CS–172 Computability & Complexity, Fall 2004

Minimizing Finite Automata

Given a DFA M, can we find an equivalent DFA (i.e., one that recognizes the same language as M) with the minimum possible number of states? This is a very natural question, and has important applications to the efficiency of procedures that use finite automata. In this note, we will see an efficient algorithm that solves this problem.

To begin, we need a couple of definitions. Let M be any DFA with alphabet Σ. Then M naturally defines an equivalence relation $\sim_M$ over Σ^*, given by

$$x \sim_M y \text{ iff } M \text{ ends in the same state on inputs } x \text{ and } y.$$

Note that the number of equivalence classes is finite (being equal to the number of states of M).

Now let $L = L(M)$ be the language recognized by M. This language also defines a natural equivalence relation $\sim_L$, as follows. Call two strings $x, y \in \Sigma^*$ *indistinguishable* by L if, for all $z \in \Sigma^*$, $xz \in L \Leftrightarrow yz \in L$. Otherwise we say that x and y are *distinguishable*. Then the relation $\sim_L$ is defined by

$$x \sim_L y \text{ iff } x \text{ and } y \text{ are indistinguishable.}$$

Our first observation is that $\sim_M$ is a *refinement* of $\sim_L$; in other words, each equivalence class of $\sim_M$ is contained inside an equivalence class of $\sim_L$ (i.e., in other words, each equivalence class of $\sim_L$ consists of the union of some equivalence classes of $\sim_M$):

Proposition 1 *$\sim_M$ is a refinement of $\sim_L$.*

Proof: Suppose that $x \sim_M y$. Then, on inputs x and y, machine M ends up in the same state. But this means that, for any $z \in \Sigma^*$, on inputs xz and yz, M must also end up in the same state, and hence xz and yz are either both in L or both not in L. Thus x and y are indistinguishable, so $x \sim_L y$. □

Note that Proposition 1 implies that the number of equivalence classes of $\sim_L$ is at most the number of states of M, which is finite.

Now we are in a position to prove our first main result, which is a version of the so-called *Myhill-Nerode Theorem*:

Theorem 2 *The relation $\sim_L$ defines a DFA M' for L whose states correspond to the equivalence classes of $\sim_L$. Moreover, this is the unique minimum DFA for L (up to isomorphism).*

Proof: We define the DFA $M' = (Q', \Sigma, q_0', F', \delta')$ as follows. The set of states is $Q' = \{[x] : x \in \Sigma^*\}$, where $[x]$ denotes the equivalence class of x under the relation $\sim_L$. The initial state is $q_0' = [\epsilon]$ (the equivalence class of the empty string), and the accepting states are $F' = \{[x] : x \in L\}$ (the equivalence classes of all strings in L). Finally, the transition function is defined by

$$\delta'([x], a) = [xa].$$

We have to be careful here! We must check that δ' is well-defined, because if y is in the same equivalence class as x then $[x] = [y]$, and our definition would imply that $\delta'([x], a) = \delta'([y], a) = [ya]$. So we need to check that $[xa] = [ya]$, so that our definition is consistent.

But this follows since if x, y are in the same equivalence class then $xaz \in L \Leftrightarrow yaz \in L$ for any $a \in \Sigma$ and any $z \in \Sigma^*$, and hence xa and ya are also in the same equivalence class, i.e., $[xa] = [ya]$. So our definition is OK.

Now it is obvious that the machine M' accepts L, since by our definition of δ', on input x it will end up in state $[x]$, and this state is accepting if and only if $x \in L$.

Finally, to see that M' has the minimum possible number of states, let M be any other machine that accepts L. By Proposition 1, $\sim_M$ is a refinement of $\sim_L$, so $\sim_M$ has at least as many equivalence classes as $\sim_L$. Hence M has at least as many states as M'. And if M has the same number of states, then the equivalence relations $\sim_M$ and $\sim_L$ are in fact identical. □

The Myhill-Nerode Theorem suggests an approach to finding the minimal DFA for a given language L. Suppose we are given a DFA M for L. By the Myhill-Nerode Theorem, we can think of each state of the minimal automaton, M', as an equivalence class of $\sim_L$, which in turn, by Proposition 1, is a collection of equivalence classes of $\sim_M$, i.e., a collection of states of M. Thus, to construct M' from M, we need to *merge together* states of M until no further merging is possible.

