State Complexity

Recall our definition of the state complexity $\text{sc}(L)$ of a regular 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 regular language. This is fairly rare, usually there are many canonical descriptions of an object.

But Why?

Humans are fairly good at constructing small DFAs that are already minimal—one naturally tends to avoid “useless” states. Unfortunately, this little reassuring fact does not help much:

- 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 algorithm typically do not produce minimal results.

Small Example

Here is the accessible part of the product automaton for $t_{aa,bb}$, built from the obvious DFAs for $aa$ and $bb$.

Redraw

Here is a slightly better diagram for this machine:
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).

![Diagram of a minimal automaton]

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

Minimal Automata

Definition
A DFA $A$ is minimal if there is no DFA equivalent to $A$ with fewer states than $A$.

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

- The existence of a minimal DFA is guaranteed by the fact that $N$ 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 $\delta(q_0, a) \neq q_1$.
- Need state $\delta(q_0, b)$ and $\delta(q_0, b) \neq q_1$, $\delta(q_0, a)$.
- Need state $\delta(q_0, aa)$ and $\delta(q_0, aa) \neq q_1$, $\delta(q_0, a), \delta(q_0, b)$.
- Need state $\delta(q_0, aaa)$ and $\delta(q_0, aaa) \neq q_1$, $\delta(q_0, a), \delta(q_0, b), \delta(q_0, aa)$.

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

In symbols:

$$[p] = L(\langle Q, \Sigma, \delta; p, F \rangle)$$

$$= \{ x \in \Sigma^* | \delta(p, x) \in F \}$$

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 states 1 and 2 produces a nondeterministic machine.

Another Step

Merging 1 and 2; and 3 and 4.

Complete Merge

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

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

Characterization

Our goal is to establish the following theorem.

**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.
The merging approach is really algebraic in nature. Given some complicated structure $S$, try to simplify matters as follows:
- Find an equivalence relation $E$ on $S$:
- that is compatible with the operations on $S$, and then
- replace $S$ by the quotient structure $S/E$.

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 operations on $S$ extend naturally to operations on $S/E$.

Congruences

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 $S$ has a binary operation $*$ then we need $aE b, cE d \implies a * c E b * d$.

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 $\text{mod } m$ relation is a congruence with respect to addition and multiplication.

Algebra of Languages

Given an alphabet $\Sigma$ we consider the carrier set of all languages over $\Sigma$:

$\mathcal{L}(\Sigma) = \mathcal{P}(\Sigma^*) = \{ L \mid L \subseteq \Sigma^* \}$

Note that $\mathcal{L}(\Sigma)$ is uncountable (same cardinality as the reals) even in the degenerate case $\Sigma = \{a\}$.

From a computational perspective $\mathcal{L}(\Sigma)$ is interesting only as a general framework, we need to restrict our attention to small (countable) subsets of $\mathcal{L}(\Sigma)$ if we want algorithms e.g. for the Membership Problem.

For example, we can study decidable languages in general or easily decidable languages such as regular ones.

Boolean Algebra

Since we are dealing with a powerset, there are the obvious Boolean operations union, intersection and complement that can be applied to languages over $\Sigma$. So we have a Boolean algebra

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

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 $L_1, L_2 \subseteq \Sigma^*$ 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 $L \cdot \{e\} = \{e\} \cdot L = L$.

Example

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

$$L^* = L^0 \cup L^1 \cup L^2 \cup \ldots$$

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

The choice of concatenation and Kleene star may seem rather arbitrary. It is justified by the following theorem (see lecture on Kleene algebras for a proof).

Theorem (Kleene)
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 finite state machine for the language, the decomposition can be generated algorithmically.

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

$$\langle L(\Sigma), \cup, \cdot, *, \emptyset, 1 \rangle$$

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

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

$$x^{-1}L = \{ y \in \Sigma^* \mid xy \in L \}.$$
It is standard to write left quotients as $x^{-1}L$.

Here is the bad news: left quotients are actually a right action of $\Sigma^*$ on $L(\Sigma)$. As a consequence, the first law of left quotients looks backward.

**Lemma**

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

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

Here we have used the abbreviation $\Delta(L)$ to simplify notation:

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

So $\Delta(L)$ is either zero or one in the language semiring.

