CDM

Quotients and Minimization

Klaus Sutner
Carnegie Mellon University
www.cs.cmu.edu/~sutner

Battleplan

- Minimal Automata
- Algebra of Languages
- Quotients
- Quotient Automaton

Minimal Automata

State Complexity

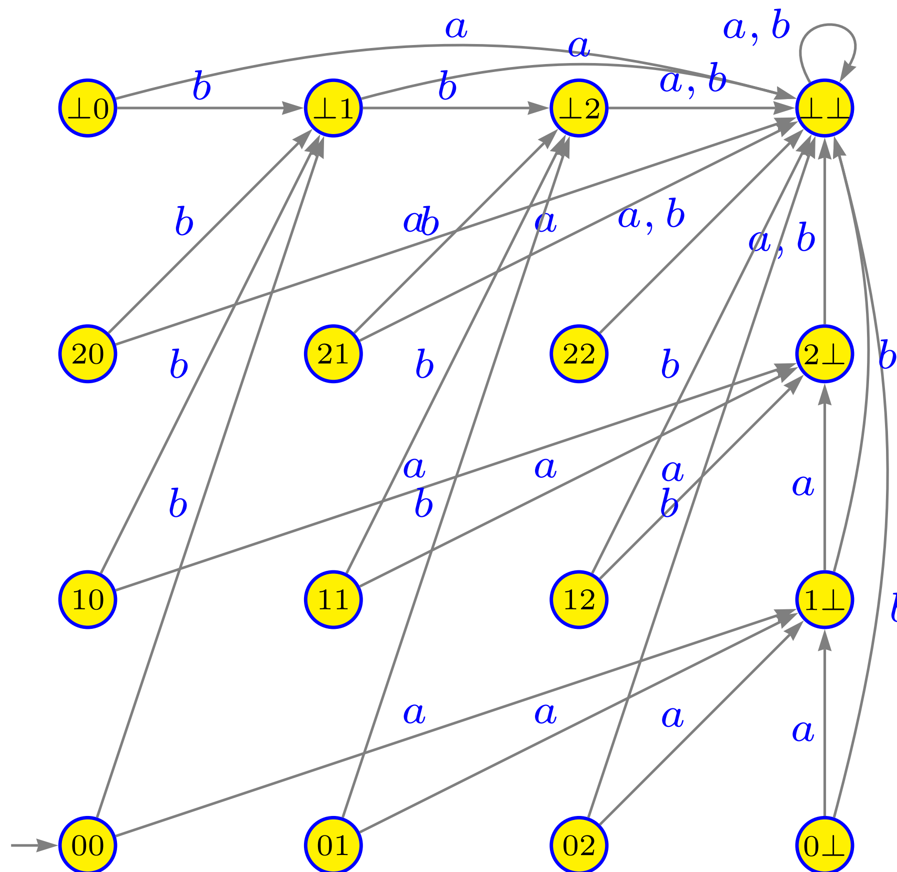
As we have seen, simple-minded application of the product machine construction leads to exponential blow-up in size. Even if the language in question is trivial (recall the example $\{aa, bb\}$ from last time), machines obtained by product automaton constructions may be exceedingly large.

We need to find ways to make the construction of DFAs (or more general finite state machines) more efficient. For example, it seems worth-while in many circumstances to try to keep the number of states in a DFA small.

Which naturally leads to the question: how many states are absolutely necessary for a given language?

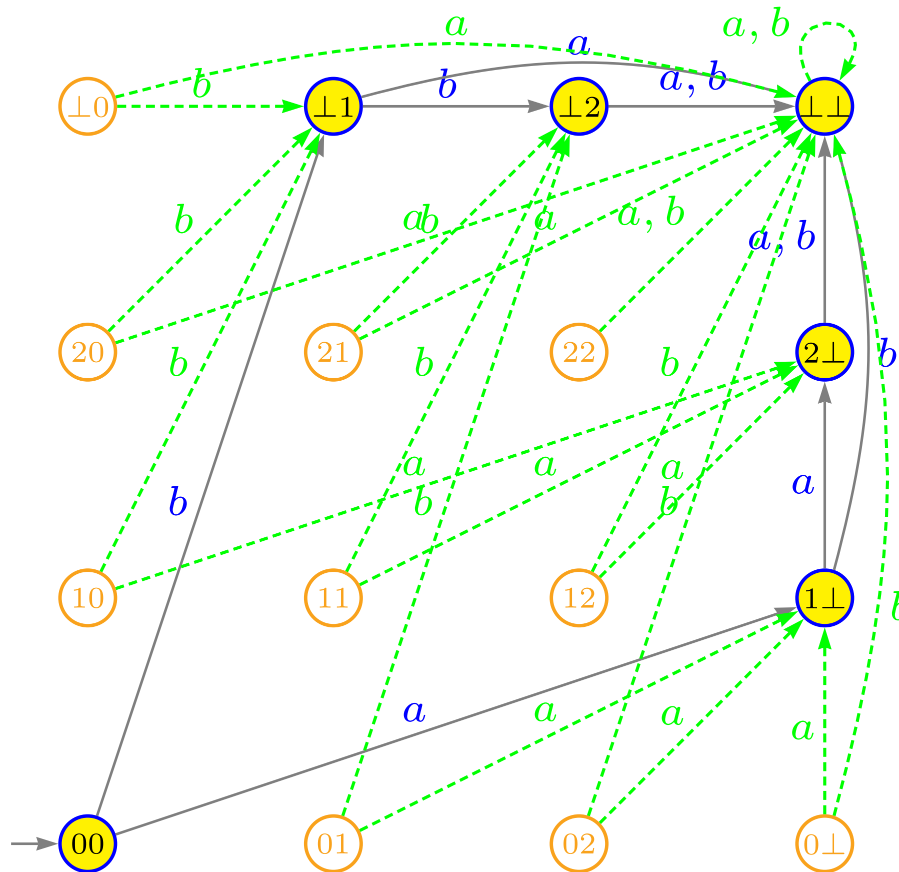
Full Product Automaton

Here is a preposterously large DFA for the language $\{aa, bb\}$.



