Engineering a Compiler

Manuscript for the Third Edition (EaC 3e)

Keith D. Cooper
Linda Torczon

Rice University
Houston, Texas

2.4.4 DFA to Minimal DFA: Hopcroft's Algorithm

As the final step in the RE→DFA construction, we can employ an algorithm to minimize the number of states in the DFA. The DFA that emerges from the subset construction can have a large set of states. While the size of the DFA does not affect is asymptotic complexity, it does determine the recognizer's footprint in memory. On modern computers, the speed of memory accesses often governs the speed of computation. A smaller recognizer may fit better into the processor's lowest level of cache memory, producing faster average accesses.

To minimize the number of states in a DFA, $(D, \Sigma, \delta, d_0, D_A)$, we need a technique to detect when two states are *equivalent*—that is, they produce the same behavior on any input string. The particular algorithm covered in this section constructs a minimal DFA from an arbitrary DFA by grouping together sets of equivalent states. The algorithm finds the largest possible sets of behaviorally-equivalent states; each set becomes a state in the minimal DFA.

The algorithm constructs a *set partition*, $P = \{ p_1, p_2, p_3, \ldots, p_m \}$, of the DFA states. The partition constructed by the minimization algorithm groups together DFA states that have equivalent behavior. More formally, it constructs a partition with the smallest number of sets, subject to the following two rules:

Set partition: A *partition* of S is a collection of disjoint, nonempty subsets of S whose union is exactly S.

(1) $\forall\, c \in \Sigma$, if $d_i, d_j \in p_s$; $d_i \xrightarrow{c} d_x$; $d_j \xrightarrow{c} d_y$; and $d_x \in p_t$;, then $d_y \in p_t$.

(2) If $d_i, d_j \in p_i$ and $d_i \in D_A$, then $d_j \in D_A$

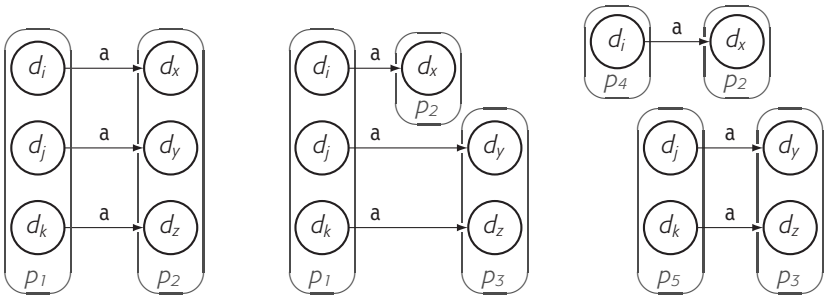
The first rule states that two states in the same set must, for every character $c \in \Sigma$, transition to states that are, themselves, grouped together in a set in the partition. The second rule states that any single set contains either accepting states or nonaccepting states, but not both.

These two properties not only constrain the final partition, P, but they also lead to a construction for P. The algorithm, often called *Hopcroft's algorithm*, starts with the coarsest partition on behavior, $P_0 = \{ D_A, \{ D - D_A \} \}$. It then iteratively "refines" the partition until both properties hold true for each set in P. To refine the partition, the algorithm splits sets based on the transitions out of DFA states in the set.

P_0 divides D into accepting and non-accepting states, a fundamental difference in behavior specified by rule 2.

Figure 2.8 shows how the algorithm uses transitions to split sets in the partition. In Figure 2.8.a, all three DFA states in set p_1 have transitions to DFA states in p_2 on the input character a. Specifically, $d_i \xrightarrow{a} d_x$, $d_j \xrightarrow{a} d_y$, and $d_k \xrightarrow{a} d_z$. Since $d_i, d_j, d_k \in p_1$, and $d_x, d_y, d_z \in p_2$, sets p_1 and p_2 conform to rule 1. Thus, p_1 and p_2 are behaviorally equivalent, and the algorithm would not split them.

