0.1 Minimizing Finite State Machines

Here we discuss the problem of minimizing the number of states of a DFA. We will show that for any regular language L, there is a *unique* DFA that recognizes L and minimizes the number of states it contains. Further, that DFA can be found by an efficient algorithm, given any DFA M that recognizes L. We will use the DFA shown in Figure 1 to illustrate the ideas and the algorithm. First, it is obvious that if DFA M has a state q that cannot be reached from the start state of M, then q can be removed and the resulting DFA recognizes exactly the same language as M does. Removing such states is the first step of minimization algorithm. As a side consequence, after such states have been removed, the resulting DFA is connected. In the example, state d should be removed, and we assume it has been.

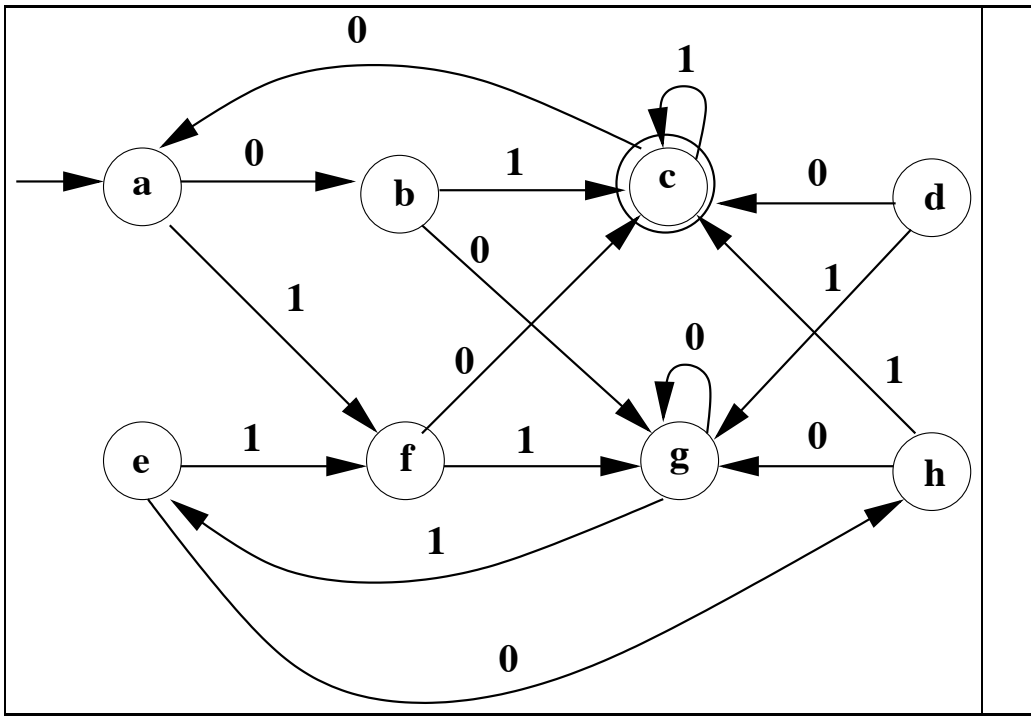

Figure 1: The DFA we will use for the running example. (Taken from Intro. to Automata Theory, Languages and Computation; by J. Hopcroft and J. Ullman, p. 68.)

Definition Two states q and q' in a DFA M are *distinguishable* if there is some string $w \in \Sigma^*$ where the computation on w from q leads to a final state, wlog, while the computation on w from q' leads to a non-final state.

For example, states a and b are distinguishable, using the $w = 01$, since final state c is reached from a on string 01, while non-final state e is reached from b

on 01.

Definition Two states q and q' in a DFA M are called *indistinguishable* if for every $w \in \Sigma^*$ the computation on w from q leads to a final state in M if and only if the computation on w from q' leads to a final state in M.

Note that we have only *defined* what it means for two states to be indistinguishable; we have ignored the question of how we can determine if two states are indistinguishable or distinguishable. That will come later.

Establishing that two states are indistinguishable is harder than establishing that two states are distinguishable, since it only takes one string to demonstrate distinguishability. However, in the example, we will see that states a and e are indistinguishable. As an exercise, you should try a few strings from states a and e and see that none of those strings show that states a and e are distinguishable.

