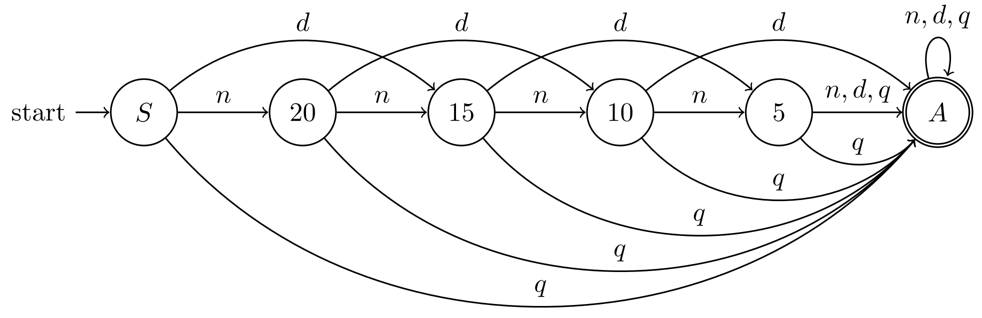
Deterministic Finite Automata
Theory of Computation (CSCI 3500)

Prof. Harley Eades (heades@augusta.edu).

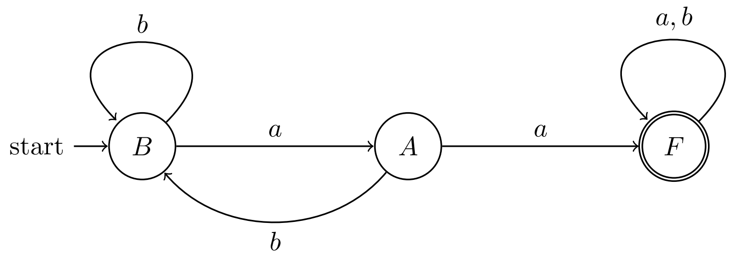
Read chapter 1.1.

1 Finite Automata by Example

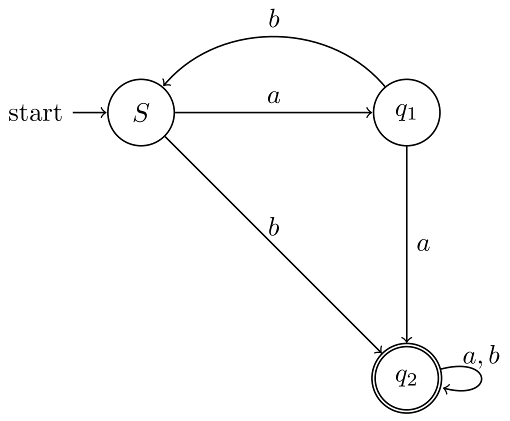
The newspaper vending machine:



Two consecutive a's:



Non-alternating subword:



2 Formal Deterministic Finite Automata

As we have seen deterministic finite automata are machine like models of computation that operate over words of a formal language. In this section we give the formal definition of deterministic finite automatas.

Definition 1. *A* ***deterministic finite automata (DFA)*** *is a 5-tuple* $(Q, \Sigma, \delta, q_0, F)$ *where:*

- Q *is a finite non-empty set of states,*
- Σ *is a finite set of alphabet symbols,*
- $\delta : Q \times \Sigma \to Q$ *is the transition function,*
- $q_0 \in Q$ *is the start state, and*
- $F \subseteq Q$ *is the set of final states.*

Lets consider how the DFA of the vending machine example is defined:

Example 2. *First, suppose:*

$$\begin{array}{rcl} Q & = & \{S, 20, 15, 10, 5, A\} \\ \Sigma & = & \{n, d, q\} \\ \delta & = & \begin{array}{l} \{((S,n),20), ((S,d),15), ((S,q),A), \\ ((20,n),15), ((20,d),10), ((20,q),A), \\ ((15,n),10), ((15,d),5), ((15,q),A), \\ ((10,n),5), ((10,d),A), ((10,q),A), \\ ((5,n),A), ((5,d),A), ((5,q),A) \\ ((A,n),A), ((A,d),A), ((A,q),A)\} \end{array} \\ F & = & \{A\} \end{array}$$

Then the vending machine DFA is defined by $V = (Q, \Sigma, \delta, S, F)$.

