DFA Decision Procedures

When studying models of computation, we are primarily interested in two kinds of results:

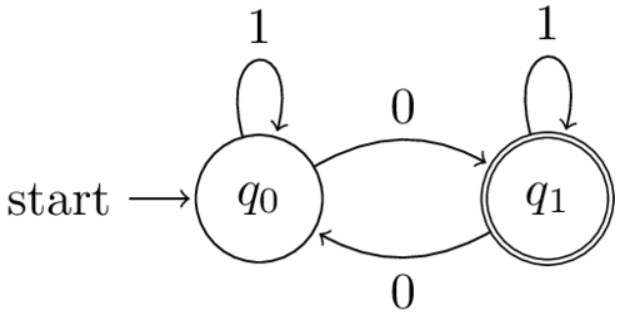
- *Closure Properties* (Sipser §1.2)—given an operation over a particular sort of language (*e.g.*, the regular languages), does the operation preserve the sort of the input languages? For example, the result of concatenating two regular languages is always a regular language.
- *Decision Procedures*—given a machine (*e.g.*, a DFA), can we determine if a proposition about that machine holds? For example, is a given DFA minimal with respect to the number of states that it contains?

Here, we are concerned with the second kind of results. In particular, we will study the following problems:

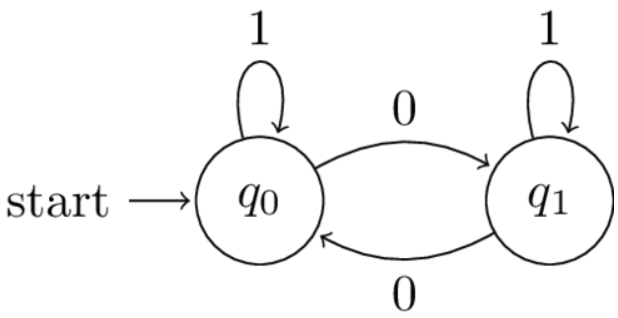
- Emptiness: given a DFA (equivalently, an NFA or regular expression), is its language empty?
- Acceptance: given a DFA (equivalently, an NFA or regular expression) and an input string, does the DFA recognize that input string?
- Minimization: given a DFA, is it minimal with respect to the number of states that it contains?
- Equivalence: given two DFAs (equivalently, NFAs or regular expressions), are the DFAs *equivalent*? That is, do they accept the same set of strings and reject the same set of strings?

Emptiness and Acceptance

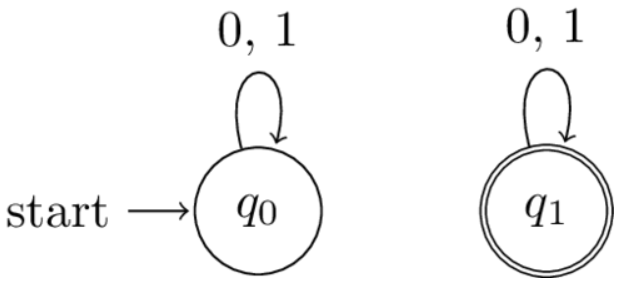
The language of a DFA D is *empty* if it is the empty set, *i.e.*, $L(D) = \emptyset$. The emptiness problem consists of an input DFA D and determining if $L(D) = \emptyset$. A naïve strategy for solving this problem is running D on every possible string $w \in \Sigma^*$, returning false if any such w is accepted by D. This does not work because there are an infinite number of such strings—we never have the opportunity to return true! Instead, we must determine whether $L(D) = \emptyset$ by inspecting the structure of D. To get a feel of how to check this, consider the following DFA:



The language of this DFA is non-empty. It at least accepts the string 10. In contrast, consider the following DFA:



This DFA is identical to the first, except that it does not have any accepting states. Because of this, no string will ever be accepted by the DFA and so its language is empty. In light of this, it is clear that the language of any DFA whose set of accept states is empty will be empty as well. However, there is one other case we must consider.