The key observation at this point is that if q and q' are indistinguishable, then we can modify M as follows: for every edge into q', redirect it into q; then remove q' and all of the edges out of it, from M. The new DFA, called M_1, now has one fewer states than M, but we claim that the language that it recognizes is the same as before. That is $L(M_1) = L(M)$. This is because any computation in M that reached q' and continued with a string w and ended in a final state of M, will now reach q and continue with w and end in the same final state of M_1. Conversely any computation that reached q' and continued with a string w and ended in a non-final state of M, will now reach q and continue with w and end in a non-final state of M_1. Further, computations in M that didn't reach q' will be unchanged in M_1, so $L(M_1) = L(M)$.

Note that the choice of which of the two states is called q and which is q' is arbitrary. Note also, that for any state $q'' \neq q'$ in M, the set of strings that lead from q'' to a final state in M is exactly the same as the set of strings that lead from q'' to a final state in M_1. As an exercise, you should do the modification on M using states a and e, and then try out several strings to see that $L(M_1) = L(M)$.

Hence, two states in M_1 will be indistinguishable in M_1 if and only if they were indistinguishable in M. This will be an important point for efficiency of the algorithm to come.

Now if there are two states in M_1 that are indistinguishable in M_1 (equivalently, in M), then we can again redirect edges and remove a state, as above. The modified DFA, call it M_2, will again recognize exactly the same language as M does. Further, two states will be indistinguishable in M_2 if and only if they are indistinguishable in M. Suppose we continue this until there are no further indistinguishable pairs of states in the resulting DFA. At that point, call the DFA $\vec{M}$ and note that $L(\vec{M}) = L(M)$.

Definition The procedure just described, creating $\vec{M}$ from M, will be called the *reduction procedure*, and we say that a DFA is *reduced* if it does not contain

any indistinguishable pair of states.

Theorem 0.1.1 *Let R and R' be two reduced DFAs such that $L(R) = L(R')$. Then R and R' are identical except for the labels used for the states. More formally, R and R' are isomorphic as directed graphs.*

Proof We can extend the notion of indistinguishably of a pair of states, q, q', to the case where q is in R and q' is in R': q and q' are indistinguishable, if for every $w \in \Sigma^*$, the computation from q leads to a final state in R if and only if the computation from q' leads to a final state in R'.

We will describe a search through R to define a one-one correspondence between indistinguishable states in R and R'. That correspondence will also be shown to define an isomorphism between R and R', proving the theorem. To start, note that the two start states, q_0 and q'_0, of R and R' respectively, are indistinguishable because $L(R) = L(R')$. So the two start states are mapped to each other in the correspondence.

Next, consider a character $c \in \Sigma$, and let q_1 and q'_1 denote the states in R and R' that are entered by a transition on c from states q_0 and q'_0 respectively. States q_1 and q'_1 must be indistinguishable, or else there is a string w such that the computation in R from q_1 on w leads to a final state, wlog, and the computation in R' from q'_1 on w leads to a non-final state. But then the computation on cw from q_0 leads to a final state in R, and the computation on cw from q'_0 leads to a non-final state in R', contradicting the fact that q_0 and q'_0 are indistinguishable. So q_1 and q'_1 are indistinguishable. Now suppose there is another character $d \in \Sigma$ such that the transition in R from q_0 on d is to state q_1 in R. Then the transition in R' from q'_0 on d must be to q'_1. To see this, suppose instead that the transition from q'_1 on d is to a state $q' \neq q'_1$. But, by the argument above, states q_1 in R and q' in R' would be indistinguishable, so by transitivity, states q'_1 and q', both in R' would be indistinguishable, contradicting the assumption that R' is a reduced DFA. It follows then that every character in Σ that causes a transition from q_0 to q_1 in R, also causes a transition from q'_0 to q'_1 in R'. A symmetric argument shows that every character in Σ that causes a transition from q'_0 to q'_1 in R' causes a transition from q_0 to q_1 in R. Thus, the desired one-one correspondence between states of R and R' contains the correspondence (q_1, q'_1). Further, the set of labeled edges from q_0 and q_1 in R is identical to the set of labeled edges from q'_0 to q'_1.

