Lecture 17

Further discussion of Turing machines

In this lecture we will discuss various aspects of decidable and Turing-recognizable languages that were not mentioned in previous lectures. In particular, we will discuss closure properties of these classes of languages, briefly discuss nondeterministic Turing machines, and prove a useful alternative characterization of Turing-recognizable languages.

17.1 Decidable language closure properties

The decidable languages are closed under many of the operations on languages that we've considered thus far in the course (although not all). While we won't go through every operation we've discussed, it is worthwhile to mention some basic examples.

First let us observe that the decidable languages are closed under the regular operations as well as complementation. In short, if A and B are decidable, then there is no difficulty in deciding the languages $A \cup B$, AB, A^*, and $\overline{A}$ in a straightforward way.

Proposition 17.1. *Let Σ be an alphabet and let $A, B \subseteq \Sigma^*$ be decidable languages. The languages $A \cup B$, AB, and A^* are decidable.*

Proof. Because the languages A and B are decidable, there must exist a DTM M_A that decides A and a DTM M_B that decides B.

The following DTM decides $A \cup B$, which implies that $A \cup B$ is decidable.

On input w:

1. Run M_A and M_B on input w.
2. If either M_A or M_B accepts, then *accept*, otherwise *reject*.

The following DTM decides AB, which implies that AB is decidable.

On input w:

1. For every choice of strings u, v satisfying $w = uv$:
2. Run M_A on input u and run M_B on input v.
3. If both M_A and M_B accept, then *accept*.
4. *Reject.*

Finally, the following DTM decides A^*, which implies that A^* is decidable.

On input w:

1. If $w = \varepsilon$, then *accept*.
2. For every way of writing $w = u_1 \cdots u_m$ for nonempty strings $u_1, \ldots, u_m$:
2. Run M_A on each of the strings $u_1, \ldots, u_m$.
3. If M_A accepts all of the strings $u_1, \ldots, u_m$, then *accept*.
4. *Reject.*

This completes the proof. □

Proposition 17.2. *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a decidable language. The language $\overline{A}$ is decidable.*

Perhaps we don't even need to bother writing a proof for this one—if a DTM M decides A, then one can obtain a DTM deciding $\overline{A}$ by simply exchanging the accept and reject states of M.

There are a variety of other operations under which the decidable languages are closed. For example, because the decidable languages are closed under union and complementation, we immediately have that they are closed under intersection and symmetric difference. Another example is string reversal: if a language A is decidable, then A^{R} is also decidable, because a DTM can decide A^{R} simply by reversing the input string, then deciding whether the string that is obtained is contained in A.

There are, however, some natural operations under which the decidable languages are not closed. The following example shows that this is the case for the prefix operation.

Example 17.3. The language $\mathrm{Prefix}(A)$ might not be decidable, even if A is decidable. To construct an example that illustrates that this is so, let us first take $B \subseteq \{0,1\}^*$ to be any language that is Turing recognizable but not decidable (such as HALT, assuming we define this language with respect to a binary string encoding of DTMs).

Let M_B be a DTM such that $\mathrm{L}(M_B) = B$, and define a language $A \subseteq \{0,1,2\}^*$ as follows:

$$A = \{w2^t \,:\, M_B \text{ accepts } w \text{ within } t \text{ steps}\}. \tag{17.1}$$

This is a decidable language, but $\mathrm{Prefix}(A)$ is not—for if $\mathrm{Prefix}(A)$ were decidable, then one could easily decide B by using the fact that a string $w \in \{0,1\}^*$ is contained in B if and only if $w2 \in \mathrm{Prefix}(A)$. (That is, $w \in B$ and $w2 \in \mathrm{Prefix}(A)$ are both equivalent to the existence of a positive integer t such that $w2^t \in A$.)

17.2 Turing-recognizable language closure properties

The Turing-recognizable languages are also closed under a variety of operations, although not precisely the same operations under which the decidable languages are closed.

Let us begin with the regular operations, under which the Turing-recognizable languages are indeed closed. In this case, one needs to be a bit more careful than was sufficient when proving the analogous property for decidable languages, as the Turing machines that recognize these languages might run forever.

Proposition 17.4. *Let Σ be an alphabet and let $A, B \subseteq \Sigma^*$ be Turing-recognizable languages. The languages $A \cup B$, AB, and A^* are Turing recognizable.*

Proof. Because the languages A and B are Turing recognizable, there must exist DTMs M_A and M_B such that $\mathrm{L}(M_A) = A$ and $\mathrm{L}(M_B) = B$.

The following DTM recognizes the language $A \cup B$, which implies that $A \cup B$ is Turing recognizable.

