1 Examples

How can we formally define what a program or algorithm is? One of the simplest models of computation is finite automata. Informally, a deterministic finite automaton (DFA) is an abstract machine that processes the input string from left to right, maintains only a constant amount of memory (independent of the input size), and accepts (outputs 1) or rejects (outputs 0) when it reaches the end of the input. We restrict our attention to input strings over the alphabet \( \{0, 1\} \), though any finite alphabet could be considered. Thus, each DFA computes a function \( f: \{0, 1\}^* \rightarrow \{0, 1\} \) (where \( \{0, 1\}^* \) is the set of all bit strings), though we will see later that not every such function can be computed by a DFA. Such a function can also be identified with a language: the set of all strings on which the function evaluates to 1 (which is why the study of DFAs is part of “formal language theory”).

Here is an example of a DFA (taken from Introduction to the Theory of Computation by Sipser).

![Example DFA Diagram]

There are states represented by nodes (this example has three), and each state must have two transitions (directed edges) coming out of it, one labeled 0 and the other labeled 1. (The lower-right edge labeled “0,1” in this example is just shorthand for two transitions that happen to have the same endpoints.) One state must be designated as the start state by having an arrow from nowhere (\( q_1 \) in this example), and an arbitrary subset of states are designated as accept states by having a double circle (\( q_2 \) is the only accept state in this example).

When this example DFA is run on input 010100, it goes through the sequence of states \( q_1, q_1, q_2, q_3, q_2, q_3, q_2, q_3 \) (it starts in \( q_1 \), then the first 0 makes it transition to \( q_1 \), then the following 1 makes it transition to \( q_2 \), etc.) and so it accepts since the computation ends in an accept state (\( q_2 \)). When run on input 11000, it goes through \( q_1, q_2, q_2, q_3, q_2, q_3 \) and so it rejects since the computation does not end in an accept state (the fact that it passed through an accept state along the way is irrelevant). Inspection reveals that this DFA accepts iff the input contains at least one 1 and has an even number of 0’s following the last 1.

Let’s design a DFA computing \( f: \{0, 1\}^* \rightarrow \{0, 1\} \) where \( f(x) = 1 \) iff \( x \mod 3 \neq 0 \) where \( x \) is interpreted as a nonnegative integer in base 2. We’ll use the states to keep track of the mod 3 value of the prefix of \( x \) we’ve seen so far. Assuming the empty string represents 0, we’ll start in the state for the value 0. The key observation is that

\[
x_1 \ldots x_{i+1} \mod 3 = ((x_1 \ldots x_i) \cdot 2 + x_{i+1}) \mod 3 = ((x_1 \ldots x_i \mod 3) \cdot 2 + x_{i+1}) \mod 3,
\]
so the next state \((x_1 \ldots x_{i+1} \mod 3)\) is uniquely determined from the current state \((x_1 \ldots x_i \mod 3)\) and the input bit about to be read \((x_{i+1})\). This leads to the following state diagram.

\[\begin{array}{c}
\text{0} & 1 & 0 \\
\text{\(q_0\)} & \text{\(q_1\)} & \text{\(q_2\)} \\
\text{1} & \text{0} & \text{1}
\end{array}\]

### 2 Minimization

Finite automata can form the basis of scanners in compilers, sequential digital logic, and network protocols, among other things. The fewer states a DFA has, the better, since this corresponds to the amount of memory needed to run a computation (and to the size of a description of the DFA). We will now design a polynomial-time algorithm for taking any DFA and converting it to an equivalent one with the minimum possible number of states. When we say two DFAs are equivalent, we mean they accept exactly the same set of strings (compute the same function). First, we assume without loss of generality that any states that are unreachable from the start state have already been removed from the DFA.

Let us define a certain *equivalence relation* on the states of a DFA; this means we partition the set of all the states into *classes* and say that two states are *equivalent* (written \(q_i \equiv q_j\)) iff they are in the same class. To describe our equivalence relation, let’s use the notation \(\delta(q_i, y)\) to mean the state the DFA would end up in if we started it in state \(q_i\) and fed it the bit string \(y\). We define:

\[q_i \equiv q_j \text{ iff for every } y \in \{0,1\}^*: (\delta(q_i, y) \text{ is an accept state iff } \delta(q_j, y) \text{ is an accept state}).\]

This means \(q_i\) and \(q_j\) are, in a sense, indistinguishable or redundant: once the DFA reaches either state, it doesn’t matter which one since the final output (acceptance or rejection) is guaranteed to be the same for both no matter what the rest of the input is. Intuitively, equivalent states can be collapsed into a single state.

