CDM

Fast Minimization

Klaus Sutner

Carnegie Mellon University

Spring 2021

Minimization is a good example where efficient computation forces one to think more carefully than in math alone.

Suppose you have a DFA and want to construct the corresponding minimal one.

There is a trivial brute-force approach that establishes computability: Enumerate all smaller DFAs and check for equivalence.

Implementing the quotient idea brute force produces $O(n^4)$. With a little more effort one can get down to $O(n^2)$ and often better (Moore's algorithm).

Breaking through the quadratic barrier is a bit more complicated.

Mathematical Thinking: behavioral equivalence. Once the concept of behavior is clear, there is a straightforward algorithm for minimization. And, it's even polynomial time.

Algorithmic Thinking: refinement of equivalence relations. A better algorithm is obtained by thinking clearly about computing with equivalence relations (Moore). The reward is a clean, quadratic time algorithm (which is often much better than quadratic).

Smart Algo Thinking: baby-steps vs. giant-steps. Now things get tricky: all sub-quadratic algorithms require a much more careful argument and deeper algorithmic methods. A bit of creative insight is required to get down to log-linear. And doing things elegantly and efficiently is quite difficult.

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, ..., 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.

Recall our abstract scenario: we have an equivalence relation $\rho \subseteq Q \times Q$ and an endofunction $f : Q \to Q$.

We want to find the coarsest refinement $\widehat{\rho}$ of ρ that is compatible with f:

$$p \,\widehat{\rho}\, q \Rightarrow f(p) \,\widehat{\rho}\, f(q)$$

This is accomplished by repeated application of a refinement operator R_f:

$$p \; \rho_f \; q \Leftrightarrow f(p) \; \rho \; f(q)$$

$$R_f(\rho) = \rho \sqcap \rho_f$$

In other words: $\widehat{\rho}$ is the fixed point of ρ under R_f.

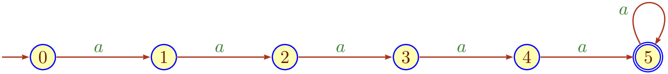
The refinement operator in Moore's algorithm works by representing all relations as canonical selector functions (aka int arrays) and scans the whole arrays for each step.

So each refinement step is $\Theta(n)$ (with good constants but still linear in n).

Quite often the algorithm uses fewer than n rounds, so the total time complexity may well be sub-quadratic.

Alas, there are cases when Moore requires $\Theta(n)$ rounds, producing quadratic running time overall.

Here is the standard example that demonstrates that Moore's algorithm may be quadratic: the minimal DFA for $\{a\}^{\geq k}$.



For this automaton, a single Moore round will split off only one state from the right end of block $D = \{0, 1, \ldots, r\}$, at a cost of $\Theta(n)$ steps.

The split occurs only because of block $B = \{r + 1\}$, nothing else matters.

Critical Idea: Maybe we could get mileage out of trying to guide the refinement by single blocks, instead of blindly hitting the whole carrier set.

Suppose ρ is a partition of Q, and consider two blocks C and B. Let $f : Q \to Q$ be some endofunction.

We say that C splits B if

$$B \cap f^{-1}(C) \neq \emptyset \quad \text{and} \quad B - f^{-1}(C) \neq \emptyset.$$

In other words, $f(B)$ intersects both C and $Q - C$ and is not f-compatible yet: stopping the refinement process at this point would produce a nondeterministic machine. We need further refinement.

Let's define a new, more complicated refinement operator $\rho' = R_f(\rho, B, C)$ as follows:

$$p \; \rho' \; q \iff (p, q \notin B \wedge p \; \rho \; q) \; \vee$$

$$\big(p, q \in B \wedge (f(p) \in C \Leftrightarrow f(q) \in C)\big)$$

In other words: outside of block B we keep the old ρ. Inside of B we check for C-equivalence of children.

So $R_f(\rho, B, C)$ is a indeed a refinement of ρ: block B is split in two.

Proposition

- $R_f(\rho) \sqsubseteq R_f(\rho, B, C)$.
- $R_f(\rho) \neq \rho$ *implies that* $R_f(\rho, B, C) \neq \rho$ *for some* B *and* C.

In other words, we make no mistakes and we can't get stuck.

Proof.

$R_f(\rho)$ is $\bigsqcap_{C,B} R_f(\rho, B, C)$ and thus finer than each part.

If $R_f(\rho) \neq \rho$ there must be some block B and $p, q \in B$ such that $\neg(f(p)\ \rho\ f(q))$.

