = \emptyset$
2. $L(a) = \{a\}$ for $a \in \Sigma$
3. $L(r^*) = (L(r))^*$ for $a \in \Sigma$. (The * on the left is a symbol used in a regex while the * on the right is the Kleene star operator on languages.)
4. $L(r \cup r') = L(r) \cup L(r')$. (The $\cup$ on the right is set union.)
5. $L(rr') = L(r)L(r')$
6. $L((r)) = L(r)$

Example 12.16.2.

1. $L((a \cup b)^*) = L(a \cup b)^* = (L(a) \cup L(b))^* = (\{a\} \cup \{b\})^* = \{a, b\}^*$
2. $L(\emptyset 0) = L(\emptyset)L(0) = \emptyset\{0\} = \{\}$
3. $L(0^* \cup (0^*10^*10^*10^*)^*)$ is the language of strings where the number of 1's is divisible by 3.

□

With parentheses, we can have very complex regex. But just like in arithmetic expressions where we do not write to write $((1+2)+3)+4)$ but use conventions to write $1 + 2 + 3 + 4$, we have operator precedence for regular expressions. The convention is

- $*$ is highest
- $\cdot$ is next
- $\cup$ is the lowest

So for instance the regular expression $a \cdot b \cup a \cdot b^*$ is really the same as

$$a \cdot b \cup a \cdot b^* = (a \cdot b) \cup (a \cdot (b^*))$$

This is important when it comes to computing the language recognized by the

regular expression since it tells you how to break down the regular expression:

$$\begin{aligned} L(a \cdot b \cup a \cdot b^*) &= L((a \cdot b) \cup (a \cdot (b^*))) \\ &= L(a \cdot b) \cup L(a \cdot (b^*)) \\ &= L(a \cdot b) \cup (L(a) \cdot L(b^*)) \\ &= \ldots \end{aligned}$$

Exercise 12.16.3. What is the language generated by the regex $aa^* \cup ba^*b$? Write down all words of length ≤ 5 of this language. □

Solution.

$$\begin{aligned} L(aa^* \cup ba^*b) &= L(aa^*) \cup L(ba^*b) \\ &= L(a)L(a^*) \cup L(b)L(a^*)L(b) \\ &= \{a\}L(a)^* \cup \{b\}L(a)^*\{b\} \\ &= \{a\}\{a\}^* \cup \{b\}\{a\}^*\{b\} \\ &= \{a^n \,|\, n \geq 1\} \cup \{ba^nb \,|\, n \geq 0\} \end{aligned}$$

It's a good idea for the above and all the following exercises to write down the corresponding DFA and NFA for comparison.

Exercise 12.16.4. Write down the regex r for the language over $\{a, b\}$ such that $L(r) = \{\}$. □

Exercise 12.16.5. Write down the regex r for the language over $\{a, b\}$ such that $L(r) = \{a\}$. □

Exercise 12.16.6. Write down the regex r for the language over $\{a, b\}$ such that $L(r) = \{a, b\}$. □

Exercise 12.16.7. Write down the regex r for the language over $\{a, b\}$ such that $L(r) = \{ab, ba\}$. □

Exercise 12.16.8. Write down the regex r for the language L over $\Sigma = \{a, b\}$ such that

$$L(r) = \{w \in \Sigma^* \mid w \text{ contains } a^3\}$$

□

Exercise 12.16.9. Write down the regex R for the language L over $\Sigma = \{a, b\}$ such that

$$L = \{w \in \Sigma^* \mid w \text{ contains exactly 3 } a\text{'s}\}$$

□

Exercise 12.16.10. Write down the regex R for the language L over $\Sigma = \{a, b\}$ such that

$$L = \{w \in \Sigma^* \mid |w| \equiv 1 \pmod 3\}$$

□

Exercise 12.16.11. Write down the regex for the language over $\{a, b\}$ containing strings which has either exact 3 a's or the substring aba. □

Here's the relationship between regex, DFA, and NFA:

Theorem 12.16.1.
(a) *If R is a regex, then $L(R)$ is a regular language*
(b) *If L is a regular language, then there is a regex R such that $L(R) = L$.*

In the next few sections, I'll show you how to convert between regex and NFA, i.e., given a regex r, how to find an NFA such that

$$L(N) = L(r)$$

and also given an NFA N, how to construct a regex r such that

$$L(r) = L(N)$$

Theoretically, the class of languages recognized by regex is exactly the same as the class of regular languages. This means that DFA, NFA, and regexes are "equal in power".

In terms of actual computation, the above tells you how to convert a regex to a DFA for computation. For instance regexes are used in compilers to recognize lexemes and tokens. Compiler tools usually accept regexes. The actual computational processes that recognizes the lexemes can be done through the corresponding DFAs.

Now just imagine you're writing a computer program using a high level programming language. Suppose the there is a function provided for you to check if a string is in a regular language. Obviously you have to specify your regular language. Clearly finite sets are easy: just specify everything in the set. But there are regular languages which are infinite. You now know that regular languages can be described using (1) a DFA, (2) an NFA, or (3) regex.

Regex is the simplest.

Here is an example of using the regex provided by the programming language Python. The regex in Python allows you to verify if a string matches a regex. (There are actually three modes of matchings). Clearly you can't find a key on your keyboard for $\cup$: In most programming languages or packages with regex facilities, the | is used in place of $\cup$. The search function in the re module accepted a regex and a string.

Example 12.16.3. Suppose you want match strings adef or bdef. You can

do the following in Python:

```
>>> import re
>>> regex = re.compile("(a|b)def")
>>> print regex.match("adef")
<$\_$sre.SRE$\_$Match object at 0x00E1F0C8>
>>> print regex.match("bdef")
<$\_$sre.SRE$\_$Match object at 0x00E1F170>
>>> print regex.match("abcd")
None
>>>
```

The `a|b` as a Python regular expression means $a \cup b$ as a mathemtical regular expression.

For a fix Σ, you have a collection of regex over Σ. You can think of $\cup$ and $\cdot$ are operators. Just like there are algebraic rules for real numbers, there are algebraic rules for regular expressions. Knowing these rules will help you simplify regulars and check quickly if two regular expressions are the same, i.e., they generated the same language.

1. $r(r'r'') = (rr')r''$
2. $r\emptyset = \emptyset = \emptyset r$
3. $\epsilon r = r = \epsilon r$
4. $r \cup r' = r' \cup r$
5. $r \cup r = r$
6. $r \cup \emptyset = r = \emptyset \cup r$
7. $r \cup \Sigma^* = \Sigma^* = \Sigma^* \cup r$
8. $r(r' \cup r'') = rr' \cup rr''$
9. $(r' \cup r'')r = r'r \cup r''r$
10. $r^{**} = r^*$
11. $\epsilon^* = \epsilon$
12. $\epsilon \cup rr^* = r^*$
13. $\epsilon \cup r \cup r^2r^* = r^*$
14. $\epsilon \cup r \cup r^2 \cup r^3r^* = r^*$
15. $\epsilon \cup r \cup r^2 \cup \cdots \cup r^n \cup r^{n+1}r^* = r^*$
16. $rr^* = r^*r$

Let's prove that $a(a^* \cup \emptyset) \cup \epsilon = a^*$.

$$(a^* \cup \emptyset)a \cup \epsilon = a^*a \cup \epsilon = aa^* \cup \epsilon = a^*$$

To make life simpler for programmers, the regex in Python has extra characters or strings to simplify programming.

- $[abcdef]$ is the same as $a|b|c|d|e|f$.

- $[0-9]$ is the same as $0|1|2|3|4|5|6|7|8|9$
- $[a-z]$ will match any ASCII character from 'a' to 'z'
- $[a-z]\{3-5\}$ will accept 3 or 4 or 5 ASCII characters from 'a' to 'z'.

This list is incomplete. For more information, use google to check regular expressions for UNIX.

Regex is used not just in work related to compilers. Regex is also used to verify for instance if a user's input is valid. In fact regex is a cornerstone in web programming.

12.17 NFA to Regex debug: nfa-to-regex.tex

The algorithm for converting an NFA N to a regex r such that

$$L(N) = L(r)$$

is actually pretty simple. Here's the algorithm.

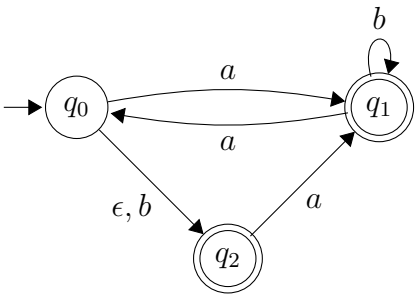
We're going to build a new NFA. Let's call this N'. In fact this NFA will not be exactly an NFA. N' will initially be N. We will make some changes.

First, in N', you create a new node, say i (for initial state) and join i to the start state of N with an ϵ–transition.

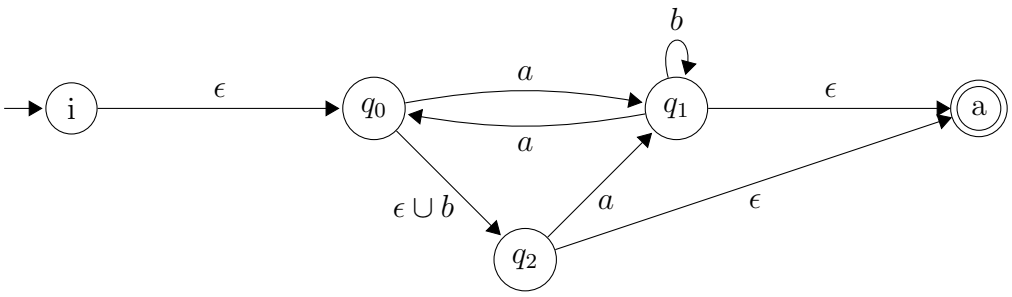
Second, you create a new node, say a (for accept state) and join all accept state of N to a by ϵ–transitions. You then make the accept states from N be non-accept states in N'.

Third, if from two states q, q' we see a, b, we replace this transition label to $a \cup b$, i.e., a regular expression. More generally if you see a list of characters from Σ of N, you replace it with the regular expression of the union of these characters.

We're not done yet, but let's look at an example right away. Consider this NFA N:



The new N' (again, I'm not done changing it yet):



Note that the state diagram is now *not* an NFA since the transitions are labeled with *regular expressions*.

generalized NFA
GNFA

The above is called a **generalized NFA GNFA**. The operation of such animals are obvious: You run them more or less like NFAs. If you read say a character a, then you follow transitions labeled with r if $a \in L(r)$. For instance you are allowed to following transitions labeled a or $a \cup b$ or $a^* \cup b$, etc.

It's clear how acceptance should be defined for such generalized NFAs.

By the way, if there's no transition from say q to q' in N, in the new generalized NFA, we add a transition labeled $\emptyset$ from q to q'. This means that you cannot go from q to q' since the language $L(\emptyset)$ is $\emptyset$.

Now you might ask: Why in the world would you want to make things even more complicated? First we have DFA. Then we have NFA. And now ... *generalized* NFA?!?

The reason is the same reason for generalizing DFA to NFA. The NFA gives us a great deal of flexibility when compared against the DFA. What about from NFA to generalized NFA?

If an NFA has a transition from q to q' labels a, b, then there are actually two transitions, one labeled a and one labeled b. In the case of the corresponding generalized NFA, there is only one transition labeled $a \cup b$.

Here's the point: It allows us to *simplify* NFAs. The above shows us that two transitions become one. Even more importantly, generalized NFA actually also provides us with the flexibility of simplifying states!!!

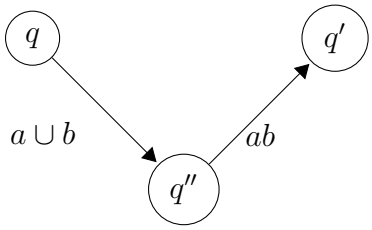
How is that?

Suppose you have a transition from q to q' labeled a and there's a transition from q' to q'' labeled b. Do you see that it's the same as combining the two transitions to one so that there's only one transition from q to q'' labeled ab.

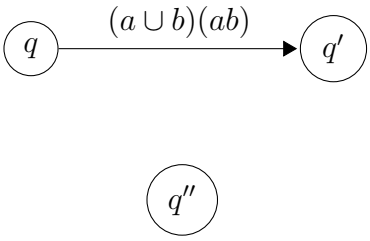
If no other transitions passed through q', do you see that now q' is useless and can be thrown away?

The main idea of the construction is to continually choose a node q in the generalized NFA other than i and a and remove q by "redirecting traffic flow" without changing the language accepted by the new generalized NFA.

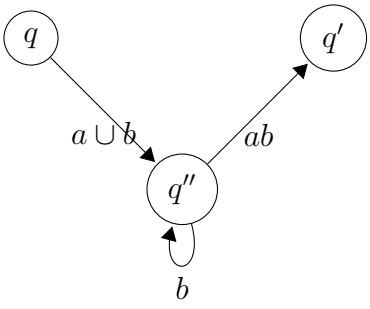
Let's look at an example. Suppose in a generalized NFA we have the following 3 states (there might be more):



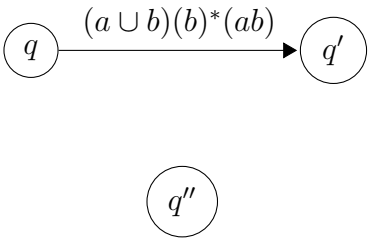
Clearly this can be replaced by



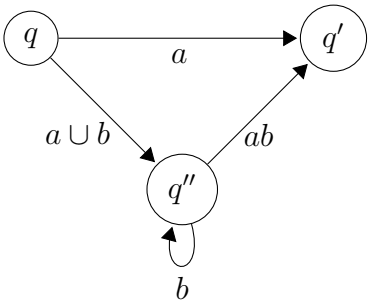
Wait ... what if there's "traffic" going from q'' to q'' , i.e., a loop? Say it's the generalized NFA before traffic redirection is this:



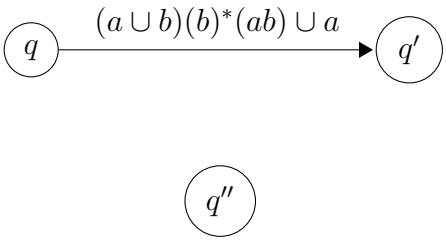
So the total "traffic" going from q to q' through q'' is $(a \cup b)(b)^*(ab)$, allowing the traffic from q'' to itself to loop as many times as we like (include none). And we get this simplification:



Wait a minute ... what if there's already a transition from q to q'? Say



In that case, the combined traffic from q to q' must be



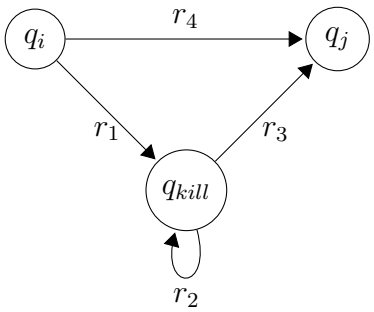
and in this case, the state q'' is redundant since now there is no "traffic" going through q''.

If we do the above for *all* possible q and q', then the resulting generalized NFA would have no transitions running through this lonely and isolated q''. This means that we can remove it. Right?

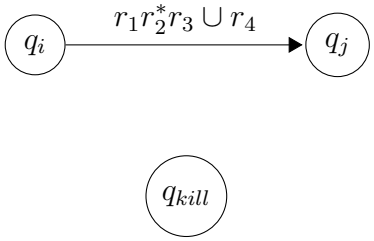
That's the whole idea.

