#1 Finite Automata, or: what does it mean to compute?

#1.1 Motivation

This lecture begins our study of the theory of computation. We will start with one of the simpler notions of computation we have available to us: finite automata. We will approach our study of automata from three equivalent angles. That is to say, in this class we will think of finite automata (and other models of computation, later) as either:

1. A special form of **state diagrams**, which give us a pictorial representation that is easy to reason about (as humans);
2. **mathematical models**, which give us a vocabulary by which we may state and deduce properties of automata; and as
3. **languages**, which we will see today are an equivalent representation of the previous bullet.

Before diving further, let's ask ourselves why we should study models of computation which are inherently limited. In particular, DFAs have no notion of memory. I open this thought to the class.

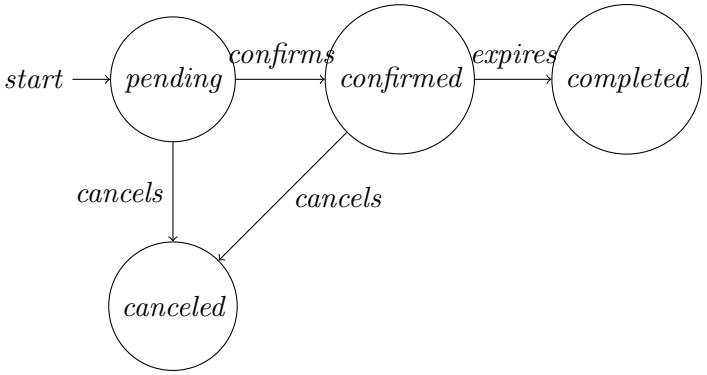
My answers are:

1. Sometimes we would like to consider hardware or environments in which memory is extremely limited;
2. studying computation *at its simplest* lets us begin to categorize the *expressivity* of successively sophisticated forms of computation; and, perhaps most importantly,
3. simpler things are more pleasant to work with.

#1.2 State Diagrams

State diagrams give us a nice way to visualize automata. The term state diagram means what you think it means: it's a collection of little nodes that represent states with *labeled* transitions. Additionally, we have some notion of a *start* state and a *set of end* states.

What might we model as state transition diagrams? Perhaps the life cycle of an AirBnB reservation.

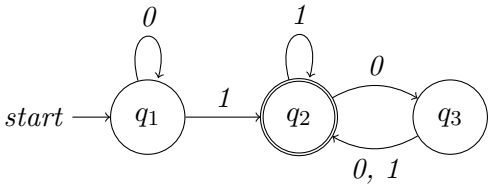


Example 1.

You can think of a DFA as a state diagram with some extra rules, namely:

1. You *must* specify the starting point (we did this above, but some state diagrams needn't have a start point);
2. You *must* specify a final set of *accept states*. Q: what could accept states be here?
3. The transitions must be *totally defined* on all nodes. See above that we do not know what the "confirms" command does to the "canceled" state.

The state diagram below is a proper DFA.



Example 2.

We have:

- states q_1, q_2, and q_3;
- *commands* or *events* 0 and 1;
- starting state q_1;
- and final state q_2.

Note too that each state is totally defined on all of the commands—for each vertex, we have an outward 0 edge and an outward 1 edge. It is standard for DFA diagrams to denote

1. the starting state with an in-arrow;
2. all accepting states with an extra outline (double circle); and
3. multi-arrows from q_i to q_j as one edge and a comma-separated list of labels. For example, we write $0, 1$ above to denote that there are actually *two* edges from q_2 and q_3.

#1.3 Strings, Languages, and Regularity

Consider strings of commands as *pathways* through the above diagram. That is, start at q_1, and follow a string such as 00101 through the diagram. The string 00101 yields the trace: q_1, q_1, q_2, q_3, q_2. In particular, it ends in an *accept state*.

1. When a string results in a path through the DFA that ends in an accept state, we say that the DFA *accepts* the string.
2. The set of strings accepted by a DFA D is called the *language* accepted by D, often written $L(D) = A$ for language A.

#1.3.1 Membership

Now, why is this considered computation? Much of this course will be concerned with questions of *membership*. That is, procedures that decide for us if a given element is in a given set. But this is precisely what a DFA tells us: whether or not a given string x is in the language L. Quite a lot can be explained this way—particularly later when we consider the question of membership using Turing Machines (and we consider strings in $\{0, 1\}^*$ to be binary-encoded natural numbers).