It can be checked that this definition indeed partitions the set of states into classes, such that any two states in the same class are equivalent and any two states in different classes are not. *Assuming we have found this partitioning* for a DFA \(D\) (we will discuss shortly how to find it in polynomial time), we can define a minimal equivalent DFA \(D'\):

- Create a state in \(D'\) for each class of states in \(D\).
- The start state of \(D'\) corresponds to the class containing the start state of \(D\).
- Make a state of \(D'\) accepting if the class contains only accepting states of \(D\). Make a state of \(D'\) non-accepting if the class contains only non-accepting states of \(D\). (This covers all cases since accept and non-accept states of \(D\) cannot be equivalent, as witnessed by \(y = \text{the empty string}\).)
- For each class \(C_1\) and each bit \(z \in \{0,1\}\), the states of \(D\) that can be reached by following the \(z\)-transitions from states in \(C_1\) must all be equivalent to each other, say from class \(C_2\) (since if \(y\) witnesses \(\delta(q_i, z) \neq \delta(q_j, z)\) then \(zy\) witnesses \(q_i \neq q_j\)). Thus in \(D'\), we make the \(z\)-transition from the state corresponding to \(C_1\) go to the state corresponding to \(C_2\).
As an example, consider the following DFA.

For the moment, trust us that the equivalence classes are \{q_1, q_2\}, \{q_3, q_4\}, \{q_5, q_6\}, \{q_7\}, \{q_8\}, \{q_9\}. Then we can build an equivalent DFA with just six states, corresponding to the six classes. Since in the original DFA, all the 0-transitions out of \{q_1, q_2\} go into \{q_3, q_4\}, we make the 0-transition out of q_1, q_2 go to q_3, q_4. Since in the original DFA, all the 0-transitions out of \{q_3, q_4\} go into \{q_1, q_2\}, we make the 0-transition out of q_3, q_4 go to q_1, q_2, and so on.

To illustrate why these DFAs are equivalent, consider the input 10001100: the original DFA goes through q_1, q_2, q_4, q_1, q_3, q_5, q_7, q_6, q_8 and rejects; the new DFA goes through the same sequence but with each state “expanded” to its class: q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_5, q_6, q_8. In general, the construction guarantees that if D goes through some sequence of states q_{i_0}, q_{i_1}, \ldots, q_{i_n} on some input x \in \{0, 1\}^n, then D’ goes through the sequence of states C_0, C_1, \ldots, C_n on x, where C_j is the class containing q_{i_j} (and thus the DFAs are equivalent: D accepts x iff D’ accepts x iff q_{i_n} is an accept state).

In summary, the D’ we have derived from D is equivalent and no larger, and may be smaller. But is it the smallest equivalent DFA? Maybe some other DFA, not derived from D in any recognizable way, could be even smaller but still equivalent to D? In fact, our D’ is guaranteed to be the smallest. Why? Suppose the number of classes (states of D’) is s, and number them from 1 to s arbitrarily. Let x^{(1)}, \ldots, x^{(s)} \in \{0, 1\}^* be some inputs such that x^{(i)} leads D from the start state to a state in the i\textsuperscript{th} class. (Such inputs exist since we assume all unreachable states were already removed.) We claim that in any DFA \hat{D} whatsoever that’s equivalent to D, x^{(1)}, \ldots, x^{(s)} must lead to different states and thus there must be at least s states. Suppose for contradiction that x^{(i)} and x^{(j)} (for some i \neq j) lead to the same state \hat{q} in \hat{D}. Then for any y \in \{0, 1\}^*, \hat{D} accepts x^{(i)}y iff \hat{D} accepts x^{(j)}y iff \delta(\hat{q}, y) is an accept state of \hat{D}. Since D and \hat{D} are equivalent, this means D
accepts \( x^{(i)} y \) iff \( D \) accepts \( x^{(j)} y \), and so the states \( x^{(i)} \) and \( x^{(j)} \) lead \( D \) to (from the start state) must be equivalent—a contradiction.

