Languages and State-Equivalence

Robert Keller
4 February 2007

We know that the language accepted by a DFA is the set of all labeled paths that take the DFA from its starting to state to some final state.

We are going to associate a language with *each* state of the DFA. If q is a state, then L(q) will be the set of all paths that take us from state q to some final state. Thus, as a special case, if q_0 is the starting state, $L(q_0)$ is the language accepted by the DFA.

What about the other states and their languages? Suppose that there is a transition from q_0 to q_1 with label $\sigma \in \Sigma^*$. If a string σx is in $L(q_0)$, then a little thought reveals that x is in $L(q_1)$, since the DFA is deterministic and any accepted string beginning with σ must first go to q_1. In fact, $L(q_1)$ relates to $L(q_0)$ as the *language derivative*, analogous to regular expression derivatives:

$$L(q_1) = L(q_0)/\sigma = \{x \in \Sigma^* \mid \sigma x \in L(q_0)\}$$

We now see that the languages associated with the states of our DFA are the same languages that could be represented by regular expression derivatives discussed in another document.

What about an accepting state, say q_2? A little reflection reveals that the empty string ε is in $L(q_2)$, since the strings in the latter are those that take the DFA from q_2 to an accepting state. But in a DFA, ε can take q_2 only to q_2. Conversely, any state q_i where $\varepsilon \in L(q_i)$ must be an accepting state. (There may be other strings in $L(q_2)$ as well.) We summarize this as

Observation 0:

$$\varepsilon \in L(q) \text{ iff } q \text{ is an accepting state}$$

State Equivalence

Two states are called *equivalent*, written $q \equiv q'$, iff their associated languages are equal, $L(q) = L(q')$. If two or more states in a DFA are equivalent, then all but one of those states is redundant in a sense. We could get rid of the redundant ones and replace all of their functions by a single state.

It is possible to determine whether two states are equivalent without using any regular expressions, just the DFA. One way to do this would be to use the product construction. But here we discuss a different way, one that deals with all states simultaneously to determine which pairs are equivalent. To explain this method, we introduce *truncated equivalence*. For a state q, define sets $L_k(q)$ for any natural number k:

$$L_k(q) = \{x \in L(q) \mid |x| \leq k\}$$

Here |x| represents the *length* of string x.

Say that two states q and q' are *k-equivalent*, written $q \equiv_k q'$, where k is a natural number, iff

$$L_k(q) = L_k(q')$$

Evidently, general equivalence is like a conjunction of truncated equivalences:

$$q \equiv q' \text{ iff } \forall k \geq 0 \;\; q \equiv_k q'$$

because $q \equiv q'$ says that their languages contain the same strings, whereas $q \equiv_k q'$ says that their languages have the same strings among the strings of length k or less.

Lemma 1:

For any two states, $q \equiv_0 q'$ iff q and q' are both accepting or both non-accepting.

Proof: $q \equiv_0 q'$ means that their languages have the same strings of length 0. But ε is the only string of length 0, so if $q \equiv_0 q'$ then either ε is in both languages or in neither. But by Observation 0, ε is in a state's language iff the state is accepting.

Observation 1:

$\forall k > 0$ ($q \equiv_k q'$ implies $q \equiv_{k-1} q'$).

Corollary to Observation 1:

$\forall k > 0$ ($q \equiv_k q'$ implies $\forall j < k \; q \equiv_j q'$).

A kind of converse to Observation 1 is the following important result:

Lemma 2:

$\forall k > 0$

If $q \equiv_{k-1} q'$

and $\forall \sigma \in \Sigma \;\; \delta(q, \sigma) \equiv_{k-1} \delta(q', \sigma)$,

then $q \equiv_k q$.

In words, if a pair of states are k-1 equivalent, and furthermore for each input letter σ, their pair of next states are also k-1 equivalent, then the states are k-equivalent.