The graphical representation of a DFA is an informal representation, and it describes the transition function. A pair in the transition function, for example like, $((10, n), 5)$ is represented graphically by a labeled directed edge from state 10 to state 5 that is labeled by an n. Then a complete DFA is one in which every pair has been drawn in this fashion. If one starts from a DFA in graphical form, then it is easy to obtain the formal definition by simply reading off each component.

The transition function tells us how to move from one state to the next given a current state and an alphabet symbol, e.g. $\delta(S, q) = A$. We can extend this to a complete word over the alphabet of a DFA. For example, we could run the DFA V over the word *ddn* which should end in the state A. We define how this is done by extending the transition function into the run function for DFAs.

Definition 3. *Suppose $M = (Q, \Sigma, \delta, q_0, F)$ is a DFA. Then we define the* ***run function*** *as follows:*

$$\begin{array}{l} \tilde{\delta} : Q \times \Sigma^* \to Q \\ \tilde{\delta}(q, \epsilon) = q \\ \tilde{\delta}(q, aw) = \tilde{\delta}(q', w) \\ \quad\quad \textit{where } \delta(q, a) = q' \end{array}$$

Example 4. *Suppose $V = (Q, \Sigma, \delta, S, F)$ is the vending machine DFA as defined above. Then we can run V on the word ddn using V's run function:*

$$\begin{aligned} \tilde{\delta}(S, ddn) &= \tilde{\delta}(15, dn) \\ &= \tilde{\delta}(5, n) \\ &= \tilde{\delta}(A, \epsilon) \\ &= A \end{aligned}$$

Using the run function we can define when a word is accepted by a DFA, and what the language of a DFA is:

Definition 5. *Suppose $M = (Q, \Sigma, \delta, q_0, F)$ is a DFA. Then a word $w \in \Sigma^*$ is* ***accepted*** *by a DFA if and only if $\tilde{\delta}(S, w) = q_f \in F$.*

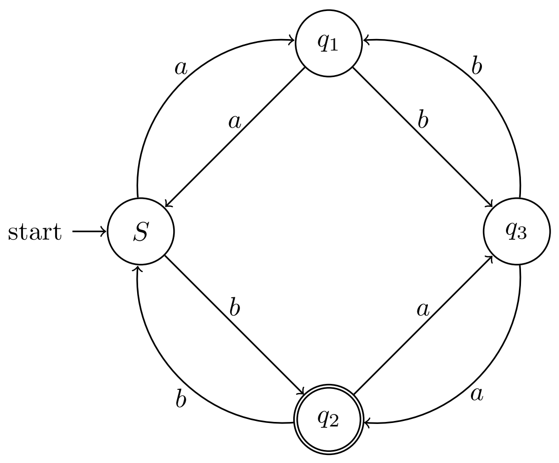
Definition 6. *The* ***language*** *of a DFA, $M = (Q, \Sigma, \delta, q_0, F)$, is defined as follows:*

$$L(M) = \{w \mid w \in \Sigma^* \textit{ and } \tilde{\delta}(q_0, w) \in F\}$$

That is, the language $L(M)$ is the set of all the words M accepts.

Here are some exercises.

- What is the language of the following DFA:



- Define a DFA that recognizes the following languages:

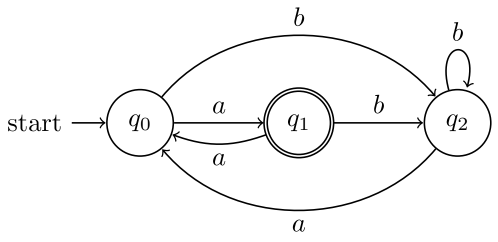
$L = \{w \mid w \in \{a,b\}^* \text{ and } w = ab^5wb^2\}$
$L = \{w \mid w \in \{a,b\}^* \text{ and } w = ab^na^m \text{ where } n \geq 2, m \geq 3\}$
$L = \{w \mid w \in \{a,b\}^* \text{ and every run of } w \text{ has length either two or three}\}$

3 Configurations and Traces of Finite Automata

A **configuration** of a deterministic finite automata (DFA) is a snapshot of where the automata is during computation. So for example, the string q_1aabb represents the configuration of some DFA that resides in state q_1 and is reading an a – the symbol just right of the state. In addition, the configuration shows the remainder of the input.