Let C be the block containing $f(p)$, done. □

Of course, from a complexity perspective this may not sound too promising: we are breaking one giant step into multiple baby steps. It is not unreasonable to suspect that this might even increase running time.

But: The baby steps provide much better control over the selection of the next refinement step: we can choose the blocks involved at will.

With a little effort this feature can be exploited sufficiently to speed up the whole process.

Since we are interested in the case where $f = \delta_a$ in a DFA, let us lighten notation a bit and write $a^{-1}C$ rather than $\delta_a^{-1}(C)$. Also, for $B, C \subseteq Q$, write

$$B/C$$

for the partition induced by splitting B via $a^{-1}C$: $B^+ = B \cap a^{-1}C$ and $B^- = B - a^{-1}C$.

Computing B/C should be very cheap.

Suppose we have $a \in \Sigma$ and we want to insure compatibility with δ_a (see below for the general case). In addition, we have a 2-coloring χ of Q.

The algorithm maintains two data structures:

- a partition P of Q, representing the equivalence relation,
- a split list S with entries some of the blocks in the partition.

Both partitions are initialized by the coloring.

We refer to the blocks C in S as active: we will refine P by $a^{-1}C$.

The algorithm extracts an active block from the split list and tries to refine the blocks in the partition accordingly.

It then updates the split list in a clever way, and stops when the split list becomes empty.

It is convenient to initialize the partition and split lists according to some arbitrary coloring $\chi : Q \to \mathbf{2}$: this includes the usual final/non-final distinction but also accommodates other situations.

Initializing P is straightforward: every color produces one block.

The split list S only gets the smaller of the two blocks (we are dealing with the single-function case here, see below on how to handle larger alphabets).

initialize partitions P and S to the given coloring

while S not empty **do**
  extract C from S
  compute $\widehat{C} = a^{-1}C$
  foreach block B split by C **do**
    $B^+ = B \cap \widehat{C}$
    $B^- = B - B^+$
    replace B by B^+ and B^- in P // update partition
    if B is in S // update split list
    then replace B by B^+, B^- in S
    else replace B by the smaller of B^+, B^- in S
end

At first glance it may seem like we are not doing enough work: in the last case it feels like both B^+ and B^-, the parts of B obtained by splitting wrto the critical block C, should be added to the split list.

Otherwise we might miss out on splitting some other block D that is not split by B but split by B^+ or B^-. The algorithm might stop without having produced an f-compatible relation.

Fortunately, this cannot happen: in the critical case, if B^+ splits block D, then so does B^-, and the other way around. By picking the smaller of the two blocks, we get the desired speedup.

The following observation by Hopcroft makes this idea of "no-missing-work" more formal.

Let B be a block in a partition, split into B^+ and B^-. Let X be a subset of a block.

Proposition

$$X/B \sqcap X/B^+ = X/B \sqcap X/B^- = X/B^+ \sqcap X/B^-$$

Call a set $Z \subseteq Q$ of states safe for partition P (or simply P-safe) if $f^{-1}(Z)$ does not split any block in P.

We will show that the following assertion is a loop invariant:

$$\forall\, X \in P - S \;\exists\, A \subseteq S \left(X \cup \bigcup A \quad P\text{-safe} \right)$$

Clearly, this assertion holds before the loop ever executes when there are only two blocks in P (and one in S, the smaller of the two blocks).

As is customary, we indicate the value of a variable after one more execution of the loop-body by attaching a prime: so P' is the partition after one more round.

In the following we argue about the state of affairs at the end of a round.

So we need to show that

$$\forall\, X \in P{-}S\; \exists\, A \subseteq S \left(X \cup \bigcup A \quad P\text{-safe} \right)$$

implies that

$$\forall\, X \in P'{-}S'\; \exists\, A \subseteq S' \left(X \cup \bigcup A \quad P'\text{-safe} \right)$$

So assume we have some arbitrary block $X \in P' - S'$.

So X is an old block that has just been removed from the split list.

Since X is old, it has not been split in the last round, yet was removed from the split list.

The only way this can happen is that $C = X$ is the critical block. But then safety is a direct result of the construction.

This time we have an induction hypothesis: $Z = X \cup \bigcup A$ is P-safe for some $A \subseteq S$.

Case 2.1: $C \notin A$

In this case we can replace split blocks in A: replace X by X^+ and X^- (which are both active) and we have $Z = X \cup \bigcup A'$ is P-safe. It is easy to check that Z is also P'-safe.

Case 2.2: $C \in A$

Again, we can replace split blocks in A; alas, we lose $C \notin S'$ to get $X \cup \bigcup A_0$. But no block $Z \in P'$ can be affected by this: it was already split wrto $f^{-1}(C)$ during the round.