Continuing as above, say with a DFS through R, we obtain the claimed one-one correspondence of states in R and R' and an isomorphism between R and R', showing that R and R' have the same number of states, and are isomorphic as directed graphs. ■

Clearly, if a DFA M has the minimum number of states of any DFA that recognizes $L(M)$, then M must be reduced. It follows that what we have really

shown in the proof of Theorem 0.1.1 is that the DFA that minimizes the number of states and recognizes $L(M)$, is unique up to relabeling of the states, and can be found from any DFA that recognizes $L(M)$ by the reduction procedure. Summarizing, we have

Corollary 0.1.1 *Let M be a DFA and $\vec{M}$ be the reduced DFA resulting from running the reduction procedure. Then $\vec{M}$ has the minimum number of states of any DFA that recognizes $L(M)$, Further, $\vec{M}$ is the unique DFA, up to isomorphism, that has the minimum number of states and recognizes $L(M)$.*

0.1.1 Efficiently implementing the reduction procedure

Now we discuss how to efficiently implement the reduction procedure. Recall that as the procedure proceeds, two states q and q' in the current DFA, say M_i after i iterations, will be indistinguishable in M_i if and only if they are indistinguishable in M. Moreover, if q is indistinguishable from q', and q' is indistinguishable from q'' in M, then q is indistinguishable from q'' in M. Hence the states of M partition into classes where in each class, every pair of states is indistinguishable, and any pair of states from two different classes are distinguishable. If we knew this partition, call it $\vec{\Pi}$, then we could form the reduced DFA $\vec{M}$ in $O(n)$ time (details left to the reader, along with the answer to the question of why we need such detail, or similar details, in order to achieve a linear time bound). Hence the remaining problem is how to find partition $\vec{\Pi}$. This can be done in $O(n \log n)$ time with the *successive refinement algorithm* due to J. Hopcroft in 1971. We will only detail an $O(n^2)$ time version of it, but may later sketch the main ideas of how to improve it to get $O(n \log n)$ time.

We will show that at any point in the successive refinement algorithm, the states of M are partitioned into classes, with the **invariant** that every state q is distinguishable from every state not in the same class as q. Note that this invariant does *not* say that every pair of states in the same class are indistinguishable, and in fact, that will not be true until the algorithm terminates.

Let Π denote the current partition of states during the execution of the successive refinement algorithm. As the algorithm proceeds, partition Π becomes refined; that is, one or more classes in Π gets subdivided, and classes are never merged or expanded. We will see that when the algorithm terminates, the current Π will be the desired $\vec{\Pi}$. As the algorithm proceeds, it also keeps a list of classes, L, of *potential splitters*. The algorithm terminates when L becomes empty.

To start, the initial partition Π consists of a class, F, containing the final states, and a class, N, containing the non-final states. Note that any final state is distinguishable from any non-final state, by the string $w = \epsilon$, i.e., the empty string. So the initial partition Π satisfies the claimed invariant. Initially, L

consists of sets F and N. In the example, class F consists of $\{c\}$ and N consists of $\{a, b, e, f, g, h\}$.

The **general step** of the successive refinement algorithm is:

1) Remove one class, call it S, from L, and consider every character in Σ to be unmarked.

2) Pick an unmarked character c in Σ, and mark it.

3) Examine each class C in the current Π and find each state q in C such that $\delta(q, c)$ is a state in S. Let $C' \subseteq C$ be the set of such states. If C' is a non-trivial subset of C, then change Π by subdividing C into classes C' and $C - C'$; and put classes C' and $C - C'$ into L.

4) Go to Step 2) if there are any unmarked characters in Σ.

The successive refinement algorithm continues executing the general step until L is empty. We will show that at termination, partition Π is $\vec{\Pi}$.

As an illustration, we show the details of the successive refinement algorithm on the running example. In the first general step, suppose the class $F = \{c\}$ is removed from L and assigned to S, and suppose the first character picked and marked is 0. The algorithm then examines each class C in Π. The first class is F, but since it only has one element, it can't subdivide further, so it then examines class N. It goes through the states in N and sees that for character 0, state f goes to state $c \in F$ (i.e., $\delta(f, 0) = c \in F$), but none of the other states of N do. So class N subdivides into classes $\{f\}$ and $\{a, b, e, g, h\}$. We will call those two classes C_3 and C_4. So at this point $\Pi = \{F, C_3, C_4\}$; i.e. $\Pi = \{\{c\}, \{f\}, \{a, b, e, g, h\}\}$. After the two new classes are put into L, L is the list (N, C_3, C_4).