### 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, we can compute the quotients of $a^*b$.

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

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

Thus $Q(a^*b)$ consists of $a^*b$, $\varepsilon$ and $\emptyset$. 

### Comments

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

<table>
<thead>
<tr>
<th>a⁻¹ a*b</th>
<th>a<em>b → a</em>b</th>
<th>a⁻¹ b</th>
<th>a*b → ε</th>
<th>ε⁻¹ ε</th>
<th>ε⁻¹ ε</th>
<th>a⁻¹ ⊘</th>
<th>⊘ → ⊘</th>
<th>b⁻¹ ⊘</th>
<th>⊘ → ⊘</th>
</tr>
</thead>
</table>

Moreover, there is a “natural” DFA for \( L \) that has six states.

\[
\begin{array}{ccc}
1 & a & b \\
2 & a & a \\
3 & b & b \\
4 & b & a \\
5 & a & b \\
6 & \epsilon & \epsilon \\
\end{array}
\]

Could this be coincidence? Nah . . .

For example, \( \delta(p_1, a)q = 2 \) and \( /llbracket 2 \rrbracket \epsilon, abu \).

Corresponding to \( a^{-1} L = t \epsilon, abu \).

A larger example, \( L = L_1 = a^*b^* \cup b^*a^* \).

\[
\begin{array}{cccccccc}
a^{-1}L_1 & a^*ba^* + b^* & b^{-1}L_1 & b^* & b^{-1}L_2 & b^* & b^{-1}L_3 & b^* & L_4 \\
b^{-1}L_1 & b^* + a^* & L_2 & a^* & L_3 & a^* & L_4 & a^* & L_5 \\
& b^{-1}L_2 & b^* + b^* & L_4 & a^{-1}L_1 & L_5 & b^* & L_6 & b^{-1}L_3 \\
& b^{-1}L_3 & b^*a*b^* & L_6 & a^{-1}L_3 & a^{-1}L_4 & b^* & L_7 & b^{-1}L_4 \\
a^{-1}L_4 & a^*b^* & L_7 & a^{-1}L_5 & a^{-1}L_6 & a^* & b^{-1}L_5 & a^{-1}L_7 & a^{-1}L_8 \\
b^{-1}L_4 & a^* & b^{-1}L_5 & a^{-1}L_6 & b^{-1}L_6 & a^* & b^{-1}L_7 & a^{-1}L_7 & a^{-1}L_8 \\
& a^{-1}L_5 & a^* & b^{-1}L_6 & a^{-1}L_6 & a^* & b^{-1}L_7 & a^{-1}L_7 & a^{-1}L_8 \\
& b^{-1}L_5 & a^* & b^{-1}L_6 & a^{-1}L_6 & a^* & b^{-1}L_7 & a^{-1}L_7 & a^{-1}L_8 \\
& b^{-1}L_6 & a^* & b^{-1}L_7 & a^{-1}L_7 & a^* & b^{-1}L_8 & a^{-1}L_8 & a^{-1}L_8 \\
& b^{-1}L_7 & a^* & b^{-1}L_8 & a^{-1}L_8 & a^* & b^{-1}L_8 & a^{-1}L_8 & a^{-1}L_8 \\
& b^{-1}L_8 & a^* & b^{-1}L_8 & a^{-1}L_8 & a^* & b^{-1}L_8 & a^{-1}L_8 & a^{-1}L_8 \\
\end{array}
\]

Exercise

Verify this table.

Here is a very different example:

\[
L = \{ a^i b^j \mid i, j \geq 0 \} = \{ \epsilon, ab, aab, aaabbb, \ldots \}
\]

This time there are infinitely many quotients.

\[
(a^i b^j)^{-1}L = \{ a^j b^i \mid i \geq 0 \}
\]

Exercise

Verify this table.

This is no coincidence: the language \( L \) is not regular.
As always, we can interpret the computation of \( Q(L) \) as a closure operation, albeit one on somewhat complicated types: the ambient set here is \( L(\sigma) \).

\[
Q(L) = \overline{L(a^{-1} \mid a \in \Sigma)}
\]

Here we have simply written \( a^{-1} \) for the operation \( K \mapsto a^{-1}K \).

In a sense, this is just another way of looking at the fixed point operation from above. But it is more appropriate in our context: it is just a somewhat abstract description of an algorithm to compute the quotient.

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

### 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 data structure 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 \( O(n^2) \) steps.