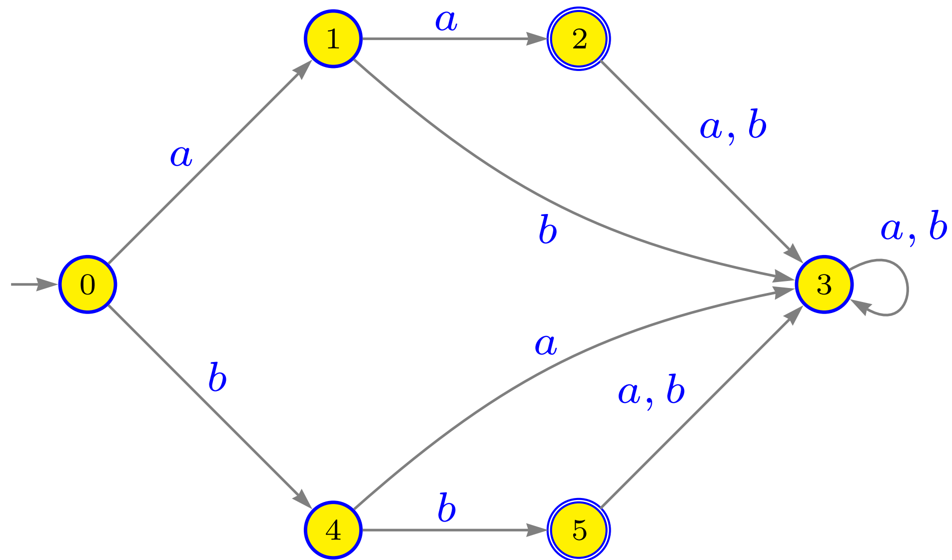
The Accessible Part

We can improve matters quite a bit by removing inaccessible states.



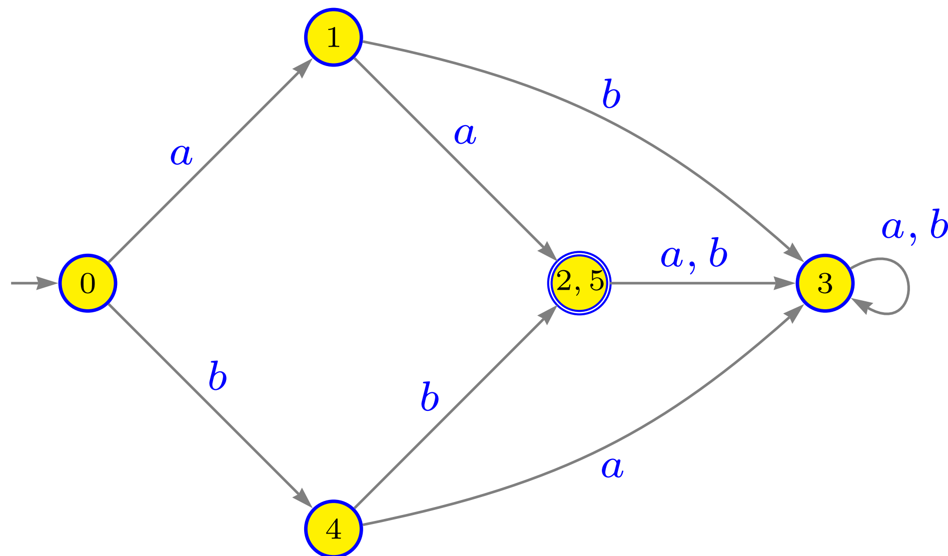
Still Too Large

The automaton we obtain by removing inaccessible states is essentially this:



A Minimal Solution

However, the state complexity of $\{aa, bb\}$ is only 5 (recall that state complexity is defined in terms of DFAs, so we have to include the sink in the count).



States 2 and 5 are merged into a single state (and the transitions rerouted accordingly).

Minimal Automata

Definition 1. *A DFA M is* minimal *if there is no DFA equivalent to M with fewer states than M.*

Thus the state complexity of M is the same as the state complexity of $\mathcal{L}(M)$. As already pointed out there are several potential problems with this definition:

- The existence of a minimal DFA is guaranteed by the fact that $\mathbb{N}$ is well-ordered, but there ought to be a more structural reason.
- There might be several minimal DFAs for the same language.
- Even if there is a unique minimal DFA, there might not be a good connection between other DFAs and the minimal one.

Really Minimal?

How do we know that 5 states are necessary for $\{aa, bb\}$?

