Minimizing Finite State Machines

1 Finite State Machines

1.1 The Basics

Recall that a *deterministic finite automaton (DFA)* is a deterministic and complete automaton. A DFA over alphabet Σ can be construed as a structure

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

where Q is the finite state set, $q_0 \in Q, F \subseteq Q$, and the transition function is given by

$$\delta : Q \times \Sigma \longrightarrow Q$$

δ extends naturally to words:

$$\begin{aligned} &\delta^* : Q \times \Sigma^* \longrightarrow Q \\ &\delta^*(p, \varepsilon) = p \\ &\delta^*(p, xa) = \delta(\delta^*(p, x), a) \end{aligned}$$

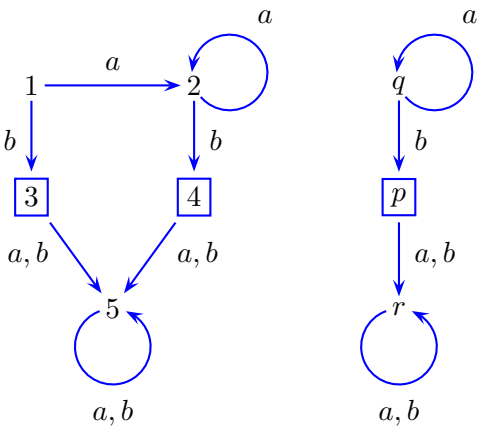
Here x is a word, and a a symbol over Σ. It is customary to write $\delta(p, x)$ instead of $\delta^*(p, x)$. Thus, a DFA accepts input x iff $\delta(q_0, x) \in F$.

Exercise 1.1 Show that for any word uv: $\delta(p, uv) = \delta(\delta(p, u), v)$.

Definition 1.1 *Two DFAs over the same alphabet are* equivalent *iff they accept the same language. Call a DFA* minimal *if it has the fewest possible states of all equivalent DFAs.*

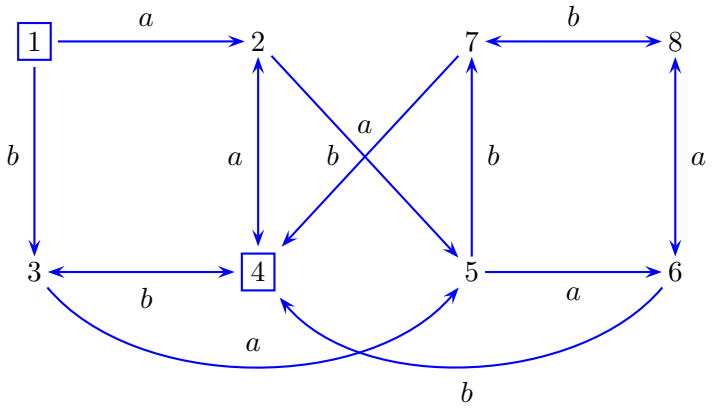
Note that according to our definition a minimal DFA need not be unique (unique up to isomorphism that is): there could be several DFAs that all realize the same minimal number of states. As it turns out, this cannat happen: up to isomorphism, there is exactly one minimal DFA for every regular language.

The following two DFAs are equivalent, both accept a^*b. The machine on the right is minimal.



DFAs lovingly designed by hand are often minimal. But finite state machine algorithms often produce non-minimal machines. A typical example is the conversion of a regular expression to a DFA. The DFA

below is produced by a standard conversion algorithm and accepts the language of all strings over $\{a, b\}$ containing an even number of a's and an even number of b's.



Exercise 1.2 Show that there is a DFA of size 4 for the even/even language, but none of size 3. Construct a regular expression for this language.

Define the languages

$$L_{a,r} = \{\, x \in \Sigma^* \mid x_r = a \,\} \subseteq \{a, b\}^*$$

For negative r, x_r is supposed to be the rth symbol from the end. E.g., $L_{a,-1}$ is the set of all strings ending in a.

Exercise 1.3 Construct a minimal automaton for $L_{a,k}$ for all $k \geq 1$. Construct a minimal automaton for $L_{a,-2}$ and $L_{a,-3}$.

1.2 Behavioral Equivalence

There are two obstructions to minimality of a DFA:

- The machine may contain inaccessible states.
- The machine may contain states that could be identified without changing the behavior of the machine.

The first obstruction is harmless, one can use graph exploration algorithms to remove all inaccessible states. To tackle the second problem, introduce the following definition.

