600.405 — Finite-State Methods in NLP
Assignment 1: Getting Started

Solution Set

Prof. J. Eisner — Fall 2000

1. *Correction:* The problem should have said $\delta : Q \times \Sigma \to Q$, not $\delta : Q \times \Sigma \to \Sigma$. Most of you caught this; apologies to those of you who were perplexed by it.

   Remark: The math notation $\delta : Q \times \Sigma \to Q$ corresponds to the C function prototype

   ```
   state delta(state q, symbol a);
   ```

   where the type `state` takes values in Q and `symbol` takes values in Σ. In other words, it says that δ is a function whose arguments are pairs of the form (q, a) where $q \in Q$ and $a \in \Sigma$, and which returns something in Q.

   Remark: The 5-tuple $(\Sigma, Q, I, F, \delta)$ encodes an automaton, so it is basically a mathematical version of a data structure. There are at least 3 reasonable data structures for describing the arcs of a finite-state automaton:

   - An *edge list*: a list L of edges like (q_i, a, q_j), meaning that there is an arc from q_i to q_j with label a. This is rather inefficient for most operations.
   - An *adjacency matrix*: A version of the edge list, but stored in a 3-dimensional array E for fast lookup. Put $E[i, a, j] =$**true** or **false** according to whether $(q_i, a, q_j) \in L$. (You may have seen a 2-dimensional version of this to represent unlabeled directed graphs.) We will actually use this representation in lecture 3, replacing the **true** and **false** here with arbitrary weights from a semiring.
   - A *transition function*: A compressed version of the adjacency matrix that doesn't bother to store all the **false** values (missing arcs). This saves both space and time if there are many missing arcs. For every state q_i and symbol a, the entry $\delta[i, a]$ stores just the state number(s) j such that $E[i, a, j] =$ **true**. Notice that in

addition to being compact, this is a very convenient representation if you are at state q_i, you read symbol a, and you want to know where to go next![1]

The transition function δ specified in the problem corresponds to the efficient last implementation option above. It's also a mathematically convenient way of defining automata, and it makes it easy to distinguish between a deterministic and a nondeterministic automaton (part of the point of this problem!).

Okay, now for the answers!

(a) Allow the transition function to be any $\delta : Q \times \Sigma \rightarrow Q \cup \{\textbf{undef}\}$.

Why? In an incomplete automaton, we must allow for the possibility that $\delta(q, a)$ is undefined: there might be no way to get from state q to any next state on input a. In this case we want $\delta(q, a)$ to return some special symbol **undef** $\notin Q$.

Remark: Most DFAs that arise in practice are incomplete. However, it is often useful to assume completeness in proofs, and the minimization construction only applies to complete DFAs. Fortunately, any DFA can be completed by adding the missing transitions: these transitions can go to a special "dead state" that is not final and just loops back to itself on any input, so that it does not accept anything.

(b) Allow the transition function to be any $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$.

Why? In a nondeterministic automaton, $\delta(q, a)$ might return multiple next states—i.e., a subset of Q rather than just one element of Q. Remember that $\mathcal{P}(Q)$ (sometimes written as 2^Q) denotes the set of all subsets of Q.

Notice that this definition obviously allows incompleteness, since $\delta(q, a)$ could return the empty set. Generally the term "complete" is only used when discussing deterministic automata.

(c) Allow the transition function to be any $\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow Q$.

[1] As a practical matter, it is common to distribute the storage of δ across states. Each state q_i stores a single array δ_i, and the value $\delta[i, a]$ is actually stored in $\delta_i[a]$. Advantages:

– No disadvantage: It is still easy for q_i to decide where to go next by consulting δ_i.

– Good cache behavior: δ_i will still be in the cache if q_i has been visited recently.

– Flexibility: Suppose the automaton is incomplete. If a given state q_i has few outgoing arcs, then δ_i is a sparse array. We can choose separately for each q_i whether to store δ_i as an array (fast lookup), as a linked list (compact storage and fast iteration), as both (extra storage but everything is fast), or as a hash table (a compromise). The decision depends on the sparsity of δ_i and how often we perform lookup and iteration operations on it.

Then not only $\delta(q, a)$ but also $\delta(q, \epsilon)$ is defined, so there are ϵ-labeled transitions. Note that the automaton is now nondeterministic, since if it is at state q and the next input symbol is a, it has a choice of reading a and going to $\delta(q, a)$, or reading nothing (yet) and going to $\delta(q, \epsilon)$.

