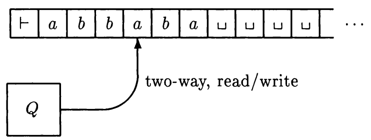
only realized in physical form several years later, the notion was definitely present in Turing's theoretical work in the 1930s.

Informal Description of Turing Machines

We describe here a deterministic, one-tape Turing machine. This is the standard off-the-shelf model. There are many variations, apparently more powerful or less powerful but in reality not. We will consider some of these in Lecture 30.

A TM has a finite set of states Q, a semi-infinite tape that is delimited on the left end by an endmarker $\vdash$ and is infinite to the right, and a head that can move left and right over the tape, reading and writing symbols.



The input string is of finite length and is initially written on the tape in contiguous tape cells snug up against the left endmarker. The infinitely many cells to the right of the input all contain a special blank symbol $\sqcup$.

The machine starts in its start state s with its head scanning the left endmarker. In each step it reads the symbol on the tape under its head. Depending on that symbol and the current state, it writes a new symbol on that tape cell, moves its head either left or right one cell, and enters a new state. The action it takes in each situation is determined by a transition function δ. It *accepts* its input by entering a special accept state t and *rejects* by entering a special reject state r. On some inputs it may run infinitely without ever accepting or rejecting, in which case it is said to *loop* on that input.

Formal Definition of Turing Machines

Formally, a *deterministic one-tape Turing machine* is a 9-tuple

$$M = (Q,\ \Sigma,\ \Gamma,\ \vdash,\ \sqcup,\ \delta,\ s,\ t,\ r),$$

where

- Q is a finite set (the *states*);
- Σ is a finite set (the *input alphabet*);
- Γ is a finite set (the *tape alphabet*) containing Σ as a subset;
- $\sqcup \in \Gamma - \Sigma$, the *blank symbol*;
- $\vdash \in \Gamma - \Sigma$, the *left endmarker*;
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$, the *transition function*;
- $s \in Q$, the *start state*;
- $t \in Q$, the *accept state*; and
- $r \in Q$, the *reject state*, $r \neq t$.

Intuitively, $\delta(p, a) = (q, b, d)$ means, "When in state p scanning symbol a, write b on that tape cell, move the head in direction d, and enter state q." The symbols L and R stand for left and right, respectively.

We restrict TMs so that the left endmarker is never overwritten with another symbol and the machine never moves off the tape to the left of the endmarker; that is, we require that for all $p \in Q$ there exists $q \in Q$ such that

$$\delta(p, \vdash) = (q, \vdash, R). \tag{28.2}$$

We also require that once the machine enters its accept state, it never leaves it, and similarly for its reject state; that is, for all $b \in \Gamma$ there exist $c, c' \in \Gamma$ and $d, d' \in \{L, R\}$ such that

$$\begin{aligned} \delta(t, b) &= (t, c, d), \\ \delta(r, b) &= (r, c', d'). \end{aligned} \tag{28.3}$$

We sometimes refer to the state set and transition function collectively as the *finite control*.

Example 28.1 Here is a TM that accepts the non-context-free set $\{a^n b^n c^n \mid n \geq 0\}$. Informally, the machine starts in its start state s, then scans to the right over the input string, checking that it is of the form $a^*b^*c^*$. It doesn't write anything on the way across (formally, it writes the same symbol it reads). When it sees the first blank symbol $\sqcup$, it overwrites it with a right endmarker $\dashv$. Now it scans left, erasing the first c it sees, then the first b it sees, then the first a it sees, until it comes to the $\vdash$. It then scans right, erasing one a, one b, and one c. It continues to sweep left and right over the input, erasing one occurrence of each letter in each pass. If on some pass it sees at least one occurrence of one of the letters and no occurrences of another, it rejects. Otherwise, it eventually erases all the letters and makes one pass between $\vdash$ and $\dashv$ seeing only blanks, at which point it accepts.

