Lecture 14

Decidable languages

In the previous lecture we discussed some examples of encoding schemes, through which various objects can be represented by strings over a given alphabet. We will begin this lecture by considering a few more encoding schemes, particularly for the models of computation we have discussed thus far in the course. We will then discuss several interesting examples of decidable languages.

14.1 Encoding for different models of computation

In this section we will discuss ways of encoding DFAs, NFAs, CFGs, and DTMs as strings. As it turns out, the specifics of these encoding schemes won't actually matter all that much to us, so we'll focus more on the general ideas than on the specific details.

Encoding DFAs

For the sake of simplicity, let us fix the alphabet $\Sigma = \{0, 1\}$, and consider one way that an arbitrary DFA $D = (Q, \Gamma, \delta, q_0, F)$ can be encoded by a string over the alphabet Σ. Let us first make some observations and clarify our expectations about what we actually mean by encoding D in this way.

1. The result of our encoding should be a binary string, which we will denote by $\langle D \rangle$. Given the string $\langle D \rangle$, it should be possible to recover a description of each component of D. Stated in more intuitive terms, you can imagine that this means that if you were given the string $\langle D \rangle$ and had enough paper, pencils, and time to devote to the task, you could draw a state diagram for D without using any creativity during the process.
2. The state set Q of D is a finite and nonempty set, but beyond this there are no restrictions on it. Of course there is no way to devise a unique binary string en-

coding for every possible state name that someone could choose—for instance, we could imagine a different state name corresponding to every snowflake that could ever exist in nature. For this reason, we will assume that the state set of Q necessarily takes the form $Q = \{q_0, \ldots, q_{n-1}\}$ for some positive integer n. Because the specific names we choose for states are irrelevant to the way a DFA functions, this assumption does not create any difficulties.

3. It is important to notice that the alphabet Γ of the DFA D does not have to be the same as Σ; the alphabet Γ could be different from Σ, and moreover we would like to use a single encoding scheme that can handle any choice of Γ. On the other hand, the same comments we just made regarding the names of the states in Q also hold for the names of the symbols in Γ, and for this reason we will assume that $\Gamma = \{0, \ldots, m-1\}$ for some positive integer m (and where we are viewing each integer in the range $0, \ldots, m-1$ as a single symbol).

   With those points in mind, we could first encode D over the alphabet

$$\Delta = \{0, 1, \#, /\} \tag{14.1}$$

along the lines suggested by this hypothetical example:

$$\underbrace{1011}_{m} \# \underbrace{01100 \cdots 011}_{F} \# \underbrace{0/0/110 \# 0/1/1001 \# \cdots \# 11001/1010/011}_{\delta} \tag{14.2}$$

Here, m is the number of alphabet symbols, encoded in binary (so we appear to have $|\Gamma| = 11$ in this example). The string labeled F indicates which states are accept states and which are reject states, and its length indicates the total number of states (so, it appears that q_1, q_2, q_{n-2}, and q_{n-1}, for whatever value n happens to take, are among the accept states in the example). If we agree that q_0 is always the start state, there is no need to indicate this in the encoding. Finally, we assume that δ is encoded by a string in some fashion. The suggestion in this example is that a string of the form $a/b/c$ (for a, b, and c being natural numbers represented in binary notation) indicates that $\delta(q_a, b) = q_c$; and we concatenate together a list of such strings separated by # symbols in order to fully describe δ.

Once we have an encoding of the form suggested above, we could then convert it to a binary string using the method we discussed last time for encoding strings over one alphabet (in this case Δ) by strings in another (in this case the binary alphabet).

Of course this is just one way to encode a DFA as a binary string. There is nothing particularly special about this encoding scheme—it was invented for the sake of the lecture, and one could easily come up with other schemes that are just as effective.

It is worth noting that if we have chosen an encoding scheme for DFAs, such as the one just suggested, then every DFA (subject to the assumptions we placed on the state set and alphabet of that DFA) will have an encoding. Nothing says, however, that the encoding is unique—there could be multiple strings that encode the same DFA—and it may be that some strings will not correspond to the encoding of any DFA at all. Neither of these things will cause us any difficulties.