At the end of the "state removal" operation, we would have only the two extra states i and a left. The regular expression from i to a is the required regular expression.

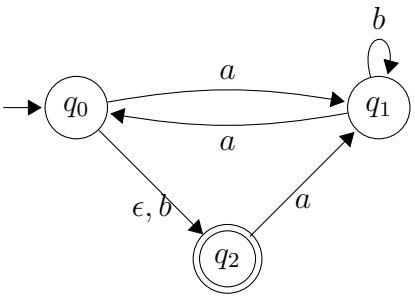
Formally, after the initial construction of the GNFA, you continually pick a state q_{kill} which is not i and not a and for every $q_i \neq q_{kill}$ and $q_j \neq q_{kill}$, with the following (relevant) regular expressions for transitions:



then this part of the GNFA can be replaced by the following without changing the resulting language accepted:



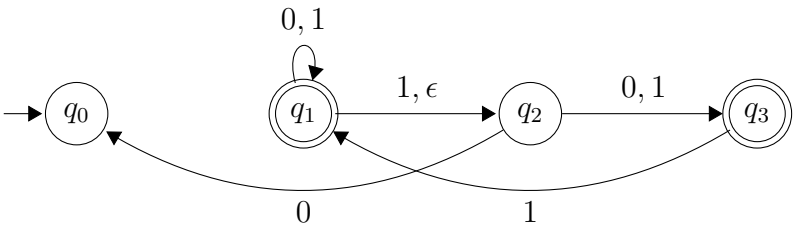
Exercise 12.17.1. Find a regex that accepts the same language as the following NFA:



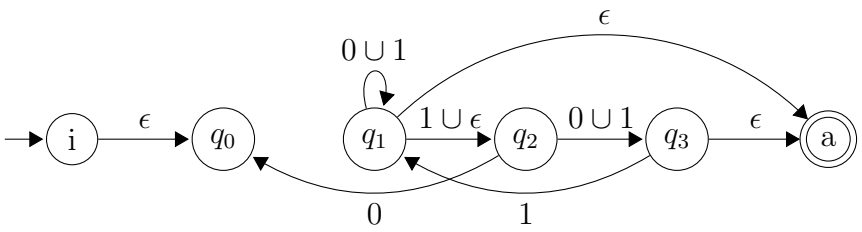
There are two examples on the regex construction for NFAs in the Sipser textbook. Make sure you try both of them.

The following are more examples.

Example. Here's an NFA:



Here's the initial GNFA:



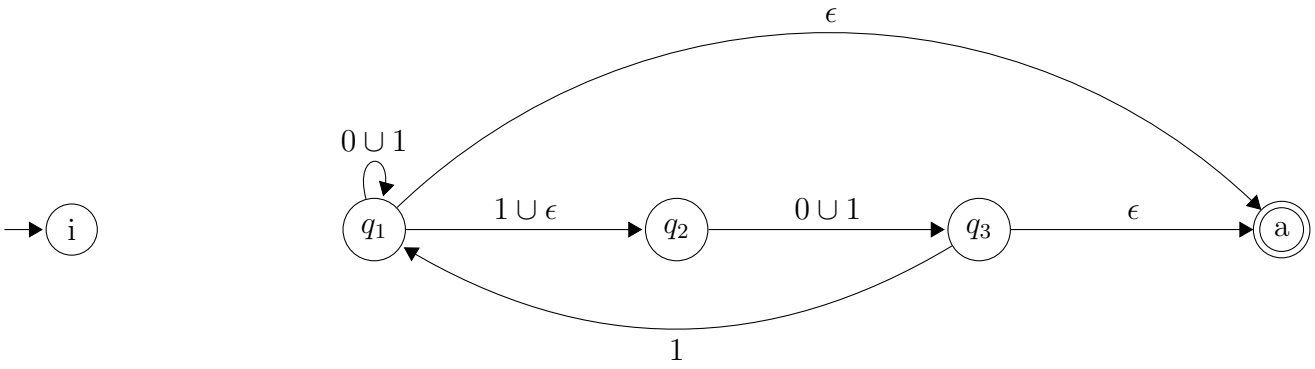
Each table below shows that computation of the resulting transistion after removing a state q_{kill}. In the third, fourth, and fifth columns,

$$r_1 = \delta(q_i, q_{kill})$$
$$r_2 = \delta(q_{kill}, q_{kill})$$
$$r_3 = \delta(q_{kill}, q_j)$$
$$r_4 = \delta(q_i, q_j)$$

Removal of q_{kill}=q_0:

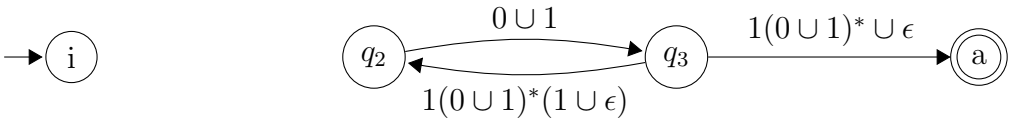
q_i	q_j	r_1	r_2	r_3	r_4	$r_1 r_2^* r_3 \cup r_4$
q_1	q_1	$\emptyset$	$\emptyset$	$\emptyset$	$0 \cup 1$	$0 \cup 1$
q_1	q_2	$\emptyset$	$\emptyset$	$\emptyset$	$1 \cup \epsilon$	$1 \cup \epsilon$
q_1	q_3	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
q_1	a	$\emptyset$	$\emptyset$	$\emptyset$	ϵ	ϵ
q_2	q_1	0	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
q_2	q_2	0	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
q_2	q_3	0	$\emptyset$	$\emptyset$	$0 \cup 1$	$0 \cup 1$
q_2	a	0	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
q_3	q_1	$\emptyset$	$\emptyset$	$\emptyset$	1	1
q_3	q_2	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
q_3	q_3	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
q_3	a	$\emptyset$	$\emptyset$	$\emptyset$	ϵ	ϵ
i	q_1	ϵ	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

i	q_2	ϵ	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
i	q_3	ϵ	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
i	a	ϵ	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$



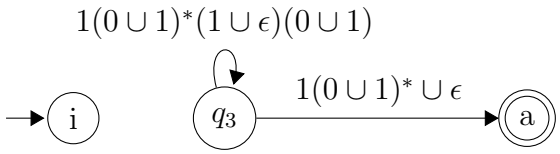
Removal of q_{kill}=q_1:

q_i	q_j	r_1	r_2	r_3	r_4	$r_1 r_2^* r_3 \cup r_4$
q_2	q_2	$\emptyset$	$0 \cup 1$	$1 \cup \epsilon$	$\emptyset$	$\emptyset$
q_2	q_3	$\emptyset$	$0 \cup 1$	$\emptyset$	$0 \cup 1$	$0 \cup 1$
q_2	a	$\emptyset$	$0 \cup 1$	ϵ	$\emptyset$	$\emptyset$
q_3	q_2	1	$0 \cup 1$	$1 \cup \epsilon$	$\emptyset$	$1(0 \cup 1)^*(1 \cup \epsilon)$
q_3	q_3	1	$0 \cup 1$	$\emptyset$	$\emptyset$	$\emptyset$
q_3	a	1	$0 \cup 1$	ϵ	ϵ	$1(0 \cup 1)^* \cup \epsilon$
i	q_2	$\emptyset$	$0 \cup 1$	$1 \cup \epsilon$	$\emptyset$	$\emptyset$
i	q_3	$\emptyset$	$0 \cup 1$	$\emptyset$	$\emptyset$	$\emptyset$
i	a	$\emptyset$	$0 \cup 1$	ϵ	$\emptyset$	$\emptyset$



Removal of q_{kill}=q_2:

q_i	q_j	r_1	r_2	r_3	r_4	$r_1 r_2^* r_3 \cup r_4$
q_3	q_3	$1(0 \cup 1)^*(1 \cup \epsilon)$	$\emptyset$	$0 \cup 1$	$\emptyset$	$1(0 \cup 1)^*(1 \cup \epsilon)(0 \cup 1)$
q_3	a	$1(0 \cup 1)^*(1 \cup \epsilon)$	$\emptyset$	$\emptyset$	$1(0 \cup 1)^* \cup \epsilon$	$1(0 \cup 1)^* \cup \epsilon$
i	q_3	$\emptyset$	$\emptyset$	$0 \cup 1$	$\emptyset$	$\emptyset$
i	a	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$



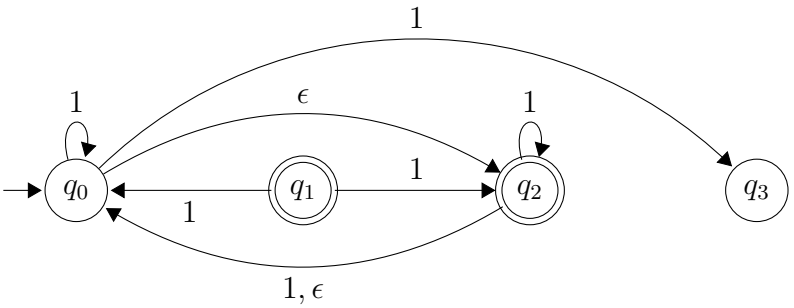
Removal of q_{kill}=q_3:

q_i	q_j	r_1	r_2	r_3	r_4	$r_1 r_2^* r_3 \cup r_4$
i	a	$\emptyset$	$1(0 \cup 1)^*(1 \cup \epsilon)(0 \cup 1)$	$1(0 \cup 1)^* \cup \epsilon$	$\emptyset$	$\emptyset$

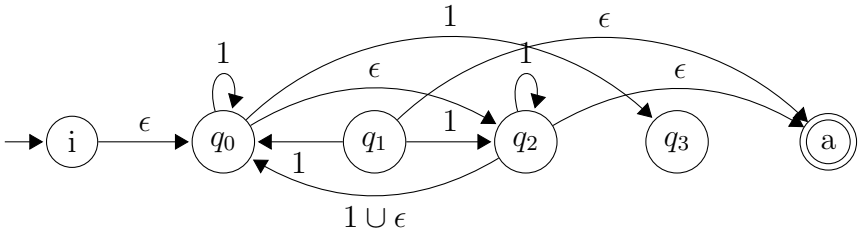
Therefore the regex that generates the same language that is accepted by the given NFA is

$$\emptyset$$

Example. Here's the NFA:



Here's the initial GNFA:



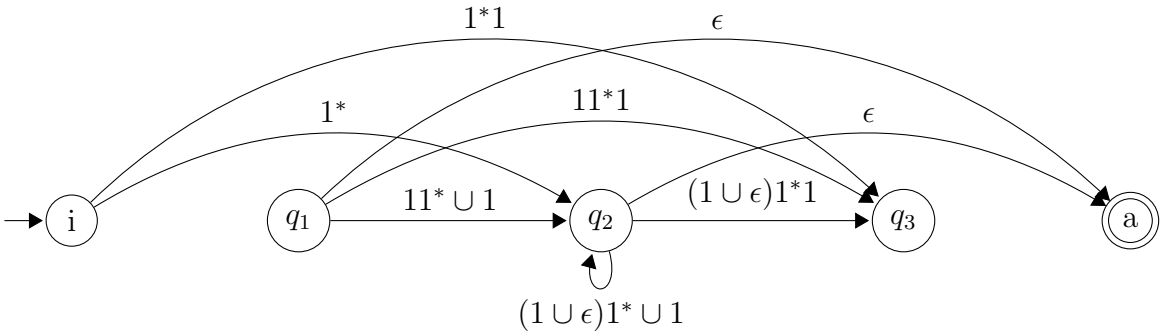
Each table below shows that computation of the resulting transistions after removing a state q_{kill}. In the third, fourth, and fifth columns,

$$r_1 = \delta(q_i, q_{kill})$$
$$r_2 = \delta(q_{kill}, q_{kill})$$
$$r_3 = \delta(q_{kill}, q_j)$$
$$r_4 = \delta(q_i, q_j)$$

Removal of q_{kill}=q_0:

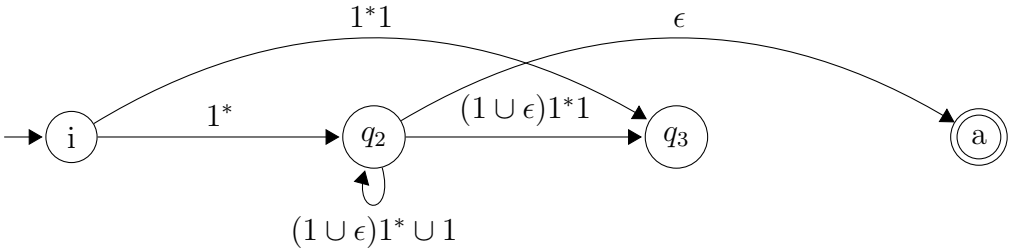
q_i	q_j	r_1	r_2	r_3	r_4	$r_1 r_2^* r_3 \cup r_4$
q_1	q_1	1	1	$\emptyset$	$\emptyset$	$\emptyset$
q_1	q_2	1	1	ϵ	1	$11^* \cup 1$
q_1	q_3	1	1	1	$\emptyset$	11^*1
q_1	a	1	1	$\emptyset$	ϵ	ϵ
q_2	q_1	$1 \cup \epsilon$	1	$\emptyset$	$\emptyset$	$\emptyset$
q_2	q_2	$1 \cup \epsilon$	1	ϵ	1	$(1 \cup \epsilon)1^* \cup 1$
q_2	q_3	$1 \cup \epsilon$	1	1	$\emptyset$	$(1 \cup \epsilon)1^*1$
q_2	a	$1 \cup \epsilon$	1	$\emptyset$	ϵ	ϵ
q_3	q_1	$\emptyset$	1	$\emptyset$	$\emptyset$	$\emptyset$
q_3	q_2	$\emptyset$	1	ϵ	$\emptyset$	$\emptyset$

q_3	q_3	$\emptyset$	1	1	$\emptyset$	$\emptyset$
q_3	a	$\emptyset$	1	$\emptyset$	$\emptyset$	$\emptyset$
i	q_1	ϵ	1	$\emptyset$	$\emptyset$	$\emptyset$
i	q_2	ϵ	1	ϵ	$\emptyset$	1^*
i	q_3	ϵ	1	1	$\emptyset$	1^*1
i	a	ϵ	1	$\emptyset$	$\emptyset$	$\emptyset$



Removal of q_{kill}=q_1:

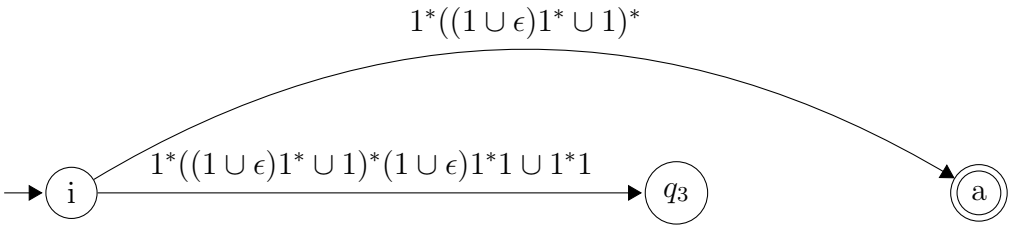
q_i	q_j	r_1	r_2	r_3	r_4	$r_1 r_2^* r_3 \cup r_4$
q_2	q_2	$\emptyset$	$\emptyset$	$11^* \cup 1$	$(1 \cup \epsilon)1^* \cup 1$	$(1 \cup \epsilon)1^* \cup 1$
q_2	q_3	$\emptyset$	$\emptyset$	11^*1	$(1 \cup \epsilon)1^*1$	$(1 \cup \epsilon)1^*1$
q_2	a	$\emptyset$	$\emptyset$	ϵ	ϵ	ϵ
q_3	q_2	$\emptyset$	$\emptyset$	$11^* \cup 1$	$\emptyset$	$\emptyset$
q_3	q_3	$\emptyset$	$\emptyset$	11^*1	$\emptyset$	$\emptyset$
q_3	a	$\emptyset$	$\emptyset$	ϵ	$\emptyset$	$\emptyset$
i	q_2	$\emptyset$	$\emptyset$	$11^* \cup 1$	1^*	1^*
i	q_3	$\emptyset$	$\emptyset$	11^*1	1^*1	1^*1
i	a	$\emptyset$	$\emptyset$	ϵ	$\emptyset$	$\emptyset$



