CMPSCI 250: Introduction to Computation

Lecture #31: What DFA’s Can and Can’t Do
David Mix Barrington
11 April 2012

What DFA’s Can and Can’t Do

- Deterministic Finite Automata
- Formal Definition of DFA’s
- Examples of DFA’s
- DFA’s in Java
- Characterizing Strings With Given Behavior
- Distinguishable Strings
- Languages With No DFA’s

Deterministic Finite Automata

- We now turn to **finite-state machines**, a model of computation that captures the idea of reading a file of text with a fixed limit on the memory we can use to remember what we have seen.
- In particular, the memory used must be **constant**, independent of the length of the file. We ensure this by requiring our machine to have a **finite state set**, so that at any time during the computation all that it knows is which state it is in.
- The **initial state** is fixed. When the machine sees a new letter, it changes to a new state based on a fixed **transition function**. When it finishes the string, it gives a yes or no answer based on whether it is in a **final state**.
- Because the new state depends only on the old state and the letter seen, the computation is **deterministic** and the machine is called a **deterministic finite automaton** or **DFA**.

Where We Are Going

- A DFA **decides** a language -- it says yes or no after reading any string over its alphabet, and its language is the set of strings for which it says yes.

- The **Myhill-Nerode Theorem** will give us a way to take an arbitrary language and determine whether there is a DFA that decides it. We'll define a particular equivalence relation on strings, based only on the language. If this relation has a finite set of equivalence classes, there is a DFA for the language, and there is a **minimal DFA** with as many states as there are classes. We'll see how to compute the minimal DFA from any DFA for the language.

- As we've mentioned, there is a DFA for a language if and only if the language is **regular** (is the language denoted by some regular expression). We'll prove this important result, called **Kleene's Theorem**, over several lectures. Our proofs will show us how to convert a DFA to a regular expression and vice versa.

Formal Definition of DFA's

- Formally a DFA is defined by its **state set** S, its **initial state** $i \in S$, its **final state set** $F \subseteq S$, its **input alphabet** Σ, and its **transition function** δ from $(S \times \Sigma)$ to S.

- We usually represent DFA's by diagrams (labeled directed multigraphs) with a node for each state, a special mark for the initial state, a double circle on each final state, and an arrow labeled "a" from node p to node q whenever $\delta(p, a) = q$.

- The **behavior function** of a particular DFA is a function called δ^* from $(S \times \Sigma^*)$ to S, such that $\delta(p, w)$ is the state of the DFA after it starts in state p and reads the string w. Formally, we say that $\delta(p, \lambda) = p$ and that $\delta^*(p, wa) = \delta(\delta^*(p, w), a)$.

- The **language of a DFA** is defined to be the set of strings w such that $\delta^*(i, w)$ is a final state. For a DFA M, we call this language L(M).

Examples of DFA's

- One of the simplest possible DFA's decides the language of binary strings with an odd number of ones. It has two states E and O, representing whether the machine has seen an even or odd number of ones so far. The initial state is E, and the final state set is {O}. The transition function has $\delta(E, 0) = E$, $\delta(E, 1) = O$, $\delta(O, 0) = O$, and $\delta(O, 1) = E$.

- We can build a four-state DFA for the language EE from Discussion #8. Its states are EE, EO, OE, and OO, where for example $\delta^*(EE, w) = EO$ if w has an even number of a's and an odd number of b's. The initial state is EE and the final state set is {EE}. An a changes the first letter of the state, a b the second.

- Another four-state DFA can decide whether the next to last letter of a binary string w exists and is 1. The state set is {00, 01, 10, 11} and the state after reading w represents the last two letters seen. The initial state is 00 and the final state set is {10, 11}.

DFA’s in Pseudo-Java

- We consider the input to be given like a file, with a method to give the next letter and one to tell when the input is done. We relabel the state set and the alphabet to be `{0,..., states - 1}` and `{0,..., letters - 1}` respectively.

```
public class DFA {
   natural states; natural letters; natural start;
   boolean [ ] isFinal = new boolean[states];
   natural [ ] [ ] delta = new natural [states] [letters];
   natural getNext( ) {code omitted}
   boolean inputDone( ) {code omitted}

   boolean decide ( )
   {// returns whether input string is in language of DFA
      natural current = start;
      while (!inputDone( ))
         current = delta [current] [getNext( )];
      return isFinal [current];}}
```

Characterizing Strings With Given Behavior

- How do we *prove* that a particular DFA has a particular language?
- With the even-odd DFA, we can say that $\delta^*(E, w) = E$ if w has an even number of ones, and $\delta^*(E, w) = O$ if it has an odd number of ones.
- Letting P(w) be the entire statement in the bullet above, we can prove $\forall w:P(w)$ by induction on all binary strings. $P(\lambda)$ says that $\delta^*(E, \lambda) = E$, because λ has no ones and 0 is even, and $\delta^*(E, \lambda) = E$ is true by definition of δ^*. Now assume that P(w) is true, so that $\delta^*(E, w)$ is E if w has an even number of ones and O otherwise. Then w0 has the same number of ones as w, so $\delta^*(E, w0)$ should be the same state as $\delta^*(E, w)$. And w1 has one more one than w, so $\delta^*(E, w1)$ should be the other state from $\delta^*(E, w)$. In each of the four cases, the new state is the one given by the δ function of the DFA.

