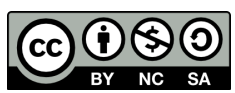
CDM

Minimization of Finite State Machines

Klaus Sutner

Carnegie Mellon University

Fall 2024



Recall the definition of the state complexity $\mathrm{st}(L)$ of a recognizable language L: the minimal number of states of any DFA accepting the language.

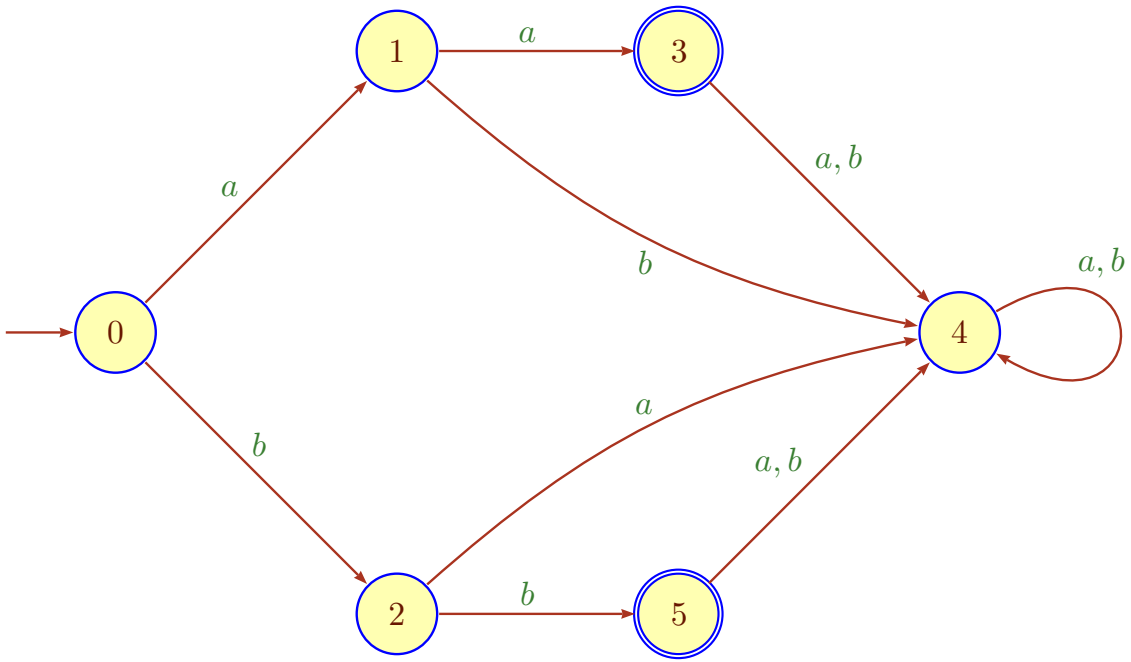
Our next goal is to show how to compute the state complexity of a language: we will construct a corresponding DFA, starting from an arbitrary machine for the language.

As it turns out, the automaton is unique, up to renaming of states. Thus, we have a normal form for any recognizable language. This is fairly rare, usually there are many canonical descriptions of an object.

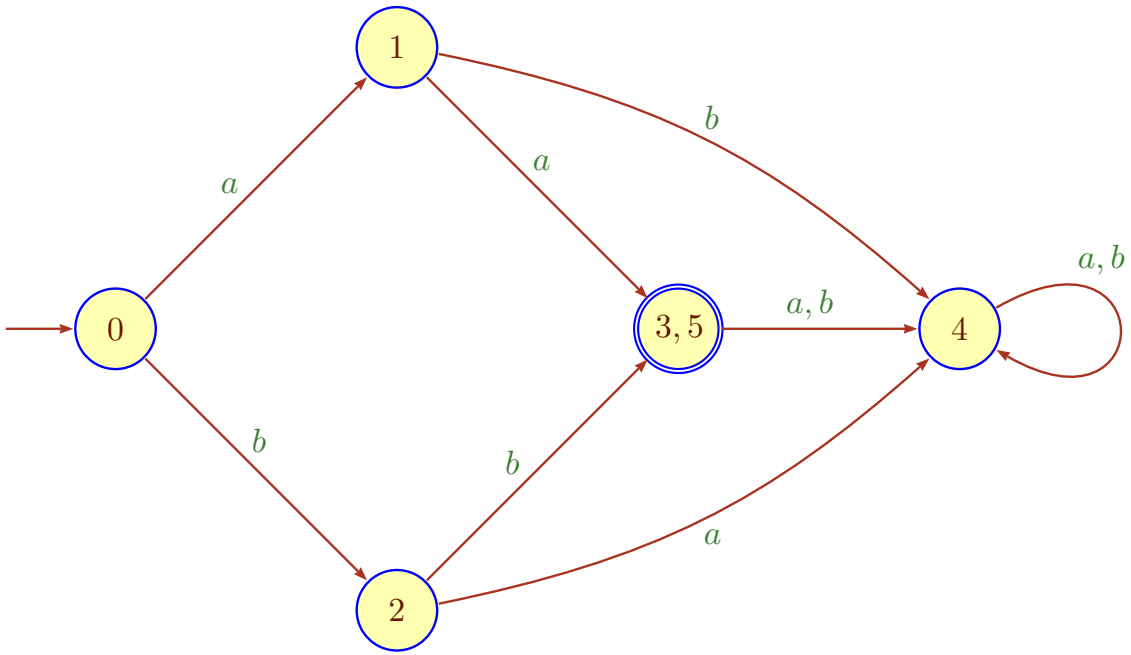
Humans are fairly good at constructing small DFAs that are already minimal–one naturally tends to avoid "useless" states. Unfortunately, this is the only good news.

- Humans fail spectacularly when the machines get large, even a few dozen states are tricky, thousands are not manageable.
- One of the most interesting aspects of finite state machines is that they can be generated and manipulated algorithmically. These algorithms typically produce nondeterministic machines, and determinization usually fails to produce minimal DFA.

Here is a DFA obtained by building the accessible part of the standard product machine for the language $\{aa, bb\}$:



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

Definition

A DFA $\mathcal{A}$ is minimal if the state complexity of $\mathcal{A}$ is the same as the state complexity of $\mathcal{L}(\mathcal{A})$.

In other words, there is no DFA equivalent to $\mathcal{A}$ with fewer states than $\mathcal{A}$. 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.

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

- Need state $q_0 = \delta(q_0, \varepsilon)$.
- Need state $q_1 = \delta(q_0, a)$ and $q_1 \neq q_0$.
- Need state $q_2 = \delta(q_0, b)$ and $q_2 \neq q_0, q_1$.
- Need state $q_3 = \delta(q_0, aa)$ and $q_3 \neq q_0, q_1, q_2$.
- Need state $q_4 = \delta(q_0, aaa)$ and $q_4 \neq q_0, q_1, q_2, q_3$.

If any of these states were equal the machine would accept the wrong language. So we need at least 5, but 5 also suffice.

Question:
What does $\{\varepsilon, a, b, aa, aaa\}$ have to do with $\{aa, bb\}$?

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

Let $\mathcal{A}$ be a DFA. The behavior of a state p is the acceptance language of $\mathcal{A}$ with initial state replaced by p. Two states are (behaviorally) equivalent if they have the same behavior.