We are left with the matter of efficiently finding the partitioning into the classes of \( \equiv \). To do this, we start with a “coarse” partitioning \( \equiv_0 \) and iteratively “refine” it (i.e., further partition each class into one or more classes) into finer and finer partitionings \( \equiv_1, \equiv_2, \ldots \) which converge to the true \( \equiv \). Specifically, the equivalence relation \( \equiv_k \) represents “indistinguishability up to \( k \) steps” (here, \( \{0,1\}^{\leq k} \) is the set of all bit strings of length at most \( k \)):

\[
q_i \equiv_k q_j \text{ iff for every } y \in \{0,1\}^{\leq k}: (\delta(q_i, y) \text{ is an accept state iff } \delta(q_j, y) \text{ is an accept state}).
\]

Note that \( \equiv_{k+1} \) is indeed a refinement of \( \equiv_k \), since if \( q_i \not\equiv_k q_j \) is witnessed by \( y \), then the same \( y \) witnesses \( q_i \not\equiv_{k+1} q_j \). Also note that since the number of classes can never exceed the number of states in \( D \), we must reach a value \( \ell \) such that \( \equiv_{\ell}, \equiv_{\ell+1}, \equiv_{\ell+2}, \ldots \), are all the same, in which case these must actually be the true \( \equiv \) (since the “\( \delta(q_i, y) \) is an accept state iff \( \delta(q_j, y) \) is an accept state” property holds for all \( y \in \{0,1\}^\ast \) iff it holds for all \( y \in \{0,1\}^{\leq k} \) for arbitrarily large \( k \)).

Note that \( \equiv_0 \) just partitions the states into two classes: accepting and non-accepting. Thus we need to efficiently (i) determine the partitioning of \( \equiv_{k+1} \) from the partitioning of \( \equiv_k \), and (ii) determine when the partitioning has converged in the sense that \( \equiv_k \) will never change again as we increase \( k \). Our procedure for (i) will have the important property that it does not depend on \( k \); this implies that as soon as we reach an iteration where \( \equiv_{k+1} \) is identical to \( \equiv_k \), it has converged since applying the procedure to derive \( \equiv_{k+2} \) from \( \equiv_{k+1} \) would result in no change (just as it resulted in no change going from \( \equiv_k \) to \( \equiv_{k+1} \)), and so on out to infinity. Moreover, this guarantees the number of iterations is at most the number of states of \( D \), so as long as our procedure for (i) is polynomial-time, the whole algorithm is polynomial-time.

Suppose the classes of \( \equiv_k \) are denoted \( C_1, C_2, \ldots, C_m \). Here’s how we can derive the classes of \( \equiv_{k+1} \). Each \( C_h \) gets further partitioned into \( m^2 \) parts which form classes in \( \equiv_{k+1} \) (some of which may be empty and hence get discarded from the list; this also allows the possibility that all parts but one are empty). Specifically, for each pair \( a, b \in \{1, \ldots, m\} \) we have a part \( C_{h,a,b} \) consisting of those states in \( C_h \) for which the 0-transition goes to a state in \( C_a \) and the 1-transition goes to a state in \( C_b \). The nonempty \( C_{h,a,b} \)’s indeed form classes of \( \equiv_{k+1} \):

- Suppose \( q_i, q_j \in C_{h,a,b} \). Then if we feed any \( y \in \{0,1\}^{\leq k+1} \) into \( D \) starting either at \( q_i \) or at \( q_j \), the first bit leads into \( C_a \) in both cases or into \( C_b \) in both cases (depending on whether the first bit is 0 or 1), and then the remaining \( \leq k \) bits of \( y \) lead to acceptance in both cases or rejection in both cases (since \( C_a \) and \( C_b \) are \( \equiv_k \) classes). Thus \( q_i \equiv_{k+1} q_j \).