So block X is new and was created by splitting a block Z in the last round.

Say, $X = Z^+$ and $Z \cup \bigcup A$ is P-safe for some $A \subseteq S$.

As before we can handle split blocks in A.

Case 3.1: $C \notin A$

Then $X \cup (\bigcup A' \cup Z^-$ is P'-safe as in Case 2.1. Note that indeed $Z^- \in S'$ by construction.

Case 3.2: $C \in A$

In this case, $X \cup \bigcup A_0 \cup Z^-$ is P'-safe as in Case 2.2.

Each block in P is represented by a doubly-linked list.

We maintain an array of pointers to these lists for P and similarly for S. We also keep track of the cardinality of each block.

Furthermore, we have an array of pointers so that `pos[p]` points to the list node containing p, plus information about the current block containing p.

The key part of each round is the computation of $\widehat{C} = a^{-1}C$. We may assume that $f^{-1}(p)$ has been precomputed for each state p. We can traverse C in time linear in $|C|$.

When a block B is hit for the first time, we start splitting it into two lists B^+ and B^-. If, in the end, $B^- = \emptyset$ we simply replace B by B^+.

All this can be handled in time $O(|a^{-1}C|)$.

Let us say that a state p is active if $p \in \bigcup S$, inactive otherwise.

At level 0, at most half of all states are active.

Each state in the critical block C becomes inactive, but maybe reactivated later. Hence we can naturally assign activation levels 0, 1, 2, … to all active states.

Recall that we only reactivate at most half the states in a block, so no state can be activated more than $\log n$ times. But then the total work computing pre-images of active states during the whole execution is just $O(n \log n)$.

Hence the running time of the whole algorithm is bounded by $O(n \log n)$.

This is a seminal paper that will bring tears to your eyes.

> J. Hopcroft
> A N log N Algorithm for Minimizing States in a Finite Automaton
> STAN-CS-71-190

We modify the split list in the algorithm to contain entries

$$(a, C) \in S$$

where C is a block and $a \in \Sigma$: the intent is that we later refine via $a^{-1}C$.

Of course, (a, X) is smaller than (a, Y) if $|X| \leq |Y|$.

Note that we need to add (a, X) for all $a \in \Sigma$, which produces a running time of $O(kn \log n)$.

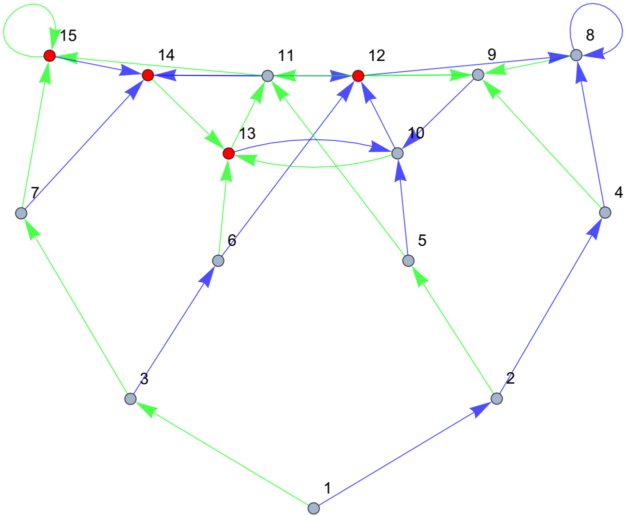
The following example uses a machine over alphabet $\{a, b\}$ with 15 states. The transition matrix is

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	2	4	6	8	10	12	14	8	10	12	14	8	10	12	14
b	3	5	7	9	11	13	15	9	11	13	15	9	11	13	15

The final states are $\{12, 13, 14, 15\}$.

The following table shows the element extracted from the split list in the first column and the blocks in the second.

Note that split list entry (a, i) means: use the ith block in the current partition with respect to $f = \delta_a$.


15
14
11
12
9
8
13
10
7
4
6
5
3
2
1

$((12,13,14,15),(1,2,3,4,5,6,7,8,9,10,11))$	$a,1;b,1$
$((14,15),(1,2,3,4,5,6,7,8,9,10,11),(12,13))$	$b,1;a,3;b,3$
$((14,15),(6,7,10,11),(12,13),(1,2,3,4,5,8,9))$	$b,1;a,3;b,3;a,2;b,2$
$((15),(6,7,10,11),(12,13),(1,2,3,4,5,8,9),(14))$	$a,3;b,3;a,2;b,2;a,5;b,5$
$((15),(7,11),(12,13),(1,2,3,4,5,8,9),(14),(6,10))$	$a,3;b,3;a,2;b,2;a,5;b,5;a,6;b,6$
$((15),(7,11),(13),(1,2,3,4,5,8,9),(14),(6,10),(12))$	$a,5;b,5;a,6;b,6;a,7;b,7$
$((15),(7,11),(13),(3,5,9),(14),(6,10),(12),(1,2,4,8))$	$a,5;b,5;a,6;b,6;a,7;b,7;a,4;b,4$

In the first step, $a^{-1}(12,13,14,15) = (6,7,10,11,14,15)$ and we split both existing blocks.