Removal of q_{kill}=q_2:

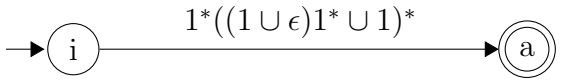
q_i	q_j	r_1	r_2	r_3	r_4	$r_1 r_2^* r_3 \cup r_4$
q_3	q_3	$\emptyset$	$(1 \cup \epsilon)1^* \cup 1$	$(1 \cup \epsilon)1^*1$	$\emptyset$	$\emptyset$

q_3	a	$\emptyset$	$(1 \cup \epsilon)1^* \cup 1$	ϵ	$\emptyset$	$\emptyset$
i	q_3	1^*	$(1 \cup \epsilon)1^* \cup 1$	$(1 \cup \epsilon)1^*1$	1^*1	$1^*((1 \cup \epsilon)1^* \cup 1)^*(1 \cup \epsilon)1^*1 \cup 1^*1$
i	a	1^*	$(1 \cup \epsilon)1^* \cup 1$	ϵ	$\emptyset$	$1^*((1 \cup \epsilon)1^* \cup 1)^*$



Removal of q_{kill}=q_3:

q_i	q_j	r_1	r_2	r_3	r_4	$r_1 r_2^* r_3 \cup r_4$
i	a	$1^*((1 \cup \epsilon)1^* \cup 1)^*(1 \cup \epsilon)1^*1 \cup 1^*1$	$\emptyset$	$\emptyset$	$1^*((1 \cup \epsilon)1^* \cup 1)^*$	$1^*((1 \cup \epsilon)1^* \cup 1)^*$



Therefore the regex that generates the same language that is accepted by the given NFA is

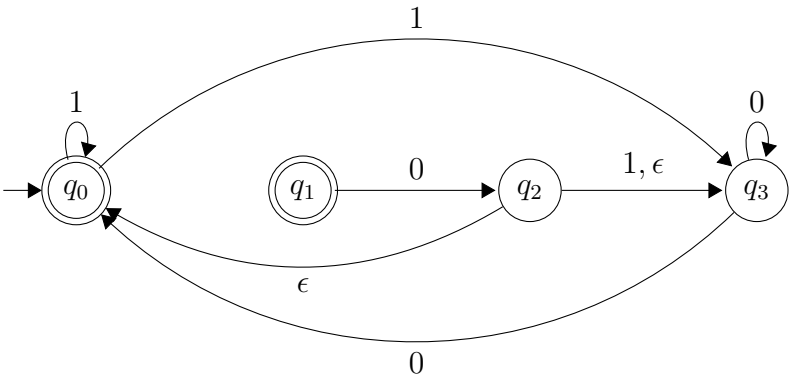
$$1^*((1 \cup \epsilon)1^* \cup 1)^*$$

You can simplify the above regex as follows:

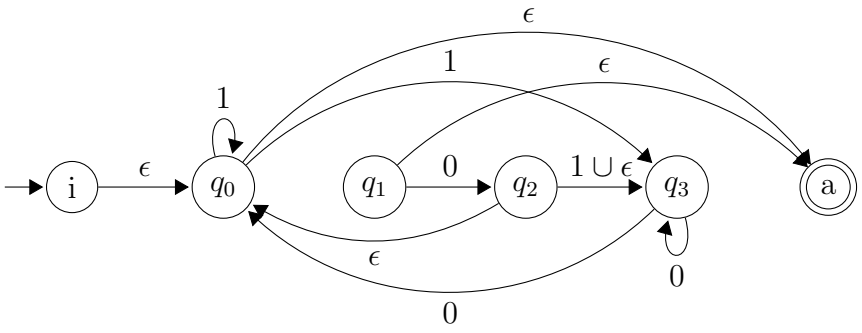
$$\begin{aligned}
1^*((1 \cup \epsilon)1^* \cup 1)^* &= 1^*((11^* \cup \epsilon 1^*)^* \cup 1)^* \\
&= 1^*((11^* \cup 1^*)^* \cup 1)^* \\
&= 1^*((1^*)^* \cup 1)^* \\
&= 1^*(1^* \cup 1)^* \\
&= 1^*(1^*)^* \\
&= 1^*1^* \\
&= 1^*
\end{aligned}$$

If you look at the original NFA, you will see that the language accepted by the NFA is indeed $\{1\}^*$.

Example. Here's the NFA:



Here's the initial GNFA:



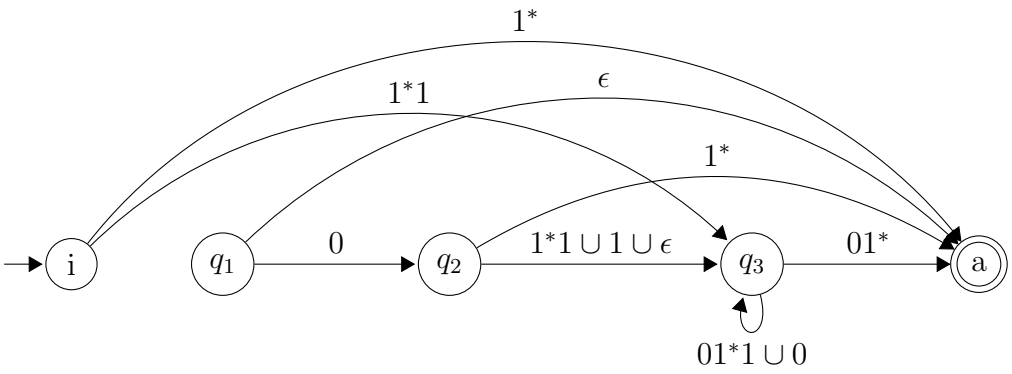
Each table below shows that computation of the resulting transistions after removing a state q_{kill}. In the third, fourth, and fifth columns,

$$r_1 = \delta(q_i, q_{kill})$$
$$r_2 = \delta(q_{kill}, q_{kill})$$
$$r_3 = \delta(q_{kill}, q_j)$$
$$r_4 = \delta(q_i, q_j)$$

Removal of q_{kill}=q_0:

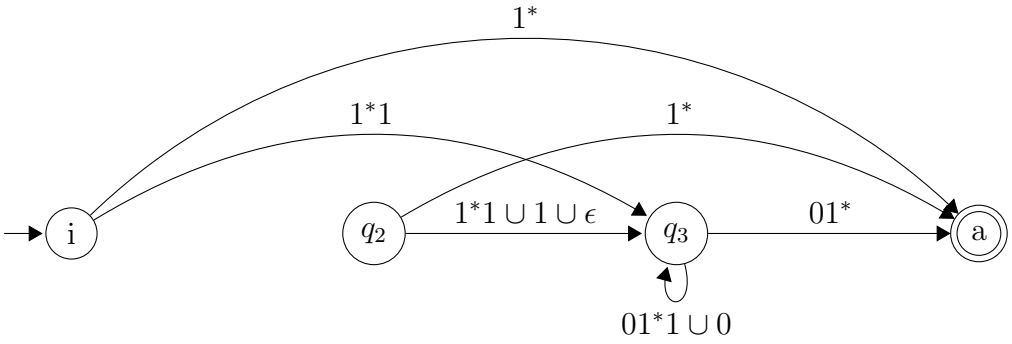
q_i	q_j	r_1	r_2	r_3	r_4	$r_1 r_2^* r_3 \cup r_4$
q_1	q_1	$\emptyset$	1	$\emptyset$	$\emptyset$	$\emptyset$
q_1	q_2	$\emptyset$	1	$\emptyset$	0	0
q_1	q_3	$\emptyset$	1	1	$\emptyset$	$\emptyset$
q_1	a	$\emptyset$	1	ϵ	ϵ	ϵ
q_2	q_1	ϵ	1	$\emptyset$	$\emptyset$	$\emptyset$
q_2	q_2	ϵ	1	$\emptyset$	$\emptyset$	$\emptyset$

q_2	q_3	ϵ	1	1	$1 \cup \epsilon$	$1^*1 \cup 1 \cup \epsilon$
q_2	a	ϵ	1	ϵ	$\emptyset$	1^*
q_3	q_1	0	1	$\emptyset$	$\emptyset$	$\emptyset$
q_3	q_2	0	1	$\emptyset$	$\emptyset$	$\emptyset$
q_3	q_3	0	1	1	0	$01^*1 \cup 0$
q_3	a	0	1	ϵ	$\emptyset$	01^*
i	q_1	ϵ	1	$\emptyset$	$\emptyset$	$\emptyset$
i	q_2	ϵ	1	$\emptyset$	$\emptyset$	$\emptyset$
i	q_3	ϵ	1	1	$\emptyset$	1^*1
i	a	ϵ	1	ϵ	$\emptyset$	1^*



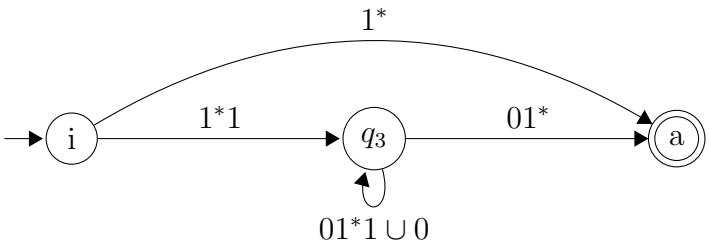
Removal of q_{kill}=q_1:

q_i	q_j	r_1	r_2	r_3	r_4	$r_1 r_2^* r_3 \cup r_4$
q_2	q_2	$\emptyset$	$\emptyset$	0	$\emptyset$	$\emptyset$
q_2	q_3	$\emptyset$	$\emptyset$	$\emptyset$	$1^*1 \cup 1 \cup \epsilon$	$1^*1 \cup 1 \cup \epsilon$
q_2	a	$\emptyset$	$\emptyset$	ϵ	1^*	1^*
q_3	q_2	$\emptyset$	$\emptyset$	0	$\emptyset$	$\emptyset$
q_3	q_3	$\emptyset$	$\emptyset$	$\emptyset$	$01^*1 \cup 0$	$01^*1 \cup 0$
q_3	a	$\emptyset$	$\emptyset$	ϵ	01^*	01^*
i	q_2	$\emptyset$	$\emptyset$	0	$\emptyset$	$\emptyset$
i	q_3	$\emptyset$	$\emptyset$	$\emptyset$	1^*1	1^*1
i	a	$\emptyset$	$\emptyset$	ϵ	1^*	1^*



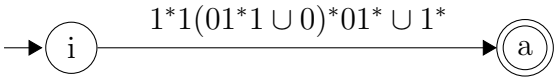
Removal of q_{kill}=q_2:

q_i	q_j	r_1	r_2	r_3	r_4	$r_1 r_2^* r_3 \cup r_4$
q_3	q_3	$\emptyset$	$\emptyset$	$1^*1 \cup 1 \cup \epsilon$	$01^*1 \cup 0$	$01^*1 \cup 0$
q_3	a	$\emptyset$	$\emptyset$	1^*	01^*	01^*
i	q_3	$\emptyset$	$\emptyset$	$1^*1 \cup 1 \cup \epsilon$	1^*1	1^*1
i	a	$\emptyset$	$\emptyset$	1^*	1^*	1^*



Removal of q_{kill}=q_3:

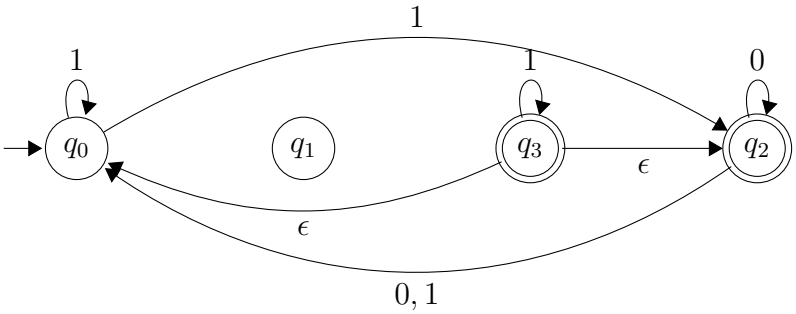
q_i	q_j	r_1	r_2	r_3	r_4	$r_1 r_2^* r_3 \cup r_4$
i	a	1^*1	$01^*1 \cup 0$	01^*	1^*	$1^*1(01^*1 \cup 0)^*01^* \cup 1^*$



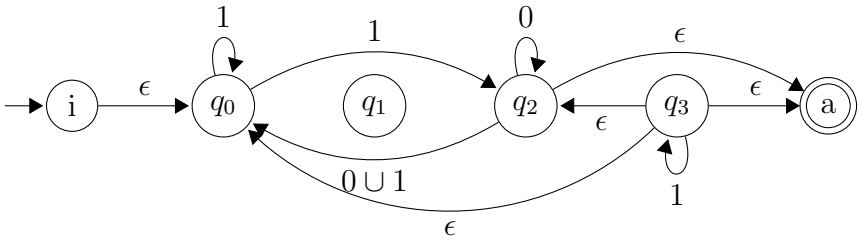
Therefore the regex that generates the same language that is accepted by the given NFA is

$$1^*1(01^*1 \cup 0)^*01^* \cup 1^*$$

Example. Here's the NFA:



Here's the initial GNFA:



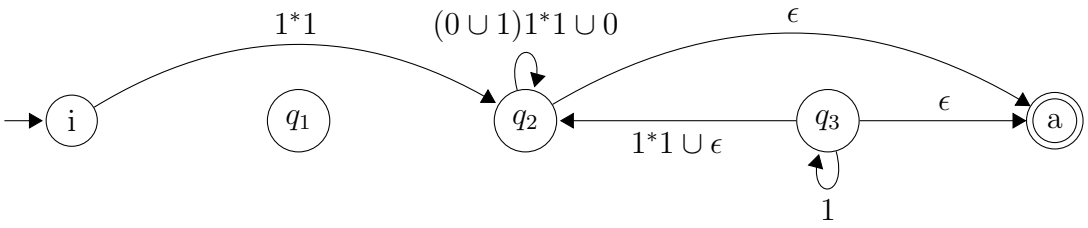
Each table below shows that computation of the resulting transistions after removing a state q_{kill}. In the third, fourth, and fifth columns,

$$r_1 = \delta(q_i, q_{kill})$$
$$r_2 = \delta(q_{kill}, q_{kill})$$
$$r_3 = \delta(q_{kill}, q_j)$$
$$r_4 = \delta(q_i, q_j)$$

Removal of q_{kill}=q_0:

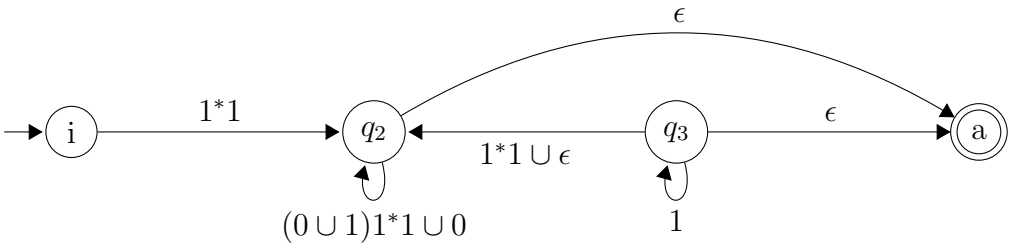
q_i	q_j	r_1	r_2	r_3	r_4	$r_1 r_2^* r_3 \cup r_4$
q_1	q_1	$\emptyset$	1	$\emptyset$	$\emptyset$	$\emptyset$
q_1	q_2	$\emptyset$	1	1	$\emptyset$	$\emptyset$
q_1	q_3	$\emptyset$	1	$\emptyset$	$\emptyset$	$\emptyset$
q_1	a	$\emptyset$	1	$\emptyset$	$\emptyset$	$\emptyset$
q_2	q_1	$0 \cup 1$	1	$\emptyset$	$\emptyset$	$\emptyset$
q_2	q_2	$0 \cup 1$	1	1	0	$(0 \cup 1)1^*1 \cup 0$
q_2	q_3	$0 \cup 1$	1	$\emptyset$	$\emptyset$	$\emptyset$
q_2	a	$0 \cup 1$	1	$\emptyset$	ϵ	ϵ
q_3	q_1	ϵ	1	$\emptyset$	$\emptyset$	$\emptyset$
q_3	q_2	ϵ	1	1	ϵ	$1^*1 \cup \epsilon$

q_3	q_3	ϵ	1	$\emptyset$	1	1
q_3	a	ϵ	1	$\emptyset$	ϵ	ϵ
i	q_1	ϵ	1	$\emptyset$	$\emptyset$	$\emptyset$
i	q_2	ϵ	1	1	$\emptyset$	1^*1
i	q_3	ϵ	1	$\emptyset$	$\emptyset$	$\emptyset$
i	a	ϵ	1	$\emptyset$	$\emptyset$	$\emptyset$



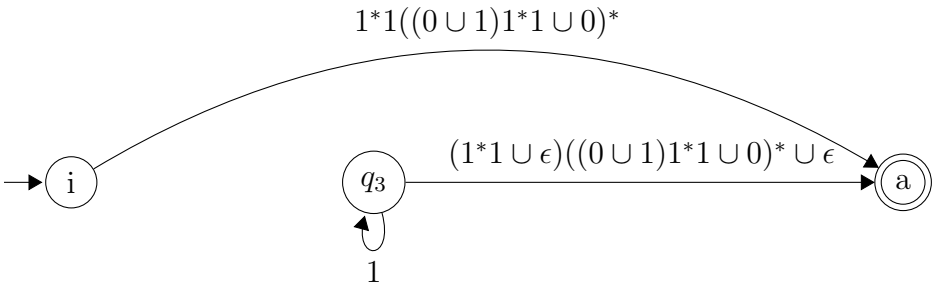
Removal of q_{kill}=q_1:

q_i	q_j	r_1	r_2	r_3	r_4	$r_1r_2^*r_3 \cup r_4$
q_2	q_2	$\emptyset$	$\emptyset$	$\emptyset$	$(0 \cup 1)1^*1 \cup 0$	$(0 \cup 1)1^*1 \cup 0$
q_2	q_3	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
q_2	a	$\emptyset$	$\emptyset$	$\emptyset$	ϵ	ϵ
q_3	q_2	$\emptyset$	$\emptyset$	$\emptyset$	$1^*1 \cup \epsilon$	$1^*1 \cup \epsilon$
q_3	q_3	$\emptyset$	$\emptyset$	$\emptyset$	1	1
q_3	a	$\emptyset$	$\emptyset$	$\emptyset$	ϵ	ϵ
i	q_2	$\emptyset$	$\emptyset$	$\emptyset$	1^*1	1^*1
i	q_3	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
i	a	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$



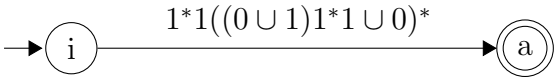
Removal of q_{kill}=q_2:

q_i	q_j	r_1	r_2	r_3	r_4	$r_1r_2^*r_3 \cup r_4$
q_3	q_3	$1^*1 \cup \epsilon$	$(0 \cup 1)1^*1 \cup 0$	$\emptyset$	1	1
q_3	a	$1^*1 \cup \epsilon$	$(0 \cup 1)1^*1 \cup 0$	ϵ	ϵ	$(1^*1 \cup \epsilon)((0 \cup 1)1^*1 \cup 0)^* \cup \epsilon$
i	q_3	1^*1	$(0 \cup 1)1^*1 \cup 0$	$\emptyset$	$\emptyset$	$\emptyset$
i	a	1^*1	$(0 \cup 1)1^*1 \cup 0$	ϵ	$\emptyset$	$1^*1((0 \cup 1)1^*1 \cup 0)^*$



Removal of q_{kill}=q_3:

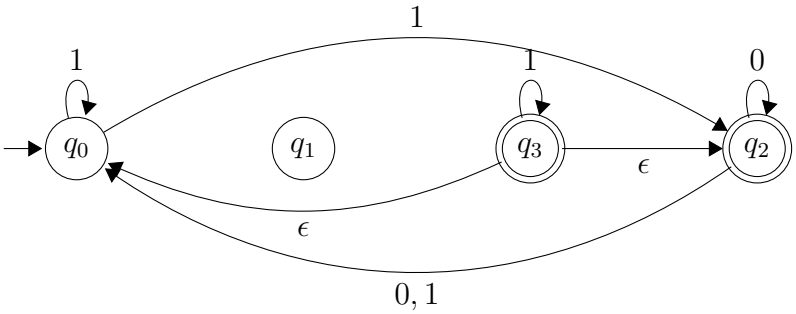
q_i	q_j	r_1	r_2	r_3	r_4	$r_1 r_2^* r_3 \cup r_4$
i	a	$\emptyset$	1	$(1^*1 \cup \epsilon)((0 \cup 1)1^*1 \cup 0)^* \cup \epsilon$	$1^*1((0 \cup 1)1^*1 \cup 0)^*$	$1^*1((0 \cup 1)1^*1 \cup 0)^*$



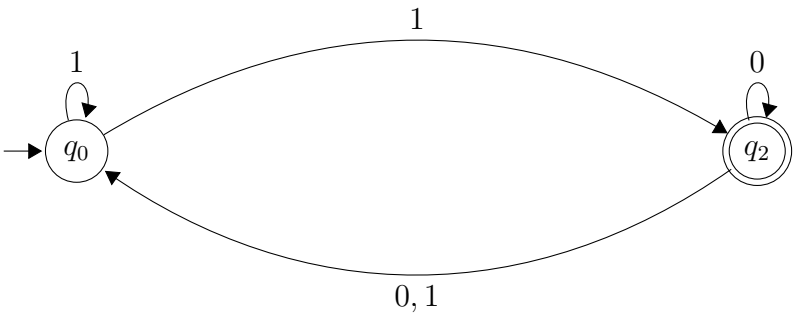
Therefore the regex that generates the same language that is accepted by the given NFA is

$$1^*1((0 \cup 1)1^*1 \cup 0)^*$$

If you look at the original NFA,



and you remove the unreachable states, you would get



You see that the most general path that reaches from q_0 to q_2 without leaving q_2 has a regex of

$$1^*1$$

Intuitively, after that, if you *leave* q_0, you can still get back to q_2, if you do

$$0$$

or

$$(0 \cup 1)1^*1$$

Intuitively, the most general path is then

$$1^*1(0 \cup (0 \cup 1)1^*1)^*$$

which is exactly what we get using the algorithm.

To give a mathematical proof, all you need to do is follow the idea behind the above examples.

First you have to define mathematically the concept of a GNFA, together with language acceptance of an GNFA.

Exercise 12.17.2. Define formally the concept of a GNFA and define what is meant a a word begin accepted by a GNFA.

Next, you have to prove that given a GNFA N, the above operation of removing a state to obtain a new GNFA N' does not change the language being accepted, i.e.,

$$L(N') = L(N)$$

You can then use induction on the number of states in a GNFA to prove the original statement above.

Exercise 12.17.3. Prove the main theorem of this section.

12.18 Regex to NFA

Now let me talk about converting a regex to an NFA, i.e., if r is a regex, I want to build an NFA N such that

$$L(N) = L(r)$$

This is actually pretty easy.

Let's try an example. Let

$$r = (a^* \cup bba)^*(bab \cup ab)^*$$

Then the language generated by r is

$$\begin{aligned} L(r) &= L((a^* \cup bba)^*(bab \cup ab)^*) \\ &= L((a^* \cup bba)^*) \cdot L((bab \cup ab)^*) \end{aligned}$$

At this point, we see that is we can build an NFA N_1 to accept

$$L((a^* \cup bba)^*)$$

and another NFA N_2 to accept

$$L((bab \cup ab)^*)$$

then, we just use the concatenation construction on N_1 and N_2, i.e., the new NFA will have the combination of N_1 followed by N_2 where the accept states of N_1 is joined to the start state of N_2 with ϵ-transitions.

Focusing on

$$L((a^* \cup bba)^*)$$

we see that

$$L((a^* \cup bba)^*) = L(a^* \cup bba)^*$$

So if we can build an NFA N_3 to accept

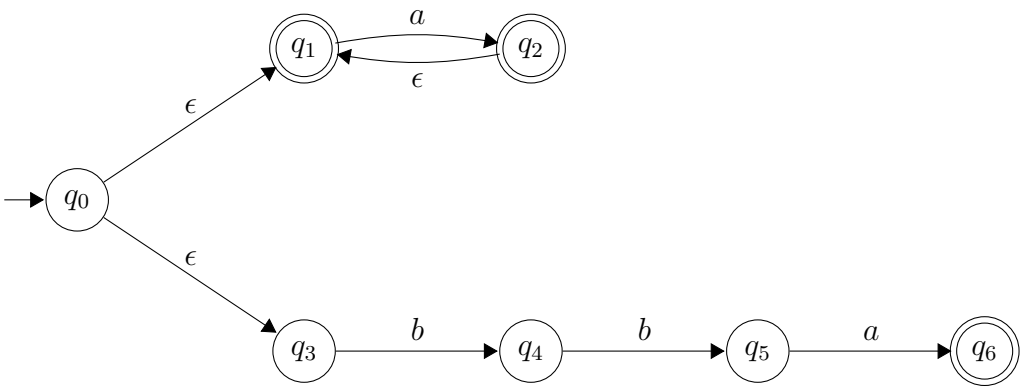
$$L(a^* \cup bba)$$

then can use the Kleene star construction on this NFA to get N_1, i.e., N_1 is built from N_3 where the start state of N_3 is changed to an accept state and the original accept state of N_3 have ϵ-transitions to this new start state. Instead

of doing this breakdown one step at a time, I think you see now that

$$L(a^* \cup bba) = L(a)^* \cup L(bba)$$

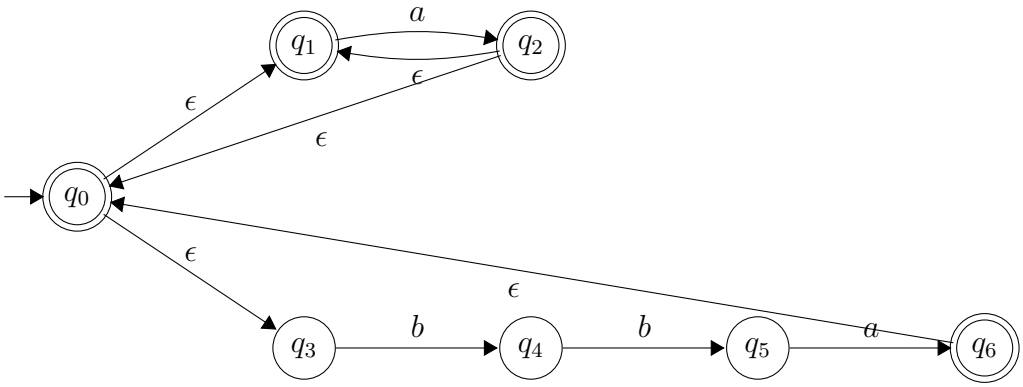
can be accepted by this NFA:



This NFA accepts

$$L(a^* \cup bba) = L(a)^* \cup L(bba)$$

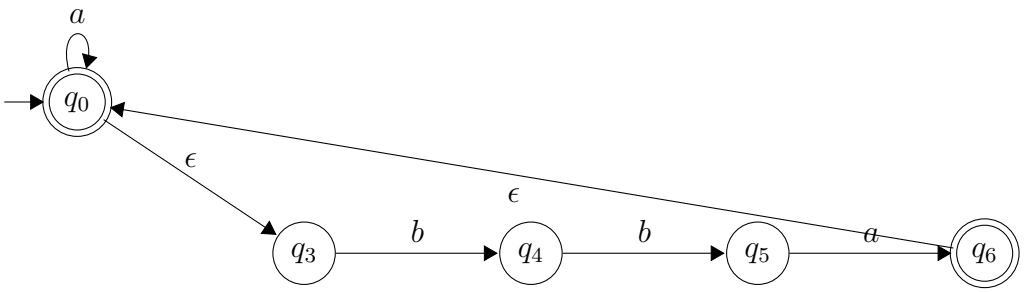
Now using the Kleene star construction, we get



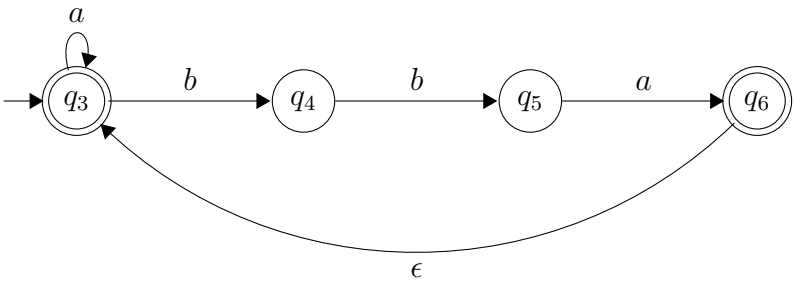
which accepts

$$L((a^* \cup bba)^*) = L(a^* \cup bba)^*$$

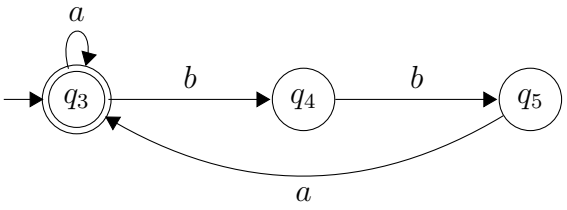
Let me simplify this a little before we continue. First I do this:



then this:



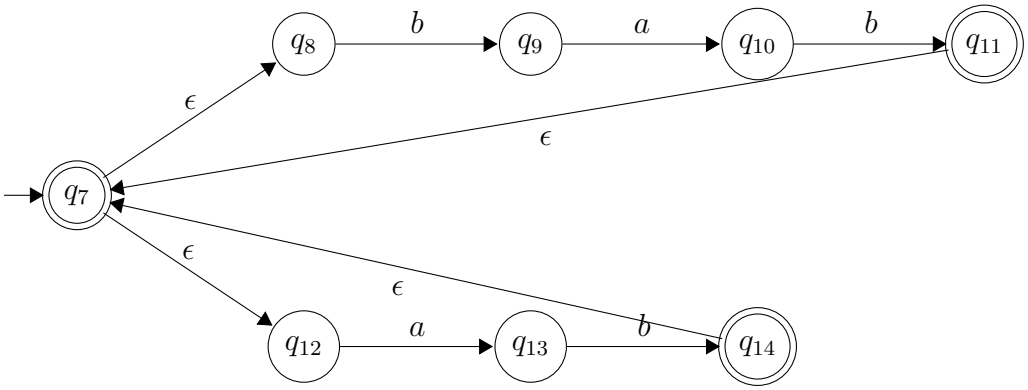
and then this:



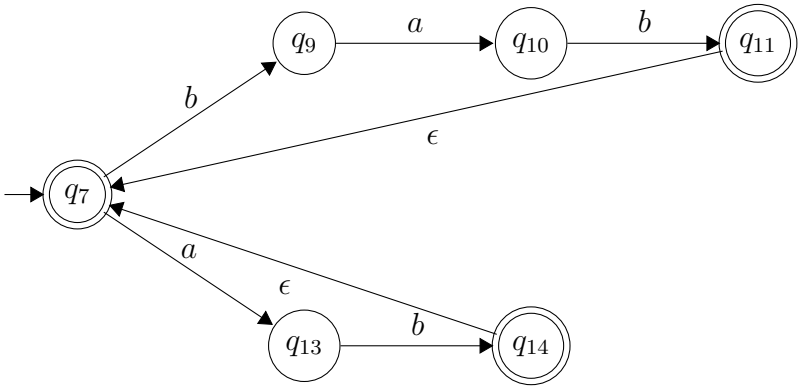
Now for

$$L((bab \cup ab)^*) = L(bab \cup ab)^*$$

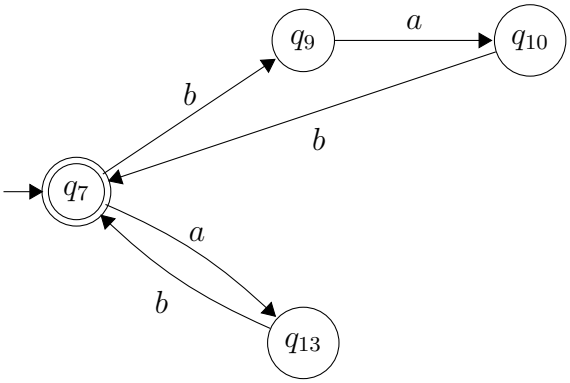
An NFA accepting this language is



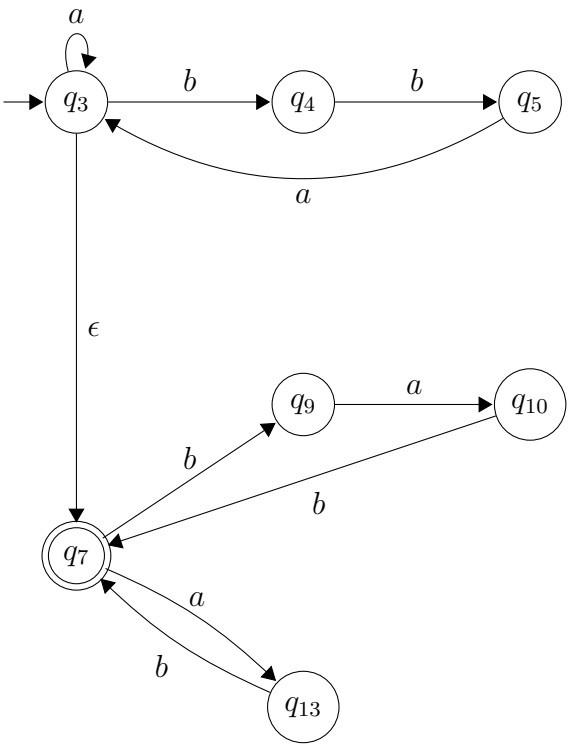
I'm going to simplify it to



and then this



And finally when I concat the two machines I get


a
b
b
q_3
q_4
q_5
a
ϵ
q_9
a
q_{10}
b
b
q_7
a
b
q_{13}

Exercise 12.18.1. Construct an NFA N such

$$L(N) = L(r)$$

where r is the regular expression

$$r = a(ba \cup abb \cup (ab)^*)^*$$

□

The big picture is this: Regular expressions are constructed starting from the symbols in Σ or $\emptyset$ (and you know how to construct NFAs for all these). You combine regular expressions recursively using unions, concatenation, and Kleene star (and you know how to construct the unions, concatenations, and Kleene stars of NFAs). Therefore the languages accepted by regex must be contained in the languages accepted by NFAs.

That's the main idea.

Formally, the right strategy is to proof the following statement

> If r is a regular expression, then there is an NFA N such that $L(N) = L(r)$.

by mathematical induction on the length of the regex (remember: a regex is a string.) In other words you should form this:

> $P(n)$: If r is a regular expression of length n, then there is an NFA N such that $L(N) = L(r)$.

You then prove this by induction. The base case starts at $n = 1$ since a regex have at least one character. The base cases are

$$r = c$$

where $c \in \Sigma$ or

$$r = \emptyset$$

It's easy to construct NFAs for these regex (of course!). So the base case is ... DONE!

Now you assume $P(1), P(2), \ldots, P(n)$ is true for some $n \geq 1$. Now you want to prove $P(n+1)$, i.e., given any regex r of length $n+1$, you want to prove that there is an NFA N such that

$$L(N) = L(r)$$

You can of course assume $n \geq 1$ since we're done with the base case. Now what?

Well, since the length of r is $n + 1 > 1$, r must be either

$$r = s^*$$

where s is a regex, or

$$r = st$$

where s and t are regexes, or

$$r = s \cup t$$

where s and t are regexes. (For simplicity, I'm ignoring the case where the regex has parenthesis, i.e., r is (s) – this is an easy case. So I'll leave that to you.) By definition, there are no other cases. Note that in the first case the length of s is $\leq n$ (in fact exactly equal to n). In the second case, the lengths of s and t are both $\leq n$. Likewise in the third case, the lengths of s and t are both $\leq n$. By induction hypothesis, for all cases, the s or the t (depending on the cases) are accepted by some NFAs N_1 and N_2.

In the first case, since

$$L(N_1) = L(s)$$

we have by definition

$$L(N_1)^* = L(s)^* = L(s^*)$$

We also know that the Kleene star operator is a closed operator. Since $L(N_1)$ is a regular language, this means that $L(N_1)^*$ is also regular. This means that there is some NFA/DFA N_1' such that

$$L(N_1') = L(N_1)^*$$

Altogether we have found some NFA/DFA N_1' such that

$$L(N_1') = L(N_1)^* = L(s^*) = L(r)$$

We have proven $P(n+1)$ for this case. All other cases are similar. (Go over the proof yourself.)

Therefore $P(n+1)$ is true for all cases.

By mathematical induction, $P(n)$ is true for all $n \geq 1$.

12.19 Myhill–Nerode theorem for regularity and DFA minimization debug: minimization.tex

PUT SOMEWHERE: Go ahead and review equivalence relations.

Here's the aim: Let M be a DFA. Can we find another DFA M' such that

- $L(M') = L(M)$
- M' is a DFA with the smallest number of states satisfying the above.

In other words I want to minimize M.

Why would you want to do that? Because a smaller DFA will execute faster and furthermore will require less memory (states are memory) and therefore cheaper to produce. This is the case whether the DFA is a piece of software or a piece of hardware. Therefore DFA minimization improves runtime, space, and cost.

I will say that two automatas are **equivalent** (whether they are DFAs or NFAs) if they accept the same language. equivalent

Exercise 12.19.1. Create three equivalent DFAs where one has two states, one has 3 states, and the last has 5 states. Prove that the one with two states is the minimal, i.e., you cannot find another DFA that accepts the same language and has 1 state. You have 2 minutes. □

The main result is this:

Theorem 12.19.1. *For any DFA, there is a minimum equivalent DFA that is unique up to relabeling of states.*

The result is not just "there is a minimal DFA". There are algorithms for converting a given DFA to a smallest one. I don't have time to talk about all of them. I'll focus on just one. And before explaining the algorithm, I'll do one example first so you get the main idea. Before that, I'll talk about isomorphism of DFAs.

In computer science and math, two objects of the same kind is said to be isomorphic (example: graph isomorphism) if they are essentially the same except for some renaming of the internals of the objects. You can also define isomorphism of DFAs. In that case, you can restate the theorem as saying that the minimal DFA exist and is unique up to isomorphism. Here's the

formal definition of DFA isomorphism: Two DFAs $M = (\Sigma, Q, q_0, F, \delta)$ and $M' = (\Sigma, Q', q_0', F', \delta')$ are **isomorphic** if there is a bijection

$$f : Q \to Q'$$

(f is the renaming) such that

$$f(q_0) = q_0'$$
$$f(F) = F'$$

and

$$\delta(q_i, x) = q_j \iff \delta(f(q_i), x) = f(q_j)$$

You can see that f is just a re-labeling of states so that the "behavior" of the states remain the same. For instance

$$f(q_0) = q_0'$$

means that the start state of M has been renamed as the start state of M'. The condition

$$f(F) = F'$$

means that if a state q in M is an accept state, then after being renamed to $f(q)$ in M', $f(q)$ is also an accept state in M'. The transition function renaming is a little bit more complicated. The condition

$$\delta(q_i, x) = q_j \iff \delta(f(q_i), x) = f(q_j)$$

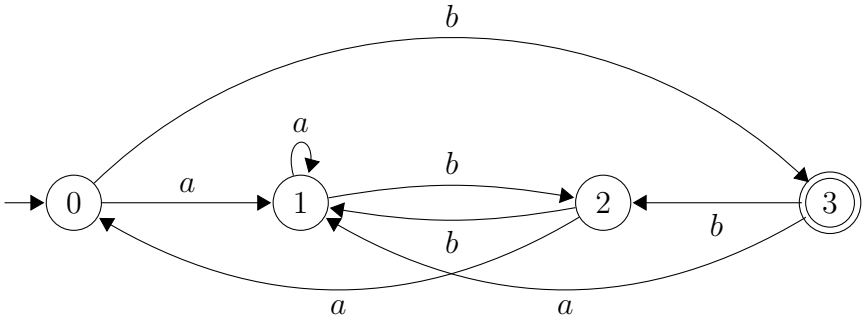
means that if at state q_i, on reading a character x, you transition to q_j, then after renaming q_i to $f(q_i)$ and q_j to $f(q_j)$, at state $f(q_i)$, on reading x, you also transition to state $f(q_j)$ (and vice versa).

Note that the alphabet is the same for M and M'. You can also talk about relabeling (or renaming) of alphabet too.

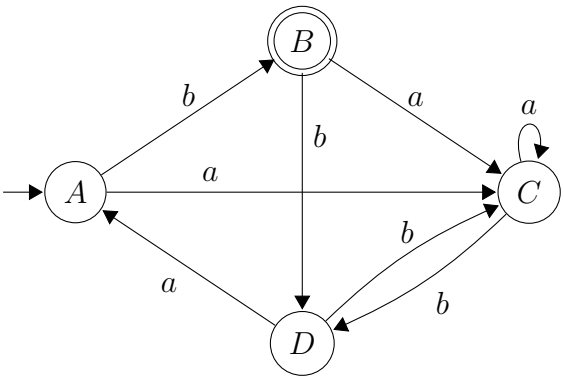
The general idea of isomorphism is extremely important in computer science and math (and therefore is also use heavily in physics, chemistry, engineering, etc.)

Note that the above talks about "sameness" in two different ways: equivalence of DFAs and isomorphism of DFAs. Isomorphism of DFAs is stronger.

Exercise 12.19.2. Consider the following two DFAs:



and



Are they isomorphic? □

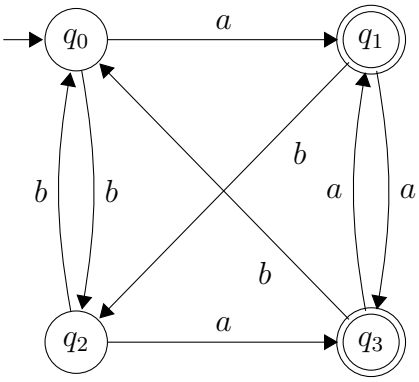
Exercise 12.19.3.

1. True or false:
   a) If M, M' are equivalent DFAs, then they are isomorphic.
   b) If M, M' are isomorphic DFAs, then they are equivalent.
2. Is equivalence of DFAs an equivalent relation?
3. Is isomorphism of DFAs an equivalent relation?

□

Exercise 12.19.4. What is the right definition of isomorphism if you want to rename the symbols in alphabets as well? (There's only one reasonable definition.) The definition should look like this: "An isomorphism between two DFAs $M = (\Sigma, Q, q_0, F, \delta)$ and $M' = (\Sigma', Q', q_0', F', \delta')$ is a pair of bijections $f : Q \to Q'$ and $g : \Sigma \to \Sigma'$ such that ..." Complete the definition. □

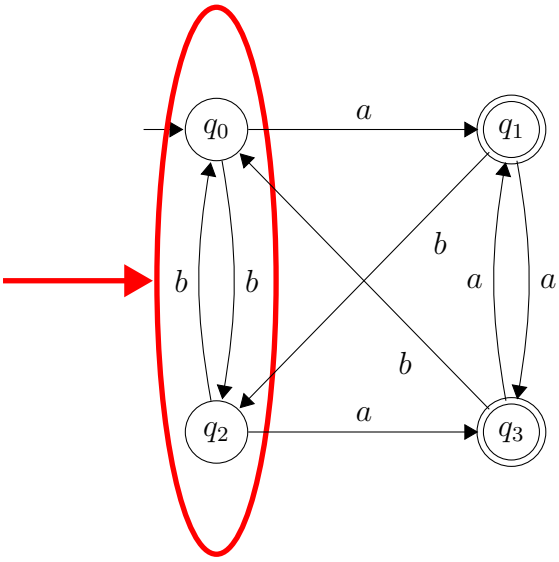
Let's look at this DFA:

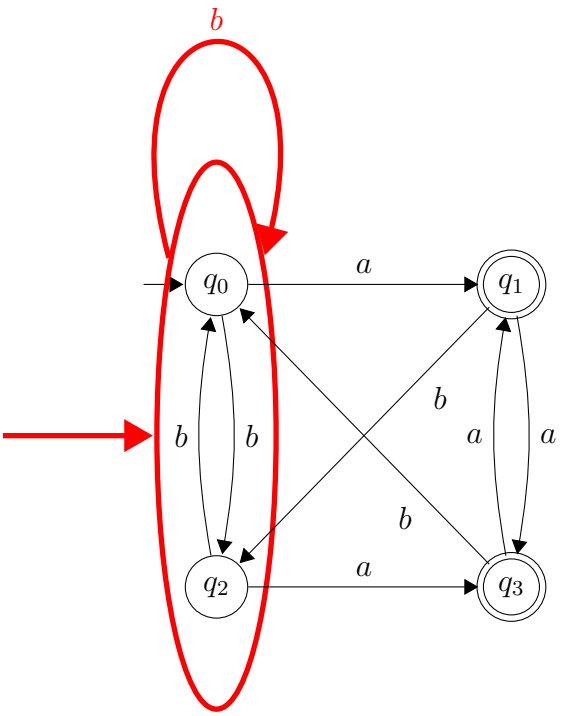


How would you simplify it? (Simplify = remove some state(s).)