To get a completely nondeterministic automaton with ϵ transitions, we would declare $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$.

(d) Allow the transition function to be any $\delta : Q \times \Sigma \to Q \times K$, where K is the set of output strings or weights.

So δ returns both a next state and an element of K. If $\delta(q_1, a) = (q_2, 5)$, this means that the a arc from q_1 goes to q_2 and has weight 5.

Another answer is to use two transition functions, $\delta : Q \times \Sigma \to Q$ (for the arc's destination) and $\delta : Q \times \Sigma \to K$ (for the same arc's weight or output). This is also correct but it would be harder to adapt this solution to nondeterministic automata.

(e) How many 5-tuples fit the conditions of the definition? We were given Σ and Q. We have an n-way choice for the initial state i. For each of the n states, we have a 2-way choice about whether to make it final or not. Finally, of each of the nk input pairs to δ, we have an n-way choice for the output.

Since all these choices are independent, there are $n \cdot 2^n \cdot n^{nk} = 2^n \cdot n^{nk+1}$ different options altogether. Asymptotically, it is $e^{n \log 2 + (nk+1) \log n} = e^{O(n \log n)}$ different automata—quite a lot!

(f) $\star$ In fact, "almost all" of these $e^{O(n \log n)}$ automata describe distinct languages. We'll show a lower bound: there must be at least $n^{n(k-1)}$ different languages recognized by these automata. Since $k > 1$ by assumption, this is fully $e^{O(n \log n)}$ distinct languages.

Let a be some arbitrary symbol in Σ. We will consider the set S of all automata with the following properties:

- initial state $I = q_1$
- final stateset $F = \{q_n\}$
- for all $i = 1, 2, \ldots n$, transition $\delta(q_i, a) = q_{\min(i+1,n)}$.
  (That is, $\delta(q_1, a) = q_2, \delta(q_2, a) = q_3, \ldots \delta(q_{n-1}, a) = q_n$, and $\delta(q_n, a) = q_n$.)

Because we are free to choose all the transitions on symbols other than a, we have an n-way choice for each of $n(k-1)$ of the input pairs to delta, so there are $n^{n(k-1)} = e^{O(n(k-1) \log n)}$ automata in this set. This is the same formula as in (1e) except that it only counts $k - 1$ of the k symbols in the alphabet (since symbol a is not free).

It remains to be shown that all these automata describe different languages. For this we need two key results about minimal automata (the assignment told you where to find these).

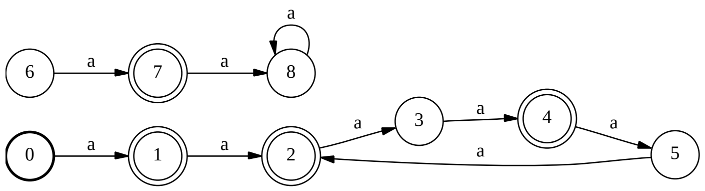
First of all, an automaton is minimal iff no states can be merged. Every automaton in S is minimal: the two states q_i and q_j (for $i < j$) can never be merged because they are distinguishable—the suffix a^{n-j} can be accepted from q_j but not from q_i.

Second, the minimal automaton for a language is unique up to isomorphism (renaming of the states). Since any two automata in S are minimal, they can only describe the same language if they are isomorphic. But renaming the states of an automaton in S never gives a different automaton in S, since the properties of S force the initial state to be named q_0, force $\delta(q_0, a)$ to be named q_1, etc. It follows that all the automata in S describe distinct languages, as desired.

(g) $\star\star$ The construction above doesn't provide a tight lower bound for the case $k = 1$. (In fact, it says only that there must be at least 1 language; by completely constraining the transitions on $a \in \Sigma$, it only allows one automaton!) So let's find better bounds.

The key insight is that for $k = 1$, the path from the initial state loops back on itself after some number of steps j, giving a "lollipop" topology. (There may be multiple lollipops but only one can be reached from the start state; the others are irrelevant!) Here is an example with $j = 6$, which accepts the language $a + aa(aa)*$:

(a single disconnected DFA)



Why is this? Given any complete deterministic n-state automaton on $\Sigma = \{a\}$, define $s_0 \in Q$ to be the initial state, $s_1 = \delta(s_0, a) \in Q$, $s_2 = \delta(s_1, a) \in Q$, etc. Since Q is finite, eventually we will run out of states: let j be the smallest integer such that $s_j = s_i$ for some $i < j$. (In the example above, $s_6 = s_2 =$ state 2.) Notice that $j \leq n$.