This trace only shows steps where the partition changes. Note there are many "useless" steps at the end.

The algorithm is quite messy to implement correctly, as can be seen from the following papers:

D. Gries
Describing an algorithm by Hopcroft
Acta Informatica, 2 (1973) 97–109.

T. Knuutila
Re-describing an algorithm by Hopcroft
Theoretical Computer Science, 250 (2001) 333–363.

Exercise

Implement Hopcroft's algorithm, correctly.

Theorem (Hopcroft 1971)

Hopcroft's algorithm minimizes a DFA in $O(kn \log n)$ steps, where n is the state complexity of the DFA and k the size of the alphabet.

Unsurprisingly, the running time of Hopcroft's algorithm depends on the size k of the alphabet.

Knuutila pointed out in 2001, one can produce cubic (in n) running time when $k = n/2$ and a poor method of choosing the "smaller" block is used.

Note that Hopcroft's algorithm is nondeterministic in several ways.

- We can extract any element from the split list (e.g., could use a stack, queue, ...).
- Likewise we can place the new entries anywhere in the split list.
- When B^+ and B^- have the same size, we can pick either one.

None of these choices effect correctness, but they may well influence running time.

It should be noted that the algorithm often takes far fewer than $n \log n$ steps.

Given a reasonable implementation, every round is linear. It turns out to be quite difficult to construct inputs where the algorithm requires $\log n$ many rounds.

The best result known today is that there are some DFAs such that the algorithm takes $n \log n$ steps for a certain choice of active blocks in the main loop.

Alas, for these machines a different choice of active blocks results in linear running time.

- When is the running time $\Omega(n \log n)$ regardless of the chosen split list protocol? (A unary example is known where the execution sequence is essentially unique and reaches the log-lin bound.)
- What is the average complexity of Hopcroft's algorithm (average with respect to input automaton and/or split list protocol)?
- Is Hopcroft faster than Moore on average? For the uniform distribution, Moore has expected behavior $O(n \log n)$ and it may be that the constants are smaller (Bassino,David,Nicaud 2009).

There are many examples where the number of transitions m in a partial DFA is much smaller than $k \cdot n$, k the size of the alphabet and n the number of states. This leads naturally to an algorithmic question:

> Is there a $O(m \log n)$ minimization algorithm that deals directly with partial transition functions?

This would also nicely encapsulate problems with alphabet size in a parameter that really reflects the size of the data structure: unused symbols do not inflate the transition function.

The high-level logic is similar as in Hopcroft's algorithm: one maintains a partition of Q and tries to refine the partition by splitting with sets of the form

$$\delta_a^{-1}(B)$$

The choice of B is a bit more complicated, though.

Special care is taken to avoid unnecessary computation when preimages of blocks are computed. To this end, the algorithm also maintains and refines a second partition of the transitions.

As an implementation detail: the partition data structure is all array-based (unlike Hopcroft's algorithm).

Write $[n]_0$ for $\{0, 1, \ldots, n-1\}$.

Suppose we wish to maintain a partition of $[n]_0$. Keep track of r, the number of blocks, and maintain two maps (arrays)

$$\mathsf{lo}, \mathsf{hi} \colon [r]_0 \longrightarrow [n]_0$$

plus an array `P[n]` such that

> B_d, the block number d, is located in $P[\mathsf{lo}[d], \mathsf{hi}[d]-1]$

We also have location and block-number maps

$$\mathsf{loc} \colon [n]_0 \longrightarrow [n]_0$$
$$\mathsf{bnum} \colon [n]_0 \longrightarrow [r]_0$$

such that $P[\mathsf{loc}[p]] = p$, $\mathsf{loc}[P[i]] = i$ and $p \in B_{\mathsf{bnum}[p]}$.

It is convenient to subdivide the splitting process into three phases:

- Pre-splitting: initialize offset pointers $\text{mrk}[d] = \text{lo}[d]$ for all blocks, create empty hit list.
- Splitting: process a sequence of elements, swap each to the "marked" part $P[\text{lo}[d], \text{mrk}[d] - 1]$ of their respective blocks. If block B_d is encountered for the first time, add to hit list.
- Post-splitting: walk through blocks in hit list and update to maintain invariants.