Formally, this machine has

$$Q = \{s, q_1, \ldots, q_{10}, t, r\},$$

$$\Sigma = \{a, b, c\},$$
$$\Gamma = \Sigma \cup \{\vdash, \sqcup, \dashv\}.$$

There is nothing special about $\dashv$; it is just an extra useful symbol in the tape alphabet. The transition function δ is specified by the following table:

	$\vdash$	a	b	c	$\sqcup$	$\dashv$
s	$(s, \vdash, R)$	(s, a, R)	(q_1, b, R)	(q_2, c, R)	$(q_3, \dashv, L)$	$-$
q_1	$-$	$(r, -, -)$	(q_1, b, R)	(q_2, c, R)	$(q_3, \dashv, L)$	$-$
q_2	$-$	$(r, -, -)$	$(r, -, -)$	(q_2, c, R)	$(q_3, \dashv, L)$	$-$
q_3	$(t, -, -)$	$(r, -, -)$	$(r, -, -)$	$(q_4, \sqcup, L)$	$(q_3, \sqcup, L)$	$-$
q_4	$(r, -, -)$	$(r, -, -)$	$(q_5, \sqcup, L)$	(q_4, c, L)	$(q_4, \sqcup, L)$	$-$
q_5	$(r, -, -)$	$(q_6, \sqcup, L)$	(q_5, b, L)	$-$	$(q_5, \sqcup, L)$	$-$
q_6	$(q_7, \vdash, R)$	(q_6, a, L)	$-$	$-$	$(q_6, \sqcup, L)$	$-$
q_7	$-$	$(q_8, \sqcup, R)$	$(r, -, -)$	$(r, -, -)$	$(q_7, \sqcup, R)$	$(t, -, -)$
q_8	$-$	(q_8, a, R)	$(q_9, \sqcup, R)$	$(r, -, -)$	$(q_8, \sqcup, R)$	$(r, -, -)$
q_9	$-$	$-$	(q_9, b, R)	$(q_{10}, \sqcup, R)$	$(q_9, \sqcup, R)$	$(r, -, -)$
q_{10}	$-$	$-$	$-$	(q_{10}, c, R)	$(q_{10}, \sqcup, R)$	$(q_3, \dashv, L)$

The symbol $-$ in the table above means "don't care." The transitions for t and r are not included in the table—just define them to be anything satisfying the restrictions (28.2) and (28.3). $\square$

Configurations and Acceptance

At any point in time, the read/write tape of the Turing machine M contains a semi-infinite string of the form $y\sqcup^\omega$, where $y \in \Gamma^*$ (y is a finite-length string) and $\sqcup^\omega$ denotes the semi-infinite string

$$\sqcup \sqcup \sqcup \sqcup \sqcup \sqcup \sqcup \sqcup \cdots.$$

(Here ω denotes the smallest infinite ordinal.) Although the string is infinite, it always has a finite representation, since all but finitely many of the symbols are the blank symbol $\sqcup$.

We define a *configuration* to be an element of $Q \times \{y\sqcup^\omega \mid y \in \Gamma^*\} \times \mathbb{N}$, where $\mathbb{N} = \{0, 1, 2, \ldots\}$. A configuration is a global state giving a snapshot of all relevant information about a TM computation at some instant in time. The configuration (p, z, n) specifies a current state p of the finite control, current tape contents z, and current position of the read/write head $n \geq 0$. We usually denote configurations by α, β, γ.

The *start configuration* on input $x \in \Sigma^*$ is the configuration

$$(s, \vdash x\sqcup^\omega, 0).$$

The last component 0 means that the machine is initially scanning the left endmarker $\vdash$.

One can define a *next configuration relation* $\xrightarrow[M]{1}$ as with PDAs. For a string $z \in \Gamma^\omega$, let z_n be the nth symbol of z (the leftmost symbol is z_0), and let $\mathrm{s}_b^n(z)$ denote the string obtained from z by substituting b for z_n at position n. For example,

$$\mathrm{s}_b^4(\vdash b\,a\,a\,a\,c\,a\,b\,c\,a\,\cdots) = \vdash b\,a\,a\,b\,c\,a\,b\,c\,a\,\cdots.$$

The relation $\xrightarrow[M]{1}$ is defined by

$$(p, z, n) \xrightarrow[M]{1} \begin{cases} (q, \mathrm{s}_b^n(z), n-1) & \text{if } \delta(p, z_n) = (q, b, L), \\ (q, \mathrm{s}_b^n(z), n+1) & \text{if } \delta(p, z_n) = (q, b, R). \end{cases}$$

Intuitively, if the tape contains z and if M is in state p scanning the nth tape cell, and δ says to print b, go left, and enter state q, then after that step the tape will contain $\mathrm{s}_b^n(z)$, the head will be scanning the $n-1$st tape cell, and the new state will be q.