The language recognized by the automaton is completely determined by j, i, and the choice of final states among $s_0, s_1, \ldots s_{j-1}$. Since $i, j \leq n$, this immediately gives a weak upper bound of $n^2 \cdot 2^n$, which is $e^{O(n)}$.

Can we match this asymptotically with an $e^{O(n)}$ lower bound? Yes. Consider just automata for which $i = 0$ and $j = n$, so that the n states form a simple cycle. There are 2^n such automata, which differ according to the choice of final states. Any two such automata recognize different languages, since they must disagree on the finality of some state s_p, and then one accepts the string a^p while the other does not.

The problem is rather harder if one allows nondeterministic automata (which was actually what I had in mind, hence the $\star\star$ rating, but of course I actually asked about deterministic ones!).

To summarize the answers to the last two questions, there are $e^{O(n)}$ different languages recognized by n-state DFAs if the alphabet has size $k = 1$, or $e^{O(n \log n)}$ if the alphabet has size $k > 1$ (indeed the exponent increases linearly with k).

(Technically speaking, I should have been writing $\Theta()$ rather than $O()$ throughout this answer, since I'm giving exact asymptotic behavior and not merely an upper bound. Alas, this distinction is commonly ignored outside the algorithms community.)

2. (no right answer so long as you wrote something that showed you did the tutorial, in which case you got full credit!)

3. (a) $1(0+1)^*00$: correct except that it rejects 0. Note that the author of this expression decided that 0100 shouldn't count as a binary number because it starts with 0.

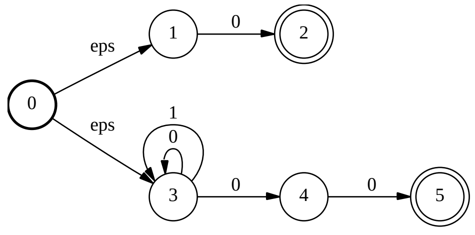
   (b) $(0+1)^*100$: rejects all numbers divisible by 8 (e.g., 1000).

   (c) $(0+1)^*00+0$: correct (unless you think that ϵ is an allowable binary representation of 0).

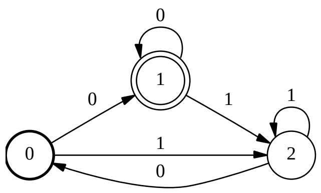
   (d) ^{*00}: not a regular expression. (I suspect the author was using * in the Unix shell sense of ?*, i.e., any sequence of characters.)

   (e) $(1^*0^*)^*00$: equivalent to $(0+1)^*00$, which is correct except that it rejects 0.

4. The following automaton corresponds to (3c):

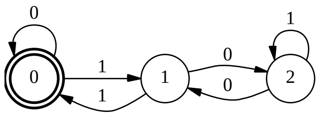


And here it is determinized, using `fsmdeterm`:



5. (a) wrong: rejects 11
   (b) wrong: rejects 1001
   (c) right: keeps track of (odd sum $-$ even sum $\pmod 3$), and odd vs. even position in string
   (d) right: merges two states from previous answer. These states can be regarded as (odd sum $-$ even sum $\equiv 0 \pmod 3$, odd position) and (odd sum $-$ even sum $\equiv 0 \pmod 3$, even position).
   (e) wrong: accepts 10

6. (a)



   (b) The refpixes that reach state i are the binary numbers that are congruent to i (modulo 3).

   Let's write $\langle w \rangle$ to mean the number represented by string w. So the claim is that if $\langle w \rangle \equiv i$, then w reaches state i.

   This claim can be justified by induction on the length of the refpix string. The claim is true for ϵ. How about longer strings? To show just one of six cases, suppose the longer string has the form $w0$ and $\langle w \rangle \equiv 2 \pmod 3$. Then $\langle w0 \rangle \equiv 2\langle w \rangle \equiv 2 \cdot 2 \equiv 1 \pmod 3$, so we must show that $w0$ reaches state 1. This is true because w reaches state 2 by inductive hypothesis, and there is a 0 arc from 2 to 1. The other cases are similar.

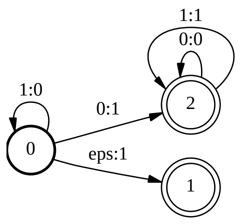
   A consequence of the claim is that refpixes that are divisible by 3 reach the final state, 0, and others do not (since the automaton is deterministic). So the automaton correctly tests divisibility by 3.