Proof: Suppose that $k > 0$, $q \equiv_{k-1} q'$, and $\forall\sigma \in \Sigma \ \ \delta(q, \sigma) \equiv_{k-1} \delta(q', \sigma)$, to show $q \equiv_k q'$.

Since $q \equiv_{k-1} q'$ gives us that L(q) and L(q') have the same strings of length k-1 or less, we only need to show that they have the same strings of length k.

Suppose that σx is an arbitrary string of length k, with $\sigma \in \Sigma$ and $|x| = k-1$. Since $\forall\sigma \in \Sigma \ \ \delta(q, \sigma) \equiv_{k-1} \delta(q', \sigma)$, we have

$$\forall\sigma \in \Sigma \ (\sigma x \in L(q) \text{ iff } \sigma x \in L(q'))$$

Since σx is arbitrary,

$$q \equiv_k q'$$

Lemmas 1 and 2 form the basis of an algorithm for computing whether or not two states are equivalent. Better yet, the algorithm classifies *all* pairs of states in a DFA as equivalent or not.

It should be apparent that the relations $\equiv_k$ and $\equiv$ are *equivalence relations*, because the characterize pairs of states as having a certain *sameness*. Recall that an equivalence relation $\equiv$ is one with the properties:

Reflexive: $\forall q$	$q \equiv q$
Symmetric: $\forall q, q'$	If $q \equiv q'$ then $q' \equiv q$
Transitive: $\forall q, q', q''$	If $q \equiv q'$ and $q' \equiv q''$, then $q \equiv q''$.

It is important for our algorithm that every equivalence relation is representable by a *partition*, a set of non-empty sets of states, such that the no two sets have elements in common and the union of the sets is the set of all states. A set within the partition simply contains all states that are equivalent to each other.

Example: If the state set were {1, 2, 3, 4, 5, 6}, then two partitions would be {{1, 2, 3}, {4, 5}, {6}} and {{1, 2}, {3, 4, 5, 6}}. In the first case, 1 is equivalent to 2 and 3, while 4 is equivalent to 5. Obviously for a set of n states a partition must have between 1 and n elements.

Let P_k denote the partition corresponding to equivalence relation $\equiv_k$. Let P* be the partition corresponding to $\equiv$. Our algorithm will provide a way of computing P*, hence determining whether any two states are equivalent, by computing "successive approximations" to P*.

Corollary to Lemma 1:

Two states q and q' are in the same element of partition P_0 iff q and q' are both accepting or both non-accepting.

Corollary to Lemma 2:

Two states q and q' are in the same element of P_k iff both are in the same element of P_{k-1} and $\forall \sigma \in \Sigma$ their next state pairs $\delta(q, \sigma)$ and $\delta(q', \sigma)$ are in the same element of P_{k-1} as well.

Example 1:

Consider this DFA:

	a	**b**
1S	6	3
2	5	6
3F	4	5
4F	3	2
5	2	1
6	1	4

Here the state set is {1, 2, 3, 4, 5, 6}, with S marking the start state and F marking the accepting states. The input alphabet is {a, b}.

By Corollary to Lemma 1, P_0 = { {1, 2, 5, 6}, {3, 4} }.

By Corollary to Lemma 2, P_1 = { {1, 6} {2, 5}, {3, 4} }.

By Corollary to Lemma 2, P_2 = { {1, 6} {2, 5}, {3, 4} }.

It can then be inferred that P* = { {1, 6} {2, 5}, {3, 4} }, since computing P_3 from P_2 will yield the same partition as computing P_2 from P_1 and so on. With P*, we know exactly which states are equivalent.

Lemma 3:

$\forall k > 0$ If $P_k = P_{k-1}$ then $\forall j \geq k$ $P_j = P_{k-1}$ and moreover $P^* = P_{k-1}$.

Proof: Based on the observation made above, we can compute P_k based on P_{k-1} and the next state function δ. If this computation of P_k yields the same partition P_{k-1}, then computing P_{k+1} from P_k etc. will yield the same partition P_{k-1}. If all are the same, then $P^* = P_k$.

