Finite Automata as Graphs

Richard Cole

September 8, 2006

Finite automata model very simple computational devices. These devices have a constant amount of memory and process their input in an online manner. By that we mean that the input is read a character at a time, and for each character, as it is read, a constant amount of processing is performed.

One might wonder whether such simple devices could be useful. In fact, they are widely used as controllers in mechanical devices such as elevators and automatic doors (see pages 31-33 of Sipser). They are also very useful as a front end in many programs, where they are used to partition the input into meaningful tokens. For example, a compiler, which is a program that processes input programs so they can then be executed, needs to partition the input into keywords, variables, numbers, operators, comments, etc.

For example, we could imagine recognizing the keywords if, then, else, end with the help of the following device, represented as a graph.

![Graph Diagram]

The processing begins at the node start and follows the path labeled by the input characters as they are read. If there is no edge to follow, then the computation fails; it recognizes a keyword if it finishes at a node represented by a double circle.

There are more details to handle in a compiler. In particular, one has to be able to decide when one has reached the end of a keyword rather than being in the middle of a variable (e.g., ended) whose name begins with the keyword. But those essentially straightforward and tedious, albeit important, details are not the focus here.

Another set to recognize are variables. Let us suppose variables consist of all strings of (lower-case) letters other than keywords, and to simplify the illustration below let us suppose
there is just one keyword: if. Then the following device will recognize variables.

Notice the graph has a self-loop; also, some of its edges have more than one label (alternatively, we could think of this as a collection of parallel edges, each with a distinct label). Again, to use the device, one follows the path spelled out by the input. Any input other than the word “if” leads to a node represented as a double circle and hence represents a variable.

We now need to introduce a little notation and terminology. An alphabet is a set of characters. For example

i. Binary = \{0,1\};
ii. Boolean = \{T,F\};
iii. English = \{a,b,\ldots,z\}

The usual way of writing an unspecified \(k\)-character alphabet is as \(\Sigma = \{a_1,a_2,\ldots,a_k\}\).

A string is a sequence of zero or more characters drawn from the alphabet. For example: 
\(aa,\ \text{the},\ \text{bat},\ldots\). The zero character string, also called the empty string is usually written as \(\varepsilon\).

The concatenation of strings \(u\) and \(v\), written as \(uv\), is simply the characters of \(u\) followed by the characters of \(v\). For example, if \(u = \text{'beef'}\), \(v = \text{'root'}\), \(uv = \text{'beetroot'}\).

A finite automaton is conventionally named \(M\) or \(M_1, M_2, \ldots\) if there are several. As we will see subsequently, other devices will also be named by \(M\). A finite automaton is a directed graph plus an alphabet \(\Sigma\) in which:

i. One vertex is designated the start vertex; conventionally, this is vertex \(q_0\).
ii. There is a collection of Recognizing or Final vertices, conventionally called \(F\).
iii. Each edge is labeled by a character of \(\Sigma\).
iv. Each vertex has \(|\Sigma|\) outgoing edges, each one labeled by a distinct character.
$M$ processes an input string $s$ as follows: it determines the end of the path, beginning at $q_1$, the start vertex, obtained by following the labels spelled out by reading $s$. If the end of the path is a vertex in $F$ then $M$ recognizes or accepts $s$; otherwise $M$ rejects $s$.

Thus $M$ partitions the set of all possible strings into those it accepts and those it rejects. The topic we will study is what sorts of collections of strings finite automata can accept.

The set of strings accepted by $M$ is often called the language accepted by $M$, and is sometimes written as $L(M)$ or $L$ for short.

Conventionally, the vertices of a finite automata are called its states and are written as $Q = \{q_1, q_2, \ldots, q_n\}$ rather than $V = \{v_1, v_2, \ldots, v_n\}$. The index $n$ tends to be reserved for the length of the input string.

For simplicity in drawing we replace multiple parallel edges by a single edge with multiple labels:

Replace $\begin{aligned} \text{a} & \quad \text{with} \quad \text{a}, \text{b} \to \text{c} \end{aligned}$

Also, it is often convenient to omit a "sink" state (a non-final state which one cannot leave).

Use $\begin{aligned} \text{a} & \quad \text{with} \quad \text{a}, \text{b} \to \text{c} \end{aligned}$

Instead $\begin{aligned} \text{a}\ldots \text{z} & \quad \text{with} \quad \text{a}, \text{b}\ldots \text{z} \end{aligned}$

Finally, we need a notation for representing labeled edges. We write $\delta(q,a) = p$ to mean that there is an edge labeled $a$ from $q$ to $p$; a more active interpretation is that starting at vertex (state) $q$ on reading a one goes to vertex (state) $p$. In other words the destination on reading a in state $q$ is state $p$. Conventionally $\delta$ is called the transition function.

It is also useful to extend the destination (or transition) function to the reading of strings; this is denoted by $\delta^*$. $\delta^*(q,s) = r$ means that state $r$ is the destination on reading string $s$ starting in state $q$. 

3