- Need initial state q_0.
- Need state $\delta(q_0, a)$ and $\delta(q_0, a) \neq q_0$.
- Need state $\delta(q_0, b)$ and $\delta(q_0, b) \neq q_0, \delta(q_0, a)$.
- Need state $\delta(q_0, aa)$ and $\delta(q_0, aa) \neq q_0, \delta(q_0, a), \delta(q_0, b)$.
- Need state $\delta(q_0, aaa)$ and $\delta(q_0, aa) \neq q_0, \delta(q_0, a), \delta(q_0, b), \delta(q_0, aa)$.

If any of these states were equal the machine would accept the wrong language.

So in a sense all these states are inequivalent, indispensable.

There is no hope to build a machine with fewer than 5 states.

Behavioral Equivalence

There is an interesting idea hiding in this argument: some states must be distinct, so the machine cannot be too small.

To make this more precise we adopt the following definition.

Definition 2. *Let M be a DFA. The* behavior *of a state p is the acceptance language of M with initial state replaced by p. Two states are* (behaviorally) equivalent *if they have the same behavior.*

In symbols:

$$\begin{aligned} [\![p]\!] &= \mathcal{L}(\langle\, Q, \Sigma, \delta, p, F \,\rangle\,) \\ &= \{\, x \in \Sigma^\star \mid \delta(p, x) \in F \,\} \end{aligned}$$

So in a DFA the language accepted by the machine is simply $[\![q_0]\!]$.

Daemons and Observations

Whenever $[\![p]\!] = [\![q]\!]$ we can identify p and q: any input that leads to acceptance from p also leads to acceptance from q (and vice versa).

Think about a little daemon sitting in the machine.

Whenever state p is reached, it may magically flip the state to q, and vice versa.

The outside observer would never notice: the daemon infested machine would still accept precisely the same set of strings as the old one.

The Main Idea

So suppose p and p' have the same behavior. We can then collapse p and p' into just one state: to do this we have to redirect all the affected transitions to and from p and q.

This is easy for the incoming transitions.

But there is a little problem for the outgoing transitions: one has to merge all equivalent states, not just a few.

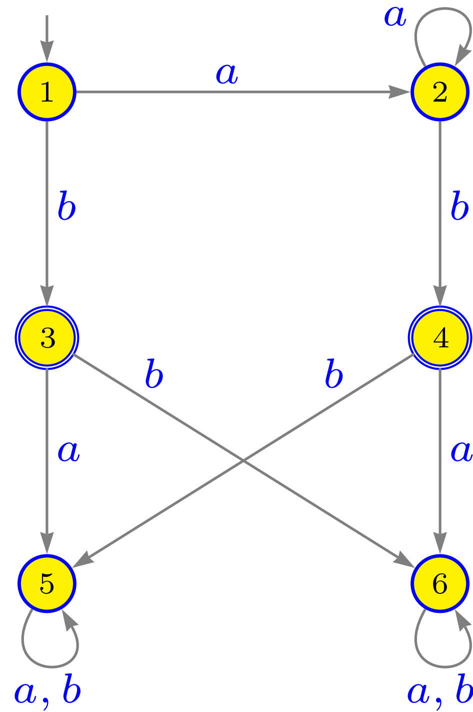
Otherwise the merged states will have nondeterministic transitions emanating from them – and we do not want to deal with nondeterministic machines here.

An Example

Language:

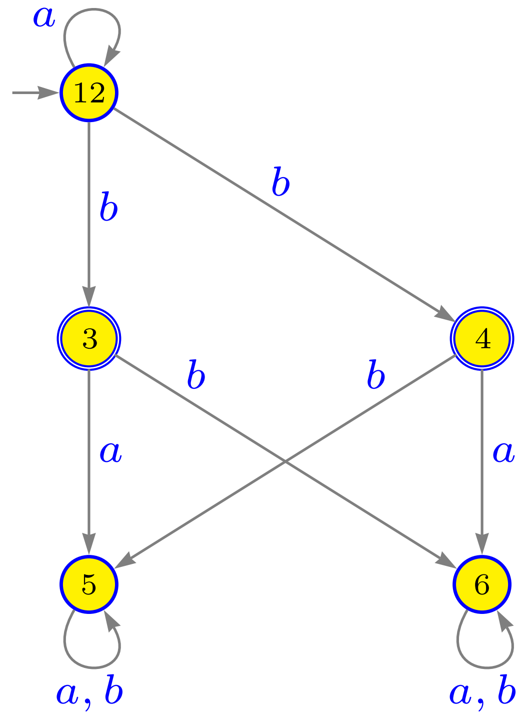
a^*b.

$[[1]] = [[2]]$
$[[3]] = [[4]]$
$[[5]] = [[6]]$



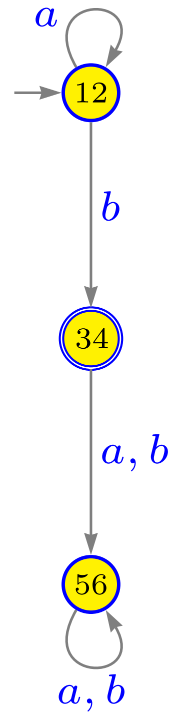
Partial Merge

Merging only 1 and 2 produces a nondeterministic machine



Complete Merge

But a complete merge produces a DFA.



Reduced Machines

In the last machine, all states are inequivalent:

$$[\![12]\!] = a^* b$$
$$[\![34]\!] = \varepsilon$$
$$[\![56]\!] = \emptyset$$

So no further state merging is possible.