Definition 1.2 *Two states p and q of a DFA M are* (behavior) equivalent, *in symbols $p \equiv q$, if*

$$\forall\, x \in \Sigma^* \big(\delta(p, x) \in F \iff \delta(q, x) \in F\big)$$

Think of an experiment conducted on a DFA:

- Put the DFA into an arbitrarily selected state p,
- feed an arbitrary input string to the machine, and
- observe whether the DFA accepts the string.

The experimenter is allowed to conduct any number of these tests, but can't pry the DFA open: the transition function is "secret". Two states of a DFA are equivalent if they cannot be distinguished by this type of experiment. Note that we can distinguish between final and non-final states with test strings of length 0.

1.3 State Merging

To obtain a minimal DFA we will merge all behaviorally equivalent states into a single state. The result will be a *reduced* automaton: all states will be pairwise inequivalent.

Technically, let $M = \langle Q, \Sigma, \delta, q_0, F\rangle$ an accessible DFA. Define a new DFA $M' = \langle Q', \Sigma, \delta', q_0', F'\rangle$ by

$$\begin{aligned} Q' &= \{\, [p] \mid p \in Q \,\} \\ \delta'([p], a) &= [\delta(p, a)] \\ q_0' &= [q_0] \\ F' &= \{\, [p] \mid p \in F \,\} \end{aligned}$$

Here $[p]$ denotes the equivalence class of p. Note that for δ' to be well-defined we need the following crucial property of equivalence:

$$p \equiv q \quad \rightarrow \quad \forall\, a \in \Sigma \left(\delta(p, a) \equiv \delta(q, a)\right)$$

The last property will be also be important in our state merging algorithm.

The following theorem establishes the uniqueness of the minimal DFA; we won't go into details here. The idea is to show that the map $p \mapsto [p]$ is a natural epimorphism from M to M'.

Theorem 1.1 *Let M_1 and M_2 be two equivalent accessible DFAs. Then the two corresponding state-merged automata M_1' and M_2' are isomorphic.*

1.4 Equivalence Relations and Partitions

It is convenient to identify an equivalence relation $\pi \subseteq Q \times Q$ with the *partition* induced by it:

$$\pi = (P_1, P_2, \ldots, P_r)$$

where the P_i are the equivalence classes of ρ; thus $Q = \bigcup P_i$, $P_i \neq \emptyset$, and $P_i \cap P_j = \emptyset$ of $i \neq j$. The P_i are often referred to as *blocks* in this context.

Definition 1.3 *Consider two quivalence relations ρ and σ on A. ρ* refines *σ if every class of ρ is contained in a class of σ. One also says that σ is* coarser *than ρ.*

In other words, $x\ \rho\ y \rightarrow x\ \sigma\ y$. Notation: $\rho \sqsubseteq \sigma$.

Meet and join operations on equivalence relations.

$\rho \sqcap \sigma$	logical and
$\rho \sqcup \sigma$	equ-closure of logical or

Note that the partition of the meet $\rho \sqcap \sigma$ can simply be obtained by computing the pairwise intersections of the blocks of ρ and σ (and discarding all empty intersections). The join is harder to compute: we have to form the transitive closure of the relation $\rho \vee \sigma$.

Exercise 1.4 Show that $\sqsubseteq$ is a partial order on the collection of equivalence relation on a set. $\sqcup$ and $\sqcap$ are the corresponding sup and inf operations.

For our purposes we need to consider the interaction between an equivalence relation π on A and a function $f : A \to A$. π is f-compatible if

$$\forall\, x, y \in A\,(x\ \pi\ y \to f(x)\ \pi\ f(y)).$$

Letting $x\ \pi(f)\ y \iff f(x)\ \pi\ f(y)$ we have: π is f-compatible iff $\pi \sqsubseteq \pi(f)$.

Lemma 1.1 *Let π be an equivalence relation on A and f a function on A. Then there is a uniquely determined coarsest equivalence relation ρ such that ρ refines π and is f-compatible.*

Proof. Consider the join

$$\rho = \bigsqcup \{\, \sigma \mid \sigma \sqsubseteq \pi, f - \text{compatible} \,\}$$

The set is not empty since it always contains the discrete relation. It is easy to see that the join of two f-compatible equivalence relations is again an f-compatible equivalence relation. Thus, ρ is as requrired. □

We will write $\mathsf{R}(\pi, f)$ for the coarsest equivalence relation refining π that is f-compatible.

2 The Standard Algorithm

In the application to minimization the one has to deal with several functions $\delta_a : Q \to Q$, but one can simply apply the algorithms for one after the other.