- Conversely, suppose \( q_i \) and \( q_j \) are in different parts \( C_{h,a,b} \) and \( C_{h',a',b'} \) respectively. If \( h \neq h' \) then some \( y \in \{0,1\}^{\leq k} \) already witnesses \( q_i \not\equiv_k q_j \) (in particular, \( q_i \not\equiv_{k+1} q_j \)) since \( C_h \) and \( C_{h'} \) are \( \equiv_k \) classes. If \( h = h' \) and \( a \neq a' \) (the case \( b \neq b' \) is analogous), then the input bit 0 leads \( q_i \) and \( q_j \) into \( C_a \) and \( C_{a'} \) respectively, and then some \( y \in \{0,1\}^{\leq k} \) leads them to different outputs (acceptance for one and rejection for the other, since \( C_a \) and \( C_{a'} \) are \( \equiv_k \) classes). That is, one of \( \delta(q_i, 0y) \) and \( \delta(q_j, 0y) \) is an accept state and the other is not, so \( q_i \not\equiv_{k+1} q_j \).

That’s it! Intuitively, this is just saying that \( q_i \) and \( q_j \) and distinguishable within \( k + 1 \) steps iff some single step can lead them to states that are distinguishable within \( k \) steps.

Let’s illustrate the whole process by applying it to our running example. We start by partitioning into accept and non-accept states:
Let’s call these classes \( C_1 \) and \( C_2 \) respectively. The procedure yields (recalling that \( C_{h,a,b} \) consists of the states in \( C_h \) with 0-transitions into \( C_a \) and 1-transitions into \( C_b \), for \( h, a, b \in \{1, 2\} \)):

\[
\begin{align*}
C_{1,1,1} &= \{q_1, q_2\} & C_{1,1,2} &= \{q_3, q_4, q_9\} & C_{1,2,1} &= \{q_7\} & C_{1,2,2} &= \emptyset \\
C_{2,1,1} &= \emptyset & C_{2,1,2} &= \emptyset & C_{2,2,1} &= \{q_5, q_6, q_8\} & C_{2,2,2} &= \emptyset
\end{align*}
\]

That is, \( C_2 \) is unchanged but \( C_1 \) gets split into \( \{q_1, q_2\} \), \( \{q_3, q_4, q_9\} \), and \( \{q_7\} \). Discarding the empty parts, we have:

\[
\equiv_1 \text{ classes: } \{q_1, q_2\}, \{q_3, q_4, q_9\}, \{q_7\}, \{q_5, q_6, q_8\}
\]

Let’s recycle notation by calling these classes \( C_1, C_2, C_3, C_4 \) respectively. We won’t list all 64 of the new \( C_{h,a,b} \)’s, just the nonempty ones:

\[
\begin{align*}
C_{1,2,1} &= \{q_1, q_2\} & C_{2,1,4} &= \{q_3, q_4\} & C_{2,3,4} &= \{q_9\} \\
C_{3,4,2} &= \{q_7\} & C_{4,4,3} &= \{q_5, q_6\} & C_{4,4,2} &= \{q_8\}
\end{align*}
\]

That is, \( C_1 \) and \( C_3 \) are unchanged, but \( C_2 \) gets split into \( \{q_3, q_4\} \) and \( \{q_9\} \), and \( C_4 \) gets split into \( \{q_5, q_6\} \) and \( \{q_8\} \).

\[
\equiv_2 \text{ classes: } \{q_1, q_2\}, \{q_3, q_4\}, \{q_9\}, \{q_7\}, \{q_5, q_6\}, \{q_8\}
\]

Another iteration of the procedure reveals that the classes don’t change going from \( \equiv_2 \) to \( \equiv_3 \), so we’ve found the \( \equiv \) classes (matching what we originally claimed).

In summary, the algorithm for minimizing a DFA is:

1. Initialize the two classes of \( \equiv_0 \) as the set of accept states and the set of non-accept states.
2. For \( k = 0, 1, 2, \ldots \) determine the classes of \( \equiv_{k+1} \) from the classes of \( \equiv_k \) as described above.
3. After the first iteration where \( \equiv_k \) and \( \equiv_{k+1} \) have exactly the same classes, use this as the \( \equiv \) partitioning to form the minimal DFA as described earlier.

### 3 Limitations

A function \( f: \{0,1\}^* \to \{0,1\} \) (or equivalently, the language consisting of its 1-inputs) is called **regular**\(^1\) if there exists a DFA that computes it. We will now see how to use ideas from the previous section to prove that even some fairly simple functions turn out to be nonregular.