The whole algorithm is then an unimpressive \( O(n^3) \).

Can we speed this up?

Exercise

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

### Example

As a small example consider \( L = \{t, ab, aaa, abbb, bbb\} \). Then the 8 quotients are

<table>
<thead>
<tr>
<th>Quotient</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( L )</td>
<td>5 {( \epsilon ), b}</td>
</tr>
<tr>
<td>2. ( {t, aa, bb} )</td>
<td>6 ( \emptyset )</td>
</tr>
<tr>
<td>3. ( {\epsilon, bb} )</td>
<td>7 {b}</td>
</tr>
<tr>
<td>4. ( {a} )</td>
<td>8 {( \epsilon )}</td>
</tr>
</tbody>
</table>

corresponding to the partial DFA (the sink 6 is omitted):

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

**Lemma**

Let \( L \subseteq \Sigma^* \) be any language. Then
\[
L = \Delta(L) \cup \bigcup_{a \in \Sigma} a \cdot (a^{-1} L)
\]

**Proof.** Duh.

More precisely, a word \( x \in L \) is either \( \epsilon \), or it looks like \( x = au \) for some \( a \in \Sigma, u \in \Sigma^* \).

**Exercise**

Give a fastidious definition of words as functions \( w : r^n s \rightarrow \Sigma, n \in \mathbb{N} \), and use this definition to give a formal proof of the Decomposition lemma.

---

The Decomposition lemma is just about trivial. For \( L \subseteq \{a,b\}^* \) 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 \( \Delta \) term determines whether a state is final.

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

Construct a DFA
\[
\mathcal{M}_L = \langle Q, \Sigma, \delta, q_0, F \rangle
\]

as follows:

\[
\begin{align*}
\delta(K, a) &= a^{-1} K \\
q_0 &= L \\
F &= \{ K \in Q \mid \epsilon \in K \}
\end{align*}
\]

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 \( \mathcal{M}_L \), we have
\[
\delta(q_0, x) = \delta(L, x) = x^{-1} L.
\]

But then
\[
x \in L \iff \epsilon \in x^{-1} L \iff \delta(q_0, x) \in F
\]
so that \( \mathcal{M}_L \) duly accepts \( L \).

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 $A$ be an arbitrary DFA, $p$ a state and $x \in \Sigma^*$. Then
$$[\delta(p, x)] = x^{-1}[p]$$

**Proof.** Straightforward induction on $x$. Use
$$(x\alpha)^{-1}L = \alpha^{-1}(x^{-1}L)$$

**Corollary**
Suppose $A$ is a DFA accepting $L$. Then for any word $x$:
$$[\delta(p, x)] = x^{-1}L$$

**State Complexity Revealed**

So now we know that for any regular language $L$ the quotient automaton $M_L$ is minimal:
$$sc(L) = \# \text{ quotients of } L$$

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.

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

**The Minimal Automaton**

**Theorem**
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.

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

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

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

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

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

Again every quotient of $L$ must appear exactly once as the behavior of a state: thus $\mathcal{A}$ 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
$$f: Q \to Q(L)$$
$$f(p) = [p]$$

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

This is a bijection since $\mathcal{A}$ 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:

\[
\begin{array}{c}
p \xrightarrow{a} \delta(p, a) \\
f \\
f(p) \xrightarrow{a} \delta(f(p), a)
\end{array}
\]

Lastly, \( f \) maps initial to initial, and final to final states.

Hence, the states in \( A \) are just “renamed” quotients: the machines \( A \) and \( M_L \) are isomorphic.

```

Preserving Computations

It is clear that for a map \( f \) from machine \( A_1 \) to machine \( A_2 \) to be a homomorphism it must preserve transitions:

\[ p \xrightarrow{a} q \text{ implies } f(p) \xrightarrow{a} f(q) \]

Moreover, we require \( f(q_0) = q_0 \) and \( f(F_1) = F_2 \).

It follows immediately that \( L(A_1) \subseteq L(A_2) \).

However, we may still have \( L(A_1) \neq L(A_2) \) (why?), 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:

\[
\begin{align*}
    f : Q &\rightarrow Q(L) \\
    f(p) &= [p]
\end{align*}
\]

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

Let \( L \) be a regular language and \( A \) an arbitrary accessible DFA for \( L \). Then there is compatible map from \( A \) onto \( M_L \).