A **trace** of a DFA is a list of configurations that begins at the start state, and ends when all the input is consumed yielding either an accept state or a reject state.

Consider the following example DFA:



Now the complete trace of the above automaton on the word $abbaa$ is the following:

0. q_0abbaa
1. q_1bbaa
2. q_2baa
3. q_2aa
4. q_0a
5. q_1

We can see that at each step in the trace the configuration changes because the machine consumes a letter, and then moves to a new state. Traces also reveal the fact that finite automata use very little memory, because as we proceed we do not keep track of the alphabet symbols we have already seen. They are simply forgotten.

A word is accepted by a DFA if and only if its trace starts in the start state, and ends in a final state with all its input consumed. This last point is important, but is it possible for a DFA to not be able to consume all of the input? Also, what forces determinism in DFAs?

4 How do DFAs determine, and can they be partial?

The formal definition of a DFA stipulates that δ must be a function from the set $Q \times \Sigma$ to the set of states Q. A **total** function, $f : X \to Y$, is one in which for every element of X, there exists some element of Y, such that, $f(x) = y$. That is, all elements of the domain map to some element in the range. We call a function **partial** if there is an element in the domain that is not mapped to an element in the range. So if a function $f : X \to Y$ is partial, then there is some $x \in X$ such that $f(x)$ is actually undefined. Now if we stipulate that δ must be total, then that would imply that for every state in the DFA there is an outgoing edge for each input symbol. This latter point has actually been true for every example DFA we have seen thus far, and so one might wonder if this is always true. The answer is debatable, and it really depends on which computer scientist you ask. I personally feel that it is completely fine to make δ partial, because it does not affect the definition of a DFA, and there is always a unique choice for when to proceed or for when to fail which is the very definition of deterministic.

4.1 Partial DFAs

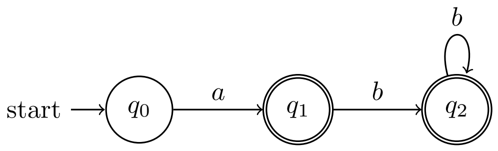
The definition of acceptance in a DFA is that the DFA ends in a final state, and all input is consumed. We made this formal by defining the run function:

Definition 7. *The* ***run function*** $\hat{\delta} : Q \times \Sigma^* \to Q$ *is defined as follows:*

$$\begin{array}{lcl} \hat{\delta}(q, \epsilon) & = & q \\ \hat{\delta}(q, aw) & = & \hat{\delta}(\delta(q, a), w) \end{array}$$

So now a word w is accepted by a DFA $M = (Q, \Sigma, \delta, q_0, F)$ if and only if $\hat{\delta}(q_0, w) = q_f$ and $q_f \in F$. The very fact that we were able to apply the run function to w and get as a result q_f means that all input was consumed, and we ended in a final state.

So does this definition of acceptance make sense when δ is partial? In fact, it does. Consider the following example:



Lets recheck our trace from above:

$$\begin{array}{ll} 0. & q_0 abbaa \\ 1. & q_1 bbaa \\ 2. & q_2 baa \\ 3. & q_2 aa \end{array}$$

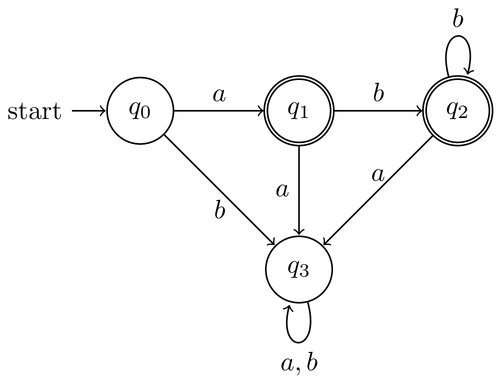
We can see that in the above example there is no transition from state q_2 to some other state when reading an a. So the above trace ends in state q_2, a final state, but still has input remaining, thus we must reject and issue failure.

What the previous example means formally is that the function

$$\delta = \{((q_0, a), q_1), ((q_1, b), q_2), ((q_2, b), q_2), ((q_2, a), a_2)\}$$

is still a function, but there are elements of $Q \times \Sigma$ that are not mapped to anything, namely, (q_0, b), (q_1, a), and (q_2, a). So we are still within the bounds of the formal definition. Because, if we were to define the run function for the example above, we would see that $\hat{\delta}(q_0, abbaa)$ is undefined, because $\delta(q_2, a)$ is undefined. Thus, we are forced to reject, because acceptance is defined as the run function producing an output, it cannot be undefined.