In symbols:

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

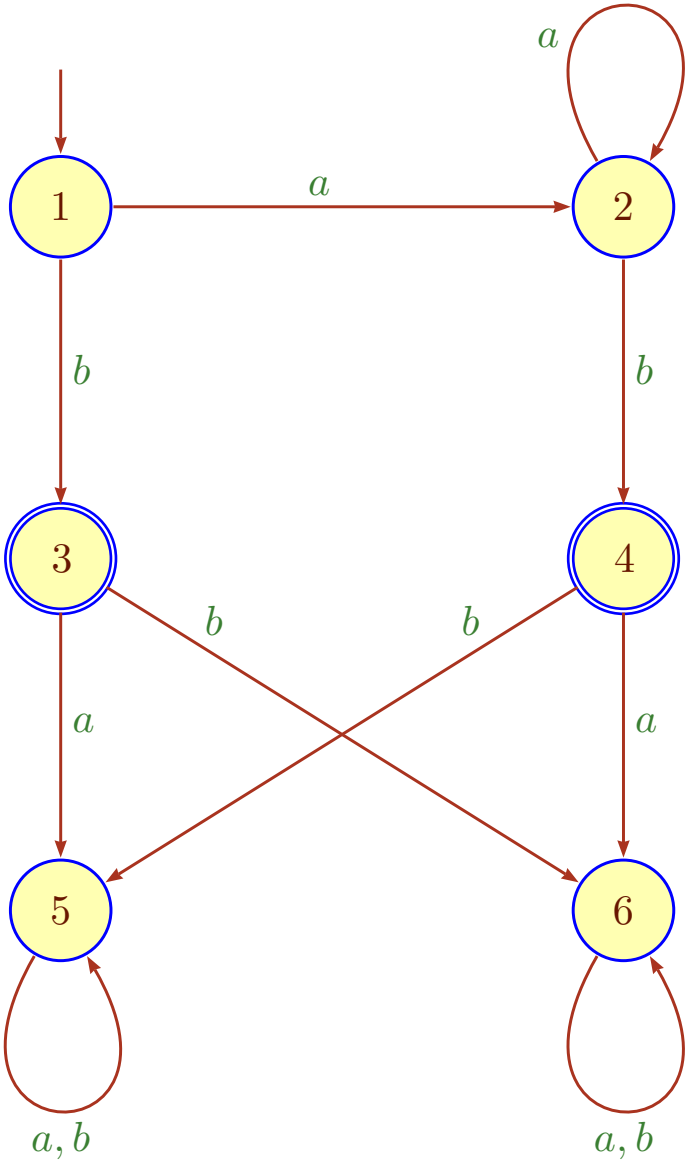
So in a DFA the language accepted by the machine is simply $[\![q_0]\!]$.

- 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.

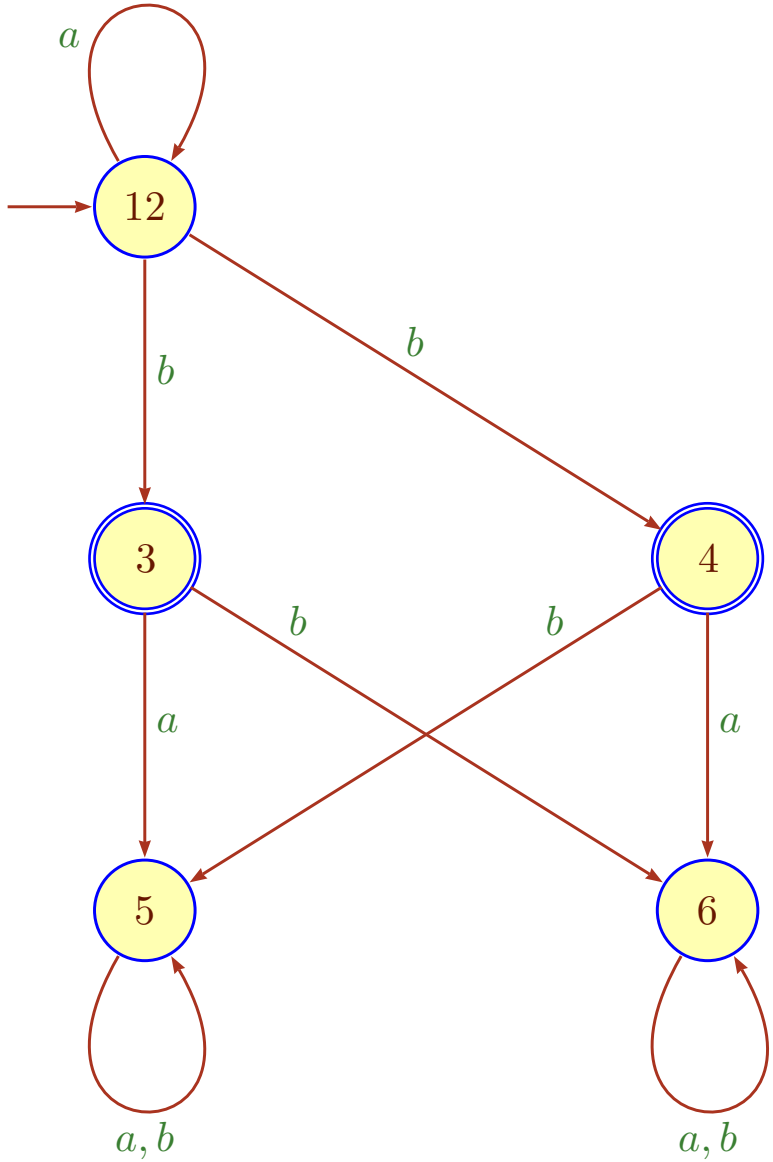
An Example

Language: a^*b.

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

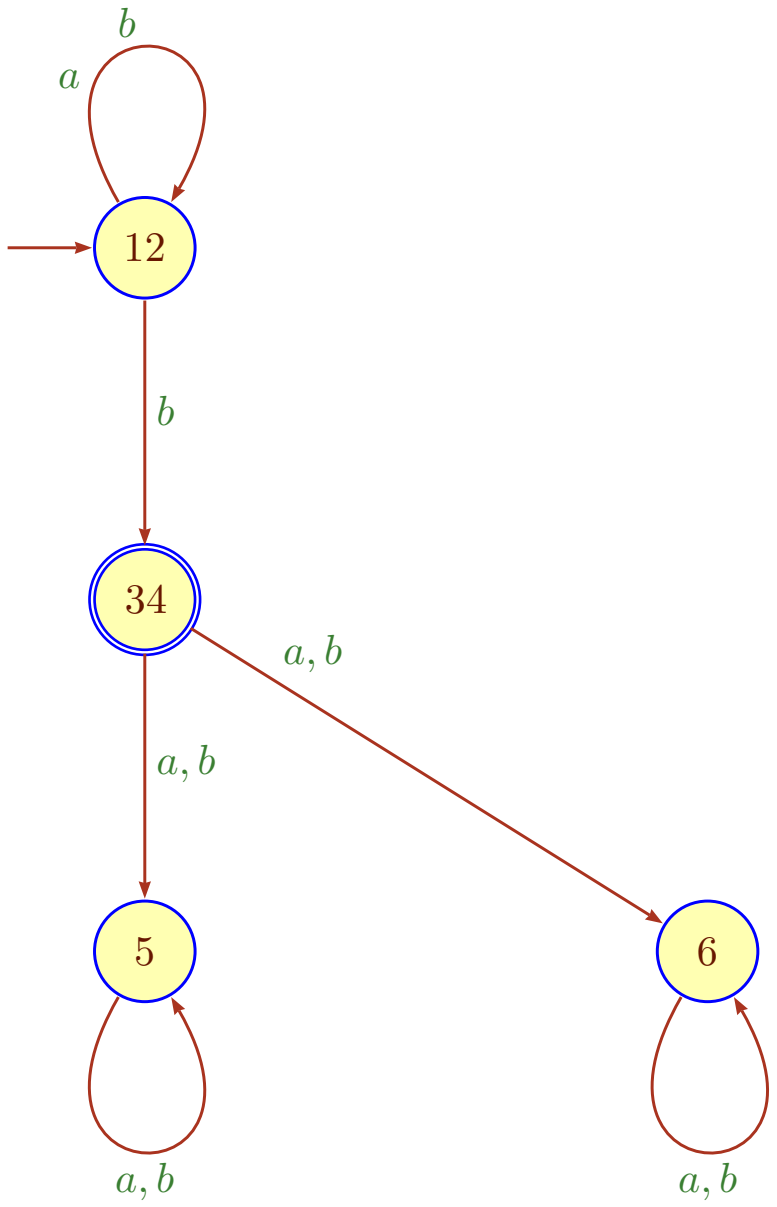


Merging only states 1 and 2 produces a nondeterministic machine.