On input w:

1. Set $t \leftarrow 1$.
2. Run M_A and M_B on input w for t steps.
3. If either M_A or M_B accepts within t steps, then *accept*.
4. Set $t \leftarrow t + 1$ and goto 2.

The following DTM recognizes AB, which implies that AB is Turing recognizable.

On input w:

1. Set $t \leftarrow 1$.
2. For every choice of strings u, v satisfying $w = uv$:
3. Run M_A on input u and run M_B on input v, both for t steps.
4. If both M_A and M_B accept within t steps, then *accept*.
5. Set $t \leftarrow t + 1$ and goto 2.

Finally, the following DTM recognizes A^*, which implies that A^* is Turing recognizable.

On input w:

1. If $w = \varepsilon$, then *accept*.
2. Set $t \leftarrow 1$.
3. For every way of writing $w = u_1 \cdots u_m$ for nonempty strings $u_1, \ldots, u_m$:
4. Run M_A on each of the strings $u_1, \ldots, u_m$ for t steps.
5. If M_A accepts all of the strings $u_1, \ldots, u_m$ within t steps, then *accept*.
6. Set $t \leftarrow t + 1$ and goto 3.

This completes the proof. □

The Turing-recognizable languages are also closed under intersection. This can be proved through a similar method to closure under union, but in fact this is a situation in which we don't actually need to be as careful about running forever.

Proposition 17.5. *Let Σ be an alphabet and let $A, B \subseteq \Sigma^*$ be Turing-recognizable languages. The language $A \cap B$ is Turing recognizable.*

Proof. Because the languages A and B are Turing recognizable, there must exist DTMs M_A and M_B such that $\mathrm{L}(M_A) = A$ and $\mathrm{L}(M_B) = B$.

The following DTM recognizes the language $A \cap B$, which implies that $A \cap B$ is Turing recognizable.

On input w:

1. Run M_A on input w.
2. Run M_B on input w.
3. If both M_A and M_B accept, then *accept*, otherwise *reject*.

If it so happens that either M_A or M_B runs forever on input w, then the DTM just described also runs forever, which is fine—what is relevant is that the DTM accepts if and only if both M_A and M_B accept. This completes the proof. □

The Turing-recognizable languages are not, however, closed under complementation. We already proved in Lecture 15 that if A and $\overline{A}$ are Turing recognizable, then A is decidable. We also know that there exist languages, such as HALT, that are Turing recognizable but not decidable—so it cannot be that the Turing-recognizable languages are closed under complementation.

Finally, there are some operations under which the Turing-recognizable languages are closed, but under which the decidable languages are not. For example, if A is Turing recognizable, then so too are the languages $\mathrm{Prefix}(A)$, $\mathrm{Suffix}(A)$, and $\mathrm{Substring}(A)$ (as one of the homework problems on Assignment 3 asked you to prove), but this is not necessarily so for decidable languages.

17.3 Nondeterministic Turing machines

One can define a nondeterministic variant of the Turing machine model along similar lines to the definition of nondeterministic finite automata, as compared with deterministic finite automata.

Definition 17.6. A *nondeterministic Turing machine* (or NTM, for short) is a 7-tuple

$$N = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}), \tag{17.2}$$

where Q is a finite and nonempty set of *states*; Σ is an alphabet, called the *input alphabet*, which may not include the blank symbol $\sqcup$; Γ is an alphabet, called the *tape alphabet*, which must satisfy $\Sigma \cup \{\sqcup\} \subseteq \Gamma$; δ is a transition function having the form

$$\delta : Q \backslash \{q_{\text{acc}}, q_{\text{rej}}\} \times \Gamma \to \mathcal{P}\big(Q \times \Gamma \times \{\leftarrow, \rightarrow\}\big); \tag{17.3}$$

$q_0 \in Q$ is the *initial state*; and $q_{\text{acc}}, q_{\text{rej}} \in Q$ are the *accept* and *reject* states, which satisfy $q_{\text{acc}} \neq q_{\text{rej}}$.

For a given state $q \in Q \backslash \{q_{\text{acc}}, q_{\text{rej}}\}$ and tape symbol a, the elements in the set $\delta(q, a)$ represent the possible moves the NTM can make. The yields relation of an NTM is then defined in a similar manner to DTMs, except this time there may be multiple choices of configurations that are reachable from a given configuration.