Definition 3. *A DFA is* reduced *if all its states are pairwise inequivalent.*

Characterization

In a while we will establish the following theorem.

Theorem 1. *A DFA is minimal if, and only if, it is accessible and reduced.*

Accessibility is computationally cheap, we can always strip off inaccessible states in linear time.

It is not so clear how we can implement the merging procedure (whose goal it is to ensure a reduced automaton).

First a slightly more algebraic perspective.

The Algebra of Languages

Merging and Algebra

The merging approach is really algebraic in nature. Given some complicated structure $\mathcal{S}$, try to simplify matters as follows:

- Find an equivalence relation E on $\mathcal{S}$,
- that is compatible with the operations on $\mathcal{S}$, and then
- replace $\mathcal{S}$ by the quotient structure $\mathcal{S}/E$.

In general one would like to make the quotient structure as small as possible, so the equivalence relation should be as coarse as possible. The important point here is that not just any equivalence will do, rather we need a *congruence*. E.g., if $\mathcal{S}$ has a binary operation $*$ then we need

$$a \; E \; b, c \; E \; d \quad \text{implies} \quad a * c \; E \; b * d.$$

It might be a good idea to take a closer look at the algebra of languages, whatever that may turn out to be.

Algebra of Languages

Given an alphabet Σ we consider the carrier set of all languages over Σ:

$$\mathcal{L}(\Sigma) = \mathfrak{P}(\Sigma^\star) = \{ \, L \mid L \subseteq \Sigma^\star \, \}$$

There are the obvious Boolean operations union, intersection and complement that can be applied to languages over Σ. Hence we have a Boolean algebra

$$\langle \, \mathcal{L}(\Sigma), \cup, \cap, ^{-} \rangle$$

That's OK, but not terribly interesting: at no point are we using the fact that the sets in question are sets of words, rather than arbitrary objects.

So the question is: what are interesting operations on $\mathcal{L}(\Sigma)$ that exploit this fact?

Concatenation

Recall that we can "multiply" two words by concatenating them.

We can lift this operations to language by applying concatenation pointwise. Given two languages $L_1, L_2 \subseteq \Sigma^\star$ we concatenate them by concatenating all possible combinations of words:

$$L_1 \cdot L_2 = \{\, xy \mid x \in L_1, y \in L_2 \,\}.$$

Note that this operation fails to commute.

Also, $L \cdot \emptyset = \emptyset \cdot L = \emptyset$ and $L \cdot \{\varepsilon\} = \{\varepsilon\} \cdot L = L$.

Example 1.

$$\{a, b\}\,\{a, b\} = \{aa, ab, ba, bb\}$$

Kleene Star

Boolean operations and concatenation when applied to finite languages produce only finite and co-finite languages and are thus insufficient to generate interesting languages. We need at least one operation that generates an infinite language from a finite one. Here is one such operation that turns out to be immensely useful.

Definition 4. *Let L be a language. The powers of L are the languages obtained by repeated concatenation:*

$$L^0 = \{\varepsilon\}$$

$$L^{k+1} = L^k \cdot L$$

The Kleene star of L is the language

$$L^\star = L^0 \cup L^1 \cup L^2 \ldots \cup L^n \cup \ldots$$

Kleene star corresponds roughly to a while-loop or iteration.

Star Examples

Example 2.

$\{a, b\}^\star$: *all words over* $\{a, b\}$

Example 3.

$\{a, b\}^\star \{a\} \{a, b\}^\star \{a\} \{a, b\}^\star$: *all words over* $\{a, b\}$ *containing at least two* a*'s*

Example 4.

$\{\varepsilon, a, aa\} \{b, ba, baa\}^\star$: *all words over* $\{a, b\}$ *not containing a subword* aaa

Example 5.

$\{0, 1\} \{0, 1\}^\star$: *all numbers in binary, with leading 0's*

$\{1\} \{0, 1\}^\star \cup \{0\}$: *all numbers in binary, no leading 0's*

Building Regular Languages

The choice of concatenation and Kleene star may seem rather arbitrary. It is is justified by the following theorem (which we won't prove here).

Theorem 2. *Every regular language can be obtained from singletons* $\{a\}$ *for* $a \in \Sigma$, *and* $\emptyset$, *by finitely many applications of the operations union, concatenation and Kleene star.*

Given a DFA for the language, the decomposition can be generated algorithmically.

In other words, the collection $\mathrm{Reg}(\Sigma)$ of all regular languages over alphabet Σ is a sub-algebra of

$$\langle\, \mathcal{L}(\Sigma), \cup, \cdot, {}^\star, 0, 1 \,\rangle$$

and is generated by singletons $\{a\}$.

How about Intersection and Complement?

Note that the theorem makes a rather surprising claim: it suffices to consider operations union, concatenation and Kleene star when one tries to construct regular languages from atomic pieces (in this case singletons $\{a\}$ and the empty set).

But regular languages are closed under interesection and complement. It is by no means clear how

$$L \cap K \qquad \text{or} \qquad \Sigma^\star - L$$

can be so generated, even if we already know how to handle K and L.

Who Cares?

The first part of this result is actually very important in applications: it provides a simple notation system for regular languages.

If we write a for the singleton language $\{a\}$ then all regular language can be written down using just $+$ for union, $\cdot$ for concatenation and * for Kleene star (we won't quibble about the empty set here).

These expressions are usually referred to as *regular expressions*.