Here, the DFA has an accepting state, q_1. However, there is no way to reach that accepting state from the start state. To formalize this idea, let's define the notion of *reachability* in a DFA.

Definition 1. *In a DFA $D = (Q, \Sigma, \delta, q_0, F)$, a state $q' \in Q$ is* reachable *from another state $q \in Q$ if there exists a $w \in \Sigma^*$ such that $\delta^*(q, w) = q'$.*

Intuitively, a state q' is reachable from a state q if there exists a series of characters we can read to go from q to q' according to the transition function δ. In the case where $q = q_0$, then the series of characters we read correspond to strings accepted by the DFA.

Thus, the emptiness problem is a matter of determining *reachability* in a DFA. We can define the solution to the emptiness problem as follows:

Theorem 1. *Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, $L(D) = \emptyset$ if and only if no $q_f \in F$ is reachable from q_0.*

Proof. In both directions of the biconditional, correctness follows from the fact that the definition of reachability is analogous to the definition of acceptance of a DFA: "There exists a chain of characters that take the machine from the start state to an accepting state." □

Note that we don't need to enumerate all possible strings to test reachability. It is sufficient to simply perform a breadth- or depth-first search starting from q_0 to see if we are able to reach an accepting state.

Likewise, the acceptance problem considers a DFA D and a string w and asks whether $w \in L(D)$. This also corresponds to reachability:

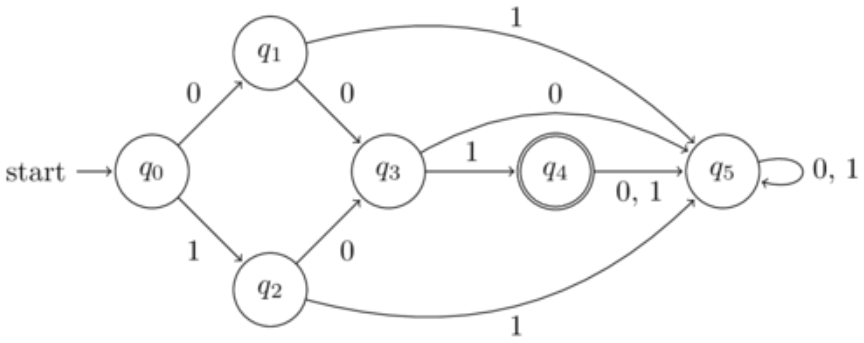
Theorem 2. *Given a DFA $D = (\Sigma, Q, \delta, q_0, F)$ and string $w \in \Sigma^*$, $w \in L(D)$ if and only if $q_f \in F$ is reachable from q_0 using w.*

Proof. Again, both directions of the biconditional are immediate from the definition of reachability and DFA acceptance. □

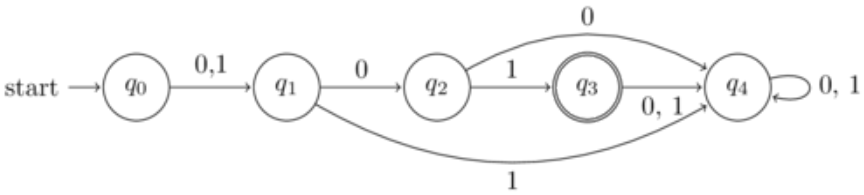
Here, we simply need to hand-simulate execution of D on w to determine acceptance.

Minimization and Equivalence

Consider the following DFA:



The language of this DFA are the strings corresponding to the following regular expression: $(0 \cup 1)01$. However, this is a simpler DFA that recognizes the same language:



This DFA is simpler in the sense that it has one fewer states than the original DFA. We could try to reduce this DFA further, but it turns out that this DFA is the *smallest* DFA recognizing the language $(0 \cup 1)01$. The process of taking a DFA that recognizes language L and finding a corresponding DFA with less states that also recognizes L is called *minimization*.