Note that membership can answer a lot of questions. For example, rather than ask if the string s is a palindrome, we might ask if it's an element of the set

$$\{s \mid s \text{ equals itself when reversed}\}$$

Set theory is a sufficiently expressive language that Most interesting questions can be rephrased into a question of membership.

#1.3.2 Regularity

You have probably heard the terms "regular expression" and "regular language" in the context of pattern matching on strings.

Definition 1 (Regular Language)**.** *A language A is called* regular *if there exists a DFA D that recognizes A.*

#1.4 DFAs and Computation, formally

Formally define a DFA as follows.

Definition 2 (Deterministic Finite Automata)**.** *A* Deterministic Finite Automata*, or* DFA*, is formally a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where*

1. *Q is a finite set of states;*
2. *Σ is a finite set called the* alphabet*;*
3. *$\delta : Q \times \Sigma \to Q$ is called* transition function*;*
4. *$q_0 \in Q$ is the* start state*;*
5. *$F \subseteq Q$ is a set of* accept states*.*

The state diagram above can be defined formally as

1. $Q = \{q_1, q_2, q_3\}$;
2. $\Sigma = \{0, 1\}$;
3. start state q_1; and
4. $F = \{q_2\}$

where we specify δ by the table

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

#1.4.1 Computation, formally

Back to what it means to "compute". Sipser defines computation as the trace of states an accepted string takes through a DFA. That is:

Definition 3 (Computation)**.** *Let M be a finite automaton and let $w = w_1 w_2 ... w_n$ be a string where each $w_i \in \Sigma$. We say that M* accepts *w if a sequence of states $r_0, r_1, ... r_n$ exists in Q with three conditions:*

1. $r_0 = q_0$,
2. $\delta(r_i, w_{i+1}) = r_{i+1}$ *for* $i = 0, ..., n-1$, *and*
3. $r_n \in F$

That is to say, we have the following property on the trace r:

1. it starts at the start state;
2. it steps in tandem with how δ acts on the input string w; and
3. it ends in an accept state.

#1.5 Regular Operations

An important part of this class is asking what benefit a formalization gives us. Our first results will be over what are called *regular operations* and *closure properties*. The premise is: we start with regular languages, A, B, C, and so forth. What can we do to them such that they stay regular? This will look familiar to what you can do with regular expressions, which is no coincidence.

The following operations are called *regular*:

1. Union: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$.
2. Concatenation: $A \circ B : \{xy \mid x \in A \text{ and } y \in B$.
3. Star (aka the *Kleene Closure*): $A^* = \{x_1 x_2 ... x_k \mid k \geq 0, x_i \in A\}$.

Note that the star operator above is over *languages*, not *alphabets*. This means that we are taking all finite strings formed from the *concatenation* of strings, not the sequencing of characters. A subtle difference.

Example 3. *Let* $A = \{"good", "bad"\}$ *and* $B = \{"boy", "girl"\}$. *Then*

$$\begin{aligned} A \cup B &= \{"good", "bad", "boy", "girl"\} \\ A \circ B &= \{"goodboy", "badboy", "goodgirl", "badgirl"\} \\ A^* &= \{\epsilon, "good", "bad", "goodgood", "goodbad", "badbad", ...\} \end{aligned}$$

Finally, we want to show that these operations preserve regularity. This style of claim (and proof) will be exceedingly common in this class.

Claim 1 (Union Regularity)**.** *Let* L_1 *and* L_2 *be regular languages. Then their union* $A \cup B$ *is regular.*

The full proof is given in the text. To give you more lab time, I will skip belaboring it in class. What I want you to remember, though, is the proof technique, which goes like this.

1. Observe that, because L_1 and L_2 are regular, we must have DFAs D_1 and D_2 that recognize L_1 and L_2.
2. Using these DFAs, construct a *new* DFA D'. Prove that D' recognizes $L_1 \cup L_2$.
3. Conclude: as $L_1 \cup L_2$ is accepted by some DFA, it is by definition regular.

We will be doing this style of proof by construction ad nauseam in this class. In particular, always observe the connection between languages and their machine counterparts. If a language is regular, there is some D for it—if there is some DFA D for a language, it is regular. You will get plenty of practice invoking this concept in your labs and homework this week.