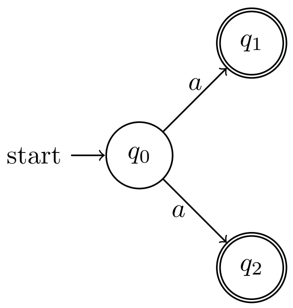
Now something interesting can be done at this point. It turns out that it is possible to convert any partial DFA into a total DFA. How would we do that? Well, any undefined elements need to be reject states. So we can simply add a sink (or trap) state for each of these undefined elements. So our previous example becomes:



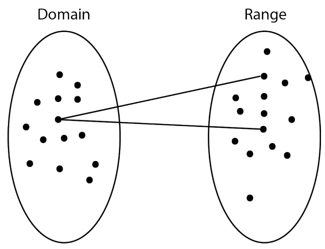
The above DFA is now total, and recognizes the same language as the partial version from above.

4.2 How do DFAs Determine?

One thing that is hidden in the definition of a DFA is what exactly makes the DFA deterministic. A finite automata is deterministic if for every edge there is at most one outgoing edge for any input symbol. The following is an example of a non-deterministic finite automata (NFA):



Does this diagram look familiar? Compare it to the following:

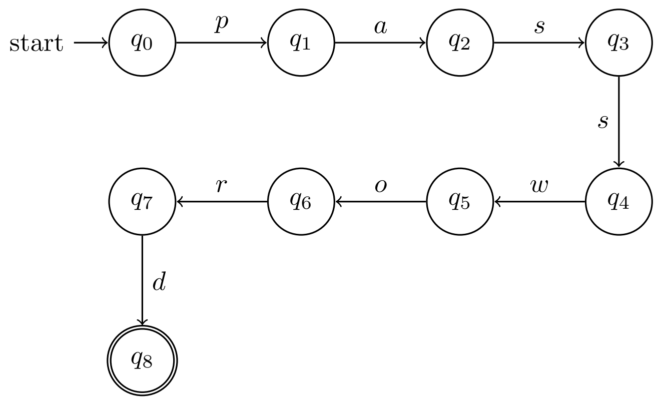


The second diagram is a counter-example to the property of being a function. So the property that is enforcing determinism is the property that the transition function δ is a function!

5 Password Verification using DFAs

In the next two sections we look at some applications of DFAs. Password verification is a natural application of a DFA, but how do we implement this?

First, lets consider using the formal model how we would verify a password using a DFA. Suppose someones password is “password” – the number two password used on the internet, see `http://www.cbsnews.com/news/the-25-most-common-passwords-of-2013/` – then a DFA recognizing “password” would be:



Clearly, this DFA will recognize only the password, and fail on anything else. So how do we implement this in general, and in such away that it can be programmed on a computer?

We first design how we can verify passwords in general using the mathematical definition, and then turn this into a program. In fact, this is how solving problems should be approached in general. First, start with a mathematical design of an algorithm, and then turn it into a program. Note that we must pay some attention to the structures we use, for example, the set $(A \times B) \times C$ is not equivalent to the function $A \to B \to C$ in most programming languages. If we represent a function using the former, then we must also define a way to run such functions.

Suppose we are given a password p. Then we first must compute a DFA recognizing p and only p. So we must define a DFA $M_p = (Q, \Sigma, \delta, q_0, F)$. The easiest parts are as follows:

$$\begin{array}{l}
Q = \{0, \ldots, |p| + 1\}, \text{ where } |p| \text{ is the number of characters in } p \\
\Sigma = \{c \mid c \text{ is a character of } p\} \\
q_0 = 0 \\
F = \{|p| + 1\}
\end{array}$$

Next we have to define the transition function for any p. To do this we define it as a subset of $(Q \times \Sigma) \times Q$:

$$\delta = \{((i, c_i), i + 1) \mid i \in \{0, ..., n\} \text{ and } p = c_0 \cdots c_n\}.$$