By definition,

$$p \equiv q \iff \forall\, x \in \Sigma^* \left(\delta(p, x) \in F \iff \delta(q, x) \in F\right)$$

It is easy to see that one only has to check words of length up to n^2. Thus, a brute force approach requires a search over some k^{n^2} words, $k = |\Sigma|$, and is highly inefficient.

Instead, we use a fixed point argument and approximate the equivalence relation in stages: the initial partition is given by $(Q - F, F)$. Then we construct the largest (in the sense of $\sqsubseteq$) fixed point under the operation $\pi \mapsto \pi \sqcap R(\pi)$ where

$$R(\pi) = \pi(\delta_{a_1}) \sqcap \pi(\delta_{a_2}) \sqcap \ldots \sqcap \pi(\delta_{a_k})$$

and $\Sigma = \{a_1, \ldots, a_k\}$. Thus, we compute

$$\begin{aligned} \pi_0 &= (Q - F, F) \\ \pi_{i+1} &= \pi_i \sqcap R(\pi_i) \end{aligned}$$

and stop as soon as $\pi_{r+1} = \pi_r$. Since $\pi_{i+1} \sqsubseteq \pi_i$ the process must terminate at $r < |Q|$.

Exercise 2.1 Show that the fixed point π_r is indeed the behavioral equivalence relation.

For the implementation, we need a representation of equivalence relations that makes it easy to compute meets $\rho \sqcap \sigma$ and the single refinement steps $\pi(f)$. Let's assume $Q = [n]$. If ρ is an equivalence relation on Q, we can represent it by the standard choice function

$$\begin{aligned} &R : Q \to Q \\ &R(p) = \min\big(\, q \in Q \mid p\ \rho\ q \,\big) \end{aligned}$$

Note that in this implementation we can test equivalence in $O(1)$ steps with small constants: $p\ \rho\ q$ iff $R(p) = R(q)$.

Exercise 2.2 Show that the functions obtained in this fashion are precisely the ones that are shrinking and idempotent: $R(p) \leq p$ and $R(R(p)) = R(p)$.

Suppose we have two relations σ and ρ, represented by S and R. To compute the representation T for $\tau = \sigma \sqcap \rho$ consider the table

1	2	3	...	p	...	n
$S(1)$	$S(2)$	$S(3)$	...	$R(p)$	...	$S(n)$
$R(1)$	$R(2)$	$R(3)$	...	$S(p)$	...	$R(n)$

Then $p \tau q$ iff $S(p) = S(q)$ and $R(p) = R(q)$. So, we only need to traverse the list, and look for new pairs (i, j) in the S/R rows. If a new pair appears, the corresponding p gets value $T(p) = p$, otherwise $T(p) = q$ where q is the least element with the same S/R pair.

```
//  meet( R, S )
for( p = 1; p <= n; p++ )
{
        i = S[p];
        j = R[p];
        if( (i,j) new  )
                T[p] = val(i,j) = p;
        else
                T[p] = val(i,j);
}
```

To implement `val` we can use a hash table, yielding expected performance $\Theta(n)$. Alternatively, we can use an auxiliary array `valarr[n][n]`, initialized to 0. Disregrading the initialization cost, T can be computed in $\Theta(n)$ steps. Note that we can reset `valarr` in linear time after T has been computed. Thus, we can perform m meet operations in time $O(n^2 + m \cdot n)$.

Given the `meet()` function, the whole state merging algorithm now looks like so.

```
initialize R to F and Q - F

cnt = 0;
while( cnt < k )
foreach  a in Sigma  do
{
        for( p = 1; p <= n; ++p )
                S[p] = R[ delta[p][a] ];

        RR = meet( R, S );
        if( RR == R )
                cnt++;
        else
        {  R = RR; cnt = 0; }
}
```

Example 2.1 Consider a DFA on 8 states with final states $\{1, 4\}$ and transition matrix

	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

If we collect the steps for symbols a and b into one step, we get the following trace.

```
     1  2  3  4  5  6  7  8
P0   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

P1   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

P2   1  2  3  1  5  3  2  5
```

Hence we have established the following result.

Theorem 2.1 *The state merged DFA M can be computed from M in time $O(k \cdot n^2)$ where n is the number of states of the machine, and k the size of the alphabet.*

Note that in practice the number of refinement steps is often bounded, so that we obtain a running time of the form $O(c \cdot k \cdot n)$. However, in general the quadratic bound is best possible.

Exercise 2.3 Build a family of DFAs that shows that in general the number of refinement steps is not bounded by a constant.