The intuition behind minimizing a DFA is that some states in a non-minimal DFA exhibit the *same behavior*. For example, consider q_1 and q_2 in the original DFA. Even though they are different states, you can see that for any string w that simulating the machine on input w starting at either q_1 or q_2 results in the same result: either both cases accept the string or reject the string. This is because for any character, q_1 and q_2 both move to the same states (q_3 on a 0 and q_5 on a 1). In this sense, q_1 and q_2 are *indistinguishable*—we can combine them without any change in the behavior of the DFA. Formally, we define two states to be distinguishable as follows:

Definition 2. *In a DFA $D = (\Sigma, Q, \delta, q_0, F)$, states $p, q \in Q$ are* distinguishable *if there exists a string $w \in \Sigma^*$ such that $\delta^*(p, w) = p'$ and $\delta^*(q, w) = q'$ and either $p' \in F, q' \notin F$ or $p' \notin F, q' \in F$.*

Indistinguishability is defined similarly:

Definition 3. *In a DFA* $D = (\Sigma, Q, \delta, q_0, F)$*, states* $p, q \in Q$ *are* indistinguishable *if for all* $w \in \Sigma^*$*,* $\delta^*(p, w) = p'$ *and* $\delta^*(q, w) = q'$ *and either* $p' \in F, q' \in F$ *or* $p' \notin F, q' \notin F$*.*

With this in mind, the *DFA minimization process* works in three steps:

1. First, remove from the DFA any states that are *unreachable* from the start state (using the definition of reachability from the previous section).
2. Discover pairs of states in the DFA that are indistinguishable.
3. Combine indistinguishable states to achieve a final, *minimal* DFA.

To discover pairs of indistinguishable states, we use a *table-filling algorithm* that tracks which pairs of states are indistinguishable. We repeatedly refine the table, discovering new states that are indistinguishable until we reach a point where we make no new discoveries. The final table denotes which states are indistinguishable and thus combinable in the final step of the algorithm.

As an example, consider running the table-filling algorithm on the original DFA. Our table has the following shape:

q_0	×	×	×	×	×	×
q_1		×	×	×	×	×
q_2			×	×	×	×
q_3				×	×	×
q_4					×	×
q_5						×
	q_0	q_1	q_2	q_3	q_4	q_5

An entry in the table is marked with a circle (∘) if we have discovered that the corresponding pair of states are *distinguishable*. If the entry is empty, we say that the states are *indistinguishable*. Entries marked with a cross (×) are ignored as they are redundant (since the ordering does not matter in these pairs of states).

First, from our definition of distinguished states, we see that any accepting state is distinguished from a non-accepting state. We therefore mark any pair of states that contains q_4 as distinguished:

q_0	×	×	×	×	×	×
q_1		×	×	×	×	×
q_2			×	×	×	×
q_3				×	×	×
q_4	∘	∘	∘	∘	×	×
q_5					∘	×
	q_0	q_1	q_2	q_3	q_4	q_5

Now, we repeat the following process until we no longer change the table:

1. For each pair of states (p, q) and for each possible character $a \in \Sigma$, consider the new pair of states $(\delta(p, a), \delta(q, a))$.

2. If (p, q) is undistinguished, but $(\delta(p, a), \delta(q, a))$ is distinguished, then mark (p, q) as distinguished.

For example, on the first iteration of our process, we first consider the pair (q_1, q_0) (the top-left corner of the table although the order does not matter) and the possible transitions from it:

- On input 0, $(\delta(q_1, 0), \delta(q_0, 0)) = (q_3, q_1)$.
- On input 1, $(\delta(q_1, 1), \delta(q_0, 1)) = (q_5, q_2)$.