We define the reflexive transitive closure $\xrightarrow[M]{*}$ of $\xrightarrow[M]{1}$ inductively, as usual:

- $\alpha \xrightarrow[M]{0} \alpha$,
- $\alpha \xrightarrow[M]{n+1} \beta$ if $\alpha \xrightarrow[M]{n} \gamma \xrightarrow[M]{1} \beta$ for some γ, and
- $\alpha \xrightarrow[M]{*} \beta$ if $\alpha \xrightarrow[M]{n} \beta$ for some $n \geq 0$.

The machine M is said to *accept* input $x \in \Sigma^*$ if

$$(s, \vdash x\sqcup^\omega, 0) \xrightarrow[M]{*} (t, y, n)$$

for some y and n, and *reject* x if

$$(s, \vdash x\sqcup^\omega, 0) \xrightarrow[M]{*} (r, y, n)$$

for some y and n. It is said to *halt* on input x if it either accepts x or rejects x. As with PDAs, this is just a mathematical definition; the machine doesn't really grind to a halt in the literal sense. It is possible that it neither accepts nor rejects, in which case it is said to *loop* on input x. A Turing machine is said to be *total* if it halts on all inputs; that is, if for all inputs it either accepts or rejects. The set $L(M)$ denotes the set of strings accepted by M.

We call a set of strings

- *recursively enumerable* (r.e.) if it is $L(M)$ for some Turing machine M,
- *co-r.e.* if its complement is r.e., and

- *recursive* if it is $L(M)$ for some *total* Turing machine M.

In common parlance, the term "recursive" usually refers to an algorithm that calls itself. The definition above has nothing to do with this usage. As used here, it is just a name for a set accepted by a Turing machine that always halts.

We will see lots of examples next time.

Historical Notes

Church's thesis is often referred to as the Church–Turing thesis, although Alonzo Church was the first to formulate it explicitly [25]. The thesis was based on Church and Kleene's observation that the λ-calculus and the μ-recursive functions of Gödel and Herbrand were computationally equivalent [25]. Church was apparently unaware of Turing's work at the time of the writing of [25], or if he was, he failed to mention it. Turing, on the other hand, cited Church's paper [25] explicitly in [120], and apparently considered his machines to be a much more compelling definition of computability. In an appendix to [120], Turing outlined a proof of the computational equivalence of Turing machines and the λ-calculus.

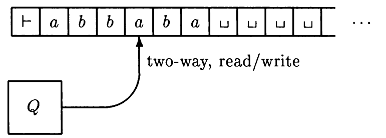
Lecture 29

More on Turing Machines

Last time we defined deterministic one-tape Turing machines:

$$M \quad = \quad (Q,\ \Sigma,\ \Gamma,\ \vdash,\ \sqcup,\ \delta,\ s,\ t,\ r)$$

states
input alphabet
tape alphabet
left endmarker
blank symbol
transition function
start state
accept state
reject state



In each step, based on the current tape symbol it is reading and its current state, it prints a new symbol on the tape, moves its head either left or right, and enters a new state. This action is specified formally by the transition function

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}.$$

Intuitively, $\delta(p,a) = (q,b,d)$ means, "When in state p scanning symbol a, write b on that tape cell, move the head in direction d, and enter state q."

We defined a *configuration* to be a triple (p,z,n) where p is a state, z is a semi-infinite string of the form $y\sqcup^\omega$, $y \in \Sigma^*$, describing the contents of the tape, and n is a natural number denoting a tape head position.