They are crucial for a lot of text searching and manipulation tools such as `grep`, `awk`, `sed` and `pearl`: it is easy to type in regular expressions but would be entirely hopeless to have to input the corresponding finite state machines.

The second part is more of theoretical interest: the algorithm usually generates a really bad decomposition (much too big, but simplification is hard).

All Behaviors

Conspicuously absent from our algebra so far is any operation resembling division. If we think of division as the inverse of multiplication (i.e., concatenation) the natural answer is the following.

Definition 5. *Let $L \subseteq \Sigma^\star$ be a regular language and $x \in \Sigma^\star$. The* left quotient of L by x *is*

$$x^{-1}L = \{\, y \in \Sigma^\star \mid xy \in L \,\}.$$

So we are simply removing a prefix x from all words in the language that start with this prefix. If there is no such prefix we get an empty quotient.

This is the reason why it is a bit more elegant to talk about quotients in the context of languages rather than words: for words x and y the quotient $x^{-1}y$ would be undefined whenever x fails to be a prefix of y.

Algebra of Quotients

To simplify notation, let

$$\Delta(L) = \begin{cases} \{\varepsilon\} & \text{if } \varepsilon \in L, \\ \emptyset & \text{otherwise.} \end{cases}$$

Lemma 1. *Let $a \in \Sigma$, $x, y \in \Sigma^\star$ and $L, K \subseteq \Sigma^\star$. Then the following hold:*

- $(xy)^{-1}L = y^{-1}x^{-1}L$,
- $x^{-1}(L \cup K) = x^{-1}L \cup x^{-1}K$,
- $x^{-1}(L \cap K) = x^{-1}L \cap x^{-1}K$,
- $x^{-1}(\Sigma^\star - L) = \Sigma^\star - x^{-1}L$,
- $a^{-1}(LK) = (a^{-1}L)K \cup \Delta(L)a^{-1}K$,
- $a^{-1}L^* = (a^{-1}L)L^*$.

Comments

Note that $(xy)^{-1}L = y^{-1}x^{-1}L$ and NOT $x^{-1}y^{-1}L$. The problem is that algebraically left quotients are a right action. Oh well.

Quotients coexist peacefully with Boolean operations, we can just push the quotients inside.

But for concatenation and Kleene star things are a bit more involved; the lemma makes no claims about the general case where we divide by a word rather than a single letter.

Exercise 1. *Prove the last lemma.*

Exercise 2. *Generalize the rules for concatenation and Kleene star to words.*

All Behaviors

The reason we are interested in quotients is that they are closely related to behaviors of states in a DFA. More precisely, consider the following question:

What are the possible behaviors of states in an arbitrary DFA for a fixed regular language?

One might think that the behaviors differ from machine to machine, but they turn out to be the same, always.

To see why, first ignore the machines and consider the acceptance language directly. Note that the language is the behavior of the initial state and thus the same in any DFA.

Quotients Example 1

Using the lemma, whe can compute the quotients of a^*b.

$$a^{-1}\, a^*b = a^*b$$
$$b^{-1}\, a^*b = \varepsilon$$
$$a^{-1}\, \varepsilon = \emptyset$$
$$b^{-1}\, \varepsilon = \emptyset$$
$$a^{-1}\, \emptyset = \emptyset$$
$$b^{-1}\, \emptyset = \emptyset$$

Note that these equations between quotients really determine the transitions in the example machine for state-merging from above.

Quotients Example 1, Contd.

$a^{-1}\, a^*b = a^*b$	$\{1,2\} \xrightarrow{a} \{1,2\}$
$b^{-1}\, a^*b = \varepsilon$	$\{1,2\} \xrightarrow{b} \{3,4\}$
$a^{-1}\, \varepsilon = \emptyset$	$\{3,4\} \xrightarrow{a} \{5,6\}$
$b^{-1}\, \varepsilon = \emptyset$	$\{3,4\} \xrightarrow{b} \{5,6\}$
$a^{-1}\, \emptyset = \emptyset$	$\{5,6\} \xrightarrow{a} \{5,6\}$
$b^{-1}\, \emptyset = \emptyset$	$\{5,6\} \xrightarrow{b} \{5,6\}$

Quotients Example 2

Let L be the finite language $\{a, aab, bbb\}$.

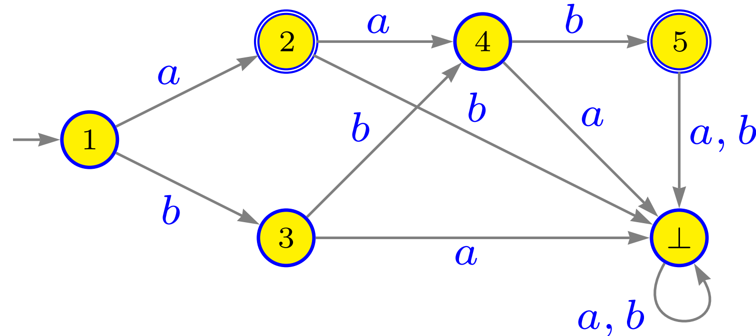
There are exactly six quotients of L:

x	$x^{-1} L$
ε	$\{a, aab, bbb\}$
a	$\{\epsilon, ab\}$
b	$\{bb\}$
bb	$\{b\}$
aab	$\{\epsilon\}$
ab	$\emptyset$

Note that the x is not uniquely determined, for example $(abz)^{-1} L = (baz)^{-1} L = \emptyset$ for any z.