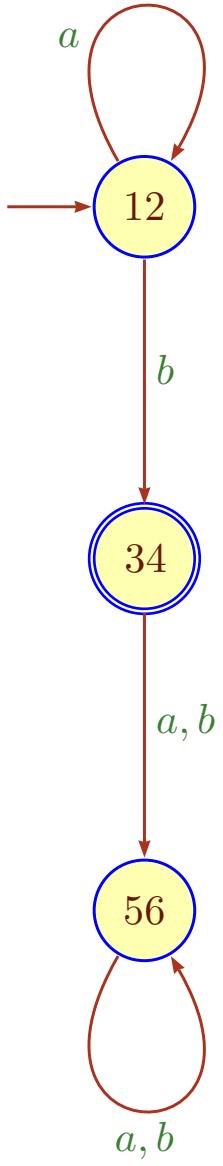


Another Step

Merging 1 and 2;
and 3 and 4.



A complete merge produces a DFA.



In the last machine, all states are inequivalent:

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

So no further state merging is possible.

Definition

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

Our goal is to exploit the following theorem for algorithmic purposes.

Theorem

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

Accessibility is computationally cheap. The merging part naturally comes in two phases:

- Determine the required partition of the state set.
- Merge the blocks into single states of the new machine.

The second phase is easy, the first requires work, in particular if one needs fast algorithms.

A Brutish Algorithm

To compute behavioral equivalence we can exploit the fact that we can check DFAs for equivalence. In its most basic incarnation this method is $O(n^2)$. There are $\binom{n}{2}$ pairs of states to check, so the total damage is $O(n^4)$.

Here is a relatively cheap improvement: exploit the fact that the machines in the equivalence test are essentially the same for all pairs. All we need is the Cartesian product of $\mathcal{A}$ with itself.

Let E be the set of pairs in hte product that cannot reach a point in $\{ (p, q) \mid p \in F \oplus q \in F \}$. Then E is the behavioral equivalence relation, given as a set of pairs.

This method is utterly simple, trivially correct, easy to implement and quadratic time. As we will see, we can do much better than that.

The state merging approach is really algebraic in nature. Given some complicated structure $\mathcal{S}$, we can always 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.

Operations on $\mathcal{S}$ extend naturally to operations on $\mathcal{S}/E$: $[x] * [y] = [x * y]$.

The important point here is that not just any equivalence will do, rather we need a congruence: an equivalence that coexists peacefully with the algebraic operations under consideration.

E.g., if $\mathcal{S}$ has a binary operation $*$ then we need

$$x \; E \; x', y \; E \; y' \quad \text{implies} \quad x * y \; E \; x' * y'$$

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

Example

The classical example is modular arithmetic: the $\mathrm{mod} m$ relation is a congruence with respect to addition and multiplication.

Useful Congruences?

Are there any relevant congruences in our case?

The problem is: congruences on what? The most tempting apprach would be to find a congruence on the state set. Alas, there really isn't much of an algebraic structure there.

Recall the $\{aa, bb\}$ example from above? The main trick is to focus on the input words $\{\varepsilon, a, b, aa, aaa\}$ that separate the states.

Why would that help? Because the set of words $\Sigma^\star$ over alphabet Σ naturally forms a monoid under concatenation. So we are looking for an equivalence relation on $\Sigma^\star$ that coexists peacefully with concatenation.

Definition

Let E be an equivalence relation on A and $B \subseteq A$. Then E saturates B if B is the union of equivalence classes of E.

In other words,

$$B = \bigcup_{x \in B} [x]_E.$$

So $(L, \Sigma^\star - L)$ is coarsest equivalence relation that saturates L, but that is not particularly interesting.

For our purposes, we need congruences on $\Sigma^\star$ that saturate L, and in particular the coarsest such congruence.

Definition

Given a language $L \subseteq \Sigma^\star$, its syntactic congruence is defined by

$$u \equiv_L v \iff \forall\, x, y \in \Sigma^\star \left(L(xuy) = L(xvy) \right)$$

Given a DFA $\mathcal{A}$, its transition congruence is defined by

$$u \approx_{\mathcal{A}} v \iff \delta_u = \delta_v$$

Here we have conflated L with its characteristic function, and $\delta_u(p) = \delta(p, u)$.

The first definition may look strange, but it turns out that syntactic congruence is indeed the coarsest congruence that saturates L.

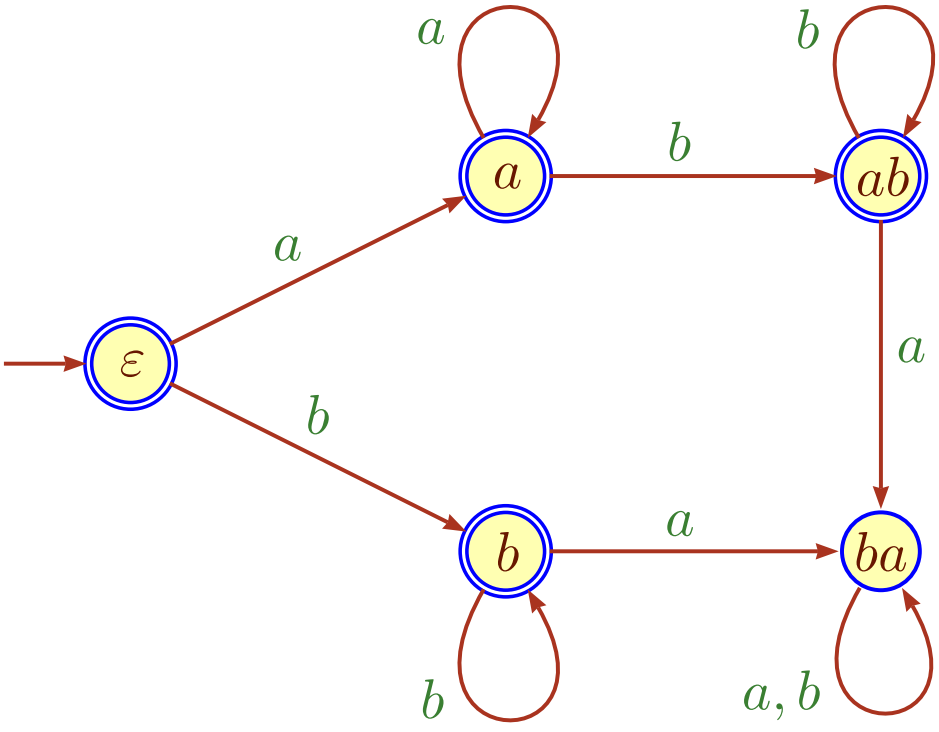
Let $L = a^\star b^\star$. By substituting appropriate values for x and y in the definition of $\equiv_L$ we can "compute" the the equivalence classes of $\equiv_L$.

For example, for $u = a$ we need $x \in a^\star$ and $y \in a^\star b^\star$ to have $xuv \in L$.

This is quite tedious because of the universal quantifier, but ultimately produces the following classes:

$$[\varepsilon] = \{\varepsilon\} \qquad [a] = a^+ \qquad [b] = b^+ \qquad [ab] = a^+ b^+ \qquad [ba] = \Sigma^\star - L$$

We can turn these equivalence classes into an automaton. Sadly, the machine is not minimal.



Works fine, but there is a DFA with just 3 states for this language.

Theorem (Myhill-Nerode 1958)

A language L is recognizable iff it is saturated by a congruence of finite index (and in particular its syntactic congruence).

Proof.

Given an accessible DFA $\mathcal{A}$ for L, its transition congruence has finite index and saturates L.