Encoding NFAs

If you use the encoding scheme suggested above for DFAs as a model, it should not be hard to come up with an encoding scheme for NFAs. So long as you have a way to represent the transition function of an NFA as a string, as opposed to the transition function of a DFA, then the rest could go unchanged. For the particular scheme that was suggested above, it would be pretty straightforward to extend it so that nondeterministic transition functions can be handled, and I will leave it to your imagination exactly how it might be done.

Encoding CFGs and DTMs

In the interest of time, we will not discuss any specific schemes for encoding CFGs and DTMs, but it is appropriate for us to observe that we could do this in a variety of different ways, perhaps using a similar style of encoding to the one suggested above for DFAs (but of course allowing for descriptions of the different components of CFGs and DTMs).

In the case of CFGs, an encoding scheme allows one to represent each CFG G by a binary string $\langle G \rangle$, from which one can uniquely identify G, obtain a description of the different components of G, and so on. Similar to our assumptions concerning encodings of DFAs, an assumption on the possible variable names and symbol names is required. For this reason, we might assume that the variable set of G takes the form $\{X_0, \ldots, X_{n-1}\}$ for some positive integer n, and the alphabet of G takes the form $\{0, \ldots, m-1\}$ for a positive integer m.

The situation is similar for DTMs, whereby any given DTM M can be encoded as a binary string $\langle M \rangle$ that allows one to recover M, describe its parts, an so on. Once again, similar assumptions should be made on the state names and alphabet symbol names (in this case including both the input alphabet and tape alphabet).

A general remark on alphabets used for encodings

In the discussion above, we described encodings over the binary alphabet $\Sigma = \{0,1\}$ for simplicity—but any other alphabet could be used instead. You could

even use an alphabet with a single symbol, such as $\Delta = \{0\}$, to encode objects like DFAs, NFAs, or anything else you might care to encode. One simple way to do this is to first come up with an encoding that uses the binary alphabet, and then translate this encoding into one that uses the alphabet Δ by placing strings in lexicographic correspondence:

$$\begin{array}{lcl} \varepsilon & \rightarrow & \varepsilon \\ 0 & \rightarrow & 0 \\ 1 & \rightarrow & 00 \\ 00 & \rightarrow & 000 \end{array} \tag{14.3}$$

and so on.

14.2 Simple examples of decidable languages

Now we will see some examples of decidable languages, beginning with some simple examples. One goal of this discussion is to develop the connections between DTMs and the intuitive notion of an algorithm, along the lines that the Church–Turing thesis suggests, and also to introduce a general style through which DTMs can be expressed at a high level.

Comparing natural numbers

As a first example, let us suppose that we have agreed upon a scheme for encoding any pair of natural numbers (n, m) into a string that we will denote[1] by $\langle n, m \rangle$. Consider the following language:

$$A = \bigl\{ \langle n, m \rangle \,:\, n \text{ and } m \text{ are natural numbers that satisfy } n > m \bigr\}. \tag{14.4}$$

We may then ask whether or not this language is decidable.

Now, it is certainly the case that the language A depends on the particular encoding scheme we selected. It may therefore seem as though the answer to the question of whether or not A is decidable also depends on the encoding scheme that is selected—but the reality is that the problem is decidable for *any reasonable encoding scheme whatsoever*.

[1] In general, whenever we wish to consider the encoding of some mathematical object X as a string, with respect to some encoding scheme that has been selected beforehand, we will denote this string by $\langle X \rangle$; if we have two objects X and Y, the string encoding both objects together is denoted $\langle X, Y \rangle$; and so on for any finite number of objects.

If the specific encoding scheme we chose when defining A was particularly simple, we could probably write down a state diagram of a DTM that decides A—but while this might be feasible, it would also be tedious and not very enlightening.