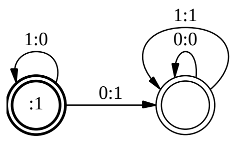
(c) $\star$ In general, divisibility by k in base b can be decided by a k-state DFA. The states are numbered $0, 1, \ldots k-1$; a refpix that is congruent to i (modulo k) reaches state i. This is arranged by the following arcs: $\delta(i, d) = j$ iff $i \cdot b + d \equiv j \pmod{k}$. (This leads to an attractively regular pattern.) The proof of correctness is as above.

The above automaton is not always minimal. For example, we saw in problem (4) that the case $k = 4, b = 2$ can be handled by a DFA with fewer than 4 states. In that machine, states 0, 1, and 2 respectively correspond to refpixes that are congruent to 2, 0, and either 1 or 3. In other words, the "1" and "3" states have been merged.[2]

If k and b are relatively prime, then the k-state automaton is minimal (it can be shown that any state pair is distinguishable). Otherwise it is not unless $k = 2$. I've checked the minimal size for a variety of k and b, and the pattern seems to be complicated, although there are subpatterns (based on the prime factors of k and b) that are easy to pick out.

7. (a) which could also be written





If the automaton is in the first state, it still has to add 1 to the rest of the string, either because it is in the start state or it is propagating a carry bit. Notice that runs out of input in the first state, it still has the obligation to output a 1 (e.g., $111 + 1 = 1000$); hence the final-state output at this state, which makes the transducer *subsequential* rather than sequential.

If the automaton is in the second state, it has discharged its obligation and can just copy the rest of the string.

(b) An equivalent regular expression is $(1:0)^*((\epsilon:1) + (0:1)(?:?)*)$. You can think of this as zeroing out the initial string of 1's and then adding 1 to what's left, which is easy because it's either ϵ or starts with 0.

(c) The reversed transducer cannot be determinized. In fact, the `fsmdeterminize` program does not halt—it deliberately omits the (polynomial but expensive)

[2] The start state is 0 rather than 1 because the author of regular expression (3c) decided not to allow ϵ as a binary representation of 0.

check for this condition.

Why is determinization impossible? It's because the transducer needs an *unbounded* amount of input lookahead to produce the correct output. (Note that in general a transducer can get a *bounded* amount of lookahead by postponing the output: i.e., have ϵ outputs for a while until it gets to a good state, then produce the postponed output. Essentially it is using the states to keep track of the postponed output—but there are limits to this as it has finitely many states!)

To be formal about this case, suppose we had a deterministic (sequential) version. Because it's deterministic, it reads a string on only one path, so that path must produce the correct output. Consider that $01^{n-2}0 \mapsto 01^{n-2}1$, whereas $01^{n-2}1 \mapsto 10^{n-1}$. On these examples, it cannot produce the correct first output symbol until it has read n input symbols—so it must wait till then to produce any output at all, and must then produce exactly n output symbols. But with only finitely many states, it cannot possibly remember (for arbitrary large n) how many output symbols to produce. So there is no such deterministic transducer.

Even though the reversed transducer is not determinizable, it has only one output per input (the word I was looking for was "**unambiguous**"). It may take multiple paths, but only one will be consistent with the rest of the input; the others will not reach a halt state.

(d) The original transducer recognizes the relation $\{(w, w+1) : w \geq 0\}$.[3] The inverted transducer therefore recognizes the relation $\{(w+1, w) : w \geq 0\}$. This relation is a function on *positive* natural numbers. It does not map 0 to anything, because 0 does not have the form $w+1$ for any $w \geq 0$.

It is worth noting that the inverted relation is *not* unambiguous, because the original relation accepts multiple representations of the same number. The inverted transducer maps 1 to both 0 and ϵ, and it maps 0001 (= 8) to both 111 and 1110.

(e) Multiplication by a fixed value k can be implemented deterministically from right to left, using k states to remember the carry (an integer in $[0, k-1]$) from one position to the next.

While this transducer can be reversed in order to do multiplication from left to right, the reversed transducer cannot be determinized. To see this, notice that in base ten, $33333333 \times 3 = 99999999$ but $33333334 \times 3 = 100000002$, and argue as in (7c).

[3] An abuse of notation, since it blurs the distinction between the string w and the number $\langle w \rangle$ that it encodes.