When we think about the yields relation of an NTM N, and the way that it computes on a given input string w, it is natural to envision a *computation tree*. This is a possibly infinite tree in which each node corresponds to a configuration. The root node is the initial configuration, and the children of each node are all of the configurations that can be reached in one step from the configuration represented by that node. If we thought about such a tree for a deterministic computation, it would not be very interesting—it would essentially be a stick, where each node has either one child (representing the next configuration, one step later) or no children (in the case of an accepting or rejecting configuration). For a nondeterministic computation, however, the tree can branch, as each configuration could have multiple next configurations. Note that we allow multiple nodes in the tree to represent the same configuration, as there could be different *computation paths* (or paths starting from the root node and going down the tree) that reach a particular configuration.

Acceptance for an NTM is defined in a similar way to NFAs (and PDAs as well). That is, if there *exists* a sequence of computation steps that reach an accepting configuration, then the NTM accepts, and otherwise it does not. In terms of the computation tree, this simply means that somewhere in the tree there exists an accepting configuration. Note that there might or might not be rejecting configurations in the computation tree, and there might also be computation paths starting

from the root that are infinite, but all that matters when it comes to defining acceptance is whether or not there exists an accepting configuration that is reachable from the initial configuration. Formally speaking, the definition of acceptance for an NTM is the same as for a DTM, albeit for a yields relation that comes from an NTM. That is, N accepts w if there exist strings $u, v \in \Gamma^*$ and a symbol $a \in \Gamma$ such that

$$(q_0, \sqcup) w \vdash_N^* u(q_{\text{acc}}, a) v. \tag{17.4}$$

As usual, we write $\mathrm{L}(N)$ to denote the language of all strings w accepted by an NTM N, which we refer to as the language recognized by N.

The following theorem states that the languages recognized by NTMs are the same as for DTMs.

Theorem 17.7. *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a language. There exists an NTM N such that $\mathrm{L}(N) = A$ if and only if A is Turing recognizable.*

It is not difficult to prove this theorem, but I will leave it to you to contemplate if you are interested. The key idea for showing that the language $\mathrm{L}(N)$ for an NTM N is Turing recognizable is to perform a breadth-first search of the computation tree of N on a given input. If there is an accepting configuration anywhere in the tree, a breadth-first search will find it. Of course, because computation trees may be infinite, the search might never terminate—this is unavoidable, but it is not an obstacle for proving the theorem. Note that an alternative approach of performing a depth-first search of the computation tree would not work: it might happen that such a search descends down an infinite path of the tree, potentially missing an accepting configuration elsewhere in the tree.

17.4 The range of a computable function

We will now consider an alternative characterization of Turing-recognizable languages (with the exception of the empty language), which is that they are precisely the languages that are equal to the *range* of a computable function. Recall that the range of a function $f : \Sigma^* \to \Sigma^*$ is defined as follows:

$$\text{range}(f) = \big\{ f(w) \,:\, w \in \Sigma^* \big\}. \tag{17.5}$$

Theorem 17.8. *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a nonempty language. The following two statements are equivalent:*

1. *A is Turing recognizable.*
2. *There exists a computable function $f : \Sigma^* \to \Sigma^*$ such that $A = \text{range}(f)$.*

Proof. Let us first suppose that the second statement holds: $A = \text{range}(f)$ for some computable function $f : \Sigma^* \to \Sigma^*$. Let us define a DTM M as follows:

On input w:

1. Set $x \leftarrow \varepsilon$.
2. Compute $y = f(x)$, and *accept* if $w = y$.
3. Increment x with respect to the lexicographic ordering of Σ^* and goto 2.

In essence, this DTM searches over Σ^* to find a string that f maps to a given input string w. If it is the case that $w \in \text{range}(f)$, then M will eventually find $x \in \Sigma^*$ such that $f(x) = w$ and accept, while M will certainly not accept if $w \notin \text{range}(f)$. Thus, we have $\text{L}(M) = \text{range}(f) = A$, which implies that A is Turing recognizable.

Now suppose that A is Turing recognizable, so that there exists a DTM M such that $\text{L}(M) = A$. We will also make use of the assumption that A is nonempty—there exists at least one string in A, so we may take w_0 to be such a string. (If you like, you may define w_0 more concretely as the first string in A with respect to the lexicographic ordering of Σ^*, but it is not important for the proof that we make this particular choice.)

We will define a computable function f for which $A = \text{range}(f)$. Here is one such function:

$$f(x) = \begin{cases} w & \text{if } x = \langle w, t \rangle \text{ for } w \in \Sigma^* \text{ and } t \in \mathbb{N}, \text{ and} \\ & M \text{ accepts } w \text{ within } t \text{ steps} \\ w_0 & \text{otherwise.} \end{cases} \tag{17.6}$$

To be a bit more precise, the condition "$x = \langle w, t \rangle$ for $w \in \Sigma^*$ and $t \in \mathbb{N}$" refers to an encoding scheme through which a string $w \in \Sigma^*$ and a natural number $t \in \mathbb{N}$ are encoded as a string $\langle w, t \rangle$ over the alphabet Σ.