Definition: Partition P' *refines* partition P, and P is *refined by* P', written $P \leq P'$, iff every set in P' is contained in a set of P.

In general, every partition refines itself. A partition *properly refines* another if it refines it and the two are not equal.

In the previous DFA example, P_1 refines P_0.

The notation $P \leq P'$ is suggestive of the fact that if P is refined by P', then the number of sets in P' will be at least the number in P, since in order for P' to refine P, either they have to be the same, or one or more of the sets in P is "split" in P'.

Observation 2: For all k, P_k refines P_{k-1}.

This follows directly from Observation 1.

Thus we have $P_0 \leq P_1 \leq P_2 \leq P_3 \leq \ldots$

Some of the refinements in the above series may be proper. But they cannot go on being proper forever. They must eventually reach a point where no proper refinement is possible. At the point P_k where P_k cannot be properly refined, we will have $P^* = P_k$. This is the key to the termination of the equivalence test.

Let |P| be the number of sets in a partition. Observe that if P' refines P, then $|P'| \geq |P|$.

Observation 3:

If $|P_0| = 1$, then $P_0 = P^*$.

Rationale: If $|P_0| = 1$, then either all states are accepting or all are non-accepting. Hence every string is accepted or none is, and all states are equivalent.

Corollary to Observation 3:

If $P_0 \neq P_1$ then $|P_0| \geq 2$.

Proof: If $P_0 \neq P_1$ then $P_0 \neq P^*$. Then by the contrapositive of Observation 3 $|P_0| \neq 1$, so $|P_0| \geq 2$.

Observation 4:

For every k, if $|P_k| = n$, where n is the number of states, then $P_k = P^*$.

Lemma 4:

For every k, if $P_k \neq P_{k+1}$ then $|P_k| \geq k+2$.

Proof: This is by induction, starting with the Corollary to Observation 3 as a basis. Informally, if the sequence $P_0 \leq P_1 \leq P_2 \leq P_3 \leq \ldots P_k$ contains only *strict* refinements, then $|P_0| \geq 2$, $|P_1| \geq 3$, $|P_2| \geq 4$, . . . Hence $|P_k| \geq k+2$.

Corollary to Lemma 4:

If n is the total number of states, then $P_{n-2} = P^*$.

Proof: If the sequence $P_0 \leq P_1 \leq P_2 \leq P_3 \leq \ldots P_n$ contains only strict refinements, then $|P_{n-2}| \geq n$, by the lemma. But then $|P_{n-2}| = n$, and by Observation 4, $P_{n-2} = P^*$. On the other hand, if the sequence is not strict, then for some $k < n-2$, $P_k = P_{k-1}$, in which case $P^* = P_k$ by Lemma 3.

We can summarize the important points of the preceding discussion with an algorithm.

Equivalent States Algorithm

Input is a DFA, output is the equivalence partition P*.

1. Let n be the number of states.
2. Compute P_0 from the information of which states are accepting.
3. Set k = 1.
4. While $k \leq n-2$:
    a. Compute P_k from P_{k-1}.
    b. If $P_k = P_{k-1}$, stop. $P^* = P_{k-1}$.
    c. Set k to k+1.
5. Stop. $P^* = P_{n-2}$.

Example 2: We essentially executed this algorithm in the previous DFA example. Here's another example:

	0	**1**
a S	b	f
b	g	c
c F	a	c
e	h	f
f	c	g
g	g	e
h	g	c

Step 2: $P_0 = \{ \{a, b, e, f, g, h\}, \{c\} \}$

Step a: $P_1 = \{ \{a, e, g\}, \{b, h\}, \{f\}, \{c\} \}$

Step a: $P_2 = \{ \{a, e\}, \{g\}, \{b, h\}, \{f\}, \{c\} \}$

Step a: $P_3 = \{ \{a, e\}, \{g\}, \{b, h\}, \{f\}, \{c\} \}$

Stop: $P^* = P_2$.