The transition function δ was used to define the *next configuration relation* $\xrightarrow[M]{1}$ on configurations and its reflexive transitive closure $\xrightarrow[M]{*}$. The machine M *accepts* input $x \in \Sigma^*$ if

$$(s, \vdash x\sqcup^\omega, 0) \xrightarrow[M]{*} (t, y, n)$$

for some y and n, and *rejects* input x if

$$(s, \vdash x\sqcup^\omega, 0) \xrightarrow[M]{*} (r, y, n)$$

for some y and n. The left configuration above is the *start configuration* on input x. Recall that we restricted TMs so that once a TM enters its accept state, it may never leave it, and similarly for its reject state. If M never enters its accept or reject state on input x, it is said to *loop* on input x. It is said to *halt* on input x if it either accepts or rejects. A TM that halts on all inputs is called *total*.

Define the set

$$L(M) = \{x \in \Sigma^* \mid M \text{ accepts } x\}.$$

This is called the set *accepted by* M. A subset of Σ^* is called *recursively enumerable* (r.e.) if it is $L(M)$ for some M. A set is called *recursive* if it is $L(M)$ for some total M.

For now, the terms *r.e.* and *recursive* are just technical terms describing the sets accepted by TMs and total TMs, respectively; they have no other significance. We will discuss the origin of this terminology in Lecture 30.

Example 29.1 Consider the non-CFL $\{ww \mid w \in \{a,b\}^*\}$. It is a recursive set, because we can give a total TM M for it. The machine M works as follows. On input x, it scans out to the first blank symbol $\sqcup$, counting the number of symbols mod 2 to make sure x is of even length and rejecting immediately if not. It lays down a right endmarker $\dashv$, then repeatedly scans back and forth over the input. In each pass from right to left, it marks the first unmarked a or b it sees with ´. In each pass from left to right, it marks the first unmarked a or b it sees with `. It continues this until all symbols are marked. For example, on input

$$a\,a\,b\,b\,a\,a\,a\,b\,b\,a$$

the initial tape contents are

$$\vdash a\,a\,b\,b\,a\,a\,a\,b\,b\,a \sqcup\sqcup\sqcup\cdots$$

and the following are the tape contents after the first few passes.

$$\vdash a\,a\,b\,b\,a\,a\,a\,b\,b\,\acute{a} \dashv \sqcup\sqcup\sqcup\cdots$$
$$\vdash \grave{a}\,a\,b\,b\,a\,a\,a\,b\,b\,\acute{a} \dashv \sqcup\sqcup\sqcup\cdots$$
$$\vdash \grave{a}\,a\,b\,b\,a\,a\,a\,b\,\acute{b}\,\acute{a} \dashv \sqcup\sqcup\sqcup\cdots$$
$$\vdash \grave{a}\,\grave{a}\,b\,b\,a\,a\,a\,b\,\acute{b}\,\acute{a} \dashv \sqcup\sqcup\sqcup\cdots$$
$$\vdash \grave{a}\,\grave{a}\,b\,b\,a\,a\,a\,\acute{b}\,\acute{b}\,\acute{a} \dashv \sqcup\sqcup\sqcup\cdots$$

Marking a with ` formally means writing the symbol $\grave{a} \in \Gamma$; thus

$$\Gamma = \{a,\ b,\ \vdash,\ \sqcup,\ \dashv,\ \grave{a},\ \grave{b},\ \acute{a},\ \acute{b}\}.$$

When all symbols are marked, we have the first half of the input string marked with ` and the second half marked with ´.

$$\vdash \grave{a}\,\grave{a}\,\grave{b}\,\grave{b}\,\grave{a}\,\acute{a}\,\acute{a}\,\acute{b}\,\acute{b}\,\acute{a} \dashv \sqcup\sqcup\sqcup\cdots$$

The reason we did this was to find the center of the input string.