When should two states of M be merged? Well, p and q should be merged iff their equivalence classes under $\sim_M$ belong to the *same* equivalence class under $\sim_L$. Let x, y be in the equivalence classes p, q respectively of $\sim_M$; i.e., on inputs x, y, M ends up in states p, q respectively. But $x \sim_L y$ iff, for all z, the strings xz, yz are either both in L or both not in L. Hence we should merge p and q iff, for all strings z, the computations of M on z starting in states p and q either both lead to an accepting state or both to a non-accepting state. We will call states p and q *indistinguishable* in this case. Otherwise we say that p and q are *distinguishable*; note that this means there exists a string z which takes M from state p to an accepting state and from state q to a non-accepting state (or vice versa).

We are now ready to present our algorithm for minimizing finite automata. The input is a DFA $M = (Q, \Sigma, q_0, F, \delta)$; the output is a minimal DFA M' that accepts the same language as M. Rather than merging indistinguishable states of M', the algorithm instead proceeds by first assuming that all states of M' are indistinguishable, and successively identifying pairs of states that are distinguishable.

The algorithm maintains a table of all unordered pairs of states of M. Each entry is binary: it is either "marked" or "unmarked." The algorithm also maintains, for each pair, a list of other pairs; the role of these lists will become clear in a moment. Initially all table entries are unmarked, and all lists are empty. The algorithm begins by marking all pairs $\{p, q\}$ such that $p \in F$ and $q \in Q - F$; clearly all these pairs are distinguishable (by the empty string). It then cycles once through all other pairs of states (in any order). For each such pair $\{p, q\}$, it considers the transitions on each letter $a \in \Sigma$. If the pair $\{\delta(p, a), \delta(q, a)\}$ is marked, then p and q are distinguishable (by the string az, where z is a string that distinguishes $\delta(p, a)$ and $\delta(q, a)$), so the algorithm marks $\{p, q\}$; if not, then the algorithm places the pair $\{p, q\}$ on a temporary list associated with the pair $\{\delta(p, a), \delta(q, a)\}$ — this means that, if this latter pair ever gets marked, then $\{p, q\}$ will be marked also. This is ensured because, whenever the algorithm marks a pair, it proceeds to mark all pairs on the associated list (and, recursively, all pairs on their lists, etc.)

Here is the full algorithm:

for $\{p, q\}$ with $p \in F$, $q \in Q \setminus F$ **do** mark $\{p, q\}$
for all other pairs $\{p, q\}$ **do**
    if $\exists\, a \in \Sigma$ such that $\{\delta(p, a), \delta(q, a)\}$ is marked **then**
        mark $\{p, q\}$
        recursively mark all pairs on the list for $\{p, q\}$ etc.
    else for each $a \in \Sigma$ **do**
        if $\delta(p, a) \neq \delta(q, a)$ **then** put $\{p, q\}$ on the list for $\{\delta(p, a), \delta(q, a)\}$

We need to check carefully that this algorithm really finds all the equivalence classes of $\sim_L$, i.e., that it identifies all pairs of distinguishable states (and only those pairs). One direction is fairly obvious: it should be clear that the algorithm only marks $\{p, q\}$ if indeed p and q are distinguishable. [Check this; to do it formally, you need an induction on the time at which the pair is marked.]

The other direction is a bit less obvious. We need to check that, if p and q are distinguishable, then they will get marked. To see this, we use induction on the minimum length of a string that distinguishes p and q, i.e., a string z that takes M from p to an accepting state, and from q to a non-accepting state. For the base case, suppose the minimum length of such a string is 0, i.e., $z = \epsilon$, the empty string. But this means that p itself is accepting and q is non-accepting, so the pair $\{p, q\}$ is marked in the initialization phase of the algorithm. For the induction step, let $z = az'$ be a minimum length distinguishing string of length at least 1. Then this means that the states $r = \delta(p, a)$ and $s = \delta(q, a)$ are distinguished by the shorter string z'. Hence, by the induction hypothesis, the pair $\{r, s\}$ will eventually get marked by the algorithm. If this marking occurs before $\{p, q\}$ is considered, then $\{p, q\}$ will get marked when it is considered; otherwise, $\{p, q\}$ will go on the list for $\{r, s\}$ and will get marked (at the latest) when the pair $\{r, s\}$ is marked. Hence, by induction, all distinguishable pairs eventually get marked.

Finally we analyze the running time of the algorithm. Let the number of states of M be n, and the number of symbols in alphabet Σ be k. The initialization phase clearly takes time $O(n^2)$. The main loop is executed $O(n^2)$ times (once for each pair), and the running time of each iteration, *excluding* the time for the recursive marking, is $O(k)$ (proportional to the number of symbols a to be checked), for a total time of $O(kn^2)$. The time for the recursive marking is clearly proportional to the sum of the lengths of all the lists; but, since each pair $\{p, q\}$ can be placed on at most k lists (one for each a), the sum of the lengths of all the lists is at most $O(kn^2)$. Putting all this together gives a total running time of $O(kn^2)$.