Of the opposite direction, let Q be the finite collection of equivalence classes of the given congruence. Define a DFA by

$$\begin{aligned} \delta([x], a) &= [xa] \\ q_0 &= [\varepsilon] \\ F &= \{\, [x] \mid x \in L \,\} \end{aligned}$$

This works since $\equiv$ is a congruence that saturates L.

□

As the example for $a^\star b^\star$ shows, we get a DFA that is not necessarily minimal, even if we start with the coarsest congruence possible (the least number of equivalence classes).

Incidentally, the transition congruence of the 5-state automaton from above is exactly the syntactic congruence.

So what to we do? We need a better kind of "congruence," called a right congruence:

$$x \equiv y \Rightarrow \forall\, z \left(x\, z \equiv y\, z\right)$$

The coarsest such right congruence is called the Nerode congruence.

Definition

Given a DFA $\mathcal{A}$, its right transition congruence is defined by

$$u \, \theta_{\mathcal{A}} \, v \iff \delta(q_0, u) = \delta(q_0, v)$$

As far as $\mathcal{A}$ is concerned, u and v are indistinguishable. In particular, we cannot find a separating string z such that $L(uz) \neq L(vz)$. One can check that our construction of a DFA from a congruence really only requires a right congruence, so we can construct a DFA of size the index of any right congruence.

Here is the same idea again, but this time in terms of algebra, without reference to automata.

Suppose we have a right congruence θ that saturates L, and $x\ \theta\ y$. Then $L(xz) = L(yz)$ for all z.

Definition

Let $L \subseteq \Sigma^\star$ be a 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.

Hence $x\ \theta\ y \iff x^{-1}\, L = y^{-1}\, L$ is a right congruence that saturates L. Note that θ is the kernel relation of the map $x \mapsto x^{-1}L$.

It is standard to write left quotients as

$$x^{-1}\,L$$

Here is the bad news: left quotients are actually a right action of $\Sigma^\star$ on $\mathcal{L}(\Sigma)$.

As a consequence, the first law of left quotients on the next slide looks backward.

Lemma

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

- $(xy)^{-1}L = y^{-1}x^{-1}L$,
- $x^{-1}(L \odot K) = x^{-1}L \odot x^{-1}K$ *where* $\odot$ *is one of* $\cup$, $\cap$ *or* $-$,
- $a^{-1}(LK) = (a^{-1}L)K \cup \mathsf{chr}_L\, a^{-1}K$,
- $a^{-1}L^\star = (a^{-1}L)\, L^\star$.

Here we have used the notation chr_L as a sort of characteristic function:

$$\mathsf{chr}_L = \begin{cases} \{\varepsilon\} & \text{if } \varepsilon \in L, \\ \emptyset & \text{otherwise.} \end{cases}$$

In the context of languages, this behaves just like the ordinary characteristic function in arithmetic.

Note that $(xy)^{-1}L = y^{-1}x^{-1}L$ and NOT $x^{-1}y^{-1}L$. As already mentioned, the problem is that algebraically left quotients are a right action.

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

Prove the last lemma.

Exercise

Generalize the rules for concatenation and Kleene star to words.

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 recognizable 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.

We write $\mathcal{Q}(L)$ for the set of all quotients of a language L.

Using the lemma (actually: just common sense), we can compute the quotients of $L = a^*b$.

$$
\begin{aligned}
a^{-1}\, a^*b &= a^*b \\
b^{-1}\, a^*b &= \varepsilon \\
a^{-1}\, \varepsilon &= \emptyset \\
b^{-1}\, \varepsilon &= \emptyset \\
a^{-1}\, \emptyset &= \emptyset \\
b^{-1}\, \emptyset &= \emptyset
\end{aligned}
$$

Thus $\mathcal{Q}(a^*b)$ consists of: a^*b, ε and $\emptyset$.

Note that these equations between quotients really determine the transitions of a DFA whose states are the quotients.

$$a^{-1}\,a^*b = a^*b \qquad a^*b \xrightarrow{a} a^*b$$

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

$$a^{-1}\,\varepsilon = \emptyset \qquad \varepsilon \xrightarrow{a} \emptyset$$

$$b^{-1}\,\varepsilon = \emptyset \qquad \varepsilon \xrightarrow{b} \emptyset$$

$$a^{-1}\,\emptyset = \emptyset \qquad \emptyset \xrightarrow{a} \emptyset$$

$$b^{-1}\,\emptyset = \emptyset \qquad \emptyset \xrightarrow{b} \emptyset$$

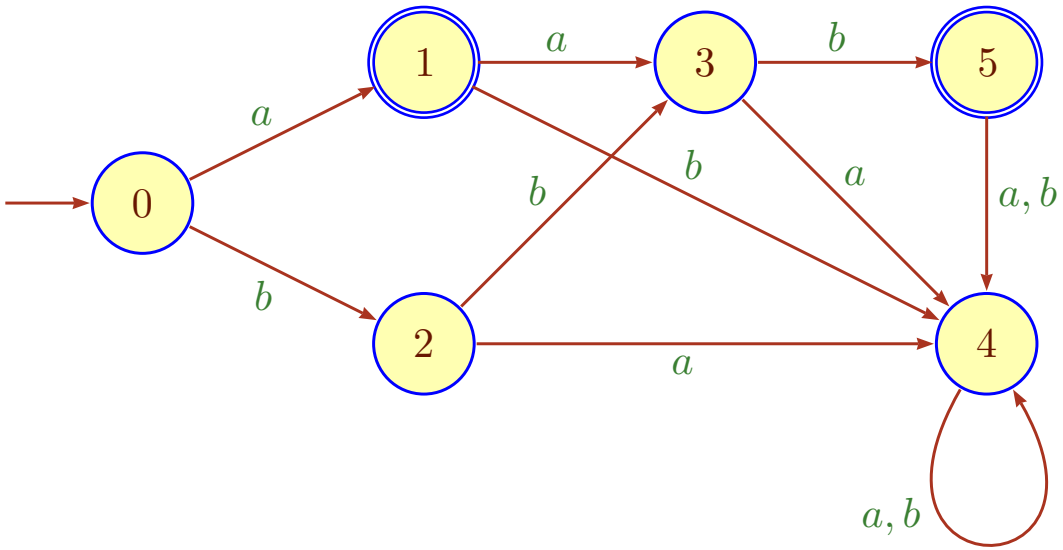
Surprise, surprise, this is exactly the machine obtained by state-merging a while ago.

Quotients Example 2

The algebra in the last example is slightly messy. However, when the language is finite, quotients are trivial to compute and implement. E.g., let $L = L_0 = \{a, aab, bbb\}$.

$a^{-1}L_0$	$\{\varepsilon, ab\}$	L_1	-
$b^{-1}L_0$	$\{bb\}$	L_2	-
$a^{-1}L_1$	$\{b\}$	L_3	-
$b^{-1}L_1$	$\emptyset$	L_4	-
$a^{-1}L_2$	$\emptyset$	-	L_4
$b^{-1}L_2$	$\{b\}$	-	L_3
$a^{-1}L_3$	$\emptyset$	-	L_4
$b^{-1}L_3$	ε	L_5	-
$a^{-1}L_4$	$\emptyset$	-	L_4
$b^{-1}L_4$	$\emptyset$	-	L_4
$a^{-1}L_5$	$\emptyset$	-	L_4
$b^{-1}L_5$	$\emptyset$	-	L_4

Hence $\mathcal{Q}(L)$ has size 6, with witnesses as follows: Moreover, there happens to be a "natural" DFA for L that has six states.



This is exactly the machine we can read off the table.

Consider our old workhorse language $L = \Sigma^\star a \Sigma \Sigma$, the third symbol from the end is an a (where $\Sigma = \{a, b\}$).

$$a^{-1} = L \cup \Sigma^2$$
$$b^{-1} = L$$

Hence any quotient of L contains L, a slightly unusual property.

Note, though, that these two equations make it fairly easy to compute all quotients.