Since – in our program – δ will not be an actual function, but something equivalent to a function we need to define a way to apply δ. We can do this by simply defining the following function:

$$\begin{array}{l}
\mathsf{stepTrans} : (Q \times \Sigma) \times Q \to Q \to \Sigma \to (Q \cup \{\bot\}) \\
\mathsf{stepTrans}(\delta, q, c) = q' \\
\quad \text{if } ((q, c), q') \in \delta \\
\mathsf{stepTrans}(\delta, q, c) = \bot \\
\quad \text{otherwise}
\end{array}$$

The set $(Q \cup \{\bot\})$ represents either returning a state or returning undefined ($\bot$). Now using stepTrans we can define the run function:

$$\begin{array}{l}
\mathsf{run} : Q \to \Sigma^* \to (Q \cup \{\bot\}) \\
\mathsf{run}(q, \epsilon) = q \\
\mathsf{run}(q, (aw)) = \mathsf{run}(q', w) \\
\quad \text{if } \mathsf{stepTrans}(q, a) = q' \\
\mathsf{run}(q, (aw)) = \bot \\
\quad \text{otherwise}
\end{array}$$

At this point we have accomplished two things: how to do define a DFA that is amendable to being programmed on a computer, and how to generate a DFA given a password. So how do we verify if a string is the correct password?

Call a DFA that recognizes a password and is defined as above a PWD-DFA. Suppose M_p is the PWD-DFA for the password p. Then we can verify passwords on M_p as follows:

$$\begin{array}{l}
\mathsf{verifyPWD} : \mathsf{String} \to \mathbb{B} \\
\mathsf{verifyPWD}(s) = \mathsf{True} \\
\quad \text{if } \mathsf{run}_{M_p}(0, s) = q \in F_{M_p} \\
\mathsf{verifyPWD}(s) = \mathsf{False} \\
\quad \text{otherwise}
\end{array}$$

Now if one gives verifyPWD the correct password, then it will end in the final state, otherwise we return False. Notice that the latter is returned only when the input has the wrong alphabet, or is two short, or two long, because in both of these cases, the run function either returns a non-final state or undefined ($\perp$).

6 Automated Theorem Proving using DFAs

Another application of DFAs is automatically proving that two functions are equivalent[1]. Consider the following functions:

$$\begin{array}{lll}
f(0) = 0 & g(0) = 1 & h(0) = 0 \\
f(n+1) = g(n) & g(n+1) = f(n) & h(n+1) = 1 - h(n)
\end{array}$$

We can use a DFA to prove that $\forall n \in \mathbb{N}.f(n) = h(n)$. The following DFA captures the input-output behavior of the previous three functions for every element of $\{0,1\} \times \{0,1\} \times \{0,1\}$:

$$\begin{tikzcd}
0,0,0 \arrow[r, bend left, "1"] & 0,0,1 \arrow[l, bend left, "1"] & \text{start} \arrow[r] & 0,1,0 \arrow[r, bend left, "1"] & 1,0,1 \arrow[l, bend left, "1"] \\
1,0,0 \arrow[r, bend left, "1"] & 0,1,1 \arrow[l, bend left, "1"] & 1,1,0 \arrow[r, bend left, "1"] & 1,1,1 \arrow[l, bend left, "1"] &
\end{tikzcd}$$

Each state of the previous DFA stands for a particular output of the functions f, g, and h given some input. The input is written in unary form:

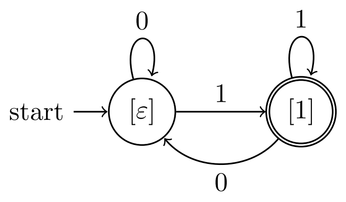
$$\begin{array}{l}
1 = 1 \\
2 = 11 \\
3 = 111 \\
4 = 1111 \\
5 = 11111 \\
\vdots
\end{array}$$

We have a transition from a state (a, b, c) to the state $(b, a, 1-c)$, and this represents applying each function. However, the initial state must be the base case. Note that $\hat{\delta}(1^n) = (f(n), g(n), h(n))$. Now to see if $f(n) = h(n)$ all we have to do is check to and make sure that $a = c$ across each transition starting from the initial state. This is clearly the case. This is called verifying an invariant.

[1]This example was based on Theirry Coquand's notes `http://www.cse.chalmers.se/~coquand/AUTOMATA/o2.pdf`

7 Useful Examples

1. DFA for $L = \{w \in \{0,1\}^* \mid w \text{ is odd}\}$:


0
1
start
[ε]
1
[1]
0