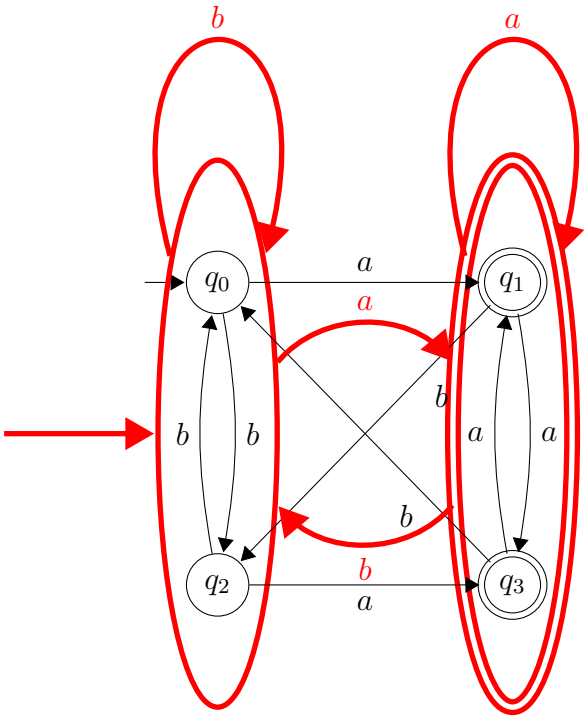
Pause here for 2 minutes and try to do it yourself ...

Now ... you can think of states are dumping ground for strings. For instance q_0 is the dumping ground (or final resting place) of the string ϵ, abb, aab, etc. If so, you can ask if you can "combine" q_0 and q_2 so that strings collected there can be combined into one dumping ground. (Of course it doesn't make sense to combine q_0 and q_1 – you can't put accepted and rejected strings in the same state!)

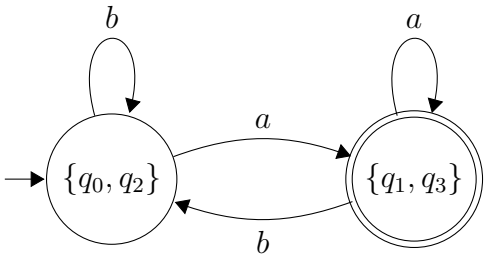




Notice that I have to draw a b–transition from $\{q_0, q_1\}$ to itself because there are b–transitions going between q_0 and q_1 in the original DFA. But notice this: In the original DFA, q_0 goes to q_1 by a and q_1 does to q_3 by a as well. But ... in the new DFA, q_0, q_1 can collapsed into one and are indistinguishable! In the new DFA, I would need to go to $\{q_0, q_1\}$ to q_1 and q_3 through a!!! That's not possible ... unless if q_1 and q_3 becomes one state. See that? At this point, that's possible since q_1 and q_3 are accept states. And I get this:



which gives me this:



As an example execution, look at the diagram with both DFAs and trace the execution of the string *abaa*. In the original DFA, the states visited are

$$q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_2 \xrightarrow{a} q_3 \xrightarrow{a} q_1$$

In the minimized DFA, the states are

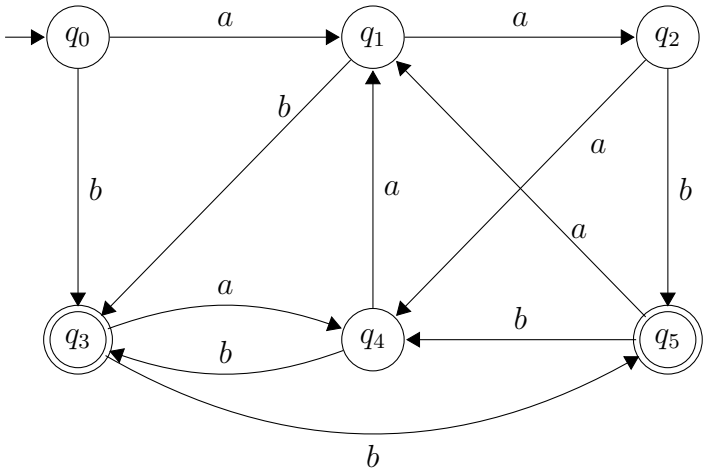
$$\{q_0, q_2\} \xrightarrow{a} \{q_1, q_3\} \xrightarrow{b} \{q_0, q_2\} \xrightarrow{a} \{q_1, q_3\} \xrightarrow{a} \{q_1, q_3\}$$

What will prevent us from grouping q_0, q_1 as a single state? Suppose x and y are two string such that x lands in q_0 and y lands in q_1 in the original DFA

M. Suppose I collapse q_0 and q_1 into one state, called it $\{q_0, q_1\}$ and call the new DFA M' which is supposed to minimize M. In this new DFA M', since x, y land in the same state $\{q_0, q_1\}$, the longest strings xz, yz must also land in the same since they are in the same state *before* z. If in the original DFA M, if $xz \in L$ and $yz \notin L$ (i.e., xz lands in an accept state and yz lands in a non-accept state), then the new DFA M' does not work correctly.

The above example of minimizing a 4–state DFA to a 2–state DFA is ad hoc (of course). We now need to create a robust algorithm to minimize *any* DFA. Before that, try to minimize the following DFA without looking at the theory below it.

Exercise 12.19.5. Try to minimize the following DFA (or is it already minimal?)



□

Let's fix some terminology and notation. Let me fix a language L (over Σ). Let $x, y \in \Sigma^*$. I will say that L **separates** x and y if

separates

$$x \in L, y \notin L \quad \text{or} \quad x \notin L, y \in L$$

I'll also say in this case that x, y are L–**separable**.

separable

I will say that x, y are L–**distinguishable** if there is *some* $z \in \Sigma^*$ such that xz and yz are L–separable, i.e.,

distinguishable

$$xz \in L, yz \notin L \quad \text{or} \quad xz \notin L, yz \in L$$

(Note the emphasize "there is *some*".) Of course x, y are not L–distinguishable if for all z, xz, yz are not L–separable. If x, y are L–distinguishable, I'll write

$$x \not\equiv_L y$$

$\not\equiv_L$

otherwise I'll write

$$x \equiv_L y$$

$\equiv_L$

(just to save on ink.)

Here's how to think of this concept: All strings in Σ^* will give you a path in a DFA (if there's one) for L. x, y are L–distinguishable, if after you walk along x and your friend walk along y, and then both of you manage to find some extra common steps z and one of you end up being *in* L and the other is *not in* L.

You can also obviously talk about a set of strings (more than two) being **pairwise** L–**distinguishable**, meaninng to say any pair of string in this set is L–distinguishable.

pairwise L–distinguishable

Example 12.19.1. Consider the following example:

$$L_1 = \{a^n b^n \mid n \geq 0\}$$

Let

$$x = a, \;\; y = b$$

Then x, y are L_1–distinguishable. Why? Because if I pick $z = b$, then

$$xz = ab \in L, \;\; yz = bb \notin L$$

However if

$$x = ba, \;\; y = b$$

then x, y are not L_1–distinguishable: Given *any* $z \in \Sigma^*$,

$$xz = baz, \;\; yz = bz$$

are both not not in L_1. □

Exercise 12.19.6. Again let

$$L_1 = \{a^n b^n \mid n \geq 0\}$$

1. Are ϵ, a L_1–separable? L_1–distinguishable?
2. Are aa, bb L_1–separable? L_1–distinguishable?
3. Are $ab, aabb$ L_1–distinguishable? L_1–distinguishable?
4. Find an example of x, y such that xz, yz are L_1–separated for z and xz', yz' are not L_1–separated for z'. (So you need to find four strings x, y, z, z'.)
5. Find three strings which are pairwise L_1–distinguishable. How about four? Five?
6. Is this true? Given any positive $n \geq 2$, it's possible to find n strings which are pairwise L_1–distinguishable.

Example 12.19.2. Here's another example. Let

$$L_2 = \{w \in \Sigma^* \mid |w|_a, |w|_b \text{ both even}\}$$

where $|w|_a$ is the number of a's in w and $|w|_b$ is the number of b's in w. For example $|abaab|_a = 3$ and $|abaab|_b = 2$. Therefore $abaab \notin L_2$. However $abbaaa \in L_2$.

(a) Are ϵ, a L_2–separable? L_2–distinguishable?
(b) Are ϵ, b L_2–separable? L_2–distinguishable?
(c) Are ϵ, aa L_2–separable? L_2–distinguishable?
(d) Are ϵ, ab L_2–separable? L_2–distinguishable?
(e) Are ϵ, bb L_2–separable? L_2–distinguishable?
(f) Are $abaa, abababa$ L_2–distinguishable?

SOLUTION. (a) $\epsilon \in L_2$ and $a \notin L_2$. Hence ϵ, a are L_2–separable. Choose $z = \epsilon$, $\epsilon z, az$ are L_2–separable. Hence ϵ, a are L_2–distinguishable.

(b) For you.

(c) $\epsilon, aa \in L_2$. Hence ϵ, aa are not L_2–separable. They are also not L_2-distinguishable.

(d) $\epsilon \in L_2$ and $ab \notin L_2$. Hence ϵ, ab are L_2–separable. They are also L_2-distinguishable.

(e), (f) For you. □

Let

$$x = abb, \quad y = ababa$$

Then x, y are not L_2–separable because the number of a's in both are odd and the number of b's in both are odd:

$$|x|_a \equiv |y|_a \equiv 1 \pmod 2$$
$$|x|_b \equiv |y|_b \equiv 1 \pmod 2$$

(Any one of the above condition is actually enough.) If z has an even number of a's and an even number of b's:

$$|z|_a \equiv 0 \pmod 2$$
$$|z|_b \equiv 0 \pmod 2$$

Then

$$|xz|_a = |x|_a + |z|_a \equiv 1 + 0 \equiv |y|_a + |z|_a = |yz|_a \pmod 2$$

and so xz, yz will both have an odd number of a's. Yes? And so again xz, yz are not L_2–separable.

Exercise 12.19.7.

(a) Suppose x, y are L_2–separable and $z \in \Sigma^*$. What can you tell me about xz, yz?
(b) Suppose x, y are not L_2–separable and $z \in \Sigma^*$. What can you tell me about xz, yz?
(c) Suppose x, y are not L_2–separable and $z \in \Sigma^*$. What can you tell me about L_s–distinguishability of xz, yz?
(d) Suppose x, y are not L_2–separable and $z \in \Sigma^*$. What can you tell me about L_s–distinguishability of xz, yz?

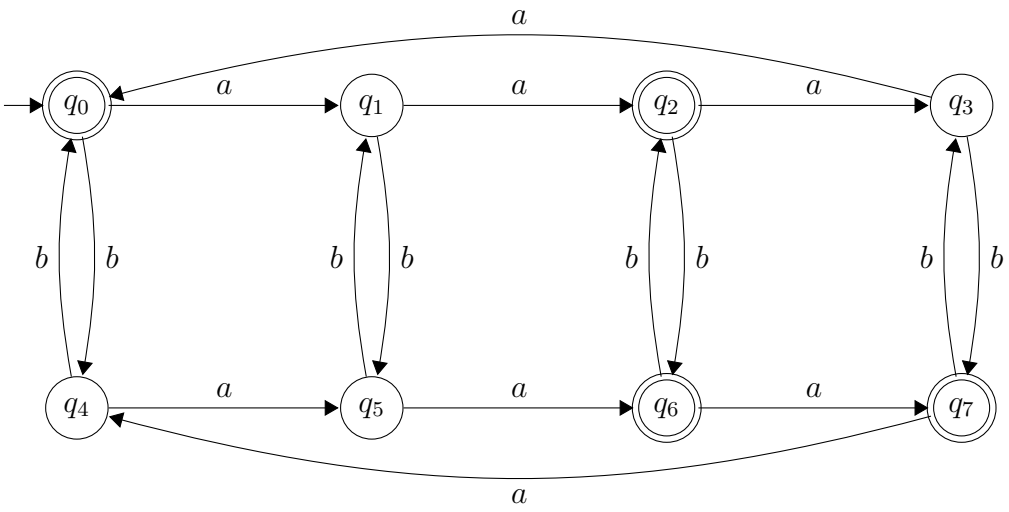
Exercise 12.19.8. Again let

$$L_2 = \{w \in \Sigma^* \mid |w|_a, |w|_b \text{ both even}\}$$

(a) Describe x, y if they are not L_2–separable.
(b) If x, y are not L_2–separable, what is the simplest z to make x, y L_2–separable.
(c) If x, y are L_2–separable, what is the simplest z to make not x, y L_2–separable.
(d) Show that ϵ, a are L_2–distinguishable.
(e) If ϵ, a, b pairwise L_2–distinguishable?
(f) Find a set of 4 words which are L_2–distinguishable.
(g) Try (if possible) to find a set of 5 words which are L_2–distinguishable.

□

Example 12.19.3. Let M be the following DFA:



Let $L = L(M)$.

(a) Are ϵ, a^4 L-distinguishable?
(b) Are ϵ, a L-distinguishable?
(c) Are ϵ, a^2 L-distinguishable?
(d) Are ϵ, a^3 L-distinguishable?
(e) Are ϵ, b L-distinguishable?
(f) Are ϵ, ba L-distinguishable?
(g) Are ϵ, ba^2 L-distinguishable?
(h) Are ϵ, ba^3 L-distinguishable?

Note that L–distinguishability does not depend on the concept of DFA. The following is a pretty obvious statement on the relationship between L–distinguishability and DFAs. Let $\text{state}_M(x)$ denotes the state that x lands in if M computes with x, i.e.,

$$\text{state}_M(x) = \delta^*(q_0, x)$$

where q_0 is the start state and δ is the transition function of M.

Proposition 12.19.1. *Let M be a DFA and $L = L(M)$. Let $x, y \in \Sigma^*$ and p, q be the respective states reached when x, y are computed with M. If x, y are L–distinguishable, then $\text{state}_M(x) \neq \text{state}_M(y)$.*

Proof. The proof is obvious. Suppose x and y lands in the same state. Then for any $z \in \Sigma^*$, xz and yz will both also land in a common state. Of course the state is either an accept state of non–accept state, which means that xz and yz cannot be separated by L. This contradicts the L–distinguishability of x, y. □

Proposition 12.19.2. *Let L be a language. If $L = L(M)$ where M is a DFA and L has k pairwise L–distinguishable strings, then M has at least k states.*

Proof. Let $x_0, \ldots, x_{k-1}$ be a set of k pairwise L–distinguishable strings. Suppose M has less than k states. Running M on the strings $x_0, \ldots, x_{k-1}$, we will obtain k states. By the pigeonhole principle, since M has less than k states, two of the strings in $x_0, \ldots, x_{k-1}$ must land in the same state. But this is a contradiction (by the previous statement) since L–distinguishable strings cannot land in the same state. □

Exercise 12.19.9.

1. Let $L_1 = \{a^n b^n \mid n \geq 0\}$. Prove that L_1 is not a regular language.
2. Let $L_2 = \{w \in \Sigma^* \mid |w|_a, |w|_b \text{ both even}\}$. What is the size of the smallest DFA? Construct a minimal DFA for L_2. Note that the results above only tells you a lower bound on the number of states of a DFA for L_2. It does not say that the number stated is attained. The results above also does not tell you how to construct a DFA.

