Lecture 15

Frederick Kellison-Linn
fjk2119

19 March 2018

1 Introduction

We left off last time with the Church-Turing thesis, that every reasonable model of computation can be simulated by a Turing machine. This means that we can use Turing machines as our canonical model of computation, and say that every algorithm can be implemented by some Turing machine. This includes everything seen in your data structures and algorithms class, everything published in theoretical CS articles, and so on.

Transitioning into the next part of this class, we're going to be looking at what types of problems Turing machines can solve, and even more interestingly, what kinds of problems they can't solve. Today, we're going to go over encodings of Turing machines (and other automata we've seen before), recall what it means for a language to be *decidable*, and look at examples of some decidable languages.

2 Encodings

As we know, Turing machines are given their input as the starting content of the tape when the machine begins execution. However, many familiar algorithms do not deal with "strings": we have trees, graphs, queues, and many other abstract mathematical structures. In order to develop Turing machines which work with these structures, we need to have a method for turning them into strings in a standardized way. The process to do this will differ for each type of structure, so we will not discuss all of them here.

Most interestingly, we can develop an encoding method for Turing machines themselves, which allows Turing machines to reason about Turing machines. This shouldn't be too shocking, since every time we write out a formal description of a Turing machine we are indeed converting it to a string. We will discuss one such method of encoding in the next few paragraphs, but know that many different encodings are possible.

Recall that the formal definition of a Turing machine M is a tuple:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, a_{accept}, q_{reject})$$

where Q is the set of states, Σ and Γ are the input and tape alphabets, respectively, and δ is the transition function. In order to make our encoding a bit easier, we will make some assumptions about the structure of our Turing machine. You can verify as an exercise that none of these assumptions reduce the power of the model of computation: any arbitrary Turing machine can be converted to one obeying these constraints:

$$Q = \{0, 1, 2, ..., n\}$$

$$q_0 = 0$$

$$q_{accept} = 1$$

$$q_{reject} = 2$$

$$\Sigma = \{0, 1\}$$

$\Gamma = \Sigma \cup \{0, 1, 2, ..., m\}$ where 2 is our blank symbol (i.e. $\llcorner = 2$)

We will also assume that the machine M is deterministic, since we have already seen the equivalence of deterministic and nondeterministic machines. Thus, the only information that we actually need to encode is the value of n, m, and the transitions of δ.

Suppose that $\delta(q_i, a) = (q_j, b, X)$. To encode this, transition, we write: $0^{i+1}10^{a+1}10^{j+1}10^{b+1}1x$ where $x = 0$ if $X = L$ and $x = 00$ if $X = R$. For example, if we had the transition $\delta(4, 0) = (3, 1, L)$, then the encoding of this transition would be 00000101000010010. Now, for every transition specified by δ, we encode that transition and append them all together, separated by 11. Thus, if δ were the function:

$$\delta(0, 1) = (1, 1, R)$$

$$\delta(0, 0) = (0, 0, R)$$

$$\delta(0, 2) = (0, 2, R)$$

We would have the encoding of δ as:

$$0100100100100110101010100110100010100 0100$$

Finally, we encode n as a string of n zeros, 0^n and append this with the encoding for δ, separated by three 1s. The same is done for m. Thus, the full encoding for the above machine would be:

$$0001110001110100100100100110101010100110100010100 0100$$

We won't again discuss the specifics of a Turing machine's encoding. When we want to talk about the encoding of a particular Turing machine, we will use the notation $\langle M \rangle$ to refer to the encoding of the Turing machine M.

This process of encoding can be applied to any (finite) mathematical structure. Thus, we can talk about the encodings of DFAs, NFAs, and PDAs (and

even graphs, tuples, etc.) just as we can talk about encodings of Turing machines. In general we will use the notation $\langle X \rangle$ to denote the encoding of the object X.

Moving forward, we will be designing Turing machines which take other automata as inputs. Since the structure of an encoding is highly specific, there are many strings which do not encode a valid object. For instance, there is no Turing machine which will generate the encoding 101. However, the format of a valid encoding is quite easy to verify, so a Turing machine which expects an encoding can always easily check whether some string is a valid encoding of an object.

3 Decidability

Recall the defintion we gave for decidability. We said that a language L is *Turing decidable* (or simply *decidable*) if there exists a Turing machine M which has the property that if $x \in L$, then M accepts x, and if $x \notin L$, then M rejects x for all $x \in \Sigma^*$. Another way to put this is that $x \in L$ if and only if $x \in L(M)$, and M eventually halts on all inputs (i.e. there are no infinite loops).

3.1 Some Decidable Languages

Now we will look at some example languages which make use of the idea of encodings. All of the following languages are decidable.

1. $A_{DFA} = \{\langle D, w \rangle \mid D \text{ is a DFA which accepts } w\}$ We will design a machine M which halts on all inputs and recognizes A_{DFA}. M, on input x, does the following:

   (a) Check that $x = \langle D, w \rangle$ for some DFA $D = (Q, \Sigma, \delta, q_0, F)$ and string $w \in \Sigma^*$. If not, reject x

   (b) Keep track of a variable `qsim` representing the current state of D, initialized to q_0.

   (c) For each symbol a of w in order:

      i. Find the transition in $\langle D \rangle$ matching the symbol a and the state `qsim`. Suppose that $\delta(\texttt{qsim}, a) = q'$.

      ii. Set $\texttt{qsim} = q'$.