An alternative is to design a two-tape DTM that decides A. Perhaps this DTM would process the encoding $\langle n, m \rangle$ in such a way that we are left with the binary representation of n on one tape and the binary representation of m on the other. The precise manner in which this is done would necessarily depend on the encoding scheme, but it is safe to say that any encoding of a pair of natural numbers that doesn't allow a DTM to extract binary representations of the two numbers is not a reasonable one. Once this is done, the lengths of these binary representations could be compared. If one string is shorter than the other, then it becomes clear which of n and m is greater; and if the binary representations are the same length, then by comparing bits from most significant to least significant we can decide if $n > m$.

This is just one way to do it, and you can imagine that even if we agreed on this general strategy, there could be variations in how a two-tape DTM implementing this strategy might behave. In any case, once we know that a two-tape DTM decides the language A, then based on the discussion from the previous lecture we conclude that A is decidable by an ordinary DTM, and is therefore decidable.

Other languages based on numbers and arithmetic

One can imagine many other languages based on the natural numbers, arithmetic, and so on. Here are just a few examples:

$$\begin{aligned} C &= \big\{ \langle a, b, c \rangle \,:\, a, b, \text{and } c \text{ are natural numbers satisfying } a + b = c \big\}, \\ D &= \big\{ \langle a, b, c \rangle \,:\, a, b, \text{and } c \text{ are natural numbers satisfying } ab = c \big\}, \\ E &= \big\{ \langle n \rangle \,:\, n \text{ is a prime number} \big\}. \end{aligned} \tag{14.5}$$

For the first two languages, we assume that $\langle a, b, c \rangle$ is an encoding of three natural numbers a, b, and c, and for the third language $\langle n \rangle$ is the encoding of a natural number n. These are all decidable languages (once again, for any reasonable encoding scheme). Obtaining an actual description of a DTM that decides the languages would be increasingly difficult—but if you think about the fact that these languages could easily be decided by a computer program in your favorite programming language, which can be broken down into a sequence of primitive computation steps, you might convince yourself that it would be possible to come up DTMs that decide these languages.

Generally speaking, when we want to describe a DTM that performs a particular task, we describe it in high-level terms—usually not bothering to say very much (or perhaps nothing at all) about tapes or tape head movements. For example, we could describe a DTM M deciding the language E in following style:

On input $\langle n \rangle$, where n is a natural number:

1. If $n \leq 1$ then *reject*. (We don't consider 0 and 1 to be prime.)
2. Set $t \leftarrow 2$.
3. If $t = n$, then *accept*.
4. If t divides n evenly, then *reject*.
5. Set $t \leftarrow t + 1$ and goto step 3.

Note that when we write "On input $\langle n \rangle$, where n is a natural number" we are implicitly defining M so that it immediately rejects if the input does not have this form. A similar interpretation should be made for DTMs in general in these notes.

Of course, if you wanted to turn this high-level Turing machine description into a formal specification of a DTM, you would have a lot of work ahead of you—but the description nevertheless explains key algorithmic ideas for deciding E. The most complicated step is step 4, and of course one could explain it in more detail. However, it is probably safe to assume that everyone reading these notes knows how to compute the remainder of one natural number divided by another, assuming the numbers are expressed in binary notation, and one could implement that method on a Turing machine.

Graph reachability

Here is one final example of a simple decidable language, which this time concerns graph reachability.

$$\text{REACHABLE} = \left\{ \langle G, u, v \rangle \ : \ \begin{array}{l} \text{there exists a path from vertex } u \text{ to} \\ \text{vertex } v \text{ in the undirected graph } G \end{array} \right\}. \tag{14.6}$$

Once again, we are not specifying the specific encoding method that is used to represent a graph G and two vertices u, v of G as a string $\langle G, u, v \rangle$—but so long as the encoding is reasonable it does not change whether or not the language is decidable. Here is a high-level description of a DTM that decides this language:

On input $\langle G, u, v \rangle$, where G is an undirected graph and u and v are vertices of G:

1. Set $S \leftarrow \{u\}$.
2. Set $F \leftarrow 1$. (F is a flag indicating we are finished adding vertices to S.)
3. For each vertex w of G do the following:
4. If w is not contained in S and w is adjacent to a vertex in S, then set $S \leftarrow S \cup \{w\}$ and $F \leftarrow 0$.
5. If $F = 0$ then goto 2.
6. *Accept* if $v \in S$, otherwise *reject*.