Creating the Minimum-State DFA Equivalent to a Given DFA

In an equivalence relation, the equivalence classes can be identified by giving a *representative* of the class, defined to be any one of the elements in the class. Generally, representatives are not unique, because the equivalence classes can contain more than one element.

It is common to use [q] to designate the class represented by a specific representative q. For example, in the example above, we would have for P*:

[a] = [e] = {a, e}
[g] = {g}
[b] = [h] = {b, h}, etc.

Once we have P* for a DFA M, define a new DFA M* as follows:

- The states of M* are the equivalence classes P*.
- The initial state of M* is the equivalence class of the initial state of M, $[q_0]$.
- Define the transition relation δ^* for M* from the transition relation δ for M as follows:

  For every state $[q] \in P^*$ and every alphabet symbol, $\sigma \in \Sigma$

  $\delta^*([q], \sigma) = [\delta(q, \sigma)]$

  In a definition of this kind, we need to establish that the definition of δ^* does not depend on the representative chosen for [q]. Indeed, if we had chosen a different representative $q' \in [q]$, it would not matter because if $q \equiv q'$ implies that $\delta(q, \sigma) \equiv \delta(q', \sigma)$, so $[\delta(q, \sigma)] = [\delta(q', \sigma)]$. Thus the right-hand side of the definition of is the same regardless of which element of [q] we choose on the left-hand side.

- Define the accepting states of M* to be those elements of P* that contain accepting states in M.

Example 3:

The minimum-state DFA equivalent to Example 2 is given as follows:

	0	**1**
{a, e} S	{b, h}	{f}
{b, h}	{g}	{c}
{c} F	{a, e}	{c}
{g}	{g}	{a, e}
{f}	{c}	{g}

Equivalence of DFA's Based on Equivalence of States

There is no requirement that the states of a DFA be connected in order to determine equivalence. An application of the disconnected case arises in determining whether two DFA's are equivalent. In this case, we require that the states be named distinctly. We construct the union of the state sets and apply the state equivalence algorithm. The DFA's are equivalent iff their initial states are equivalent.

Strings as States

Much of the reasoning presented in the preceding sections can be applied to any language, not just regular languages. (The part that can't be applied has to do with whether the series of partitions terminates.)

Let $L \subseteq \Sigma^*$ be a language. We can regard the strings Σ^* as *states* of a possibly-infinite state automaton in the following sense:

- The initial state is ε.
- The accepting states are those strings in L.
- The transition function δ is defined by $\delta(x, \sigma) = x\sigma$.

It is then possible to define equivalence $\equiv_L$ on this state set just as for the finite-state case. The only issue is that there is no algorithm for computing $\equiv_L$, because we are dealing with an infinite set of states. But in principle, the equivalence relation, and thus equivalence classes, still exist. Furthermore the *minimal* equivalent automaton could be defined exactly as for the finite-state case, with its states being equivalence classes of $\equiv_L$.

The language L is regular iff this minimal automaton has a finite set of state equivalence classes. Classically, this idea is captured by the Myhill-Nerode theorem:

Myhill-Nerode Theorem:

L is a regular language iff $\equiv_L$ has a finite set of equivalence classes.

Corollary 1:

L is *not* regular iff the set of equivalence classes is infinite.

Corollary 2:

L is *not* regular there is an infinite set of strings that are pair-wise *inequivalent* (also called **pairwise-distinguishable**).

Corollary 2 follows from corollary 1 because in order for there to be an infinite set of equivalence classes, there needs to be an infinite set of strings, no two of which are equivalent. Conversely, if there is such an infinite set of strings, then there must be an infinite set of equivalence classes in which to place them.

The Myhill-Nerode theorem provides one of two standard methods for showing that a language is not regular (the other being the "pumping lemma").

Example 4

Consider the language $L = \{0^n1^n \mid n \geq 0\}$. We can readily see that each string of the form 0^n determines a different equivalence class. For example, consider 0^n vs. 0^m where $n \neq m$. In the former case $0^n1^n \in L$, while $0^m1^n \notin L$. Since there are infinitely many distinct n, the number of equivalence classes is infinite: there is no DFA accepting this language.