The machine then repeatedly scans left to right over the input. In each pass it erases the first symbol it sees marked with ` but remembers that symbol in its finite control (to "erase" really means to write the blank symbol $\sqcup$). It then scans forward until it sees the first symbol marked with ´, checks that that symbol is the same, and erases it. If the two symbols are not the same, it rejects. Otherwise, when it has erased all the symbols, it accepts. In our example, the following would be the tape contents after each pass.

$$\grave{a}\,\grave{a}\,\grave{b}\,\grave{b}\,\grave{a}\,\acute{a}\,\acute{a}\,\acute{b}\,\acute{b}\,\acute{a} \dashv \sqcup\sqcup\sqcup\cdots$$
$$\sqcup\,\grave{a}\,\grave{b}\,\grave{b}\,\grave{a}\,\sqcup\,\acute{a}\,\acute{b}\,\acute{b}\,\acute{a} \dashv \sqcup\sqcup\sqcup\cdots$$
$$\sqcup\sqcup\,\grave{b}\,\grave{b}\,\grave{a}\,\sqcup\sqcup\,\acute{b}\,\acute{b}\,\acute{a} \dashv \sqcup\sqcup\sqcup\cdots$$
$$\sqcup\sqcup\sqcup\,\grave{b}\,\grave{a}\,\sqcup\sqcup\sqcup\,\acute{b}\,\acute{a} \dashv \sqcup\sqcup\sqcup\cdots$$
$$\sqcup\sqcup\sqcup\sqcup\,\grave{a}\,\sqcup\sqcup\sqcup\sqcup\,\acute{a} \dashv \sqcup\sqcup\sqcup\cdots$$
$$\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup \dashv \sqcup\sqcup\sqcup\cdots$$

□

Example 29.2 We want to construct a total TM that accepts its input string if the length of the string is prime. This language is not regular or context-free. We will give a TM implementation of the *sieve of Eratosthenes*, which can be described informally as follows. Say we want to check whether n is prime. We write down all the numbers from 2 to n in order, then repeat the following: find the smallest number in the list, declare it prime, then cross off all multiples of that number. Repeat until each number in the list has been either declared prime or crossed off as a multiple of a smaller prime.

For example, to check whether 23 is prime, we would start with all the numbers from 2 to 23:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23

In the first pass, we cross off multiples of 2:

$\cancel{2}$ 3 $\cancel{4}$ 5 $\cancel{6}$ 7 $\cancel{8}$ 9 $\cancel{10}$ 11 $\cancel{12}$ 13 $\cancel{14}$ 15 $\cancel{16}$ 17 $\cancel{18}$ 19 $\cancel{20}$ 21 $\cancel{22}$ 23

The smallest number remaining is 3, and this is prime. In the second pass we cross off multiples of 3:

$\cancel{2}$ $\cancel{3}$ $\cancel{4}$ 5 $\cancel{6}$ 7 $\cancel{8}$ $\cancel{9}$ $\cancel{10}$ 11 $\cancel{12}$ 13 $\cancel{14}$ $\cancel{15}$ $\cancel{16}$ 17 $\cancel{18}$ 19 $\cancel{20}$ $\cancel{21}$ $\cancel{22}$ 23

Then 5 is the next prime, so we cross off multiples of 5; and so forth. Since 23 is prime, it will never be crossed off as a multiple of anything smaller, and eventually we will discover that fact when everything smaller has been crossed off.

Now we show how to implement this on a TM. Suppose we have a^p written on the tape. We illustrate the algorithm with $p = 23$.

$$\vdash a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a \sqcup\sqcup\sqcup\cdots$$

If $p = 0$ or $p = 1$, reject. We can determine this by looking at the first three cells of the tape. Otherwise, there are at least two a's. Erase the first a, scan right to the end of the input, and replace the last a in the input string with the symbol \$. We now have an a in positions $2, 3, 4, \ldots, p - 1$ and \$ at position p.

$$\vdash \sqcup a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,\$ \sqcup\sqcup\sqcup\cdots$$

Now we repeat the following loop. Starting from the left endmarker $\vdash$, scan right and find the first nonblank symbol, say occurring at position m. Then m is prime (this is an invariant of the loop). If this symbol is the \$, we are done: $p = m$ is prime, so we halt and accept. Otherwise, the symbol is an a. Mark it with a $\widehat{\ }$ and everything between there and the left endmarker with $\acute{\ }$.