In contrast, Figure 2.8.b shows a situation where the input a induces a split in set p_1. As before, $d_i \xrightarrow{a} d_x$, $d_j \xrightarrow{a} d_y$, and $d_k \xrightarrow{a} d_z$, but $d_x \in p_2$ while $d_y, d_z \in p_3$. This situation violates rule 1, so the algorithm would



(a) a does not split p_1 (b) a splits p_1 (c) Sets from (b) after the split

FIGURE 2.8 Splitting a Set around a

split p_1 into two sets, $p_4 = \{ d_i \}$ and $p_5 = \{ d_j, d_k \}$, as shown in Figure 2.8.c.

The algorithm in Figure 2.9 uses these two rules and the notion of splitting sets to construct a partition that maps into the minimal DFA. At each stage, the algorithm holds the current approximation in *Partition*. It constructs the next approximation into *NextP*, which allows it to update *NextP* without interfering in the current iteration.

As its first approximation, the algorithm constructs the coarsest partition consistent with rule 2, $\{ D_A, \{ D - D_A \}\}$. This choice has two consequences. First, since each set in the final partition is constructed by splitting a set in an earlier approximation, it ensures that no set in the final partition will contain both accepting and nonaccepting states. Second, by choosing the largest sets consistent with rule 2, it imposes the minimum constraints on the splitting process which, in turn, can lead to larger sets in the final partition. (Larger sets means fewer states in the corresponding DFA.)

Starting with the largest possible sets and splitting them is an optimistic approach to building the sets; see, for example, Section 9.3.6 or [348].

The algorithm operates from a worklist of states, starting with the initial partition $\{ D_A, \{ D - D_A \}\}$. It repeatedly picks a set s from the worklist and uses that set to refine the partition in *NextP* by splitting sets based on their transitions into s.

To identify states that must split because of a transition into set s on some character c, the algorithm inverts the transition function. It computes the set of DFA states that can reach a state in set s on a transition labelled c and assigns that set to *Image*. It then systematically examines each set $q \in P$ to see if *Image* divides q. If *Image* divides q into non-empty sets q_1 and q_2, it removes q from both *Partition* and *NextP* and then adds both q_1 and q_2 to *NextP*.

All that remains, in processing q with respect to c, is to update the worklist. If q is on the worklist, then the algorithm replaces q with

```
Partition  ← {D_A, {D − D_A}};
NextP      ← {D_A, {D − D_A}};
Worklist   ← {D_A, {D − D_A}};
while( Worklist ≠ ∅ )
    select a set s from Worklist and remove it
    for each character c ∈ Σ
        Image ← {x | δ(x,c) ∈ s}
        for each set q ∈ Partition
            q1 ← q ∩ Image
            q2 ← q − q1
            if q1 ≠ ∅ and q2 ≠ ∅ then    // split q around s and c
                remove q from Partition
                remove q from NextP
                NextP ← NextP ∪ q1 ∪ q2
                if q ∈ Worklist then        // and update the Worklist
                    remove q from Worklist
                    WorkList ← WorkList ∪ q1 ∪ q2
                else if |q1| ≤ |q2|
                    then WorkList ← Worklist ∪ q1
                    else WorkList ← WorkList ∪ q2
                if s = q                    // need another s
                    then break
    Partition ← copy of NextP               // set up for the next iteration
```

FIGURE 2.9 DFA Minimization Algorithm

both q_1 and q_2. The rationale is simple: q was on the worklist for some potential effect; that effect might be from some character other than c, so all of the DFA states in q need to be represented on the worklist.

If, on the other hand, q is not on the worklist, then the only effect that splitting q can have on other sets is to split them. Assume that some set r has transitions on letter e into q. Dividing q might create the need to split r into sets that transition to q_1 and q_2. In this case, either of q_1 or q_2 will induce the split, so the algorithm can choose between them. Using the smaller set will lead to faster execution; for example, computing *Image* takes time proportional to the size of the set.