Some Quotients

Let $L_0 = \Sigma^\star a \Sigma \Sigma$.

$a^{-1}L_0$	$L_0 \cup \Sigma^2$	L_1	-
$b^{-1}L_0$	L_0	-	L_0
$a^{-1}L_1$	$L_0 \cup \Sigma^2 \cup \Sigma$	L_2	-
$b^{-1}L_1$	$L_0 \cup \Sigma$	L_3	-
$a^{-1}L_2$	$L_0 \cup \Sigma^2 \cup \Sigma \cup \varepsilon$	L_4	-
$b^{-1}L_2$	$L_0 \cup \Sigma \cup \varepsilon$	L_5	-
$a^{-1}L_3$	$L_0 \cup \Sigma^2 \cup \varepsilon$	L_6	-
$b^{-1}L_3$	$L_0 \cup \varepsilon$	L_7	-

We have 8 quotients so far. Fortunately, this is it.

$a^{-1}L_4$	$L_0 \cup \Sigma^2 \cup \Sigma \cup \varepsilon$	-	L_4
$b^{-1}L_4$	$L_0 \cup \Sigma \cup \varepsilon$	-	L_5
$a^{-1}L_5$	$L_0 \cup \Sigma^2 \cup \varepsilon$	-	L_6
$b^{-1}L_5$	$L_0 \cup \varepsilon$	-	L_7
$a^{-1}L_6$	$L_0 \cup \Sigma^2 \cup \Sigma$	-	L_2
$b^{-1}L_6$	$L_0 \cup \Sigma$	-	L_3
$a^{-1}L_7$	$L_0 \cup \Sigma^2$	-	L_1
$b^{-1}L_7$	L_0	-	L_0

Note that every one of the 8 quotients appears twice as a result, so in the diagram every node has indegree 2 (and trivially outdegree 2).

This should look eminently familiar, it is the same as the DFA obtained from the canonical NFA by determinization. Ponder deeply.

Here is a very different example, a counting language that is a standar example for a language that is context-free but not regular.

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

This time there are infinitely many quotients.

$$\begin{aligned} (a^k)^{-1} L &= \{ a^i b^{i+k} \mid i \geq 0 \} \\ (a^k b^l)^{-1} L &= \{b^{k-l}\} && 1 \leq l \leq k \\ (a^k b^l)^{-1} L &= \emptyset && l > k \end{aligned}$$

This is no coincidence: recognizability is equivalent to having finitely many quotients.

What is really going on in all these examples?

We are computing the reachable part of vertex L in the ambient graph Reg_Σ, with edges given by $L \to s^{-1}L$, $s \in \Sigma$: the reachable part is none other than $\mathcal{Q}(L)$.

In this case, it is arguable more natural to use BFS, but any other graph exploration algorithm would work just as well.

Wurzelbrunft (who is not much of a hacker) feels very uneasy about this: can we actually implement this, or is it just pure, inherently un-implementable math?

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

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

Are all these problems surmountable?

If we were dealing with arbitrary languages we'd be in trouble. But since we are only dealing with recognizable languages we should be able to work with finite state machine.

- Use FSMs to represent the languages.
- Quotients are then easy to implement: just move the initial state(s).
- The equality test comes down to checking Equivalence, we already know how to do this.

Of course, one needs to worry about efficiency. Brute force is not going to be pretty.

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

Proposition

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

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

Proof. Duh. □

The last proposition is totally obvious to anyone who understand the definitions.

But: To convince a theorem prover one would need a precise definition of a word and a language. This is not as easy as it may seem.

Exercise

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 decomposition lemma may be blindingly obvious, but, from the right point of view, this little observation is quite helpful.

- Think of the quotients as states in a finite state machine.
- Then the Decomposition lemma describes the transitions on these states:

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

- The chr term determines whether a state is final.

Moreover, these machines are automatically DFAs.

In other words, we can build a DFA out of the quotients. To see how, suppose $Q = \mathcal{Q}(L)$ is a finite list of all the quotients of some language L.

Construct a DFA

$$\mathfrak{Q}_L = \langle Q, \Sigma, \delta; q_0, F \rangle$$

as follows:

$$\delta(K, a) = a^{-1} K$$

$$q_0 = L$$

$$F = \{ K \in Q \mid \varepsilon \in K \}$$

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 $\mathfrak{Q}_L$ 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 $\mathfrak{Q}_L$ duly accepts L.

Exercise

We can implement the quotient computation for regular languages using DFAs to represent the languages. What is the running time of the brute-force implementation?

Exercise

A simple special case occurs when the initial language is finite: we can compute quotients by word processing. What is the running time of this method? How does it compare to other methods of computing the minimal DFA for a finite language?

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.

Lemma

Let $\mathcal{A}$ 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

Suppose $\mathcal{A}$ is a DFA accepting L. Then for any word x:

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

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

Every recognizable language has only finitely many left quotients.

Corollary

Every DFA accepting a recognizable language has at least as many states as the number of quotients of the language.

Corollary

The quotient machine for a recognizable language has the lowest possible state complexity.

So now we know that for any recognizable language L the quotient automaton $\mathfrak{Q}_L$ is minimal:

$$\mathsf{st}(L) = \#\text{ quotients of } L$$

So, computing state complexity comes down to generating all quotients. We know more or less how to do this algebraically, and we have a clumsy algorithm based on manipulating DFAs.

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

Theorem

A DFA for a recognizable 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 recognizable language in question and suppose that L has n quotients.

First assume that $\mathcal{A}$ is an accessible and reduced DFA for L. Then every quotient of L must appear exactly once as the behavior of a state in $\mathcal{A}$, hence $\mathsf{st}(\mathcal{A}) = n$.

By the corollary every DFA for L has at least n states, so $\mathcal{A}$ is minimal.

Proof, contd.

For the opposite direction, clearly any minimal automaton $\mathcal{A}$ for L must be accessible.

From the corollary, $\mathsf{st}(\mathcal{A}) \geq n$ and we know how to construct a DFA with exactly n states.

But $\mathcal{A}$ is minimal, so $\mathsf{st}(\mathcal{A}) = n$.

Again every quotient of L must appear exactly once as the behavior of a state: thus $\mathcal{A}$ is reduced.

It remains to show that all DFAs for L of size n are essentially the same as the quotient machine $\mathfrak{Q}_L$ – we can rename the states, but other than that the machine is fixed.

To see this note we can define a bijection

$$f : Q \to \mathcal{Q}(L)$$
$$f(p) = [\![p]\!]$$

from the states of $\mathcal{A}$ to the states of $\mathfrak{Q}_L$ (the quotients of L).

This is a bijection since f is surjective and Q and $\mathcal{Q}(L)$ both have size n.

Suppose we have two DFA $\mathcal{A}_1$ and $\mathcal{A}_2$ and a map $f : Q_1 \to Q_2$.

We would like for f to map computations in $\mathcal{A}_1$ to computations in $\mathcal{A}_2$. For this to work, we need additional properties:

- f must preserve transitions:

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

- f must preserve in inital and final states

$$f(q_{10}) = q_{20} \qquad f(F_1) = F_2$$

Definition

A map with these properties is a DFA homomorphism.

In other words, a homomorphism maps transitions to transitions:

$$\begin{array}{lll} p & \xrightarrow{a} & \delta_1(p,a) \qquad \mathcal{A}_1 \\ \downarrow f & & \downarrow f \\ f(p) & \xrightarrow{a} & \delta_2(f(p),a) \qquad \mathcal{A}_2 \end{array}$$

By chaining together boxes of this kind we can see how whole computations in $\mathcal{A}_1$ are mapped to computations in $\mathcal{A}_2$.

Claim: If there is a homomorphism from $\mathcal{A}_1$ to $\mathcal{A}_2$, then $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$.

Dire Warning: It may still be the case that $\mathcal{L}(\mathcal{A}_1) \neq \mathcal{L}(\mathcal{A}_2)$.