$$\vdash \acute{\sqcup}\widehat{a}\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,\$ \sqcup\sqcup\sqcup\cdots$$

We will now enter an inner loop to erase all the symbols occurring at positions that are multiples of m. First, erase the a under the $\widehat{\ }$. (Formally, just write the symbol $\widehat{\sqcup}$.)

$$\vdash \acute{\sqcup}\widehat{\sqcup}a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,\$ \sqcup\sqcup\sqcup\cdots$$

Shift the marks to the right one at a time a distance equal to the number of marks. This can be done by shuttling back and forth, erasing marks on the left and writing them on the right. We know when we are done because the $\widehat{\ }$ is the last mark moved.

$$\vdash \sqcup\sqcup\acute{a}\widehat{a}\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,\$ \sqcup\sqcup\sqcup\cdots$$

When this is done, erase the symbol under the $\widehat{\ }$. This is the symbol occurring at position $2m$.

$$\vdash \sqcup\sqcup\acute{a}\widehat{\sqcup}a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,a\,\$ \sqcup\sqcup\sqcup\cdots$$

Keep shifting the marks and erasing the symbol under the $\widehat{\ }$ in this fashion until we reach the end.

$$\vdash \sqcup\sqcup a \sqcup a \sqcup a \sqcup a \sqcup a \sqcup a \sqcup a \sqcup a \sqcup a \sqcup a \sqcup \acute{\$}\,\widehat{\sqcup}\sqcup\sqcup\cdots$$

If we find ourselves at the end of the string wanting to erase the \$, reject—p is a multiple of m but not equal to m. Otherwise, go back to the left and repeat. Find the first nonblank symbol and mark it and everything to its left.

$$\vdash \acute{\sqcup}\acute{\sqcup}\widehat{a} \sqcup a \sqcup a \sqcup a \sqcup a \sqcup a \sqcup a \sqcup a \sqcup a \sqcup a \sqcup \$ \sqcup\sqcup\sqcup\cdots$$

Alternately erase the symbol under the $\widehat{\ }$ and shift the marks until we reach the end of the string.

$$\vdash \sqcup\sqcup\sqcup\sqcup a \sqcup a \sqcup\sqcup\sqcup a \sqcup a \sqcup\sqcup\sqcup a \sqcup a \sqcup\sqcup\acute{\sqcup}\acute{\$}\,\widehat{\sqcup}\sqcup\sqcup\cdots$$

Go back to the left and repeat.

$$\vdash \acute{\sqcup}\acute{\sqcup}\acute{\sqcup}\acute{\sqcup}\widehat{a} \sqcup a \sqcup\sqcup\sqcup a \sqcup a \sqcup\sqcup\sqcup a \sqcup a \sqcup\sqcup\sqcup \$ \sqcup\sqcup\sqcup\cdots$$

If we ever try to erase the \$, reject—p is not prime. If we manage to erase all the a's, accept. $\square$

Recursive and R.E. Sets

Recall that a set A is *recursively enumerable* (r.e.) if it is accepted by a TM and *recursive* if it is accepted by a *total* TM (one that halts on all inputs).

The recursive sets are closed under complement. (The r.e. sets are not, as we will see later.) That is, if A is recursive, then so is $\sim A = \Sigma^* - A$. To see this, suppose A is recursive. Then there exists a total TM M such that $L(M) = A$. By switching the accept and reject states of M, we get a total machine M' such that $L(M') = \sim A$.

This construction does not give the complement if M is not total. This is because "rejecting" and "not accepting" are not synonymous for nontotal machines. To reject, a machine must enter its reject state. If M' is obtained from M by just switching the accept and reject states, then M' will accept the strings that M rejects and reject the strings that M accepts; but M' will still loop on the same strings that M loops on, so these strings are not accepted or rejected by either machine.