To construct the new DFA from the final partition P, we can create a single state to represent each set $p_i \in P$ and add the appropriate transitions between these new representative states. For the state representing p_m, we add a transition to the state representing p_n on character c if some $d_j \in p_m$ has a transition on c to some $d_k \in p_n$. The construction

ensures that, if $d_j \xrightarrow{c} d_k$, where $d_j \in p_m$ and $d_k \in p_n$, then every state in p_m has a similar transition on c to a state in p_n. If this condition did not hold, the algorithm would have split p_m around the transitions on c. The resulting DFA is minimal; the proof is beyond our scope.

Examples

As a first example, consider the DFA in Figure 2.10a. It recognizes the language *fee* | *fie*. Figure 2.10.b shows the progress of the minimization algorithm on this DFA.

The first line shows the initial configuration of the algorithm, with *Partition*, *NextP*, and *Worklist* all set to contain $\{ D_A, \{D - D_A\} \}$, which is $\{\{s_3, s_5\}, \{s_0, s_1, s_2, s_4\} \}$. The algorithm enters the while loop and selects a first set for s; it chooses p_0. Next, it iterates over the characters in the alphabet, in order, *f*, *e*, *i*. With *f*, p_0 does not split either p_0 or p_1. With *e*, p_0 splits p_1 into two sets: p_2: $\{s_0, s_1\}$ and p_3: $\{s_2, s_4\}$. The algorithm removes p_1 from *Partition* and *NextP*. It adds p_2 and p_3 to *NextP*. Then, it updates the worklist, removing p_1 and adding p_2 and p_3, before advancing to the final character, *i*.

The final iteration of the character loop generates no further splits. The only set it considers for splitting is $q = p_0$ because the algorithm removed p_1 from *Partition* when it was split. Note that the new sets were added to *NextP* rather than to *Partition*. After processing $s = p_0$, $c = i$, and $q = p_0$, the code updates *Partition* from *NextP* and proceeds into the second iteration of the while loop.

It makes no sense to split the other sets based on a set that no longer exists.

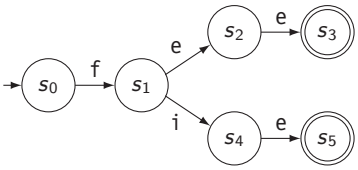
The second iteration proceeds in a similar fashion, until it considers $q = p_2$, $c = f$, and $q = p_2$. This combination splits p_2. Since p_2 is both the set being split, q, and the set that induces the split, s, the algorithm breaks out of the rest of the second iteration.

When we draw the minimal DFA, we combine the edges for *e* and *i* and label the edge with both letters. The algorithm might be viewed as creating separate edges for *e* and *i* that have the same behavior.

The third, fourth, and fifth iteration of the while loop systematically go through the worklist and the alphabet. They discover no further splits, so the algorithm halts when the worklist is empty after the fifth iteration of the outer loop. The resulting DFA is shown in Figure 2.10.c. It has four states.

As a second example, consider the DFA for $a\,(b \mid c)^*$ produced by Thompson's construction and the subset construction, shown in Figure 2.11a. The first step of the minimization algorithm constructs an initial approximation to the partition as $\{ \{s_1, s_2, s_3\}, \{s_0\} \}$. The algorithm selects p_0 as q and tries to split p_0 based on each of *a*, *b*, and *c*. While p_1 has a transition into p_0 on *a*, it causes no split. (First, every DFA state in p_1 has the same transition on *a*. Second, the algorithm cannot split a singleton state.) For *b* and *c*, all of the transitions into p_0 originate inside p_0 and neither *b* nor *c* split p_0.