To start, recall how we defined two states being equivalent with respect to a DFA. We similarly define two inputs \( x, x' \) to be equivalent with respect to a function \( f: \{0,1\}^* \to \{0,1\} \) iff for every \( y \in \{0,1\}^* \), \( f(xy) = f(x'y) \). Observe that for any DFA computing \( f \), if \( x, x' \) lead to equivalent states

\[^1\]This terminology was chosen a long time ago for no particular reason, and it stuck.
then $x, x'$ are equivalent with respect to $f$ (formally, if they lead to states $q_i \equiv q_j$ then: $f(xy) = 1$ iff $\delta(q_i, y)$ is an accept state iff $\delta(q_j, y)$ is an accept state iff $f(x'y) = 1$). Contrapositively, if $x, x'$ are inequivalent then they lead to inequivalent (in particular, distinct) states in any DFA computing $f$.

Thus, to show that $f$ is nonregular, it suffices to exhibit an infinite set of inputs $x^{(1)}, x^{(2)}, \ldots$ that are pairwise inequivalent with respect to $f$. This is because in any DFA computing $f$, these inputs would lead to distinct states, so there would need to be infinitely many states, contradicting the “F” in DFA. Let’s apply this method to show that various functions are nonregular.

- Suppose $f(x) = 1$ iff $x$ has the same number of 0’s as 1’s. The inputs 0, 00, 000, 0000, \ldots are pairwise inequivalent with respect to this $f$. To see this, consider any two of them, say 0$^i$ (i.e., $i$ 0’s) and 0$^j$ (i.e., $j$ 0’s) where $i \neq j$. Then $y = 1^i$ witnesses that they are inequivalent, since $f(0^i1^i) = 1$ but $f(0^j1^j) = 0$. Intuitively, $f$ is nonregular since a DFA should need to “remember” how many 0’s and/or 1’s it has seen, which is an unbounded amount of information—but that is not a rigorous proof. Indeed, consider the function that returns 1 iff the input has the same number of occurrences of 01 and 10—the same intuition might suggest that this is nonregular, but in fact it is regular (convince yourself!)

- Suppose $f(x) = 1$ iff $x$ is a palindrome. The inputs 01, 001, 0001, \ldots are pairwise inequivalent with respect to this $f$. To see this, consider any two of them, say 0$^i$1 and 0$^j$1 where $i \neq j$. Then $y = 0^i$ witnesses that they are inequivalent, since $f(0^i10^i) = 1$ but $f(0^j10^j) = 0$.

- Suppose $f(x) = 1$ iff $x$ consists of any number of 0’s followed by a larger number of 1’s. The inputs 0, 00, 000, \ldots are pairwise inequivalent with respect to this $f$. To see this, consider any two of them, say 0$^i$ and 0$^j$ where $i < j$. Then $y = 1^{i+1}$ witnesses that they are inequivalent, since $f(0^i1^{i+1}) = 1$ but $f(0^j1^{i+1}) = 0$ (because $i + 1 \leq j$).

- Suppose $f(x) = 1$ iff the length of $x$ is a perfect square. Then the inputs 1$^2$, 1$^2$, 1$^3$, \ldots are pairwise inequivalent with respect to this $f$. To see this, consider any two of them, say 1$^2$ and 1$^2$ where $i < j$. Then $y = 1^{2i+1}$ witnesses that they are inequivalent since $f(1^2y) = 1$ (because $i^2 + 2i + 1$ is the perfect square $(i + 1)^2$ but $f(1^{j^2}y) = 0$ (because $j^2 + 2i + 1 > j^2$ and $< j^2 + 2j + 1 = (j + 1)^2$ and is thus not a perfect square).