Although we might not be able to construct the minimal automaton for an arbitrary language in its entirety, we can still get insight about its structure from the equivalence classes.

For any string $x \in \Sigma^*$, define

$$L/x = \{z \in \Sigma^* \mid xz \in L\}$$

In other words, L/x is the set of extensions of string x that give us members of L. So

$$z \in L/x \text{ iff } xz \in L$$

In particular, since $\varepsilon z = z$

$$L/\varepsilon = L$$

$$\varepsilon \in L/x \text{ iff } x \in L$$

Thus the sets L/x, as x ranges over all elements of Σ^*, *are* the equivalence classes of $\equiv_L$: L/x = [x], the equivalence class of x. It is altogether possible that L/x = L/y although x and y differ. This just says that x and y are equivalent.

I thus dub the sets L/x the ***abstract states*** of the language L, and from earlier discussion, they are also the states of the minimal automaton, which thus has transitions of the form:

$$\delta(L/x, \sigma) = L/(x\sigma)$$

The initial state is $L/\varepsilon = L$, and the accepting states are those sets L/x where $\varepsilon \in L/x$.

These abstract states are related to the derivatives of regular expressions discussed in another document, in that

$$L(R/\sigma) = L(R)/\sigma$$

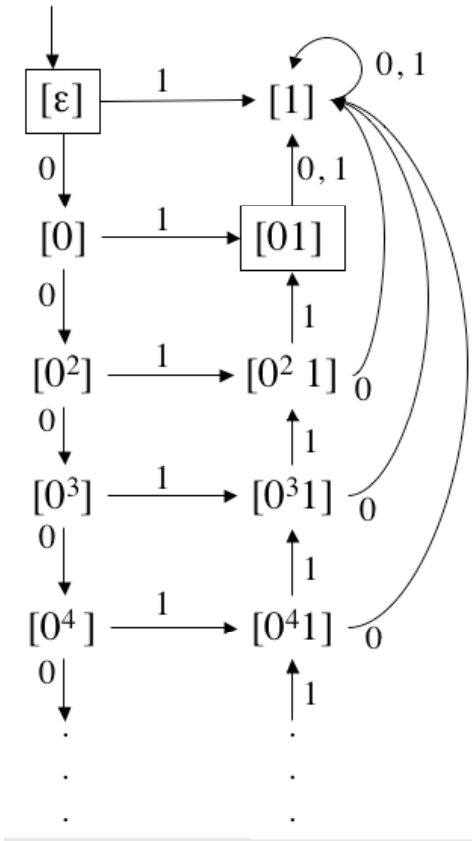
L(R) is the language of regular expression R. On the left, $L(R/\sigma)$ is the language of the derivative regular expression R/σ, while on the right, $L(R)/\sigma$ operates on language L(R) using $/\sigma$.

We can diagram these states in a manner that suggests their structure of an *abstract acceptor* (possibly one with an infinite state set) for the language. The set of abstract states will be finite iff the language is regular.

Nonetheless, we can still determine the *form* of the states and the connections between them. For Example 4, we have:

- $[0^n]$ is a state for each $n \geq 0$.
- $[0^n1^m]$ is a state for each m, n where $m < n$, since extending with 1^{n-m} uniquely gives a string in the language.
- [01] is a state. It contains exactly strings of the form 0^n1^n, $n > 0$, as all are in the language. It does not contain ε, as ε has already been mentioned as 0^0. ε is not equivalent to 01, since there are many different extensions of ε that are accepted, while no proper extension of 01 is accepted.
- All other strings are in the same equivalence class. They are not in the language and cannot be extended to strings that are.

The diagram of abstract states and their transitions is shown below. Accepting states are in boxes.