It is straightforward to arrange the post-splitting phase so that whenever B splits into B_1 and B_2, the larger part replaces B and the smaller part receives the higher block index (the current value of r).

As already mentioned, the minimization algorithm uses two PRDS:

- P: represents a partition of Q; initialized to $(F, Q - F)$, as usual.
- T: represents a partition of the transitions; initialized to blocks containing all transitions with the same label.

The main loop of the algorithm looks like this:

> **foreach** T-block C **do**
>   split blocks in P via C
>   **foreach** P-block B **do**
>   split blocks in T via B

New blocks are “appended” in both partitions, so the traversals end when all blocks have been processed.

Two Splitting Procedures

pre-split P
foreach transition $p \xrightarrow{a} q$ in block C **do**
    mark source p in P-partition
post-split P

pre-split T
foreach state p in block B **do**
    foreach transition $t : q \xrightarrow{a} p$ **do**
        mark t in T-partition
post-split T

Note that for the transition splitting operation one needs to be able to traverse all transitions with a fixed target.

To this end one precomputes two arrays

$$\mathsf{trn}\colon [m]_0 \longrightarrow [m]_0$$
$$\mathsf{fst}\colon [n] \longrightarrow [m]_0$$

such that for each state p

the transitions with target p are located in $\mathsf{trn}[\mathsf{fst}[p], \mathsf{fst}[p+1]-1]$

This is easy via counting sort.

Recall that bnum(p) is the block number of state p. By abuse of notation, bnum(t) is the block number of transition t.

Proposition

- *States:* bnum$(p) \neq$ bnum(q) *implies* $[\![p]\!] \neq [\![q]\!]$
- *Transitions:* bnum$(s) \neq$ bnum(t) *and* lab$(s) =$ lab(t) *implies* $[\![\text{trg}(s)]\!] \neq [\![\text{trg}(t)]\!]$

Proposition

Let s, t *be two transitions in the same* T*-block.*
If $\mathsf{src}(s)$ *and* $\mathsf{src}(t)$ *are in different* P*-blocks, then at least one of the blocks is unprocessed.*

Proposition

Let p, q *be two states in the same* P*-block and* $s : p \xrightarrow{a} p'$, $t : q \xrightarrow{a} q'$ *two transitions in different* T*-blocks. Then at least one of these blocks is unprocessed.*
If $s : p \xrightarrow{a} p'$ *and there is no* a*-transition with source* q*, then* s *is in an unprocessed* T*-block.*

Theorem

The algorithm correctly minimizes a trim partial DFA in $O(m \log n)$ steps.

Proof.

It follows from the propositions that two states have the same behavior iff, upon completion, they are in the same P-block.

For running time, note that since we are dealing with a trim automaton, we have $n \leq m + 1$.

As in Hopcroft's algorithm, each state can be active at most $\log n$ times and, likewise, a transition can be active and at most $\log m$ times.

So the total running time is $O(m \log n)$.

This is another paper that will bring tears to your eyes, but for entirely different reasons.

> Antti Valmari
>
> Fast brief practical DFA minimization
>
> Information Processing Letters 112 (2012) 213–217

Let's return to our old problem of testing two DFAs for equivalence.

Problem:	**Equivalence**
Instance:	Two DFAs M_1 and M_2.
Question:	Are the two machines equivalent?

This is of course decidable, but the interesting question is whether there are any fast algorithms.

Mathematical Thinking. Directly apply the fact that we have an effective Boolean algebra, plus a simple Emptiness algorithm.

Algorithmic Thinking. Exploit the particular normal form for DFAs, and the fast algorithms that can be used to compute it.

Smart Algorithm Thinking. Compute a direct comparison between the two given machines, exploiting Union/Find.

Two DFAs M_1 and M_2 are isomorphic if there is a bijection $f : Q_1 \to Q_2$ such that

$$f(q_{10}) = q_{20}$$
$$f(\delta_1(p, a)) = \delta_2(f(p), a)$$
$$f(F_1) = F_2$$

As usual, there is an associated decision problem.

Problem:	**Isomorphism of DFA**
Instance:	Two DFAs M_1 and M_2.
Question:	Are M_1 and M_2 isomorphic?

To check whether two DFAs are isomorphic one can use a variant of depth-first-search. We may safely assume the machines have the same size.