Note that the previous statements tell you when you have the smallest DFA for a language. But suppose a language has 5 distinguishable strings. At this point, you only know that a DFA has at least 5 states. Does the minimal DFA have *exactly* 5? The above statement does not address this question.

Another thing to note is that we already have many techniques for designing DFAs. What we need is to use those techniques to build a DFA, and *then* minimize *that* DFA just built.

So here we go ...

First, recall that I have a relation $\equiv_L$ on strings in Σ^*. Recall that $\not\equiv_L$ means L–distinguishability. In other words

$$x \not\equiv_L y$$

if there is some $z \in \Sigma^*$ such that L separates xz, yz. In other words

$$x \equiv_L y$$

if for all $z \in \Sigma^*$, L does not separate xz, yz.

Proposition 12.19.3. *$\equiv_L$ is an equivalence relation on Σ^*. In other words:*

(a) *Reflexivity: For all $x \in \Sigma^*$, $x \equiv_L x$.*
(b) *Symmetry: For all $x, y \in \Sigma^*$, if $x \equiv_L y$, then $y \equiv_L x$.*
(c) *Transitivity: For all $x, y, w \in \Sigma^*$, if $x \equiv_L y, y \equiv_L w$, then $x \equiv_L w$.*

Proof. Easy exercise. □

This means that Σ^* is partitioned into a collection of disjoint sets of strings. The fancy term for such a set is **equivalence class**. In other words the collection of equivalence classes look like $[x_0], [x_1], [x_2], ..., [x_{k-1}]$ where

equivalence class

$$[x_i] = \{x \in \Sigma^* \mid x_i \equiv_L x\}$$

Here's I'm assuming the number of equivalence classes is finite. It can be infinite – see exercise on showing $L_1 = \{a^n b^n \mid n \geq 0\}$ is not regular.

Remember what I said: If there are k equivalence classes, it means that a DFA for L (if there's one at all) has at least k states. It turns out the k *is* actually the correct number of states for the smallest DFA:

Theorem 12.19.2. Myhill–Nerode (1958) *Let L be a language (over Σ^*).* Myhill–Nerode (1958)

(a) *If $\equiv_L$ (over Σ^*) has infinitely many equivalence classes then L is not regular.*

(b) *If $\equiv_L$ (over Σ^*) has finitely many equivalence classes, say k, then L is regular and there is a DFA M with exactly k states, where each state correspond to an equivalence class of $\equiv_L$. Specifically if $[x_0], [x_1], [x_2], ..., [x_{k-1}]$ where $x_0 = \epsilon$, are the equivalence classes of $\equiv_L$, then we define $M = (\Sigma, Q, q_0, F, \delta)$ to be*

a) The set of states Q is $\{[x_0], [x_1], [x_2], ..., [x_{k-1}]\}$.
b) The start state is $[x_0]$.
c) The set of final states F consists of those $[x_i]$ where $x_i \in L$.
d) The transition function $\delta : Q \times \Sigma \to Q$ is defined as follows. If $c \in \Sigma$,

$$\delta([x_i], c) = [x_i c]$$

then $L(M) = L$.

Before proving Myhill–Nerode's theorem, let me use it ...

Example 12.19.4. Let's try to compute the minimal DFA for

$$L_2 = \{w \in \Sigma^* \mid |w|_a, |w|_b \text{ both even}\}$$

SOLUTION. Recall from the earlier exercises, there are exactly four L_2–distinguishable strings:

$$\epsilon, a, b, ab$$

Hence $\equiv_{L_2}$ partitions Σ^* into four equivalence classses:

$$[\epsilon], [a], [b], [ab]$$

where

$$\begin{aligned}
[\epsilon] &= \{x \in \Sigma^* \mid \epsilon \equiv_L x\} = \{x \in \Sigma^* \mid |x|_a, |x|_b \equiv 0, 0 \pmod 2\} \\
[a] &= \{x \in \Sigma^* \mid a \equiv_L x\} = \{x \in \Sigma^* \mid |x|_a, |x|_b \equiv 1, 0 \pmod 2\} \\
[b] &= \{x \in \Sigma^* \mid b \equiv_L x\} = \{x \in \Sigma^* \mid |x|_a, |x|_b \equiv 0, 1 \pmod 2\} \\
[ab] &= \{x \in \Sigma^* \mid ab \equiv_L x\} = \{x \in \Sigma^* \mid |x|_a, |x|_b \equiv 1, 1 \pmod 2\}
\end{aligned}$$

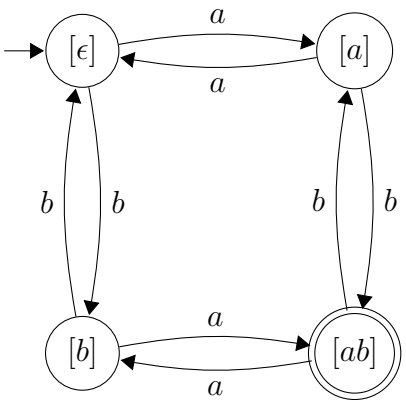
The start state is $[\epsilon]$. The set of accept states is

$$F = \{[\epsilon]\}$$

since among the string ϵ, a, b, ab, ϵ is the only string in L. The transition function δ is given by the following:

- $\delta([\epsilon], a) = [\epsilon a] = [a]$.
- $\delta([\epsilon], b) = [\epsilon b] = [b]$.
- $\delta([a], a) = [aa] = [\epsilon]$. This is because $aa \equiv_{L_2} \epsilon$.
- $\delta([a], b) = [ab]$.
- $\delta([b], a) = [ba] = [ab]$. This is because $ba \equiv_{L_2} ab$.
- $\delta([b], b) = [bb] = [\epsilon]$. This is because $bb \equiv_{L_2} \epsilon$.
- $\delta([ab], a) = [aba] = [b]$. This is because $aba \equiv_{L_2} b$.
- $\delta([ab], b) = [abb] = [a]$. This is because $abb \equiv_{L_2} a$.

The above describes the DFA completely. Here's the diagram:


$[\epsilon]$
$[a]$
$[b]$
$[ab]$
a
a
b
b
b
b
a
a

Now let's prove Myhill–Nerode's theorem.

Proof. (a) If $\equiv_L$ has infinitely many equivalence classes, that means that there are infinitely many L–distinguishable strings and we have already mentioned that this means that for every $k > 0$, a DFA for L (if it exists) must have at least k states. But this means that M has arbitrarily many states. That's impossible since M must have a finite number of states by definition. (I actually already talked about this earlier.)

(b) First let me show that δ is well–defined, i.e., I need to show that if $x_i \equiv_L x$, then

$$\delta([x_i], c) = [x_i c] = [x'c] = \delta([x], c)$$

In other words, I need to show

$$x_i c \equiv_L x' c$$

Again, I need to show

$$x_i \equiv_L x \implies x_i c \equiv_L x' c$$

$x_i \equiv_L x$ means that for all $z \in \Sigma^*$, L does not separate $x_i z, xz$. Let $z' \in \Sigma^*$. Then $x_i(cz'), x(cz)$ are not separated by L. Hence $(x_i c)z', (xc)z$ are not separated by L, which implies that $x_i c, xc$ are not L–distinguishable, i.e., $x_i c \equiv_L xc$.

Clearly M is a DFA. Now I need to show $L(M) = L$. Let $x \in \Sigma^*$.

$$x \in L(M) \iff \delta^*([\epsilon], x) \in F$$

Note that

$$\delta^*([\epsilon], x) = [x]$$

for any $x \in \Sigma^*$. (This can be proven by induction.) Hence

$$\begin{aligned} x \in L(M) &\iff [x] \in F \\ &\iff \exists x_i \in L \text{ such that } [x] = [x_i] \\ &\iff \exists x_i \in L \text{ such that } x \equiv_L x_i \\ &\iff \exists x_i \in L \text{ such that } \forall z, L \text{ does not separate } xz, x_i z \end{aligned}$$

In particular, if $x \in L(M)$, then there is some $x_i \in L$ such that for $z = \epsilon$, $xz = x$ and $x_i z = x_i$ are not L separable. Since $x_i \in L$ and L does not separate x, x_i, $x \in L$. I have just shown

$$x \in L(M) \implies x \in L$$

Hence $L(M) \subseteq L$. Now let $x \in L$, say $[x] = [x_i]$.

$$\delta^*([\epsilon], x) = [x] = [x_i]$$

This implies that for all z, L does not separate $xz, x_i z$. In particular when $z = \epsilon$, L does not separate x, x_i. Since $x \in L$, then $x_i \in L$. Hence

$$\delta^*([\epsilon], x) = [x] = [x_i] \in F$$

Therefore $x \in L$. Hence $L \subseteq L(M)$. Altogether I have shown $L = L(M)$. $\square$

The easiest thing to do is to remove states which are not reachable from the start state. Simply perform a breadth-first traversal. For instance (this is not optimized):

```
Let S be the set of reachable states. Initialize is with the start state.

Repeatedly, put all states reachable from S into S.
Repeat the above until S is not changed.
```

After the above is done, only the states in S need to be considered. The states in $Q - S$ are thrown away. Of course F is replaced by $F - S$ and δ is updated so that only states in $Q - S$ are considered part of δ.

First we'll find a smaller DFA. Later we'll prove that it's the smallest using Myhill-Nerode's Theorem.

Throughout this section, $M = (\Sigma, Q, q_0, F, \delta)$ will be a DFA where $Q = \{q_0, \ldots, q_{n-1}\}$

The idea is to collapse states ... equivalence relations to the rescue!

Let L be a language (over Σ). Given two strings x, y in Σ^*, I will say that L separates x, y if there is some $z \in \Sigma^*$ such that either $xz \in L, yz \notin L$ or $xz \notin L, yz \in L$. I will further write

$$x \equiv_L y$$

if there is no z such that L separates xz, yz. In other words

$$x \equiv_L y$$

iff

$$\text{for all } z \in \Sigma^*, xz \in L \iff yz \in L$$

Proposition 12.19.4. *$\equiv_L$ is an equivalence relation on Σ^**

Proof. Exercise. □

What is the intuition here?

Consider the example of

$$L = \{x \in \{a, b\}^* \mid |x| \equiv 1 \bmod 8\}$$

Let M be the DFA with 8 states that accepts $w \in \Sigma^* = \{a, b\}^*$ where the number of a's in w is $\equiv 1$ or $5 \pmod 8$. We do know that L is regular with a DFA with 8 states. Notice that the string Σ^* will land in the states. So we can think of classes of strings as being same as states, i.e., each state can be thought of as a collection of strings from Σ^*. For instance if $|x| = 6$ and $|y| = 8k + 6$, then

$$x \equiv_L$$

Hence Σ^* is partitioned into 8 sets of strings in the obvious way. The partitions are

- $[\epsilon]$
- $[a] = [b]$
- $[a^2] = [b^2] = [ab] = [ba]$
- $[a^3] = [b^3]$
- $[a^4]$
- $[a^5]$
- $[a^6]$
- $[a^7]$

Exercise 12.19.10. Prove that $\equiv_M$ is an equivalence relation on Σ^*. Therefore Σ^* is partitioned into equivalence classes of strings. We will write $[x]_M$ for the equivalence class containing x, i.e.,

$$[x]_M = \{y \in \Sigma^* \mid x \equiv_M y\}$$

We will write index($\equiv_M$) for the number of equivalence classes of Σ^* under $\equiv_M$.

$\equiv_M$ is more than just an equivalence relation. Note that if $x \equiv_M y$, then they remain equivalence (wrt $\equiv$) even when you extend them "on their right":

Exercise 12.19.11. Let M be a DFA. Prove that $\equiv_M$ is right invariant. □

The main idea is that the equivalence classes of L under $\equiv_M$ will be the states. [?]

Now we define an equivalence relation on *any* language L. Note that L need not be regular:

Definition 12.19.1. Let L be any language. Define the following relation on Σ^*. For $x, y \in \Sigma^*$,

$$x \equiv_L y \quad \text{if} \quad \text{for all } z,\ xz \in L \iff yz \in L$$

This is the same as saying that $x \equiv_L y$ if for all $z \in \Sigma^*$, either

- xz, yz both in L, or
- xz, yz both not in L.

We will write $[x]_L$ for the equivalence class of x under $\equiv_L$.

Example 12.19.5. Let $\Sigma = \{a, b\}$. For simplicity we will write $|w|_a$ for the number of a's in w and $|w|_b$ for the number of b's in w. Let $L = \{w \in \Sigma^* \mid |w|_a, |w|_b \text{ are even}\}$.

Look at the conditions: $|w|_a$ is even and $|w|_b$ is even. Notice that if you define the following sets

1. $L_{ee} = \{w \in \Sigma^* \,|\, |w|_a \text{ even}, |w|_b \text{ even}\}$
2. $L_{eo} = \{w \in \Sigma^* \,|\, |w|_a \text{ even}, |w|_b \text{ odd}\}$
3. $L_{oe} = \{w \in \Sigma^* \,|\, |w|_a \text{ odd}, |w|_b \text{ even}\}$
4. $L_{oo} = \{w \in \Sigma^* \,|\, |w|_a \text{ even}, |w|_b \text{ even}\}$

Obviously $L_{ee}, L_{eo}, L_{oe}, L_{oo}$ forms a partition of L (Recall: this means they are pairwise disjoint and their union is L.)

When we look the equivalence relation on L, we see that

1. $L_{ee} = [\epsilon]$
2. $L_{eo} = [b]$
3. $L_{oe} = [aaabb]$
4. $L_{oo} = [baaabb]$

Not only that, we can actually use these equivalence classes of strings to create a DFA in the following way. If $x \in \Sigma^*$ and $a \in \Sigma$, we just define

$$\delta([x]_L, a) = [xa]_L$$

This is the smallest (in terms of number of states) DFA that accepts L.

Example 12.19.6. Now consider $L = \{w \in \Sigma \,|\, w \text{ does not contain aab }\}$ where $\Sigma = \{a, b\}$.

- Is $ba \equiv_L baa$?
- Suppose w, w' both contain aab. Is it true that $w \equiv_L w'$?

OK. Here's the big theorem of this section:

Theorem 12.19.3. *(Myhill-Nerode) Let L be a language over Σ. The following are equivalent:*

(a) L is regular
(b) There is a right invariant equivalence relation $\equiv$ on L such that $\equiv$ has finite index and L is the union of some equivalence classes of $\equiv$.
(c) $\equiv_L$ has finite index.

I will not give a proof. However the important thing is that the proof of (c) $\implies$ (a) is constructive. In particular, the DFA constructed uses equivalence classes of $\equiv_L$ as states.

The important thing you should realize by reading this theorem is that this is another way of characterizing regular languages. This characterization does not rely on the definition of DFA. In other words you can now re-define what is meant by regular in terms of $\equiv_L$ and the finiteness of the equivalence classes of $\equiv_L$. Theorems like these are extremely important because they reveal connections between different concepts and reveal different points of view. Note also that if you can write down infinitely many strings which are not equivalent to each other, then L is not regular.

Theorem 12.19.4. *(DFA Minimization) If L is regular and M is a DFA with the minimum number of states, then M is the "same" as the DFA constructed in the Myhill-Nerode theorem.*

Here "same" can be formalize in terms of isomorphisms of DFA. It means "the same up to renaming the states". For instance if you take a DFA with states labeled $q_0, q_1, \ldots, q_{n-1}$ and relabel them as $s_0, s_1, \ldots, s_{n-1}$, then obviously the way they both operate is exactly the same except for the names of the states.