To make sure that the languages are the same we need to strengthen the conditions a bit:

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

Surjective homomorphisms that have this stronger property are often called covers or covering maps.

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

$$f : Q \to \mathcal{Q}(L)$$
$$f(p) = [\![p]\!]$$

Hence we have the following lemma which shows that an arbitrary DFA for a given recognizable 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

Let L be a recognizable language and $\mathcal{A}$ an arbitrary accessible DFA for L. Then there is covering map from $\mathcal{A}$ onto $\mathfrak{Q}_L$.

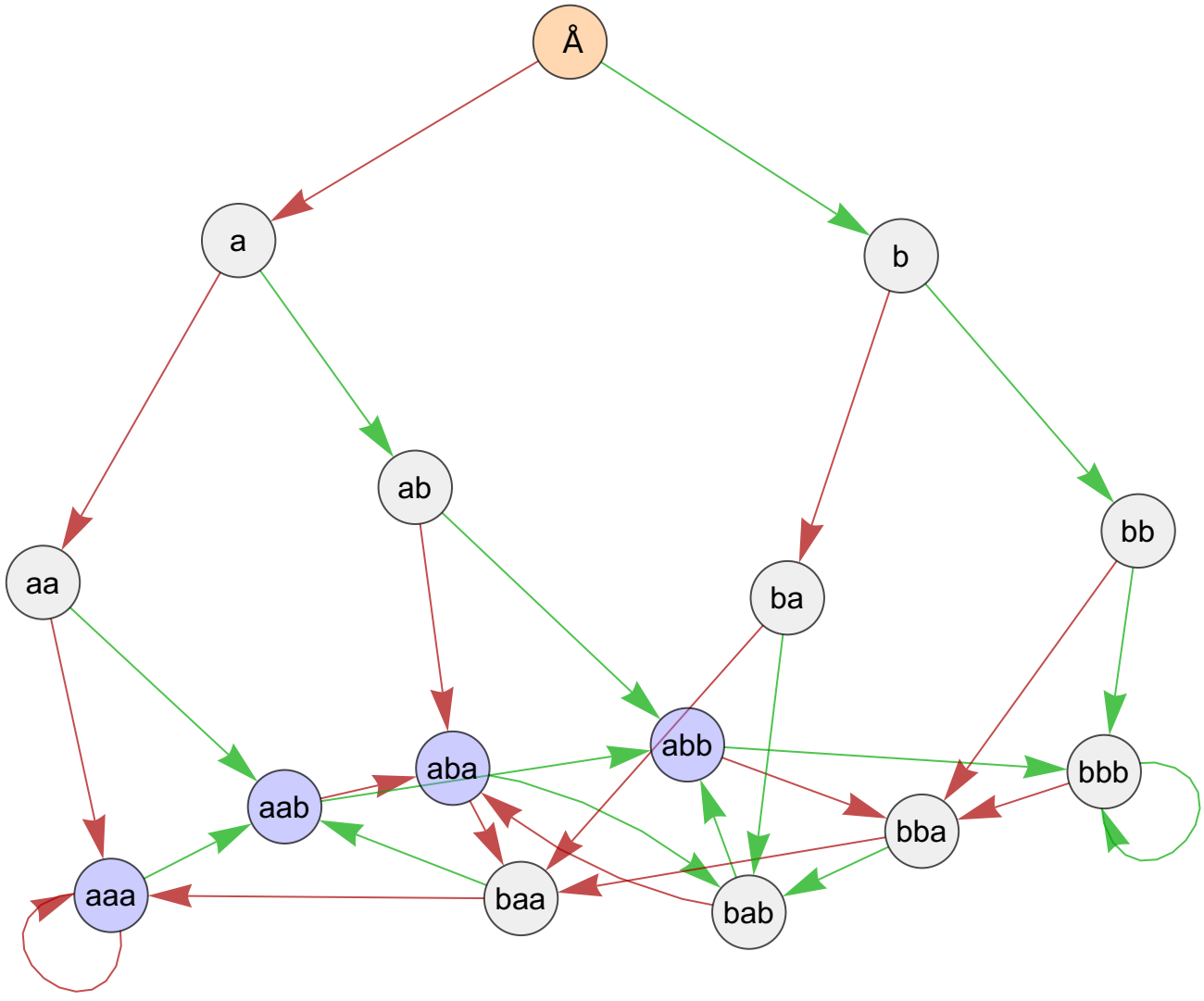
Example

Let's return to our old example $L_{a,-3} = \Sigma^\star a \Sigma \Sigma$.

Perhaps the most natural way to construct a DFA for the language is to start with ε and then remember letters until we get to 3. From then on, we drop and add one letter.

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

The initial state is ε and the final states are $\{aaa, aab, aba, abb\}$.


Å
a
b
ab
aa
ba
bb
abb
aba
bbb
aab
bba
aaa
baa
bab

We already know that the states Σ^3 suffices, we only need the de Bruijn graph but not the tree (the initial state moves to bbb). Here is the corresponding cover:

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

From the last section, we can directly derive a minimization algorithm.

Suppose we have some accessible DFA $\mathcal{A}$ for a language L.

We know that the behavioral map $p \mapsto [\![p]\!]_{\mathcal{A}}$ is a covering map onto the minimal DFA for L.

All we need to do is to compute the equivalence classes determined by the cover and then merge these blocks in $\mathcal{A}$.

That's it!

As usual, the devil is in the details.

We know we can brute force the computation of the equivalence classes since we can compare the behavior of states is $\mathcal{A}$.

- Compute the product automaton $\mathcal{A}^2 = \mathcal{A} \times \mathcal{A}$.
- Determine all pairs $(p, q) \in Q \times Q$ that can only reach points in

$$F \times F \cup (Q - F) \times (Q - F)$$

These pairs describe behavioral equivalence, done.

Certainly we can compute the points co-reachable from the bad set $F \times (Q - F) \cup (Q - F) \times F$ in quadratic time using standar graph algorithms.

In fact, we could use unordered pairs for a constant speedup.

This method is $\Theta(n^2)$: it takes that long to build the full product automaton, everything else is just linear time in the size of that machine.

Not terrible, but not great either.

One annoying aspect of this approach is that it also requires quadratic space.

To break through the quadratic barrier, we need to find a better way to compute behavioral equivalence.

The main idea is to use an iterative method: we start with the partition $(F, Q-F)$ and refine it untill we obtain behavioral equivalence. This is not particularly hard and leads to a reasonable algorithm that is often faster than quadratic, but still quadratic in the worst case. Space requirement drops down to linear, though.

One can achieve log-linear worst case, but that is quite a challenge. The method is still iteration based, but the refinement steps are substantially more complicated.

We will switch back and forth between two natural representations of the same concept.

Equivalence Relations

A relation $\rho \subseteq A \times A$ that is reflexive, symmetric and transitive.

Partition

A collection $B_1, B_2, \ldots, B_k$ of pairwise disjoint, non-empty subsets of A such that $\bigcup B_i = A$ (the blocks of the partition).

As always, we need to worry about appropriate data structures and algorithms that operate on these data structures.

Definition

Given a map $f : A \to B$ the kernel relation induced by f is the equivalence relation

$$x \; K_f \; y \iff f(x) = f(y).$$

Note that K_f is indeed an equivalence relation.

This may seem somewhat overly constrained, but in fact every equivalence relation is a kernel relation for some appropriate function f: just let $f(x) = [x]$. The codomain here is $\mathfrak{P}(A)$ which is not attractive computationally.

But, we can use a function $f : A \to A$: just choose a fixed representative in each class $[x]$.

In general we need to assume the existence of such a choice function axiomatically, but in any context relevant to us things are much simpler: we can always assume that A carries some natural total order.

In fact, usually $A = [n]$ and we can store f as a simple array: this requires only $O(n)$ space and equivalence testing is $O(1)$ with very small constants.