- Set $f(q_{01}) = q_{02}$.
- Extend the domain of f according to δ_1, and the range according to δ_2.
- Stop with failure if there ever is a clash (new point on one side, old point on the other; two different old points).
- Check if f maps the final states properly.
  If so, return Yes, otherwise return No.

Exercise

Figure out the details of this isomorphism testing algorithm.

Theorem

Given two DFAs one can test whether they are equivalent in log-linear time.

Proof.

Given M_1 and M_2 we can compute the corresponding minimal automata M_1' and M_2' in log-linear time. Then we can check in linear time whether M_1' and M_2' are isomorphic. $\square$

Note that even using Moore's algorithm this will often produce sub-quadratic running time.

Since isomorphism testing part of our algorithm is very fast, the real cost is incurred in minimization.

Wild Idea: Can we somehow avoid minimization?

Of course, we also want to avoid a product machine construction.

Suppose the two given DFAs have disjoint state sets Q_1 and Q_2 and set $Q = Q_1 \cup Q_2$. For simplicity write δ for $\delta_1 \cup \delta_2$.

We could try to define an equivalence relation E on Q that will tell us (among other things) whether the two initial states have the same behavior. And do it fast.

```
set E = identity;
set active = ( (q01,q02) );

while( active != empty )
     (p,q) = active.extract();
     if( ! p E q ) {
        set p E q;
        foreach a in S do
            set (delta(p,a),delta(q,a)) active;
```

The algorithm returns false whenever two states $p \in Q_1$ and $q \in Q_2$ are defined to be equivalent under E but

$$p \in F_1 \text{ xor } q \in F_2.$$

Otherwise we return true.

We can use a simple stack for the active list, the interesting question is how to maintain the equivalence relation E.

Note that this is the dynamical situation: initially E is just the identity relation, but as we go along we discover more related pairs.

So really we need to compute the equivalential closure of all the pairs (p, q) we have discovered. Union/Find is perfect for this and produces essentially linear running time.

By initialization $q_{01} \; E \; q_{02}$.

If $p \in Q_1$ and $q \in Q_2$ such that $p \; E \; q$ then $\delta(p, a) \; E \; \delta(q, a)$ for each $a \in \Sigma$.

By induction for any word x

$$\delta(q_{01}, x) \; E \; \delta(q_{02}, x).$$

Using a loop invariant one can show that in fact

$$E \sqcap Q_1 \times Q_2 = \{ \, (\delta(q_{01}, x), \delta(q_{02}, x)) \mid x \in \Sigma^\star \, \}$$

Exercise

Give a detailed proof for the correctness of this algorithm.

We have seen $O(n \log n)$ minimization algorithms. However, no better algorithm is known to date.

Exercise

Why can't this equivalence testing algorithm (or rather: some slight modification thereof) be used to compute the minimal automaton in nearly linear time?

Note that we do not claim that equivalence is polynomial time decidable for nondeterministic machines. In fact, one can show that equivalence testing is $\mathbb{NP}$-hard (even PSPACE-hard) for nondeterministic machines, even if one of the two machines simply accepts all words over the input alphabet.

Likewise it is computationally hard to find minimal nondeterministic machines for a given language. Moreover, the minimal nondeterministic machine is not unique in general.

Recall that our original definition of recognizability was based on DFAs: a language is recognizable if it is accepted by some DFA.

As Rabin-Scott showed, one can naturally generalize the underlying machines:

Theorem

A language is recognizable iff it is accepted by an NFA (or even and NFAE).

We have already seen the following.

Theorem

A language $L \subseteq \Sigma^\star$ *is recognizable iff there is a finite monoid* M, $M_0 \subseteq M$ *and a monoid homomorphism* $f : \Sigma^\star \to M$ *such that* $L = f^{-1}(M_0)$.

A similar result is the following.

Theorem

A language $L \subseteq \Sigma^\star$ *is recognizable iff there is a congruence on* $\Sigma^\star$ *of finite index that saturates* L.

The last theorem also holds when congruence is replaced by right congruence.

Also, our quotient approach to minimization provides as a pleasant side effect yet another characterization.

Lemma

A language is recognizable iff it has finitely many quotients.

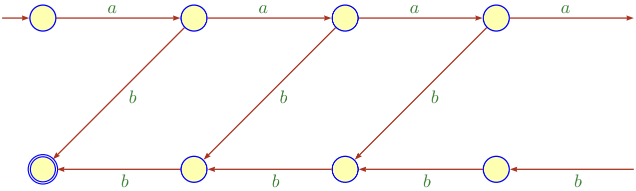
This follows immediately from the preceding results.

Interesting point: the quotient automaton makes perfect sense even for non-recognizable languages, it just has infinitely many states in this case.