The strings represented by the equivalence classes for this example, along with the set of strings that extend them to an accepting state, follows:

Class by representative	Equivalent strings	Extensions to accepting
[ε] (accepting)	$\{0^n1^n \mid n \geq 0\}$	$\{0^n1^n \mid n \geq 0\}$
[01] (accepting)	$\{0^n1^n \mid n > 0\}$	$\{\varepsilon\}$
$[0^n]$ where $n > 0$	$\{0^n\}$	$\{0^m1^{n+m} \mid m \geq 0\}$
$[0^n1]$ where $n > 1$	$\{0^{n+m}1^n \mid m > 0\}$	$\{1^{n-1}\}$
[1]	$\{1\}\Sigma^* \cup \{0^n1^n \mid n > 0\}\Sigma^+$	$\varnothing$

It is important to note the difference between notation such as "$\{0^n\}$, where $n > 0$" and "$\{0^n \mid n > 0\}$". In the first case, we are speaking of an infinite family of sets of one element, whereas in the second, we are speaking of a single infinite set.

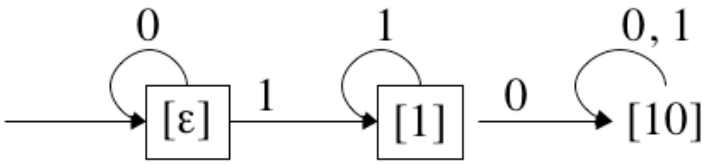
Note that in the case that the language is regular, the sets of extensions to accepting states will be regular. They correspond to the derivatives of the language.

Example 5:

Consider the language $L = \{0^m1^n \mid m, n \geq 0\}$. This language is represented by the regular expression 0*1*. The derivatives are the extensions to accepting strings. The equivalence classes are also regular.

Derivatives = Extension to accepted strings	Equivalent Strings	Representative
0*1* = 0*1*/0	0*	[ε] (accepting)
1* = 0*1*/1	0*11*	[1] (accepting)
∅ = 1*/0	10(0+1)*	[10]

The abstract states for Example 5 are diagrammed below. As before, the accepting states are boxed.



Showing Non-Regularity by the Pumping Lemma

The following gives a *necessary*, but not sufficient, condition for a language to be regular. It is thus used for proving languages to be *not* regular, by showing that the language violates the condition.

Pumping Lemma for Regular Languages

For any regular language L, there is a natural number p = *pump*(L) such that:

If $x \in L$ and $|x| \geq p$

then there exists strings u, v, w such that

a. $x = u\ v\ w$ (i.e. the concatenation)

b. $v \neq \varepsilon$

c. $|u\ v| \leq p$

d. for all $i \geq 0$ $u\ v^i\ w \in L$

In English, if L is a regular language, then for all sufficiently-long strings in L, there is an infinite set of additional strings also in L, with additional qualifications known for those strings.

Proof

Let L be regular. We claim that the condition is satisfied by p = pump(L) = the number of states of a DFA M accepting L.

Consider a string x such that $|x| \geq p$. Each letter of the string takes M to a state. So within a string of length p, p+1 states have been visited. But there are only p states total, so some state must have been visited at least twice.

Let q_0 be the initial state. Let q_1 be the first state that is repeated in the sequence of state traversed due to x. Let

- u be the prefix of x that takes q_0 to q_1.
- v be the infix that takes the first instance of q_1 to the second instance of q_1.
- w be the suffix that takes the second instance of q_1 to an accepting state.

Conditions a and b are clearly satisfied. Also, the repetition of must occur within the first p+1 states, corresponding to a sequence of length p, so condition c is satisfied.

Finally, since v takes q_1 to q_1, v can be repeated any number of times and still take q_1 to q_1. Thus $u\ v^i\ w \in L$ for any i, giving condition d. (Note that since v is non-empty, L is infinite.)

Note that it is possible for there to be *no* $x \in L$ such that $|x| \geq$ pump(L). This would be the case when the language L is finite. In this case, the main premise of the pumping lemma is not satisfied, so there is no need for the conclusions, which would imply that L is infinite, be satisfied.