It is evident that the function f is computable: a DTM M_f can compute f by checking to see if the input has the form $\langle w, t \rangle$, simulating M for t steps on input w if so, and then outputting either w or w_0 depending on the outcome. If M accepts a particular string w, then it must hold that $w = f(\langle w, t \rangle)$ for some sufficiently large natural number t, so $A \subset \text{range}(f)$. On the other hand, every output of f is either a string w accepted by M or the string w_0, and therefore $\text{range}(f) \subseteq A$. It therefore holds that $A = \text{range}(f)$, which completes the proof. □

Remark 17.9. The assumption that A is nonempty is essential in the previous theorem because it cannot be that $\text{range}(f) = \varnothing$ for a computable function f. Indeed, it cannot be that $\text{range}(f) = \varnothing$ for any function whatsoever. (If we were to consider *partial functions*, which are functions that may be undefined for some inputs,

the situation would be different—but the standard interpretation of the word *function*, at least as far as this course is concerned, means *total function* and not *partial function*.)

Theorem 17.8 provides a useful characterization of the Turing-recognizable languages. For instance, you can use this theorem to come up with alternative proofs for all of the closure properties of the Turing-recognizable languages stated in the previous section.

For example, suppose that $A \subseteq \Sigma^*$ is a nonempty Turing-recognizable language, so that there must exist a computable function $f : \Sigma^* \to \Sigma^*$ such that $\text{range}(f) = A$. Define a new function

$$g(x) = \begin{cases} f(u_1) \cdots f(u_m) & \text{if } x = \langle u_1, \ldots, u_m \rangle \text{ for } m \geq 0 \text{ and} \\ & u_1, \ldots, u_m \in \Sigma^* \\ \varepsilon & \text{otherwise,} \end{cases} \tag{17.7}$$

where the statement "$x = \langle u_1, \ldots, u_m \rangle$ for $m \geq 0$ and $u_1, \ldots, u_m \in \Sigma^*$" refers to an encoding scheme in which zero or more strings $u_1, \ldots, u_m \in \Sigma^*$ are encoded as a single string over Σ. One sees that g is computable, and $\text{range}(g) = A^*$, which implies that A^* is Turing recognizable.

Here is another example of an application of Theorem 17.8.

Corollary 17.10. *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be any infinite Turing-recognizable language. There exists an infinite decidable language $B \subseteq A$.*

Proof. Because A is infinite (and therefore nonempty), there exists a computable function $f : \Sigma^* \to \Sigma^*$ such that $A = \text{range}(f)$.

We will define a language B that satisfies the requirements of the corollary by first defining a DTM M and then arguing that $B = \text{L}(M)$ works. This will require proving that M never runs forever (so that B is decidable), M only accepts strings that are contained in A (so that $B \subseteq A$), and M accepts infinitely many different strings (so that B is infinite). We'll prove each of those things, but first here is the DTM M:

On input w:

1. Set $x \leftarrow \varepsilon$.
2. Compute $y = f(x)$.
3. If $y = w$ then *accept*.
4. If $y > w$ (with respect to the lexicographic ordering of Σ^*) then *reject*.
5. Increment x with respect to the lexicographic ordering of Σ^* and goto 2.

The fact that M never runs forever follows from the assumption that A is infinite. That is, because A is infinite, the function f must output infinitely many different strings, so regardless of what input string w is input into M, the loop will eventually reach a string x so that $f(x) = w$ or $f(x) > w$, either of which causes M to halt.

The fact that M only accepts strings in A follows from the fact that the condition for acceptance is that the input string w is equal to y, which is contained in $\text{range}(f) = A$.

Finally, let us observe that M accepts precisely the strings in this set:

$$\left\{ w \in \Sigma^* \,:\, \begin{array}{l} \text{there exists } x \in \Sigma^* \text{ such that } w = f(x) \\ \text{and } w > f(z) \text{ for all } z < x \end{array} \right\}. \tag{17.8}$$

The fact that this set is infinite follows from the assumption that $A = \text{range}(f)$ is infinite—for if the set were finite, there would necessarily be a maximal output of f with respect to the lexicographic ordering of Σ^*, contradicting the assumption that $\text{range}(f)$ is infinite.

The language $B = \text{L}(M)$ therefore satisfies the requirements of the corollary, which completes the proof. □