4 Nondeterminism

In a DFA, each state must have exactly one outgoing transition for each symbol of the input alphabet (which we’re assuming is \{0, 1\} for convenience). In a nondeterministic finite automaton (NFA), for each state and each symbol there can be transitions to an arbitrary subset of states (zero or more, as opposed to exactly one in the case of a DFA). Here’s an example.

\[\text{Look up “Myhill–Nerode Theorem” for more information on this.}\]
Note that \( q_2 \) has two 1-transitions (to \( q_2 \) and \( q_3 \)), meaning that if the NFA reads a 1 in the input while in state \( q_2 \), the NFA “guesses” whether to stay in \( q_2 \) or go to \( q_3 \). In such a case, the computation is said to split into multiple “computation paths”, and further splits down the line can result in a whole tree of possible computation paths. Note that \( q_3 \) has no 1-transitions, which means that if the NFA reads a 1 while in state \( q_3 \), it becomes “stuck”, and the current computation path fails to finish reading the input. For example, the input 01101 has four computation paths: \( q_1, q_2, q_2, q_1, q_1, q_2, q_3, q_2, q_2, q_3, q_2, q_3, q_2, q_3, q_1, q_2, q_3 \) (the last path gets stuck). The input 1110 has three computation paths: \( q_1, q_2, q_2, q_1, q_1, q_2, q_3, q_2, q_3, q_2, q_1, q_2, q_3 \) (the last path gets stuck).

In general, an NFA is said to accept an input if there exists a computation path that accepts after reading the whole input (not getting stuck). Our example NFA accepts 01101 since the third computation path we listed reads the whole input and ends in the accept state \( q_3 \), but the NFA rejects 1110 since all the computation paths either get stuck or lead to a non-accept state.

Every function computable by an NFA is regular, because every NFA can be converted to an equivalent DFA. The idea is to have a state of the DFA keep track of the entire subset of states the NFA could possibly be in at the moment. Specifically, to carry out this conversion, create a state in the DFA for each subset \( S \) of states in the NFA, and have a 0-transition to the subset of NFA states that are reachable from \( S \) via a 0-transition (and similarly for 1-transitions). The start state of the DFA is the singleton set containing the start state of the NFA, and the accept states of the DFA are those subsets that contain at least one accept state of the NFA.

Applying this construction to convert our example NFA to a DFA, we will have states \( q_1, q_2, q_2, 3, q_1, 2 \) representing the subsets \( \{q_1\}, \{q_2\}, \{q_2, q_3\}, \{q_1, q_2\} \) respectively (in general, states representing the other subsets \( \emptyset, \{q_3\}, \{q_1, q_3\}, \{q_1, q_2, q_3\} \) would also need to be included, but in our particular example they are unreachable from the start state, so we safely omit them).

For example, the 0-transition out of \( q_{2,3} \) goes to \( q_{1,2} \) since using a single 0-transition in the NFA, \( q_1 \) is reachable from \( q_2 \), and \( q_2 \) is reachable from \( q_3 \), but \( q_3 \) is reachable from neither \( q_2 \) nor \( q_3 \).

This DFA makes it slightly easier (than the NFA does) to see what function is being computed: it accepts iff the input has a 1 in an even-index position and ends with a 1. Also, note that this function cannot be computed by a DFA with fewer than four states, since the states of the above DFA are pairwise inequivalent (\( q_{2,3} \) is distinguishable from the others by \( y = \text{the empty string}; q_1 \) and \( q_2 \) are distinguishable by \( y = 1; q_1 \) and \( q_{1,2} \) are distinguishable by \( y = 1; q_2 \) and \( q_{1,2} \) are distinguishable by \( y = 01 \)). This shows that, although DFAs and NFAs are equally powerful in terms of which functions they can compute, they are not equally powerful in terms of size: three-state NFAs can compute functions that three-state DFAs cannot. In fact, the NFA to DFA conversion generally results in an exponential blow-up in the number of states, and it is possible to show that for some functions, this exponential blow-up is necessary (which is kind of like a \( P \neq \text{NP} \) result for finite automata size).

It is also common to define NFAs allowing so-called \( \varepsilon \)-transitions (\( \varepsilon \) is notation for the empty string), which can be used “for free” at any time, without consuming the next bit of the input.
Such ε-transitions can be very handy when designing an NFA, but we leave it as an exercise to convince yourself that they are inessential, in the sense that they can be removed (bypassed) without increasing the number of states of the NFA.

Speaking of number of states: we know how to efficiently minimize the number of states in any given DFA—how about for NFAs? Although we will not prove so, this turns out to be a so-called PSPACE-complete problem (which is an even higher level of complexity than NP-complete).