Quotients Example 2.5

Moreover, there is a "natural" DFA for L that has six states.



Could this be coincidence? Nah . . .

For example, $\delta(1, a) = 2$ and $[\![2]\!] = \{\varepsilon, ab\}$.

Corresponding to $a^{-1} L = \{\varepsilon, ab\}$.

Quotients Example 3

A larger example, $L = L_1 = a^*b^* \cup bab$.

$a^{-1}L_1$	a^*b^*	L_2
$b^{-1}L_1$	$b^* \cup ab$	L_3
$a^{-1}L_2$	L_2	
$b^{-1}L_2$	b^*	L_4
$a^{-1}L_3$	b	L_5
$b^{-1}L_3$	L_4	
$a^{-1}L_4$	$\emptyset$	L_6
$b^{-1}L_4$	L_4	
$a^{-1}L_5$	L_6	
$b^{-1}L_5$	ε	L_7
$a^{-1}L_{6/7}$	L_6	
$b^{-1}L_{6/7}$	L_6	

Exercise 3. *Verify this table.*

Quotients Example 4

An even larger example, $L = L_1 = a^*aa^* \cup b^*ab^*$.

$a^{-1}L_1$	$a^*ba^* + b^*$	L_2
$b^{-1}L_1$	$b^*ab^* + a^*$	L_3
$a^{-1}L_2$	a^*ba^*	L_4
$b^{-1}L_2$	$a^* + b^*$	L_5
$a^{-1}L_3$	$a^* + b^*$	
$b^{-1}L_3$	b^*ab^*	L_6
$a^{-1}L_4$	a^*ba^*	
$b^{-1}L_4$	a^*	L_7
$a^{-1}L_5$	a^*	
$b^{-1}L_5$	b^*	L_8
$a^{-1}L_6$	b^*	
$b^{-1}L_6$	b^*ab^*	
$a^{-1}L_7$	b^*	
$b^{-1}L_7$	$\emptyset$	L_9
$a^{-1}L_8$	$\emptyset$	
$b^{-1}L_8$	b^*	
$a^{-1}L_9$	$\emptyset$	
$b^{-1}L_9$	$\emptyset$	

Quotients Example 5

$$L = \{\, a^i b^i \mid i \geq 0 \,\} = \{\varepsilon, ab, aabb, aaabbb, \ldots\}$$

Here things become a bit complicated: we obtain infinitely many quotients.

$$(a^k)^{-1}L = \{\, a^i b^{i+k} \mid i \geq 0 \,\}$$

$$(a^k b^l)^{-1}L = \left\{ b^{k-l} \right\} \qquad 1 \leq l \leq k$$

$$(a^k b^l)^{-1}L = \emptyset \qquad l > k$$

This is no coincidence: the language L is not regular.

Computing Quotients

These calculations can be described less informally by the following algorithm.

```
i = j = 1;
L[1] = L;
while( i <= j ) {
    foreach a in Sigma do
        K = left_quotient( a, L[i] );
        if( K is new ) L[++j] = K;
    i++;
}
```

Note that this is again a closure operation: we generate the smallest collection $\mathcal{L}$ of languages that contains L and is closed under quotients:

- $L \in \mathcal{L}$,
- $K \in \mathcal{L}$ implies $a^{-1}K \in \mathcal{L}$.

Is this an Algorithm?

Can we really compute the number of quotients of a given regular language? For example, could we write a primitive recursive function to do this? Or is this just un-implementable pseudo-code?

The logical control structure is easy: just a while-loop. But we need to represent the basic objects and operations:

- represent languages by some datastructure,
- implement the operations $a^{-1}K$,
- implement the equality test $K = K'$.

Are all these problems surmountable?

Real Implementation

Naturally, we represent languages by machines (we don't have much choice at this point).

- Since we are only dealing with regular languages we can use DFAs as representation.
- Quotients are then easy to implement: just move the initial state.
- The equality test comes down to checking Equivalence of DFAs, we already know how to do this.

Note how the choice of datastructure really settles the whole issue: if a regular language is represented by a DFA we know how to compute quotients, and we know how to check equality.

The question arises: how efficient is this approach?

Running Time Analysis

Suppose the DFA representing L has n states. For simplicity we ignore the size of the alphabet.

Clearly, there will be at most n quotients to compute.

For each one, we have to test equality against $O(n)$ others.

Doing this the obvious way requires $O(n^2)$ steps for each equality check, so each new quotient requires $O(n^3)$ steps.

The whole algorithm is then an unimpressive $O(n^4)$.

Can we speed this up?

Exercise 4. *Explain the running time of this method in detail. Take into account the size of the alphabet.*

A Special Case

Before we discuss better algorithms, note that for finite languages

$$L = \{w_1, w_2, \ldots, w_n\}$$

we don't need a DFA to represent the language: we can use any container datastructure to represent finite sets of strings. A particularly good choice often is a trie.

The operations of taking quotients and checking equality are straightforward to implement in any reasonable programming language. Note that except for the sink the diagram of the automaton must be a DAG.