14.3 Languages concerning DFAs, NFAs, and CFGs

Now let us turn our attention toward languages that concern the models of computation we have studied previously in the course. The first language we will consider is this one:

$$\mathrm{A}_{\mathrm{DFA}} = \bigl\{ \langle D, w \rangle \;:\; D \text{ is a DFA and } w \in \mathrm{L}(D) \bigr\}. \tag{14.7}$$

Along the same lines as the previous examples in this lecture, we understand $\langle D, w \rangle$ to be the encoding of a DFA D together with a string w. The alphabet of D and w is not assumed to be fixed ahead of time, but as discussed earlier in the lecture we will assume that for each particular choice of D and w, the alphabet symbols appearing are drawn from the set $\{0, \ldots, n-1\}$ for some positive integer n.

Here is a high-level description of a DTM M that decides $\mathrm{A}_{\mathrm{DFA}}$:

On input $\langle D, w \rangle$, where D is a DFA and w is a string:

1. If w contains symbols not in the alphabet of D, then *reject*.
2. Simulate the computation of D on input w. If this simulation ends on an accept state of D, then *accept*, and otherwise *reject*.

By specifying that this DTM should "simulate the computation of D on input w," it might seem like we are cheating somehow—but it really is quite simple and direct to simulate a DFA on a given input string, so stating that this process should be performed by a computer is reasonable. It follows from the fact that the DTM M decides $\mathrm{A}_{\mathrm{DFA}}$ that this language is decidable.

We may also consider a variant of this language for NFAs in place of DFAs:

$$\mathrm{A}_{\mathrm{NFA}} = \bigl\{ \langle N, w \rangle \;:\; N \text{ is an NFA and } w \in \mathrm{L}(N) \bigr\}. \tag{14.8}$$

This language is also decidable. However, it would not be reasonable to simply define a DTM that "simulates the computation of N on input w," because it really isn't clear how one simulates a nondeterministic computation with a DTM. Here is an alternative description of a DTM that decides $\mathrm{A}_{\mathrm{NFA}}$:

On input $\langle N, w \rangle$, where N is an NFA and w is a string:

1. Convert N into an equivalent DFA D using the subset construction described in Lecture 3.
2. If w contains symbols not in the alphabet of D, then *reject*.
3. Simulate the computation of D on input w. If this simulation ends on an accept state of D, then *accept*, and otherwise *reject*.

One could also define a similar language

$$\mathrm{A}_{\mathrm{REX}} = \bigl\{\langle R, w\rangle \,:\, R \text{ is a regular expression and } w \in \mathrm{L}(R)\bigr\}, \tag{14.9}$$

and conclude that it is decidable through a similar argument (referring to a construction that converts a regular expression into a DFA).

Here is a different example of a language, which we will also prove is decidable:

$$\mathrm{E}_{\mathrm{DFA}} = \bigl\{\langle D\rangle \,:\, D \text{ is a DFA and } \mathrm{L}(D) = \varnothing\bigr\}. \tag{14.10}$$

In this case, one cannot decide this language simply by "simulating the DFA D," because we don't necessarily know what string to simulate it on. Here is a Turing machine that decides this language based on a different approach, which essentially is to test to see if there is an accept state that is reachable from the start state. The approach is similar to the graph reachability example from before.

On input $\langle D\rangle$, where D is a DFA:

1. Set $S \leftarrow \{q_0\}$, for q_0 the start state of D.
2. Set $F \leftarrow 1$.
3. For each state q of D do the following:
4. If q is not contained in S and there exists a transition from any state in S to q, then set $S \leftarrow S \cup \{q\}$ and $F \leftarrow 0$.
5. If $F = 0$ then goto 2.
6. *Reject* if there exists an accept state in S, otherwise *accept*.