The algorithm is not finished with the first general step, because character 1 is still unmarked. Classes F and C_3 each contain only a single element, so they won't subdivide further, but in class C_4, states b and h go to a state in class N (i.e., state c) on character 1, while the other states in C_3 do not. Hence C_3 subdivides into classes $C_5 = \{b, h\}$ and $C_6 = \{a, e, g\}$. So, the updated $\Pi = \{F, C_3, C_5, C_6\} = \{\{c\}, \{f\}, \{b, h\}, \{a, e, g\}\}$. The updated $L = (N, C_3, C_4, C_5, C_6)$. Only at this point is the processing of the first class S removed from L complete, ending the first general step.

The second general step removes class N from L, making it S. What happens when the chosen character is 0? Class F can't subdivide, so it is skipped. Class $C_5 = \{b, h\}$ does not subdivide on 0, because both $\delta(b, 0)$ and $\delta(h, 0)$ equal state c, which is not in S, so $C' = \emptyset$ in this case. Similarly, class $C_6 = \{a, e, g\}$ does not subdivide on 0, because $\delta(a, 0) = b \in S$, $\delta(e, 0) = h \in S$ and $\delta(g, 0) = g \in S$, so $C' = C_6$. As an exercise, verify that no class in Π subdivides when the marked character is 1. So, after the second general step, Π is unchanged, but the updated $L = (C_3, C_4, C_5, C_6)$.

The third general step removes $C_3 = \{f\}$ from L, making it S. When the picked character is 0, no class subdivides, but when the marked character is 1, class C_6 subdivides into classes $C_7 = \{a, e\}$ and $C_8 = \{g\}$. As an exercise, verify this. So after the third general step $\Pi = \{F, C_3, C_5, C_7, C_8\}$ and $L = (C_4, C_5, C_6, C_7, C_8)$.

In each of the next five general steps, Π does not change, so when L is empty, $\Pi = \{F, C_3, C_5, C_7, C_8\} = \vec{\Pi}$, and the successive refinement procedure is complete on this example.

After $\vec{\Pi}$ has been found, we use it in the reduction procedure on the DFA M in the running example. The reduced DFA $\vec{M}$ is shown in Figure 2. $\vec{M}$ is the unique DFA that recognizes language $L(M)$ and minimizes the number of states.