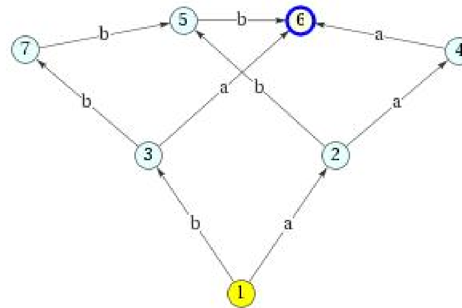
There are better algorithms for maintaining machines for finite languages (hugely important in computational linguistics), but let's not get involved with efficiency at the moment.

Example

As a small example consider $L = \{ab, aaa, abb, bbbb\}$. Then the 7 quotients other than L are

2	$\{aa, bb\}$	6	$\{\varepsilon\}$
3	$\{a, bbb\}$	7	$\{bb\}$
4	$\{a\}$	8	$\emptyset$
5	$\{b\}$		

corresponding to the DFA (the sink is omitted):



The Quotient Machine

The Decomposition Lemma

Here is simple observation about the relationship between languages (not just regular) and their quotients.

Lemma 2. *Let $L \subseteq \Sigma^\star$ be any language. Then*

$$L = \Delta(L) \cup \bigcup_{a \in \Sigma} a \cdot (a^{-1} L)$$

Proof.

Well, a word $x \in L$ is either empty, or it looks like $x = au$ for some $a \in \Sigma, u \in \Sigma^\star$.

$\Box$

Exercise 5. *Give a fastidious definition of words as functions $w : [n] \to \Sigma$, $n \in \mathbb{N}$, and use this definition to give a formal proof of the Decomposition Lemma.*

The Crucial Insight

The Decomposition Lemma is just about trivial. For $L \subseteq \{a, b\}^\star$ we get

$$L = a \cdot (a^{-1} L) \cup b \cdot (b^{-1} L) \cup \Delta(L)$$

But, from the right point of view this little observation is quite helpful:

- Think of the quotients as states.
- Then the Decomposition Lemma describes the transitions on these states:

$$L \xrightarrow{a} a^{-1} L$$

- The Δ term determines whether a state is final.

The Quotient Machine

In other words, we can build a DFA out of the quotients. To see how, suppose $Q = \{L_1, L_2, \ldots, L_n\}$ is a list of all the quotients of some language L.

Construct a DFA

$$M_L = \langle\, Q, \Sigma, \delta, q_0, F \,\rangle$$

as follows:

$$\begin{aligned} \delta(K, a) &= a^{-1} K \\ q_0 &= L \\ F &= \{\, K \in Q \mid \varepsilon \in K \,\} \end{aligned}$$

Quotient Machine, contd.

This is perfectly in keeping with our definitions: the state set has to be finite, but no one said the states couldn't be complicated.

At any rate, in M_L we have

$$\delta(q_0, x) = \delta(L, x) = x^{-1} L.$$

But then

$$x \in L \iff \varepsilon \in x^{-1} L \iff \delta(q_0, x) \in F$$

so that M_L duly accepts L.

How About Other DFAs

It is clear by now that there is a very close link between behaviors and quotients of the acceptance language.

More precisely, it follows from the Decomposition Lemma that in any DFA whatsoever

$$[\![\delta(p,a)]\!] = a^{-1}\,[\![p]\!]$$

Note that it is critical here that DFAs are deterministic: there is only one path in the diagram starting at the initial state labeled by any particular word x.

The theory of nondeterministic machines is much more complicated.

Quotients and Behaviors

Lemma 3. *Let M be an arbitrary DFA, p a state and $x \in \Sigma^\star$. Then*

$$[\![\delta(p,x)]\!] = x^{-1}\,[\![p]\!]$$

Proof.

Straightforward induction on x. Use

$$(xa)^{-1}L = a^{-1}(x^{-1}L)$$

□

Corollary 1. *Suppose M is a DFA accepting L. Then for any word x:*

$$[\![\delta(q_0,x)]\!] = x^{-1}L$$

So What?

Hence all accessible states have as behavior one of the quotients of L.

Conversely, all quotients appear as the behavior of at least one state in any DFA for L.

This may not sound too impressive, but it has some very interesting consequences.

Corollary 2. *Every regular language has only finitely many left quotients.*

Corollary 3. *Every DFA accepting a regular language has at least as many states as the number of quotients of the language.*

Corollary 4. *The quotient machine for a regular language has the lowest possible state complexity.*

But Is It Computationally Relevant?

It follows that that the quotient machine is a minimal DFA.

Unfortunately, computing a list of all quotients is not all that easy, so the quotient machine M_L may not seem like much of an accomplishment, at least from the computational point of view.

However, the quotient machine is very helpful in constructing perfectly efficient minimization algorithms.

As we will see later, quotients are often also useful in describing and analyzing finite state machines in general.

More on the Quotient Automaton

So the quotient automaton M_L is minimal: it has the same state complexity as its acceptance language L.

And we know abstractly how to compute the quotients of language and thus how to construct the quotient automaton.

Several issues remain:

- What exactly is the relationship between arbitrary DFAs for L and the quotient automaton for L?
- How do we turn the pseudo-algorithm from above into a real algorithm?

The Minimal Automaton

Theorem 3. *A DFA for a regular language is minimal with respect to the number of states if, and only if, it is accessible and reduced.*

Moreover, there is only one such minimal DFA (up to isomorphism): the quotient automaton of the language.

Proof.

Let L be the regular language in question and suppose that L has n quotients.

Let M be any accessible and reduced DFA for L.

Every quotient of L must appear exactly once as the behavior of a state in M, hence $|M| = n$.