Another Characterization Example

- The language No-aba is the set of strings that never have an aba substring. We can build a DFA M for No-aba with state set {1, 2, 3, 4}, start state 1, final state set {1, 2, 3}, and transition function $\delta(1, a) = 2$, $\delta(1, b) = 1$, $\delta(2, a) = 2$, $\delta(2, b) = 3$, $\delta(3, a) = 4$, $\delta(3, b) = 1$, and $\delta(4, a) = \delta(4, b) = 4$. (We call 4 a **death state**.) We can see that an aba will take us from any state to 4.

- Let L_1 be the set of strings that have no aba and don't end in a or ab. Let L_2 be the set of strings that don't have an aba and end in a. L_3 is the set of strings that don't have an aba and end in ab, and L_4 is the set that have aba.

- We can make eight checks, one for each value of δ. If $\delta(i, x) = j$, we check that any string in L_i, followed by the letter x, yields a string in L_j. Then we can do an inductive proof, where our statement P(w) is the *entire* statement in the bullet above: "For all states i, $\delta^*(1, w) = i$ if and only if $w \in L_i$" where each L_i is as defined. This proves that w is in L(M) if and only if w is in No-aba.

Distinguishable Strings

- Is it possible that another DFA with only three states could decide No-aba?

- We divided all possible strings into four sets. Suppose a DFA reads w and does not know which of the four sets w is in. We'll show that in this case it is doomed -- for some string x, it will be wrong if it sees x and has to decide wx.

- Look at the four strings λ, a, ab, and aba. If the DFA has $\delta^*(i, \lambda) = \delta^*(i, a)$, we say that it **cannot distinguish between** λ and a. If this is true, the DFA must also have $\delta^*(i, b) = \delta^*(i, ab)$ because a b will take the same state to the same state. Then as well $\delta^*(i, ba) = \delta^*(i, aba)$. But now we know that the DFA cannot decide No-aba, because it gives the same answer on the strings ba (in No-aba) and aba (not in No-aba).

- We can call this an **experiment** that **distinguishes** the two strings λ and a.

Sets of Distinguishable Strings

- Let L be any language. We say that two strings u and v are **L-distinguishable** (also called **L-inequivalent**) if there exists a string w such that $uw \in L$ and $vw \notin L$, or vice versa. They are **L-equivalent** if $\forall w$: $uw \in L \leftrightarrow vw \in L$.

- **Lemma**: If M is a DFA with transition function δ, L is any language, u and v are two L-distinguishable strings, and $\delta^*(i, u) = \delta^*(i, v)$, then $L(M) \neq L$.

- **Proof**: We can prove by induction that if $\delta^*(i, u) = \delta^*(i, v)$, then for any string w, $\delta^*(i, uw) = \delta^*(i, vw)$. For the particular w that distinguishes u and v, then, the single state $\delta^*(i, uw) = \delta^*(i, vw)$ needs to be both final and non-final if $L(M) = L$.

- **Theorem**: If there exists a set of k pairwise L-distinguishable strings, no DFA that decides L can have fewer than k states.

- **Proof**: With more strings than states, by the **Pigeonhole Principle** there must exist two strings mapped from i to the same state by the function δ^*.

Languages With No DFA's

- Consider the balanced parenthesis language Paren, which we will write as a subset of {L, R}* with L for left parens and R for right parens. We can prove that there is no DFA *at all* that decides this language.

- Look at the infinite set of strings {λ, L, LL, LLL,...}. I claim that this set is pairwise Paren-distinguishable, because if i and j are two naturals with $i \neq j$, then L^i and L^j are distinguished by R^i, since L^iR^i is in Paren and L^jR^i is not.

- So for any natural k, we can find more than k pairwise Paren-distinguishable strings, and by our theorem there cannot be a k-state DFA.

- Our real-life algorithm to decide Paren is to remember the number of L's we have seen, minus the number of R's. If this number ends at 0, without ever going negative, we are in Paren. But this requires more than constant memory -- potentially a state for every natural from 0 through n.

The Language Prime Has No DFA

- Let Prime be the language $\{a^n: n \text{ is a prime number}\}$. It doesn't seem likely that any DFA could decide Prime, but this is a little tricky to prove.

- Let i and j be two naturals with $i > j$. We'd like to show that a^i and a^j are Prime-distinguishable, by finding a string a^k such that $a^i a^k \in$ Prime and $a^j a^k \notin$ Prime. We need a natural k such that $i + k$ is prime and $j + k$ not, or vice versa.

- Pick a prime p bigger than both i and j (since there are infinitely many primes). Does $k = p - j$ work? It depends on whether $i + (p - j)$ is prime -- if it isn't we win because $j + (p - j)$ *is* prime. If it is prime, look at $k = p + i - 2j$. Now $j + k$ is the prime $p + (i - j)$, so if $i + k = p + 2(i - j)$ is not prime we win.

- We find a value of k that works unless *all* the numbers p, $p + (i - j)$, $p + 2(i - j)$,..., $p + r(i - j)$,... are prime. But $p + p(i - j)$ is not prime as it is divisible by p.