```

Example

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

\[
\delta(w, s) = \begin{cases} 
    ws & \text{if } |w| < 3, \\
    w_2w_3s & \text{otherwise.}
\end{cases}
\]

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

- \( aaa \rightarrow aaa \)
- \( aab \rightarrow aab \)
- \( aba \rightarrow aba \)
- \( abb \rightarrow abb \)
- \( \varepsilon, b, bb, bbb \rightarrow bba \)

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

```

Application: Minimization

The covering map provides a way to minimize a DFA \( A \): all we need to do is to merge all the states that map to the same quotient: behavioral equivalence is the kernel relation defined by the cover map.

But note that there is a bit of a vicious cycle: to compute the cover \( f \) directly we need \( M_L \). If we have the latter there is no need to minimize \( A \).

Nonetheless, covers indicate the right approach to efficient algorithms:

- Start with any DFA \( A \) for \( L \).
- Remove inaccessible states from \( A \).
- Compute the behavioral equivalence relation for \( A \).
- Lastly, merge states with the same behavior.
In principle, all we need to do is to compute the equivalence relation associated with the behavioral map: we already know that, given any accessible DFA $A = (Q, \Sigma, \delta, q_0, F)$, the behavioral equivalence relation $E$ is the kernel relation of the behavior map:

$$[\cdot] : Q \to \mathcal{L}(\Sigma), \quad p \mapsto [p]$$

While this mathematically an elegant characterization, it is a bit abstract; we need to determine its computational meaning.

This is very similar to the problem of computing quotients from above: we can represent behaviors as DFAs, we just need to make sure that all the necessary operations can be implemented.

The wary reader will notice that the product machine constructed in DFA Equivalence testing is essentially the same for all $A$ and we are moving the initial state $(p, q)$ around.

So, we could compute the full product only once and then perform a graph exploration algorithm with different starting points.

Alas, though the coefficients will be better, asymptotically the running time is still $O(n^4)$: each run of, say, DFS is potentially quadratic in $n$.

Given a DFA $A = (Q, \Sigma, \delta, q_0, F)$ on $n$ states define

$$A_p = (Q, \Sigma, \delta, p, F)$$

so that $[p] = \mathcal{L}(A_p)$.

We already know how to solve DFA Equivalence in quadratic time, so we can compute behavioral equivalence for $A$ as follows:

For each pair of states $p$ and $q$, test if $A_p$ and $A_q$ are equivalent.

Of course, the running time looks like $O(n^4)$. That’s fine if we are only interested in polynomial time, but it’s not enough for an efficient algorithm.
Implementing Equivalence Relations

Consider an equivalence relation $\rho \subseteq A \times A$.

Suppose $|A| = n$ and $|\rho| = m$, so $n \leq m \leq n^2$.

<table>
<thead>
<tr>
<th>data structure</th>
<th>test space</th>
</tr>
</thead>
<tbody>
<tr>
<td>list of pairs</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>sorted list of pairs</td>
<td>$\Theta(m)$</td>
</tr>
<tr>
<td>Boolean matrix</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>selector function</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>union/find</td>
<td>$\approx O(1)$</td>
</tr>
</tbody>
</table>

Only the last two representations are of interest if we are looking for fast algorithms.

Silly Example

Consider the equivalence relation $E$ on $[10]$ with blocks $\{1,3,5,7,9\}, \{2,6,10\}, \{4\}, \{8\}$

List of pairs

$(1,1), (1,3), (1,9), \ldots, (9,1), (2,2), (2,6), \ldots, (10,2), (4,4), (8,8)$

Sorted list of pairs: well, just sort the thing . . .

Example, contd.

A pretty picture of a Boolean matrix.

Total Recall, I

Definition

Given a map $f : A \rightarrow B$ 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 $\mathcal{P}(A)$ which is not attractive computationally.

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

The Canonical Selector Function

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 selector function for an equivalence relation $\mathcal{R}$ on $A$ is

$$\text{sel}_\mathcal{R}(x) = \min \{ z \in A \mid x \mathcal{R} z \}$$

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.

Silly Example

Consider the equivalence relation $E$ on $[10]$ with blocks

$(1,3,5,7,9), (2,6,10), (4), (8)$

List of pairs

$(1,1), (1,3), (1,9), \ldots, (9,1), (2,2), (2,6), \ldots, (10,2), (4,4), (8,8)$

Sorted list of pairs: well, just sort the thing . . .

Example, contd.

Kernel representation

Note that $p E q \iff \nu_2(p) = \nu_2(q)$ where $\nu_2(x) = \max \{ k \mid 2^k \text{ divides } x \}$.

Hence $E$ is the kernel relation of $\nu_2$.

The canonical selector function is

$(1,2,1,4,1,2,1,8,1,2)$
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,
\[
B = \bigcup_{x \in B} [x]_E.
\]

Proposition
In any DFA, the behavioral equivalence relation saturates the set of final states.

This means that behavioral equivalence is a refinement of the basic partition \( (F, Q - F) \).
We can use this as the starting point in an approximation algorithm.

Join
The dual notion of meet is join.

Definition (Join of Equivalence Relations)
Let \( \rho \) and \( \sigma \) be two equivalence relations on \( A \). Then \( \rho \cup \sigma \) denotes the finest equivalence relation coarser than both \( \rho \) and \( \sigma \).

Note that \( \rho \cup \sigma \) is required to be an equivalence relation, so we cannot in general expect \( \rho \cap \sigma = \rho \cup \sigma \) in the sets-of-pairs model: the union typically fails to be transitive. Hence, we have to take the transitive closure:
\[
\rho \cup \sigma = \text{tcl}(\rho \cup \sigma)
\]

Meet Algorithm
Let’s take a closer look at the problem of computing the meet of two equivalence relations.
We may safely assume that the carrier set is \( A = \{1, \ldots, n\} \) and that both relations \( \rho \) and \( \sigma \) are given by their canonical selectors (implemented as two arrays \( r \) and \( s \) of size \( n \)).

Let \( \tau = \rho \cap \sigma \). Then
\[
p \tau q \iff \text{sel}_r(p) = \text{sel}_s(q) \land \text{sel}_s(p) = \text{sel}_s(q)
\]
so we are really looking for identical pairs in the table

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

The selector \( t \) for \( \tau \) then looks like so:

```c
for( p = 1 .. n )
  t[p] = val(i,j) = p;
else
  t[p] = val(i,j);
```

Here are some basic ideas involving equivalence relations.

Definition
Let \( \rho \) and \( \sigma \) be two equivalence relations on \( A \). \( \rho \) is finer than \( \sigma \) (or: \( \sigma \) is coarser than \( \rho \)), in symbols \( \rho \subseteq \sigma \), if \( x \rho y \) implies \( x \sigma y \).

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

In terms of blocks this means that every block of \( \rho \) is included in a block of \( \sigma \).
If we think of equivalence relations as sets of pairs then
\[
\rho \subseteq \sigma \iff \rho \subseteq \sigma.
\]
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.

Moore's Algorithm
This method goes back to a paper by E. F. Moore from 1956.
The main idea is to start with the very rough approximation $p_{F,Q}$ and then refine this equivalence relation till we get behavioral equivalence.

More precisely, consider all the maps $F \cdot \{ \delta_a \mid a \in \Sigma \}.$
We need the coarsest equivalence relation finer than $p_{F,Q}$ that is $F$-compatible with respect to $F$.

The constraint "coarsest" is important, otherwise we could just refine $\rho$ to the identity.

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

Let $\rho$ be some equivalence relation and write $\rho^F$ for the coarsest refinement of $\rho$ that is $\mathcal{F}$-compatible. Note that $\rho^F \supseteq \sigma \supseteq \rho$ implies $\sigma$ for all $f \in \mathcal{F}$.
Of course, we need a real algorithm to compute this join.

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

\[ p \sigma q \iff f(p) \rho f(q) \]
\[ R_f(\sigma) = \sigma \cap \sigma_f \]

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

Lemma
- $\rho^F \subseteq \sigma \subseteq \rho$ implies $\rho^F \subseteq R_f(\sigma) \subseteq \sigma$ for all $f \in \mathcal{F}$.
- $\rho^F \subseteq \sigma \subseteq \rho$, $\sigma$ not $\mathcal{F}$-compatible implies $R_f(\sigma) \subseteq \sigma$ for some $f \in \mathcal{F}$.

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

Since $\sigma$ 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 $R_f(\sigma) \neq \sigma$.

According to the lemma, we can just apply the operations $R_f$ repeatedly until we get down to $\rho^F$. 

Minor Automata
The Algebra of Languages
The Quotient Machine
Computing with Equivalences
Moore’s Algorithm
Surprise, surprise, this is Yet Another Fixed Point problem. Let
\[ R(p) = \prod_{j} R_j(p) \]
Then behavioral equivalence is the fixed point of \((F, Q - F)\) under \(R\).

Alas, this giant-step method is not a good approach algorithmically, it is
preferable to perform a sequence of \(k\) baby-steps \(\rho \to R(p)\) and cycle through
the functions \(f = \delta_a\).

Exercise
Show how to streamline the fixed point algorithm by running through a