By the corollary every DFA for L has at least n states, so M is minimal.

Proof, contd.

For the opposite direction, clearly any minimal automaton M for L must be accessible.

From the corollary, $|M| \geq n$ and we know how to construct a DFA with exactly n states.

But M is minimal, so $|M| = n$.

Again every quotient of L must appear exactly once as the behavior of a state: thus M is reduced.

Uniqueness

It remains to show that all DFAs for L of size n are essentially the same as the quotient machine M_L – we can rename the states, but other than that the machine is fixed.

To see this note we can define a bijection

$$\begin{aligned} f : Q &\to Q_L \\ p &\mapsto [\![p]\!] \end{aligned}$$

from the states of M to the states of M_L (all quotients of L).

This is a bijection since M has size n and we know that all quotients must appear as the behavior of at least one state.

Compatibility

Moreover, this bijection is compatible with the transitions in the machines in the sense that $f(\delta(p,a)) = \delta(f(p),a)$. As a diagram:

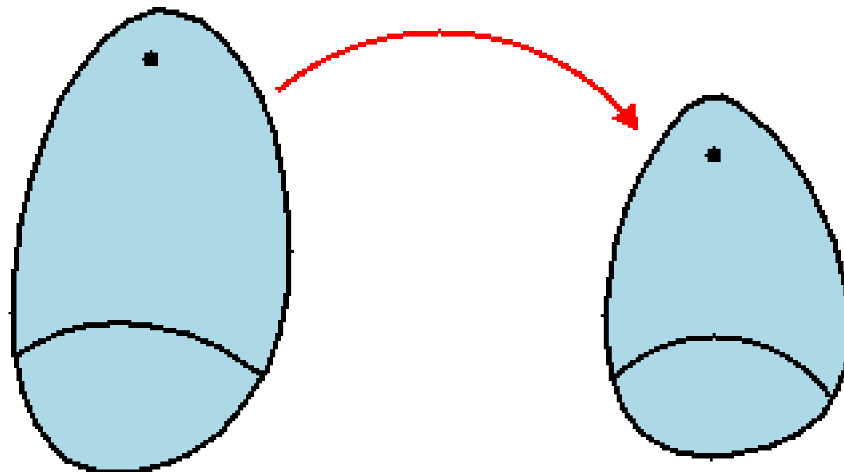
$$\begin{array}{ccc} p & \xrightarrow{a} & \delta(p,a) \\ \Big\downarrow f & & \Big\downarrow f \\ f(p) & \xrightarrow{a} & \delta(f(p),a) \end{array}$$

Lastly, f maps initial to initial, and final to final states.

Hence, the states in M are just "renamed" quotients: the machines M and M_L are isomorphic. □

Machine Homomorphisms

The isomorphism from above leads to a more general question: is there a good notion of a structure preserving map between two finite state machines? For simplicity, let's only consider DFAs.



Preserving Computations

It is clear that for a map f from machine M_1 to machine M_2 to be a homomorphism it must preserve transitions:

$$p \xrightarrow{a} q \qquad \text{implies} \qquad f(p) \xrightarrow{a} f(q)$$

Moreover, we require $f(q_{10}) = q_{20}$ and $f(F_1) \subseteq F_2$.

It follows immediately that $\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$.

However, we may still have $\mathcal{L}(M_1) \neq \mathcal{L}(M_2)$, so if we are interested in equivalent machines we need to strengthen the conditions a bit:

$$f^{-1}(F_2) = F_1$$

Homomorphisms that have this stronger property and are also surjective are often called *covers* or *covering maps*.

Covers

Thus, a covering map can identify some states in the first machine while preserving the language.

Needless to say, the classical example of a cover is the behavioral map:

$$f : Q \to Q_L$$
$$f(p) = [\![p]\!]$$

Hence we have the following lemma which shows that an arbitrary DFA for a given regular language is always an “inflated” version of the minimal DFA. There always is a close connection between an arbitrary DFA and the minimal automaton.

Lemma 4. *Let L be a regular language and M an arbitrary accessible DFA for L. Then there is compatible map from M onto M_L.*

Example

There is a natural DFA M for all words $x \in \{a,b\}^\star$ such that $x_{-3} = a$. The states in M are words over $\{a,b\}$ of length at most 3 and the transitions are of the form

$$\delta(w,s) = \begin{cases} ws & \text{if } |w| < 3, \\ w_2 w_3 s & \text{otherwise.} \end{cases}$$

The initial state is ε and the final states are $\{aaa, aab, aba, abb\}$. The covering map to the quotient automaton has the form

$$\begin{array}{lllllll} aaa & \mapsto & aaa & \quad & aa, baa & \mapsto & baa \\ aab & \mapsto & aab & & ab, bab & \mapsto & bab \\ aba & \mapsto & aba & & a, ba, bba & \mapsto & bba \\ abb & \mapsto & abb & & \varepsilon, b, bb, bbb & \mapsto & bbb \end{array}$$

Note that the transition diagram of the minimal automaton is a binary de Bruijn graph (of order 3).

Summary

- Inaccessible states can be removed from a DFA without changing the accepted language.
- Behavioral equivalence can be used to identify states in a DFA that have the same function (state merging).
- Combining both operations we obtain the minimal DFA for a given regular language.
- The minimal DFA can be characterized abstractly in terms of the quotient automaton.
- Sometimes quotients are useful to construct the minimal DFA directly.