The important thing for us is how do we simplify a DFA? First of all the simplest thing to do first is to remove states that cannot be reached from the initial state. For a human being, if the DFA is simple, then we can do it visually. In terms of an algorithm, you need to process the states in the following manner. First you need a list of states called B. This will be the list of states you will visit. You also need a list of visited states; let's call this A. You initialize B with the initial state and initialize A to be empty. Now in a while loop, take a state q out of B, visit all the states you can go to from q and put them into B, and put q into A. Repeat this loop as long as B is not empty. When you're done, A contains all the states that can be reached from the initial state.

The other question is how do we merge the visited states?

We will define another relation on states $\equiv$ so that states that should merge are equivalence to each other. Let p, q be states. Then we say that p and q are indistinguishable, and we write

$$p \equiv q$$

if there is some z such that either

- $\delta(p, z) \in F$ and $\delta(q, z) \notin F$, or
- $\delta(p, z) \notin F$ and $\delta(q, z) \in F$

All you need to do is to merge indistinguishable states into a new state.

Here's a systematic way of telling if pairs of states are distinguishable or not:

[insert]

There is a more efficient algorithm (I won't do it here) that will build the minimum DFA with time complexity $O(n \lg n)$ where $n = |Q|$.

12.20 Pumping Lemma debug: pumpinglemma.tex

So far we have to been working with regular languages. The question now to ask is this: Are there languages which are not regular? If the answer is no, then in a sense we know how to compute with language: we simply use DFAs (or NFAs or regex.) However if the answer is YES, then we would need to ask if there are other devices which are more "powerful" than DFA, NFA and regex?

It turns out that there are in fact non-regular languages. Recall that Myhill-Nerode's theorem can tell you exactly when a language is not regular. However Myhill–Nerode is hard to use. It's simple to use this theorem:

Pumping Lemma

Theorem 12.20.1. Pumping Lemma**.** *Let L be a regular language over Σ. There is a number n such that for each $x \in L$ with $|x| \geq n$, there exists $u, v, w \in \Sigma^*$ such that*

(a) $x = uvw$
(b) $|uv| \leq n$
(c) $|v| > 0$
(d) $uv^i w \in L$ *for all* $i \geq 0$.

That's quite a long theorem! By the way, do you realize that the conclusion of the theorem says something like this:

$$\exists n \left(\forall x \left(\exists u, v, w(...)\right)\right)$$

Yikes! Sure looks a little like the ϵ–δ definition of limits in calculus: $\forall \epsilon(\exists \delta(...))$

Corollary 12.20.1. *Let L be a language over Σ. Then L is non-regular if for any $n \geq 0$, there exists $x \in L$ with $|x| \geq n$ such that if $x = uvw$ for any $u, v, w \in \Sigma^*$*

(a) $|uv| \leq n$
(b) $|v| > 0$
(c) $uv^{i_0} w \notin L$ *for some* $i_0 \geq 0$.

I repeat: If you want to prove L is not regular, then you need to show the following. For each positive integer $n \geq 0$, you must write down a string $x \in L$ such that

(a) $|x| \geq n$
(b) x satisfied the following property: Let $u, v, w \in \Sigma^*$ be any strings such

that $x = uvw$, $|uv| \leq n$ and $|v| > 0$, you must find some integer $i_0 \geq 0$ such that $uv^{i_0}w \notin L$.

So here's a template of the proof that a language L is not regular.

We want to show L is not regular.

Let $n \geq 0$ be a positive integer. Let $x =$ _____. x is in L because _____ (EXPLAIN ONLY IF NOT OBVIOUS).

Let u, v, w be any string in Σ^* such that $x = uvw$ and

(a) $|uv| \leq n$
(b) $|v| > 0$
(c) $uv^{i_0}w \notin L$ for some $i_0 \geq 0$.

_____(ANALYZE u, v, w)_____

Let $i_0 =$ _____ and consider $uv^{i_0}w$.

_____(ANALYZE $uv^{i_0}w$)_____

Therefore $uv^{i_0}w \notin L$.

Hence by the Pumping Lemma for regular languages, L is not regular. □

Lemma 12.20.1. *Let $G = (V, E)$ be a graph with n nodes. If P is a path with n node, then P must contain a cycle. Specifically if $P = (e_1, \ldots, e_{n-1})$ where $e_i \in E$ for $1 \leq n \leq n-1$, then there exists k, ℓ such that $1 \leq k \leq \ell \leq n-2$ if $P_0 = (e_1, \ldots, e_k)$, $P_1 = (e_{k+1}, \ldots, e_\ell)$, $P_2 = (e_{\ell+1}, \ldots, e_{n-1})$ then $P_0P_1^iP_2$ is a path in G for all $i \geq 0$.*

Exercise 12.20.1. Prove the above lemma.

Exercise 12.20.2. Prove the pumping lemma.

When we prove certain languages are non-regular, we will frequently look at substrings within a larger strings. For instance suppose x is a string such that

$$x = x_1x_2 = y_1y_2$$

where x_1, x_2, y_1, y_2 are strings. Suppose further that $|x_1| = 5$ and $|y_1| = 10$, then x_1 must clearly be a substring of y_1. Right?

The general (obvious) fact is this: If

$$x_1x_2 = y_1y_2$$

where x_1, x_2, y_1, y_2 are strings and

$$|x_1| \leq |y_1|$$

then x_1 is a substring of y_1. (In fact it's a left substring, i.e. a prefix, of y_1.)

In particular, if

$$x_1x_2 = a^nb^n$$

and $|x_1| \leq n$, then x_1 must be a substring of a^n. This means that x_1 can only be made up of a's. Right? Since the length of x_1 is written $|x_1|$, we must have

$$x_1 = a^{|x_1|}$$

Note that since x_1 takes up $|x_1|$ number of a's, the amount of a's left is $n-|x_1|$. Therefore the above also imply that

$$x_2 = a^{n-|x_1|}b^n$$

Here's another example. Let

$$x_1x_2 = a^mb^{2m}$$

where $|x_1| < m$. Do you see that

$$x_1 = a^{|x_1|}$$

and

$$x_2 = a^{m-|x_1|}b^{2m}$$

Example 12.20.1. $L = \{a^i b^i \mid i \geq 0\}$ is not regular.

Solution. Let $n \geq 0$. We choose $x = a^n b^n$. Note that $x \in L$ and $|x| = 2n \geq n \geq 0$. Let u, v, w be strings in Σ^* such that

(a) $x = uvw$
(b) $|uv| \leq n$
(c) $|v| > 0$

Since $|uv| \leq n$, and uv is a prefix of $x = a^n b^n$. This implies that uv is made up of only a's. Therefore

$$u = a^{|u|}, \quad v = a^{|v|}$$

We also have

$$w = a^{n-|u|-|v|} b^n$$

Let $i_0 = 2$. Note that

$$uv^{i_0}w = uv^2w = a^{|u|}(a^{|v|})^2 a^{n-|u|-|v|} b^n = a^{|u|+2|v|+|n|-|u|-|v|} b^n = a^{n+|v|} b^n$$

Since $|v| > 0$, we have

$$n + |v| > n$$

Therefore

$$uv^{i_0}w = a^{n+|v|} b^n \notin L$$

Therefore by the Pumping Lemma for regular languages, L is not regular. $\square$

[Informally, since $|uv| \leq n$, uv must be a substring of a^n in $x = a^n b^n$. Also note that since $|v| > 0$, uv^2w would increase the number of a's in in $a^n b^n$ without changing the number of b's.]

Exercise 12.20.3. Prove that $L = \{a^n b^{2n} \mid n \geq 0\}$ is not regular.

Solution.

Let $n \geq 0$. We choose $x = a^n b^{2n}$. Note that $x \in L$. Furthermore $|x| = 3n \geq n$. Let u, v, w be strings in Σ^* such that

(a) $x = uvw$
(b) $|uv| \leq n$
(c) $|v| > 0$

Since uv is a prefix of $x = a^n b^{2n}$ and $|uv| \leq n$, uv must be a substring of a^n. Hence u, v are both substrings of a^n and we have

$$u = a^{|u|}, \quad v = a^{|v|}$$

We also have

$$w = a^{n-|u|-|v|} b^{2n}$$

Let $i_0 = 2$. Then

$$uv^{i_0} w = uv^2 w = a^{|u|}(a^{|v|})^2 a^{n-|u|-|v|} b^{2n} = a^{|u|+2|v|+n-|u|-|v|} b^{2n} = a^{n+|v|} b^{2n}$$

Therefore by the Pumping Lemma for regular languages, we conclude that L is not regular.

Exercise 12.20.4. Prove that $L = \{a^n b^{2n+1} \mid n \geq 0\}$ is not regular.

Exercise 12.20.5. Prove that $L = \{ab^n c^{2n} \mid n \geq 0\}$ is not regular.

Exercise 12.20.6. Is $L = \{b^n a b^n \mid n \geq 0\}$ regular? If you think it is, you must either construct a DFA/NFA/regex for L. If not, then you must prove that it's not using for instance the pumping lemma (for regular languages.)

Exercise 12.20.7. Is $L = \{a^n b a^n \mid n \geq 0\}$ is regular?. Prove your claim.

Exercise 12.20.8. Is $L = \{a^m b^n \,|\, 0 \leq m \leq n\}$ is regular?. Prove your claim.

Exercise 12.20.9. Is $L = \{a^m b^n \,|\, 0 \leq m < n\}$ is regular?. Prove your claim.

Exercise 12.20.10. Is $L = \{a^n b^n a^m b^m \mid n \geq 0, \ \ m \geq 0\}$ regular? Prove your claim.

Example 12.20.2. Prove that $L = \{a^{i^2} \mid i \geq 0\}$ is not regular.

Solution. Let $n \geq 0$ and $x = a^{n^2}$. Note that $x \in L$ and $|x| \geq n$.

Let $x = uvw$ where $u, v, w \in \Sigma^*$, $|uv| \leq n$ and $|v| > 0$. We will show that $uv^2w \notin L$.

Consider $uv^2w = a^{n^2+|v|}$. We now analyze $n^2 + |v|$. Note that $|v| \leq |uv| \leq n$. By assumption $|v| > 0$. Therefore altogether we have $0 < |v| \leq n$ and hence

$$n^2 < n^2 + |v| \leq n^2 + n$$

Now we note that

$$n^2 + n < n^2 + 2n + 1 = (n + 1)^2$$

Altogether we have shown that

$$n^2 < n^2 + |v| < (n + 1)^2$$

There are obviously no squares of integers in the interval $(n^2, (n+1)^2)$ (why?). Hence $n^2 + |v|$ cannot be a square. Therefore $wv^2w = a^{n^2+|v|} \notin L$.

Therefore by the Pumping Lemma for regular languages, L is not regular.

Exercise 12.20.11. Prove that $L = \{a^{n^2+1} \mid n \geq 0\}$ is not regular.

Exercise 12.20.12. Prove that $L = \{a^{n^3} \mid n \geq 0\}$ is not regular.

Exercise 12.20.13. Is $L = \{a^{2n^3} \mid n \geq 0\}$ regular? Prove your claim.

Example 12.20.3. Prove $L = \{a^p \,|\, p \text{ is prime } \}$ is not regular.

Solution. Let $n \geq 0$. Let $p \geq n$ be a prime; this is possible since there are infinitely many primes. Consider $x = a^p$. Note that $x \in L$ and $|x| = p \geq n$.

Let $u, w, v \in \Sigma$ such that $x = uvw$. Now we consider $uv^{k+1}w$; we will choose $k \geq 0$ in a minute so that the length of $uv^{k+1}w$ is not a prime. Note that

$$|uv^{k+1}w| = |uvw| + k|v|$$

Choose $k = |uvw|$ and we get

$$|uv^{k+1}w| = |uvw| + |uvw||v| = |uvw|(1 + |v|)$$

which is not a prime and so $uv^{1+|uvw|}w \notin L$.

By the Pumping Lemma for regular languages, L is not regular.

Exercise 12.20.14. An integer is composite if it is not prime. Prove that $L = \{a^n \mid n \text{ is composite}\}$ is not regular.

Exercise 12.20.15. Is $L = \{a^p b^{p+2} \mid p,p+2 \text{ are twin primes}\}$ regular? $(p, p+2$ are twin primes if p and $p+2$ are both primes.)

You can also combine regular operators with PL to prove that a language is not regular. Here, a regular operators means a set operator which is closed for regular languages, meaning to say that regular languages remains closed for such operators. You already know that the following closed:

1. If L, L' are regular, then $L \cup L'$ is also regular
2. If L, L' are regular, then $L \cap L'$ is also regular
3. If L, L' are regular, then LL' is also regular
4. If L are regular, then L^* is also regular
5. If L is regular, then $\overline{L}$ is also regular

Example 12.20.4. Let $\Sigma = \{a, b\}$. Is

$$L = \{x \in \Sigma^* \mid x \text{ has same number of } a\text{'s and } b\text{'s}\}$$

regular?

Solution. Assume L is regular.

$L(a^*b^*) = \{a^m b^n \mid m \geq 0, n \geq 0\}$ is regular since languages generates by regex's are also regular. Now note that

$$L \cap L(a^*b^*) = \{a^n b^n \mid n \geq 0\}$$

$L(a^*b^*)$ is regular since it is the language generated by a regex. Since intersection is a closed regular operator and we assumed that L is regular, we conclude that $L \cap L(a^*b^*)$ is also regular.

However we have already shown that $\{a^n b^n \mid n \geq 0\}$ is *not* regular.

This gives us a contradiction. Therefore our assumption that L is regular cannot hold. Hence L is not regular. □

Exercise 12.20.16. Let $\Sigma = \{a, b\}$. Is

$$L = \{w \mid \text{the number of } a\text{'s in } w \text{ is a prime}\}$$

regular? Prove your claim.

Exercise 12.20.17. Let $\Sigma = \{a, b\}$. Is

$$L = \{w \mid \text{there are more } a\text{'s than } b\text{'s in } w\}$$

regular? Prove your claim.

Exercise 12.20.18. Let $\Sigma = \{a, b\}$. Is

$$L = \{w \mid \text{there are 3 times more } a\text{'s than } b\text{'s in } w\}$$

regular? Prove your claim.

Exercise 12.20.19. Let $\Sigma = \{a, b\}$. Is $L = \{ww \mid w \in \Sigma^*\}$ regular? Prove your claim.

Exercise 12.20.20. Is the class of non-regular languages closed under union, intersection, complement, concatenation, Kleene star? For instance is it true that if L and L' are not regular, then $L \cup L'$ is also not regular.

12.21 Methods to Prove a Language is Regular or not Regular debug: showregular.tex

Here's a quick summary of methods to show a language L is regular:

- Construct a DFA or NFA and show that the language accepted is the same as L. Remember that an NFA is usually easier.
- Construct a regex and show that the language accepted is L
- Use closure rules

To show a language is not regular

- Use pumping lemma
- Assume it is regular, use closure rules to simplify it to something that you know is not regular
- Assume it is regular, use closure rules to simplify it, and use pumping lemma to show it is not regular

If you're not told if the language is regular or not, just remember that a regular language is accepted by a DFA with finitely many states. If you have to assume that the machine needs to remember arbitrarily huge amount of information, then it's very likely not regular. For instance the language $\{a^n b^n \mid n \geq 0\}$ is not regular. However note that sometimes it's because of your bad design that requires infinitely many states! You can't blame your bad design on the language!

Index