If the language is simple, the structure of the diagram will be very recognizable. This leads naturally to machines more powerful as finite state machines yet not as strong as Turing machines (such as pushdown automata).

Example: $a^i b^i$

Part of the quotient machine for the non-recognizable language $L = \{ a^i b^i \mid i \geq 0 \}$ looks like this (we omit the pesky sink).



The picture very much suggests the augmentation necessary for a plain DFA to deal with L: a single counter will do (a stack with just one stack symbol).

Exercise

Figure out how to extend a DFA so it can deal with L, keeping the additions a simple as possible.

Alas, making sure that there are indeed infinitely many quotients can be difficult, a less challenging test would be helpful.

Lemma (Pumping Lemma)

For every recognizable language L there is a constant n such that for all words $x \in L$ with $|x| \geq n$ we have $x = uvw$ where $v \neq \varepsilon$, $|uv| \leq n$ and $uv^t w \in L$ for all $t \geq 0$.

Proof.

Consider the minimal DFA M for L and let n be the number of states of M. Then any word in L of length at least n must trace a loop in the diagram of M. The claim follows.

□

The Pumping lemma is useless to establish recognizability but often the weapon of choice to refute it.

$L = \{ a^i b^i \mid i \geq 0 \}$ fails to be recognizable.

Assume otherwise.

Let n be as in the PL and consider $x = a^n b^n \in L$.

Then $x = uvw$ and $v = a^i$ for some $i > 0$.

But then $uv^t w \notin L$ for all $t \neq 1$, contradiction.

It follows that the language P of balanced parentheses is also non-recognizable.

For otherwise $P \cap a^* b^* = L$ would also be recognizable, which assertion we already know to be false.

$L = \{ zz \mid z \in \Sigma^\star \}$ fails to be recognizable.

Assume otherwise.

Let n be as in the PL and consider $x = ab^n ab^n \in L$.

Then $x = uvw$ and $|uv| \leq n$. If $v = a$ we get a contradiction with $t = 0$. If $v = b^i$ for some $i > 0$ we also get a contradiction with $t = 0$.

The problem here really is that a DFA cannot remember an arbitrarily long prefix z which is needed to check the remainder of the input.

$L = \{ zz^r \mid z \in \Sigma^\star \}$ fails to be recognizable.

Assume otherwise.

Let n be as in the PL and consider $x = (ab)^n (ba)^n \in L$.

Then $x = uvw$ and $|uv| \leq n$.

A straightforward but tedious argument shows that $uv^t w$ cannot be a palindrome for any $t \neq 1$.

As in the last example, a DFA cannot remember an arbitrarily long prefix z which is needed to check the remainder of the input.

Incidentally, between

$$L_1 = \{\, zz \mid z \in \Sigma^\star \,\}$$

and

$$L_2 = \{\, zz^r \mid z \in \Sigma^\star \,\}$$

the second one (even length palindromes) is much simpler:

To recognize these words one only needs to attach a stack to a FSM (context free language).

For L_1, a simple stack is not sufficient (context sensitive language).

Back to quotients. Suppose we want to check divisibility by 5 for numbers given in reverse binary notation where the MSD comes last. In this case the value function ν^R looks like this:

$$\nu^R(x_0 x_1 \ldots x_k) = \nu(x_k x_{k-1} \ldots x_0) = \sum_{i \leq k} x_i 2^i$$

Appending a new digit changes the numerical value in a more complicated way here than in ordinary binary:

$$\nu^R(xa) = \nu^R(x) + a 2^{|x|}$$

Cop-out: We could push the digits on a stack and then use our old DFA for standard binary notation, but that requires extra memory.

So how do we construct a DFA, a memory-less device, for reverse binary directly?

Note that the real issue here is that we read input strings

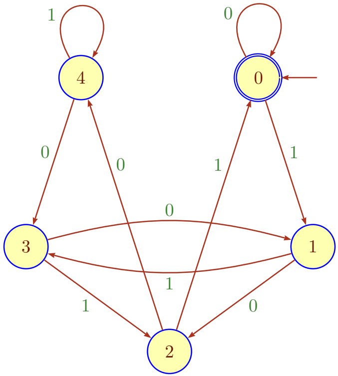
$$x_0 x_1 x_2 \dots x_{n-1} x_n$$

from left to right; this convention is hard-wired into our definition of finite state machine. Of course, the choice of direction is completely arbitrary (more precisely, a western cultural artifact).

Of course, we have a theorem that shows that L^{op} is recognizable whenever L is. However, the result relies on nondeterminism and determinization; we would really like a structural understanding of the machine for L^{op}.