   (d) Scan $\langle D \rangle$ to see if $\texttt{qsim} \in F$. If so, accept x, otherwise, reject x.

   We can see that M is a decider for A_{DFA} as follows. If $x \in L$, then $x = \langle D, w \rangle$ for some DFA D and string w, and D accepts w. This means that following the transitions of δ will take D to an accepting state. Thus, q_f in step (d) above will find that $\texttt{qsim} \in F$, so M will accept x.

   If $x \notin L$, then either x is not a valid encoding, in which case M will reject x or $x = \langle D, w \rangle$ but D does not accept w. Then, when M reaches step (d), it will find that $\texttt{qsim} \notin F$, so M will reject x.

In the future, it will be our convention that rather than always specifying step (a) where we check that the input is a valid encoding, we will say (e.g.) "M, on input $\langle D, w\rangle$..." and it will be implicit that if the input is not of this form then the machine should reject.

2. $A_{NFA} = \{\langle N, w\rangle \mid N \text{ is an NFA which accepts } w\}$ We design the following decider for A_{NFA}. M, on input $\langle N, w\rangle$:

   (a) Convert N to an equivalent DFA D.

   (b) Check via the algorithm in (1) if w is accepted by D. If so, accept; otherwise, reject.

   From our unit on regular languages, we know that there is a deterministic algorithm for converting an NFA to an equivalent DFA, so step (a) will take finite time. Furthermore, N and D are equivalent, so M accepts $\langle N, w\rangle$ if and only if D accepts w if and only if N accepts w. Thus, M is a decider for A_{NFA}

3. $E_{DFA} = \{\langle D\rangle \mid D \text{ is a DFA and } L(D) \text{ is empty}\}$ The following machine M is a decider for E_{DFA}. On input $\langle D\rangle$:

   (a) Initialize a list of 'marked' states, starting with q_0.

   (b) Repeat the following until the list remains unchanged.

      i. Go through each state q in the marked list. For each transition $\delta(q, a) = q'$, check if q' has been marked. If not, add it to the list.

   (c) Check if the marked list contains any state $q \in F$. If so, reject. If not, accept.

   Now we will show that M decides E_{DFA}. That M halts on every input can be seen from the fact that at least one state is added to the marked list on each execution of step (i), so this loop will run at most $|Q|$ times. Then, we will reach step (c) and either accept or reject.

   If $\langle D\rangle \in E_{DFA}$ then there is no string which will take D to an accepting state. Thus, because the marking process follows only transitions beginning from q_0, there will be no path which reaches some state in F. Thus, for every state q in the marked list, it will be the case that $q \notin F$, and so M will accept.

   If $\langle D\rangle \notin E_{DFA}$, then there is at least one string $w \in L(D)$. This means that when D runs on w, it reaches an accepting state $q_f \in F$. Thus, there is a path in D from q_0 to q_f, so q_f will eventually be marked. Then, in step (c), M will reject.

4. $EQ_{DFA} = \{\langle D_1, D_2\rangle \mid D_1 \text{ and } D_2 \text{ are DFAs and } L(D_1) = L(D_2)\}$ We will not give an explicit construction of a machine M here. Instead, we will note that $L(D_1) = L(D_2)$ if and only if $(L(D_1) \cap \overline{L(D_2)}) \cup (L(D_2) \cap$

$\overline{L(D_1)}) = \emptyset$ (see midterm review notes for a proof that the class of regular languages is closed under symmetric difference). Thus, in a similar fashion to A_{NFA}, we can build a new DFA D' which accepts $L(D_1) \oplus L(D_2)$, and use the algorithm from (3) to check if this language is empty.

5. $A_{CFG} = \{\langle G, w\rangle \mid G \text{ is a CFG and } G \text{ generates } w\}$ To prove this, we will rely on two facts which we will not prove:

   - There is some algorithm M_{CNF} which can convert a grammar G to a grammar G' in Chomsky Normal Form (we won't go into detail about this here, see the end of **Section 2.1** in Sipser for a full definition and proof).
   - If a string w can be generated by a grammar in CNF, then in can be generated in at most $2|w| - 1$ steps.

   Now, we can give a decider M for A_{CFG} as follows. On input $\langle G, w\rangle$:

   (a) Convert G to G' in CNF using M_{CNF}.

   (b) G' has some finite number of rules k. Thus, there are $k^{2|w|-1}$ possible derivations of length $2|w|-1$. Enumerate all of these rule sequences, trying all of them, discarding invalid sequences.

   (c) If any sequence produces w, accept.

   (d) Reject.

   If $\langle G, w\rangle \in A_{CFG}$ if and only if G can generate w, which is true if and only if G' can generate w, which is true if and only if G' can generate w in $2|w| - 1$ steps. Thus, if $\langle G, w\rangle \in A_{CFG}$ then some rule sequence will generate w, and M will accept. Otherwise, no such rule sequence will generate w, and M will reach step (d) and reject.

6. $E_{CFG} = \{\langle G\rangle \mid G \text{ is a CFG and } L(G) \text{ is empty}\}$ We will not give an explicit proof here, and instead sketch out the construction briefly. We use a similar marking approach to E_{DFA}, beginning with all terminals marked, and then marking A if $A \to x$ is a rule for some terminal x, or if $A \to BC$ is a rule for marked variables BC. When finished, reject if S is marked and accept otherwise.