By combining this DTM with ideas from the previous examples, one can prove that analogously defined languages $\mathrm{E}_{\mathrm{NFA}}$ and $\mathrm{E}_{\mathrm{REX}}$ are decidable.

One more example of a decidable language concerning DFAs is this language:

$$\mathrm{EQ}_{\mathrm{DFA}} = \bigl\{\langle D, E\rangle \,:\, D \text{ and } E \text{ are DFAs and } \mathrm{L}(D) = \mathrm{L}(E)\bigr\}. \tag{14.11}$$

Here is a high-level description of a DFA that decides it:

On input $\langle D, E\rangle$, where D and E are DFAs:

1. Construct a DFA M for which it holds that $\mathrm{L}(M) = \mathrm{L}(D) \triangle \mathrm{L}(E)$.
2. If $\langle M\rangle \in \mathrm{E}_{\mathrm{DFA}}$, then *accept*, otherwise *reject*.

One natural way to perform the construction in step 1 of the previous DTM description is to use the so-called Cartesian product construction, which we discussed in lecture earlier in the course.

Next let us turn to a couple of examples of decidable languages concerning context-free grammars. Following along the same lines as the examples discussed above, we may consider these languages:

$$\begin{aligned} \mathrm{A}_{\mathrm{CFG}} &= \bigl\{ \langle G, w \rangle \,:\, G \text{ is a CFG and } w \in \mathrm{L}(G) \bigr\}, \\ \mathrm{E}_{\mathrm{CFG}} &= \bigl\{ \langle G \rangle \,:\, G \text{ is a CFG and } \mathrm{L}(G) = \varnothing \bigr\}. \end{aligned} \tag{14.12}$$

Here is a DTM that decides the first language:

On input $\langle G, w \rangle$, where G is a CFG and w is a string:

1. Convert G into an equivalent CFG H in Chomsky normal form.
2. If $w = \varepsilon$ then *accept* if $S \to \varepsilon$ is a rule in H and *reject* otherwise.
3. Search over all possible derivations by H having $2|w| - 1$ steps (of which there are finitely many). *Accept* if a valid derivation of w is found, and *reject* otherwise.

It is worth noting that this is a ridiculously inefficient way to decide the language $\mathrm{A}_{\mathrm{CFG}}$, but right now we don't care! We're just trying to prove it is decidable. There are, in fact, much more efficient ways to decide this language, but we will not discuss them now.

Finally, the language $\mathrm{E}_{\mathrm{CFG}}$ can be decided using a variation on the reachability technique. In essence, we keep track of a set containing variables that generate at least one string, and then test to see if the start variable is contained in this set.

On input $\langle G \rangle$, where G is a CFG:

1. Set $T \leftarrow \Sigma$, for Σ the alphabet of G.
2. Set $F \leftarrow 1$.
3. For each rule $X \to w$ of G do the following:
4. If X is not contained in T, and every variable and every symbol of w is contained in T, then set $T \leftarrow T \cup \{R\}$ and $F \leftarrow 0$.
5. If $F = 0$ then goto 2.
6. *Reject* if the start variable of G is contained in T, otherwise *accept*.

Now, you may be wondering about this next language, as it is analogous to one concerning DFAs from above:

$$\mathrm{EQ}_{\mathrm{CFG}} = \bigl\{ \langle G, H \rangle \,:\, G \text{ and } H \text{ are CFGs and } \mathrm{L}(G) = \mathrm{L}(H) \bigr\}. \tag{14.13}$$

As it turns out, this language is not decidable. (We won't go through the proof, because it would take us a bit too far off the path of the rest of the course, but it would not be too difficult to prove sometime after the lecture following this one.)

Some other examples of undecidable languages concerning context-free grammars are as follows:

$$\begin{aligned}
&\{\langle G\rangle \,:\, G \text{ is a CFG that generates all strings over its alphabet}\},\\
&\{\langle G\rangle \,:\, G \text{ is an ambiguous CFG}\},\\
&\{\langle G\rangle \,:\, G \text{ is a CFG and } \mathrm{L}(G) \text{ is inherently ambiguous}\}.
\end{aligned} \tag{14.14}$$