We note that (q_3, q_1) and (q_5, q_2) are not marked (*i.e.*, are considered undistinguished), so we do not change the entry for (q_1, q_0). In contrast, consider the pair (q_2, q_3):

- On input 0, $(\delta(q_2, 0), \delta(q_3, 0)) = (q_3, q_5)$.
- On input 1, $(\delta(q_2, 1), \delta(q_3, 1)) = (q_5, q_4)$.

While (q_3, q_5) is not marked, (q_5, q_4) is marked, so we mark (q_2, q_3). Continuing this process, we arrive at the following updated table

q_0	×	×	×	×	×	×
q_1		×	×	×	×	×
q_2			×	×	×	×
q_3	○	○	○	×	×	×
q_4	○	○	○	○	×	×
q_5				○	○	×
	q_0	q_1	q_2	q_3	q_4	q_5

Note that because the only pairs of states involving q_4 were marked as distinguished initially, then only pairs of states that transitioned into a pair containing q_4 are marked as distinguished in this first round. The only such state is q_3, so only pairs of states involving q_3 were marked in this round.

Since we modified the table in the first round, we repeat the process again for all the unmarked states. On the next iteration, the updated table is:

q_0	×	×	×	×	×	×
q_1	○	×	×	×	×	×
q_2	○		×	×	×	×
q_3	○	○	○	×	×	×
q_4	○	○	○	○	×	×
q_5		○	○	○	○	×
	q_0	q_1	q_2	q_3	q_4	q_5

And on the final iteration, the updated table is:

q_0	×	×	×	×	×	×
q_1	○	×	×	×	×	×
q_2	○		×	×	×	×
q_3	○	○	○	×	×	×
q_4	○	○	○	○	×	×
q_5	○	○	○	○	○	×
	q_0	q_1	q_2	q_3	q_4	q_5

(I recommend stepping through the algorithm and checking your work with the tables above.) In the final iteration, the pair (q_2, q_1) is left unmarked. To see why we'll never mark (q_2, q_1), look at its transitions:

- On input 0, $(\delta(q_2, 0), \delta(q_1, 0)) = (q_3, q_3)$.
- On input 1, $(\delta(q_2, 1), \delta(q_1, 1)) = (q_5, q_5)$.

And note that the pairs (q_3, q_3) and (q_5, q_5) are not marked as distinguished.

Because of this, successive iterations of the process do not change the table. Therefore, this is the final result and we note that q_2 and q_1 are indistinguishable. Collapsing these two states into a single state results in the minimized DFA given above. A surprising but important fact about our minimization algorithm is that it produces a *unique* DFA.

Theorem 3. *Consider a DFA D. There exists a unique DFA $D' = (Q', \Sigma, \delta', q_0', F')$ (up to renaming of states) such that $L(D) = L(D')$ and for any other DFA $D'' = (Q'', \Sigma, \delta'', q_0'', F'')$ such that $L(D) = L(D'')$ and $|Q'| < |Q''|$. The table-filling algorithm above derives this D' for some given DFA D.*

We'll withhold the proof of this claim until later when we have additional machinery to reason about the *equivalence classes* of states discovered by our algorithm. With this theorem, we can tackle the closely-related problem of DFA *equivalence*:

Definition 4. *Let D_1 and D_2 be DFAs. D_1 is* equivalent *to D_2, written $D_1 \equiv D_2$, if $L(D_1) = L(D_2)$. That is D_1 accepts the same strings as D_2, and D_1 rejects the same strings as D_2.*

Luckily, because the minimal DFA D' for some DFA D is unique, we can use the following procedure to see if two DFAs D_1 and D_2 are equivalent:

1. Run the minimization procedure on D_1 and D_2 to produce minimal DFAs D_1' and D_2'.
2. If D_1' and D_2' are identical (up to renaming of states), then $D_1 \equiv D_2$.

Acknowledgments

This reading was written by Dr. Peter-Michael Osera. The only changes made were in regard to spacing, and in notation to match *Introduction to the Theory of Computation*, by Sipser, 3rd edition.