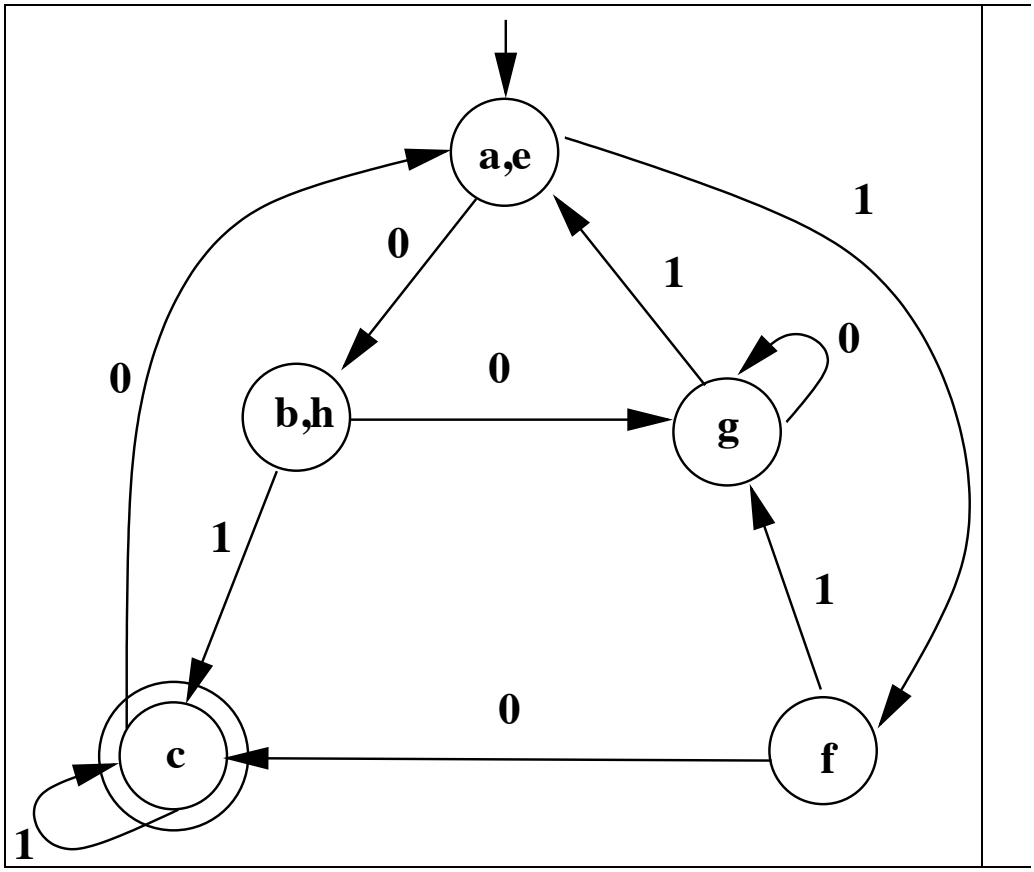

Figure 2: The unique minimumn state DFA $\vec{M}$ that recognizes $L(M)$, where M is shown in Figure 1. Each state shows the set of states of M merged into a single state in $\vec{M}$.

Implementation details We now discuss some implementation details for the successive refinement algorithm. When Π is implemented as a set of linked-lists, one for each class in Π, each general step needs only $O(n)$ time. In more detail, we assume that the states of M are numbered consecutively, so that we can use the state number as an index into an array Q. When we remove S from L, we use S to find and mark all the states in Q that are in S. This takes $O(|S|) = O(n)$ time. Having the marked Q will allow us to check if a given state q is in S, in constant time. Then, for a character $c \in \Sigma$, and a class C in Π, we create a pointer p to the start of the list for C, and traverse the linked-list for C, examining each state $q \in C$. We use the transition table to determine the state $\delta(q, c)$ and check, in constant time using Q, if that state is in S. If it is, then we move it to the position in the linked-list for C that is pointed to by p, and increment p by one position. When the scan of C is complete, C' consists of the states in C from position 1 to $p - 1$, and $C - C'$ consists of the states from position p to the end of the linked-list. The two lists are then separated at point p. The time for this is $O(|C|)$. Therefore, the time over all the classes in Π, is $O(n)$, for the fixed character c and the class S. This scan of the classes of Π has to be repeated for each of the $|\Sigma|$ characters, but since $|\Sigma|$ is assumed to be a constant number, the total time for the scans in the general step, for a given S, is still $O(n)$. Finally, after all of the scans are finished, for that S, we use S again to find and unmark all of the marked states in Q. That again only takes $O(n)$ time, so the total time for the general step, for each S removed from L, is $O(n)$.

We claim that if C is already in L, and C' is non-trivial, then the algorithm will still be correct if we remove C from L when C' and $C - C'$ are added to L. With the correct data-structures, this can be done in constant time, and clearly it improves the running time to have fewer potential splitters to check. However, removing C will complicate the proof of correctness of the successive refinement algorithm, and the removal of C is not needed for correctness or for the $O(n^2)$ time bound, so we leave C in L. (We leave it to the reader as a good exercises to prove that the algorithm remains correct if C is removed from L in this case, and how to find and remove C from L in constant time.)

Correctness and time analysis of the successive refinement algorithm: Recall that the invariant holds when the algorithm starts, i.e., that every state q in DFA M is distinguishable from every state not in the same class as q. Suppose that the invariant holds up until some point in the algorithm; let Π be the partition at that point, and let S be the next class removed from L. Note that S may not be a current class in Π, since the partition may have been refined since the time when S was put into L. Still, at the point when S was put into L, every state q in S was distinguishable from every state not in S (by the invariant), and that property can not be changed by further refining of the

states of the partition (even if S was subdivided). So at the point when S is removed from L, every state in S is distinguishable from every state not in S.