sequence of baby refinement steps.

State Merging Algorithm
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
\[ \delta'(p \rho \ast, a) = [\delta(p, a)]_E. \]

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

Exercise
Show that this merging really produces a DFA (rather than some random finite
state machine.

Running Time
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.

Computing Behavioral Equivalence
Transition matrix
\[
\begin{array}{cccccc}
    & 1 & 2 & 3 & 4 & 5 \\
\hline
    a & 1 & 2 & 3 & 4 & 5 \\
b & 2 & 3 & 4 & 5 & 6
\end{array}
\]
final states \(\{3, 4\}\):
\[
\begin{array}{cccccc}
    & 1 & 2 & 3 & 4 & 5 \\
\hline
    E_0 & 1 & 1 & 1 & 1 & 1 \\
a & 1 & 1 & 1 & 1 & 1 \\
b & 3 & 3 & 1 & 1 & 1
\end{array}
\]
\[
\begin{array}{cccccc}
    & 1 & 2 & 3 & 4 & 5 \\
\hline
    E_1 & 1 & 1 & 3 & 3 & 5 \\
a & 1 & 1 & 5 & 5 & 5 \\
b & 3 & 3 & 5 & 5 & 5
\end{array}
\]
\[
\begin{array}{cccccc}
    & 1 & 2 & 3 & 4 & 5 \\
\hline
    E_2 & 1 & 1 & 3 & 3 & 5 \\
a & 1 & 1 & 5 & 5 & 5 \\
b & 3 & 3 & 5 & 5 & 5
\end{array}
\]

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.

Another Example
Consider the DFA with final states \(\{1, 4\}\) and transition table
\[
\begin{array}{cccccccccccc}
    & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
    a & 2 & 4 & 3 & 5 & 6 & 8 & 4 & 6 \\
b & 3 & 5 & 4 & 3 & 6 & 8 & 4 & 7
\end{array}
\]
produces the trace:
\[
\begin{array}{cccccccccccc}
    & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
    E_0 & 1 & 2 & 2 & 1 & 2 & 2 & 2 & 2 \\
a & 2 & 1 & 2 & 2 & 2 & 1 & 2 & 2 \\
b & 2 & 2 & 1 & 2 & 2 & 1 & 2 & 2 \\
E_1 & 1 & 2 & 3 & 1 & 5 & 3 & 2 & 5 \\
a & 2 & 1 & 5 & 2 & 3 & 5 & 1 & 3 \\
b & 3 & 5 & 1 & 3 & 2 & 1 & 5 & 2 \\
E_2 & 1 & 2 & 3 & 1 & 5 & 3 & 2 & 5
\end{array}
\]
Digression: Brzozowski’s Method

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

Thus \( \text{pow} \) preserves the acceptance language but \( \text{rev} \) reverses it.

Key Lemma

Lemma
If \( A \) is an accessible DFA, then \( A' = \text{pow}(\text{rev}(A)) \) is reduced.

Proof.
Let \( A = (Q, \Sigma, \delta, q_0, F) \).
\( A' \) is accessible by construction, so we only need to show that any two states have different behavior.
Let \( P = \delta^{-1}(F) \neq P' = \delta'^{-1}(F) \) in \( A' \) for some \( x, y \in \Sigma^* \).
We may safely assume that \( p \in P - P' \).
Since \( A \) is accessible, there is a word \( z \) such that \( p = \delta_z(q_0) \).
Since \( A \) is deterministic, \( z^{op} \) is in the \( A' \)-behavior of \( P \) but not of \( P' \).

Application: Determining Minimal Automata

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

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

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

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 \( O(n \log \log n) \).

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.