3 Hopcroft's Algorithm

We now discuss an $O(k \cdot n \log n)$ minimization algorithm due to Hopcroft. As before, the algorithm uses stepwise refinement of the original final-nonfinal partition. One step in the standard algorithm is of the form

$$\pi \mapsto \pi \sqcap \pi(f)$$

where $f = \delta_a$. By contrast, Hopcroft's algorithm focuses on a single block of $\pi(f)$ in a single step, and thus provides better control over the selection of the next refinement step.

Suppose π is a partition of Q, and consider a subset $B \subseteq Q$. We say that B *splits* π if for some block X of π:

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

Define the equivalence relation

$$\pi(f, B) = (f^{-1}(B), Q - f^{-1}(B))$$

Observe that

$$\pi(f) = \prod_{B \text{ in } \pi} \pi(f, B)$$

Note that a partition all of whose blocks are non-splitting must be f-compatible. Hopcroft's algorithm proceeds in steps

$$\pi \mapsto \pi \sqcap \pi(f, B)$$

where as before $f = \delta_a$ for some $a \in \Sigma$ and B is a block of π. More importantly, the algorithm exercises close control over which blocks are chosen.

activate all blocks
while there is an active block B
compute $C = f^{-1}(B)$
inactivate B
foreach block D split by B **do**
compute $D' = D \cap C$
compute $D'' = D - C$
remove D, add D' and D''
if D was active
then mark both D', D'' active
else mark smaller of D', D'' active

Note that unlike with the standard algorithm the stopping condition here does not guarantee that a fixed point has been reached, a correctness proof is definititely necessary here.

Theorem 3.1 *Hopcroft's algorithm correctly computes the coarsest partition refining π that is f-compatible in time $O(n \lg n)$ steps.*

Proof. Note that

$$\pi \sqcap \pi(f) \sqsubseteq \pi \sqcap \pi(f, B) \sqsubseteq \pi$$

so that $\mathsf{R}(\pi, f)$ is certainly a refinement of the relation computed by Hopcroft's algorithm. Hence, for correctness it suffices to show that the algorithm produces a relation that is f-compatible.

To this end we establish the following loop invariant: every inactive block X has a non-splitting extension with respect to the current partition of the form

$$X \cup X_1 \cup X_2 \cup \ldots \cup X_r$$

where all the blocks X_i are active. This assertion is clearly satisfied after initialization.

Now consider a round of the algorithm that starts with a partition π, splits according to block B, and produces π'. We have to show that every inactive π'-block X has the required property.

Note that we may assume that X is already an inactive π-block. Otherwise, either $X = B$ or X is one of Y' or Y'' where Y was an inactive π-block that was split in the last round. In the first case we are done by construction. In the second case, since exactly one of Y' and Y'' is active, we can use the other part in the union.

By IH we know that $X \cup \bigcup X_i$ is π-non-splitting for some choice of active π-blocks X_i.

Case 1: $X_i \neq B$ for all i.

Any block X_i that splits at all will split into two active π'-blocks and we can simply replace the one old block by two new blocks. Thus, we obtain an extension $X \cup \bigcup \widehat{X}_i$ that is π-non-splitting where all the $\widehat{X}_i$ are active π' blocks. But then $X \cup \bigcup \widehat{X}_i$ is also π'-non-splitting.

Case 2: Without loss of generality, $X_1 = B$.

Consider any π' block Z. The only way Z could be split by $\widetilde{X} = X \cup \bigcup_{i>1} \widehat{X}_i$ is that $Z \subseteq f^{-1}(\widetilde{X} \cup B)$, $Z \cap f^{-1}(B) \neq \emptyset$ and $Z \cap f^{-1}(\widetilde{X}) \neq \emptyset$. But then Z is split by B, and will have been broken up into pieces Z' and Z'' in the construction. Neither of these pieces is split by $\widetilde{X}$.

The running time analysis is a bit complicated; here is the main idea. One execution of the loop can be implemented in $|C| = \sum_{x \in B} \left| f^{-1}(x) \right|$ steps. It is possible to show that each element in A can be activated at most $\lg n + 1$ times (we always choose the smaller part of a split inactive block to become active). But then the total running time is at most $O(n \lg n)$.

□

To deal with a k-symbol alphabet one has can modify this algorithm by using k activity tokens (rather than just one in the case of a single function). Also note that one has to be careful to implement the loop properly so that the running time is really linear in the size of the splitting set $f^{-1}(B)$.