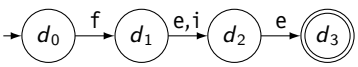
When it considers splitting on p_1, the algorithm discovers that there are no transitions into p_1—that is *Image* is empty. Thus, p_1



(a) DFA for "fee | fie

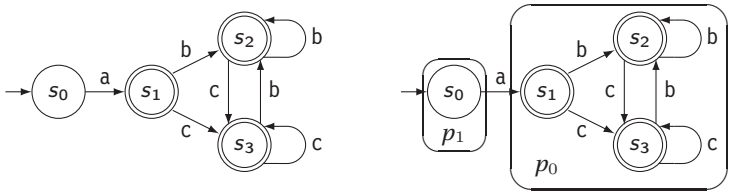
Step	Partition	Worklist	s	c	q	Action
–	$\{ p_0: \{s_3, s_5\}, p_1: \{s_0, s_1, s_2, s_4\} \}$	$\{ p_0, p_1 \}$	—	—	—	—
1	$\{ p_0: \{s_3, s_5\}, p_1: \{s_0, s_1, s_2, s_4\} \}$	$\{ p_1 \}$	p_0	*f*	*all*	*none*
		$\{ p_1 \}$	p_0	*e*	p_1	*split* p_1
		$\{ p_2, p_3 \}$	p_0	*i*	p_0	*none*
2	$\{ p_0: \{s_3, s_5\}, p_2: \{s_0, s_1\}, p_3: \{s_2, s_4\} \}$	$\{ p_3 \}$	p_2	*f*	p_0	*none*
		$\{ p_3 \}$	p_2	*e*	p_2	*split* p_2
3	$\{ p_0: \{s_3, s_5\}, p_3: \{s_2, s_4\}\ p_4: \{s_0\}, p_5: \{s_1\} \}$	$\{ p_4, p_5 \}$	p_3	*f*	*all*	*none*
		$\{ p_4, p_5 \}$	p_3	*e*	*all*	*none*
		$\{ p_4, p_5 \}$	p_3	*i*	*all*	*none*
4	$\{ p_0: \{s_3, s_5\}, p_3: \{s_2, s_4\}\ p_4: \{s_0\}, p_5: \{s_1\} \}$	$\{ p_5 \}$	p_4	*f*	*all*	*none*
		$\{ p_5 \}$	p_4	*e*	*all*	*none*
		$\{ p_5 \}$	p_4	*i*	*all*	*none*
5	$\{ p_0: \{s_3, s_5\}, p_3: \{s_2, s_4\}\ p_4: \{s_0\}, p_5: \{s_1\} \}$	$\{\ \}$	p_5	*f*	*all*	*none*
		$\{\ \}$	p_5	*e*	*all*	*none*
		$\{\ \}$	p_5	*i*	*all*	*none*
final	$\{ p_0: \{s_3, s_5\}, p_3: \{s_2, s_4\}\ p_4: \{s_0\}, p_5: \{s_1\} \}$	$\{\ \}$	*algorithm halts*			

(b) Critical Steps in Minimizing the DFA



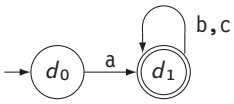
(c) The Minimal DFA

FIGURE 2.10 Applying the DFA Minimization Algorithm



(a) Original DFA (b) Initial Partition

FIGURE 2.11 DFA for $a\,(b \mid c^*)$



induces no splits on any of *a*, *b*, or *c*, and the two set partition is the final partition. The final DFA has two states, as shown in the margin.Recall that this is the DFA that we suggested a human would derive. After minimization, the automatic techniques produce the same result.

Hopcroft's DFA minimization algorithm is another example of a fixed-point computation. *Partition* is finite; at most it can contain $|D|$ elements. The while loop splits sets in *Partition*, but never combines them. Thus, *Partition* grows monotonically. The loop halts when some iteration performs no splits. The worst-case behavior occurs when each state in the DFA has distinct behavior; in that case, the while loop halts when *Partition* consists of a singleton set for each d_i in D. This worst case arises when the input DFA is already a minimal DFA.