Now consider what happens in the general step, for a chosen character $c \in \Sigma$, when a class C subdivides into classes C' and $C - C'$. That means that for every state q in C', $\delta(q, c)$ is a state, q_S, in S, while for every state $\overline{q}$ in $C - C'$, $\delta(\overline{q}, c)$ is a state $\overline{q}_S$ not in S. But any state in S is distinguishable from any state not in S (by the discussion above), so states q_S and $\overline{q}_S$ are distinguishable. It follows that q and $\overline{q}$ are distinguishable, and more generally, every state in C' is distinguishable from every state in $C - C'$. Now before the subdivision of C, every state in C was distinguishable from every state not in C (by the invariant), and so every state in C' or $C' - C$ is distinguishable from every state not in C' or $C' - C$, and so the invariant continues to hold after the subdivision of C. This proves that the invariant holds throughout the algorithm.

Now to finish the proof of correctness, i.e., that the successive refinement algorithm finds $\vec{\Pi}$, we claim that at termination, for every class C in Π, every pair of states in C are indistinguishable. Suppose not, so that there is some counterexample to the claim. That is, there is a class C in Π that contains states q and q' that are distinguishable. That means that for some string w, the computation from q on w reaches a final state, wlog, but the computation from q' on w does not. Over all such counterexamples to the claim, let C, q, q' be the class and pair of states that has the *shortest* string w demonstrating that q and q' are distinguishable. Clearly, w is not ϵ, for if it were, then exactly one of q or q' would be a final state, but classes all final states and non-final states were separated into classes F and N respectively, at the start of the algorithm, and classes are only refined during the algorithm. So $|w| \geq 1$. Let w_1 be the first character of w; let $\delta(q, w_1) = q_1$, and $\delta(q', w_1) = q'_1$. If q_1 and q'_1 are in the same class of Π, then there is a counterexample with a string of length $|w| - 1$, contradicting the choice of w. One the other hand, if q_1 is in a class $\tilde{C}$ and q'_1 is not, then when $\tilde{C}$ was removed from L (and it has been removed since L is empty), and the picked was set to w_1, the algorithm should have subdivided class C, putting q and q' in different classes. This is again a contradiction, so there is no counter-example to the claim. So, at termination, every pair of states in the same class in Π are indistinguishable, and every pair of states that are in different classes in Π are distinguishable. Hence, the partition at the end of the successive refinement algorithm is the desired $\vec{\Pi}$, and the algorithm is correct.

For the time analysis, note that the number of possible subdivisions is bounded by $n - 1$, since we start out with two classes in the partition, and each subdivision increases the number of classes in Π by one, and the number of classes can be at most n. Hence, at most $2n$ classes are ever put into L, so at most $2n$ classes are ever removed from L. When a class S is removed from L, the general step needs $O(n)$ time, so the total time for the successive refinement algorithm is $O(n^2)$.

$O(n \log n)$ time is possible The reduction of the time to $O(n \log n)$ involves two clever ideas (one algorithmic and one in the time analysis), which we may talk about later. But it also involves several small, somewhat annoying data-structure details to properly implement the idea, and those kinds of data-structure details are outside of the scope of this course[1]. So, for now, we will end with the $O(n^2)$ time bound, although we state the following exercises which lead to the $O(n \log n)$ time bound.

Exercise 1: Show that the algorithm remains correct if we modify it so that when a class C in L is subdivided into C' and $C - C'$, class C is replaced in L with C' and $C - C'$.

Exercise 2: Assuming the modification stated in Exercise 1, show that the algorithm remains correct if we next modify it so that when a class C not in L is subdivided into C' and $C - C'$, either C' or $C - C'$ is put into L, but not both. So in particular, it remains correct if we put into L the smaller of the two sets C' and $C - C'$.

Exercise 3: Show that if the modifications described in Exercises 1 and 2 can each be implemented in constant time, then the successive refinement algorithm runs in $O(n \log n)$ time. This has a very simple answer if you view things just right (that is why it is considered a clever answer), but is otherwise a hard problem.

Exercise 4: Show how to implement the modifications described in Exercises 1 and 2, so that each can be implemented in constant time.

[1] Hopcroft first suggested the $O(n \log n)$ method in 1971, but a further paper was published that fully explained the implementation details.