Definition

The canonical choice function or canonical selector for an equivalence relation R on A is

$$\mathsf{can}_R(x) = \min\Big(z \in A \mid x \ \rho \ z \Big) = \min [x]$$

So each equivalence class is represented by its least element.

To test whether $a, b \in A$ are equivalent we only have to compute $f(a)$ and $f(b)$ and test for equality. If the values of f are stored in an array this is $O(1)$, with very small constants.

Here are some basic ideas involving equivalence relations.

Definition

Let ρ and σ be two equivalence relations on A. ρ is finer than σ (or: σ is coarser than ρ), if $x\ \rho\ y$ implies $x\ \sigma\ y$. In symbols $\rho \sqsubseteq \sigma$.

Note that the index of σ is at most the index of ρ.

To avoid linguistic dislocations, we mean this to include the case where ρ and σ are the same. We will say that ρ is strictly finer than σ if we wish to exclude equality.

In terms of blocks, this means that every block of ρ is included in a block of σ (does not cut across boundaries).

If we think of equivalence relations as sets of pairs then

$$\rho \sqsubseteq \sigma \iff \rho \subseteq \sigma.$$

We need to be able to manipulate equivalence relations.

Definition (Meet of Equivalence Relations)

Let ρ and σ be two equivalence relations on A. Then $\rho \sqcap \sigma$ denotes the coarsest equivalence relation finer than both ρ and σ.

This is just logical *and*:

$$x \; (\rho \sqcap \sigma) \; y \iff x \; \rho \; y \wedge x \; \sigma \; y$$

Meet is sometimes written $\rho \cap \sigma$; which is fine if we think of the relations as sets of pairs, but a bit misleading otherwise.

The dual notion of meet is join.

Definition (Join of Equivalence Relations)

Let ρ and σ be two equivalence relations on A. Then $\rho \sqcup \sigma$ denotes the finest equivalence relation coarser than both ρ and σ.

Note that $\rho \sqcup \sigma$ is required to be an equivalence relation, so we cannot in general expect $\rho \sqcup \sigma = \rho \cup \sigma$ in the sets-of-pairs model: the union typically fails to be transitive. Hence, we have to take the transitive closure:

$$\rho \sqcup \sigma = \mathsf{tcl}(\rho \cup \sigma)$$

Let's take a closer look at the problem of computing the meet $\tau = \rho \sqcap \sigma$ of two equivalence relations:

$$p \; \tau \; q \iff \mathsf{can}_\rho(p) = \mathsf{can}_\rho(q) \wedge \mathsf{can}_\sigma(p) = \mathsf{can}_\sigma(q)$$

We may safely assume that the carrier set is $A = [n]$ and that all the relations are represented by their canonical selectors, integer vectors of length n.

$$\rho \rightsquigarrow \mathsf{can}_\rho = (r_1, r_2, \ldots, r_n)$$

But then we are really looking for identical pairs in the table

1	2	3	$\ldots$	p	$\ldots$	n
r_1	r_2	r_3	$\ldots$	r_p	$\ldots$	r_n
s_1	s_2	s_3	$\ldots$	s_p	$\ldots$	s_n

Meet Algorithm, II

```
// meet τ of ρ and σ
for p = 1, ..., n do
    i = can_ρ(p)                    // table lookup
    j = can_σ(p)                    // table lookup
    if (i, j) is new
        t_p = val(i, j) = p         // hash table
    else
        t_p = val(i, j)
return (t_1, ..., t_n)
```

Here is an example:

	1	2	3	4	5	6	7	8
ρ	1	1	1	1	5	5	1	5
σ	1	2	2	2	1	1	1	2
τ	1	2	2	2	5	5	1	8

The algorithm uses only trivial data structures except for the "new" query: we have to check if a pair has already been encountered.

The natural choice here is a hash table, though other fast container types are also plausible.

Proposition

Using array representations, we can compute the meet of two equivalence relations in expected linear time.

Exercise

Show how to implement the algorithm in linear time (not just expected) using a quadratic precomputation.

This method goes back to a paper by E. F. Moore from 1956.

The main idea is to start with the very rough approximation $(F, Q - F)$ and then iteratively refine this equivalence relation till we get behavioral equivalence.

More precisely, consider the curried transition maps $\mathcal{F} = \{ \delta_a \mid a \in \Sigma \}$ where $\delta_a : Q \to Q$, $\delta_a(p) = \delta(p, a)$.

We need the coarsest equivalence relation finer than $(F, Q - F)$ that is compatible with respect to $\mathcal{F}$. Compatible means: the δ_a do not mangle the blocks of the partition.

The constraint "coarsest" is important, otherwise we could just refine ρ to the identity (and get back the same machine).

Definition

Let $f : A \to A$ be an endofunction and $\mathcal{F}$ a family of such functions. An equivalence relation ρ on A is f-compatible if $x \; \rho \; y$ implies $f(x) \; \rho \; f(y)$.

ρ is $\mathcal{F}$-compatible if it is f-compatible for all $f \in \mathcal{F}$.

Let ρ be some equivalence relation and write $\rho^{\mathcal{F}}$ for the coarsest refinement of ρ that is $\mathcal{F}$-compatible. From the definitions

$$\rho^{\mathcal{F}} = \bigsqcup \{ \, \sigma \sqsubseteq \rho \mid \sigma \; \mathcal{F}\text{-compatible} \, \}$$

Of course, we need a real algorithm to compute this join, the set-theoretic description is not directly useful from a computational perspective.

To compute $\rho^{\mathcal{F}}$ first define for any $f \in \mathcal{F}$ and any equivalence relation σ:

$$p \; \sigma_f \; q \Leftrightarrow f(p) \; \sigma \; f(p)$$

$$\mathsf{ref}_f(\sigma) = \sigma \sqcap \sigma_f$$

It is easy to see that $\mathsf{ref}_f(\sigma)$ is indeed an equivalence relation and is a refinement of σ. The following lemma shows that we cannot make a mistake by applying these refinement operations.

Lemma

Suppose $\rho^{\mathcal{F}} \sqsubseteq \sigma \sqsubseteq \rho$. Then

- $\rho^{\mathcal{F}} \sqsubseteq \mathsf{ref}_f(\sigma) \sqsubseteq \sigma$ *for all* $f \in \mathcal{F}$.
- *If* σ *not* $\mathcal{F}$*-compatible, then* $\mathsf{ref}_f(\sigma) \sqsubsetneq \sigma$ *for some* $f \in \mathcal{F}$.

Proof

Let $\tau \sqsubseteq \rho$ be $\mathcal{F}$-compatible and assume $x\ \tau\ y$. By assumption, $\tau \sqsubseteq \sigma$. By compatibility, $f(x)\ \tau\ f(y)$, whence $f(x)\ \sigma\ f(y)$. But then $x\ \mathrm{ref}_f(\sigma)\ y$.

Since σ fails to be $\mathcal{F}$-compatible there must be some $f \in \mathcal{F}$ such that $x\ \sigma\ y$ but not $f(x)\ \sigma\ f(y)$. Hence $\mathrm{ref}_f(\sigma) \neq \sigma$.

□

Since our carrier sets are finite, the lemma guarantees that we can just apply the operations ref_f repeatedly until we get down to $\rho^{\mathcal{F}}$. The exact order of refinement steps does not matter (at least logically, efficiency is another matter).

Once we have computed the behavioral equivalence relation E (or, for that matter, any other compatible equivalence relation on Q) we can determine the quotient structure: we replace Q by Q/E, and q_0 and F by the corresponding equivalence classes.

Define

$$\delta'([p]_E, a) = [\delta(p, a)]_E$$

Proposition

This produces a new DFA that is equivalent to the old one, and reduced.

As we have seen, each refinement step is $O(n)$, so a big step is $O(kn)$ where k is the cardinality of the alphabet.

Thus the running time will be $O(knr)$ where r is the number of refinement rounds. In many cases r is quite small, but one can force $r = n - 2$.