In ordinary base 2 the state set is $Q = \{0, \ldots, 4\}$ and the transition function of the canonical Horner automaton is determined by

$$\nu(xa) = 2 \cdot \nu(x) + a \pmod 5$$



It is tempting to use the same state set $Q = \{0, \dots, 4\}$ for reverse binary. But the transition function now is given by

$$\nu^R(xa) = \nu^R(x) + a \cdot 2^{|x|} \pmod 5$$

We can represent $\nu^R(x)$ as a state, but not the $|x|$.

There are two basic solutions:

- Make the state set more complicated so we can keep track of the missing information. Quite possible, but leads to a larger machine.
- Try to find another way of describing a transition function that works for Q directly (assuming that such a thing really exists).

What state set do we need to implement the transition function

$$\nu^R(xa) = \nu^R(x) + a \cdot 2^{|x|} \pmod 5?$$

We want to keep track of

- The current value $\nu^R(x) \bmod 5$, and
- the current multiplier $2^{|x|} \bmod 5$.

Hence we have a state set $Q = \{0, 1, 2, 3, 4\} \times \{1, 2, 3, 4\}$ and transitions

$$\delta((p, q), a) = (p + aq, 2q) \bmod 5.$$

This is the solution anyone understanding the basics should be able to come up with.

Here is a more elegant line a of attack. Write

$$L_r = \{\, x \in \mathbf{2}^\star \mid \nu^R(x) = r \pmod{m} \,\}$$

for $r = 0, \ldots, 4$, so that L_0 is the language we want to recognize.

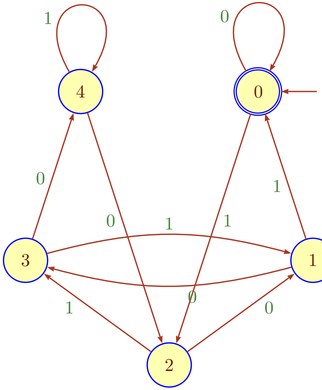
Can we perhaps compute its quotients?

$$\begin{aligned}
a^{-1}L_r &= \{\, x \in \mathbf{2}^\star \mid ax \in L_r \,\} \\
&= \{\, x \in \mathbf{2}^\star \mid \nu^R(ax) = r \pmod{5} \,\} \\
&= \{\, x \in \mathbf{2}^\star \mid a + 2\,\nu^R(x) = r \pmod{5} \,\} \\
&= \{\, x \in \mathbf{2}^\star \mid \nu^R(x) = 3r + 2a \pmod{5} \,\} \\
&= L_{3r+2a \bmod 5}
\end{aligned}$$

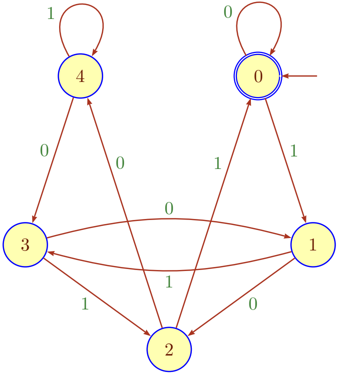
The Diagram

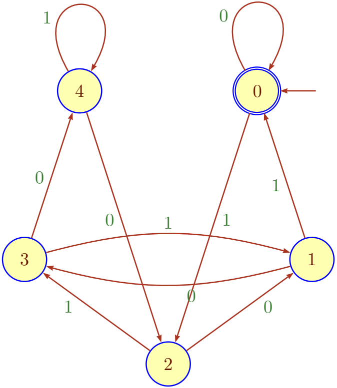
So we can actually construct the quotient automaton directly in this case.

$$\nu^R(xa) = 3 \cdot \nu^R(x) + 2 \cdot a \pmod 5$$



It is worth drawing the machines right next to each other.





Observations?

We could have done this much cheaper . . .

Here is a good test for implementations: determine the state complexity of of

$$L_k = \{\, uv \in \{a,b\}^\star \mid |u| = |v| = k, u \neq v \,\}.$$

Note that L_k is finite, so these languages are trivially recognizable.

Question: What is the state complexity of L_k?

k	1	2	3	4	5	6
sc	5	12	25	50	99	196

Exercise

Conjecture the state complexity of $\mathcal{L}_k$ from the last table and prove your conjecture.

- DFA minimization can be handled in time $n \log n$.
- Moore's quadratic time algorithm may be competitive on average.
- Equivalence testing of DFAs can be handled in nearly linear time without minimization.
- Quotients can be useful to describe machines for recognizable languages.