Lemma

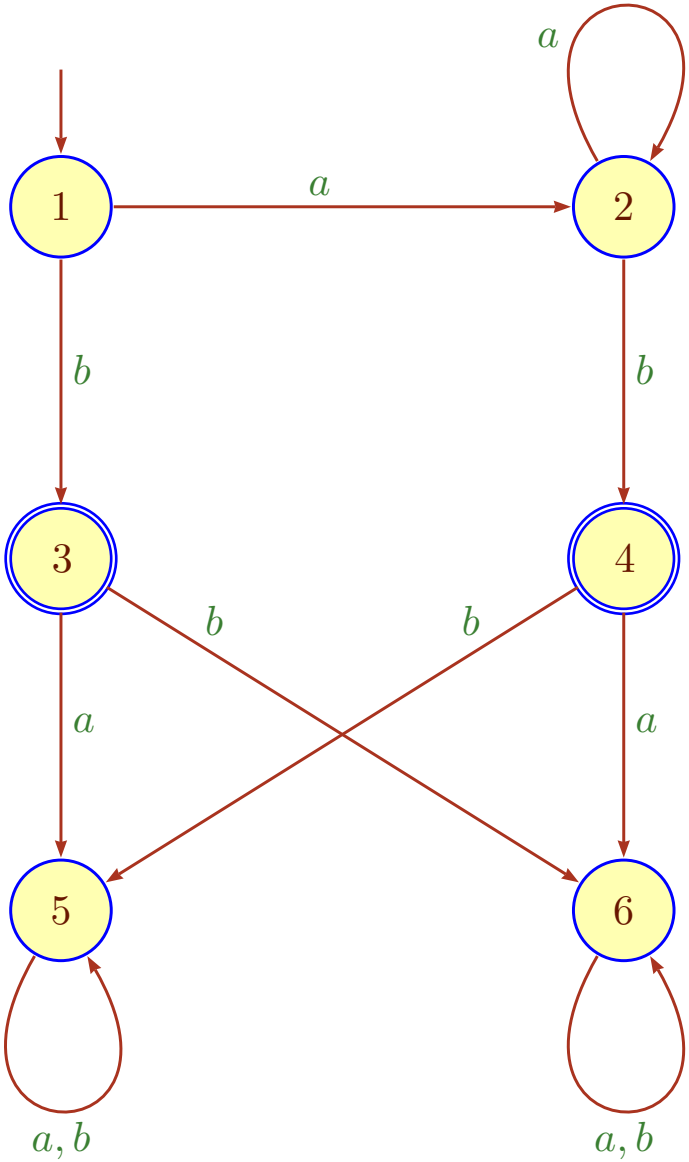
Moore's minimization algorithm runs in (expected) time $O(kn^2)$.

Exercise

Figure out how to guarantee linear time for each stage at the cost of a quadratic time initialization. Discuss advantages and disadvantages of this method.

Merging Example

The 6-state DFA for $a^\star b$.



The last DFA has transition matrix

	1	2	3	4	5	6
a	2	2	5	6	5	6
b	3	4	6	5	5	6

and final states $\{3, 4\}$.

Hence, the original approximation E_0 and the two child relations E_{0,δ_a} and E_{0,δ_b} look like so:

	1	2	3	4	5	6
E_0	1	1	3	3	1	1
E_{0,δ_a}	1	1	1	1	1	1
E_{0,δ_b}	3	3	1	1	1	1

Note that we have to perform a double table lookup to get the kids: first in the transition matrix and then in (the canonical selector for) E_0.

	1	2	3	4	5	6
E_0	1	1	3	3	1	1
a	1	1	1	1	1	1
b	3	3	1	1	1	1
E_1	1	1	3	3	5	5
a	1	1	5	5	5	5
b	3	3	5	5	5	5
E_2	1	1	3	3	5	5

Hence $E_2 = E_1$ and the algorithm terminates.

Merged states are $\{1, 2\}$, $\{3, 4\}$, $\{5, 6\}$.

To save space, we have performed giant refinement steps using

$$\mathsf{ref}(\rho) = \bigsqcap_{f \in \mathcal{F}} \mathsf{ref}_f(\rho)$$

Another Example

Consider the DFA with final states $\{1, 4\}$ and transition table

	1	2	3	4	5	6	7	8
a	2	4	5	2	6	8	4	6
b	3	5	4	3	7	4	8	7

produces the trace:

	1	2	3	4	5	6	7	8
E_0	1	2	2	1	2	2	2	2
a	2	1	2	2	2	2	1	2
b	2	2	1	2	2	1	2	2
E_1	1	2	3	1	5	3	2	5
a	2	1	5	2	3	5	1	3
b	3	5	1	3	2	1	5	2
E_2	1	2	3	1	5	3	2	5

The last minimization method may be the most canonical, but there are others. Noteworthy is in particular a method by Brzozowski that uses reversal and Rabin-Scott determinization to construct the minimal automaton.

Write

- $\mathrm{rev}(\mathcal{A})$ for the reversal of any finite state machine, and
- $\mathrm{pow}(\mathcal{A})$ for the accessible part obtained by determinization.

Thus pow preserves the acceptance language but rev reverses it.

Lemma

If $\mathcal{A}$ is an accessible DFA, then $\mathcal{A}' = \mathsf{pow}(\mathsf{rev}(\mathcal{A}))$ is reduced.

Proof.

Let $\mathcal{A} = \langle Q, \Sigma, \delta; q_0, F\rangle$.

$\mathcal{A}'$ is accessible by construction, so we only need to show that any two states have different behavior.

Let $P = \delta_x^{-1}(F) \neq P' = \delta_y^{-1}(F)$ in $\mathcal{A}'$ for some $x, y \in \Sigma^\star$.

We may safely assume that $p \in P - P'$.

Since $\mathcal{A}$ is accessible, there is a word z such that $p = \delta_z(q_0)$.

Since $\mathcal{A}$ is deterministic, z^{op} is in the $\mathcal{A}'$-behavior of P but not of P'.

□

On occasion the last lemma can be used to determine minimal automata directly.

For example, if $\mathcal{A} = \mathcal{A}_{a,-k}$ is the canonical NFA for the language "kth symbol from the end is a", then rev(pow(rev($\mathcal{A}$))) is $\mathcal{A}$ plus a sink. Hence pow($\mathcal{A}$) must be the minimal automaton.

The same holds for the natural DFA $\mathcal{A}$ that accepts all words over $\{0, 1\}$ whose numerical values are congruent 0 modulo some prime p. Then rev($\mathcal{A}$) is again an accessible DFA and pow(rev(pow(rev($\mathcal{A}$)))) is isomorphic to $\mathcal{A}$.

More generally, we can use the lemma to establish the following surprising minimization algorithm.

Theorem (Brzozowski 1963)

Let $\mathcal{A}$ be a finite state machine. Then the automaton pow(rev(pow(rev($\mathcal{A}$)))) *is (isomorphic to) the minimal automaton of $\mathcal{A}$.*

Proof.

$\widehat{\mathcal{A}} =$ pow(rev($\mathcal{A}$)) is an accessible DFA accepting $\mathcal{L}(\mathcal{A})^{\mathrm{op}}$.

By the lemma, $\mathcal{A}' =$ pow(rev($\widehat{\mathcal{A}}$)) is the minimal automaton accepting $\mathcal{L}(\mathcal{A})^{\mathrm{op}\,\mathrm{op}} = \mathcal{L}(\mathcal{A})$.

□

Which is Better?

One might ask whether Moore or Brzozowski is better in the real world. Somewhat surprisingly, given a good implementation of Rabin-Scott determinization, there are some examples where Brzozowski's method is faster.

Theorem (David 2012)

Moore's algorithm has expected running time $O(n \log n)$.

Theorem (Felice, Nicaud, 2013)

Brzozowski's algorithm has exponential expected running time.

These results assume a uniform distribution, it is not clear whether this properly represents "typical" inputs in any practical sense (say, for pattern matching algorithms).