Every recursive set is r.e. but not necessarily vice versa. In other words, not every TM is equivalent to a total TM. We will prove this in Lecture 31. However, if both A and $\sim A$ are r.e., then A is recursive. To see this, suppose both A and $\sim A$ are r.e. Let M and M' be TMs such that $L(M) = A$ and $L(M') = \sim A$. Build a new machine N that on input x runs both M and

M' simultaneously on two different tracks of its tape. Formally, the tape alphabet of N contains symbols

$$\boxed{\begin{array}{c} a \\ \hline c \end{array}} \quad \boxed{\begin{array}{c} \widehat{a} \\ \hline c \end{array}} \quad \boxed{\begin{array}{c} a \\ \hline \widehat{c} \end{array}} \quad \boxed{\begin{array}{c} \widehat{a} \\ \hline \widehat{c} \end{array}}$$

where a is a tape symbol of M and c is a tape symbol of M'. Thus N's tape may contain a string of the form

$$\begin{array}{|c|c|c|c|c|c|c|c|} \hline b & \widehat{a} & b & a & b & a & b & a \\ \hline c & c & c & d & d & c & \widehat{c} & d \\ \hline \end{array} \; \cdots$$

for example. The extra marks $\widehat{\ }$ are placed on the tape to indicate the current positions of the simulated read/write heads of M and M'. The machine N alternately performs a step of M and a step of M', shuttling back and forth between the two simulated tape head positions of M and M' and updating the tape. The current states and transition information of M and M' can be stored in N's finite control. If the machine M ever accepts, then N immediately accepts. If M' ever accepts, then N immediately rejects. Exactly one of those two events must eventually occur, depending on whether $x \in A$ or $x \in {\sim}A$, since $L(M) = A$ and $L(M') = {\sim}A$. Then N halts on all inputs and $L(N) = A$.

Decidability and Semidecidability

A property P of strings is said to be *decidable* if the set of all strings having property P is a recursive set; that is, if there is a total Turing machine that accepts input strings that have property P and rejects those that do not. A property P is said to be *semidecidable* if the set of strings having property P is an r.e. set; that is, if there is a Turing machine that on input x accepts if x has property P and rejects or loops if not. For example, it is decidable whether a given string x is of the form ww, because we can construct a Turing machine that halts on all inputs and accepts exactly the strings of this form.

Although you often hear them switched, the adjectives *recursive* and *r.e.* are best applied to sets and *decidable* and *semidecidable* to properties. The two notions are equivalent, since

$$P \text{ is decidable} \iff \{x \mid P(x)\} \text{ is recursive.}$$
$$A \text{ is recursive} \iff \text{“}x \in A\text{” is decidable,}$$

$$P \text{ is semidecidable} \iff \{x \mid P(x)\} \text{ is r.e.,}$$
$$A \text{ is r.e.} \iff \text{“}x \in A\text{” is semidecidable.}$$

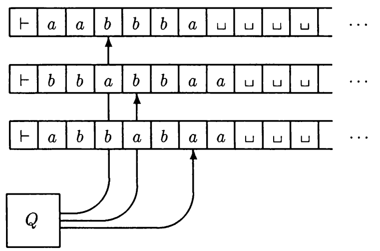
Lecture 30

Equivalent Models

As mentioned, the concept of computability is remarkably robust. As evidence of this, we will present several different flavors of Turing machines that at first glance appear to be significantly more or less powerful than the basic model defined in Lecture 29 but in fact are computationally equivalent.

Multiple Tapes

First, we show how to simulate multitape Turing machines with single-tape Turing machines. Thus extra tapes don't add any power. A three-tape machine is similar to a one-tape TM except that it has three semi-infinite tapes and three independent read/write heads. Initially, the input occupies the first tape and the other two are blank. In each step, the machine reads the three symbols under its heads, and based on this information and the current state, it prints a symbol on each tape, moves the heads (they don't all have to move in the same direction), and enters a new state.



Its transition function is of type

$$\delta : Q \times \Gamma^3 \to Q \times \Gamma^3 \times \{L, R\}^3.$$

Say we have such a machine M. We build a single-tape machine N simulating M as follows. The machine N will have an expanded tape alphabet allowing us to think of its tape as divided into three tracks. Each track will contain the contents of one of M's tapes. We also mark exactly one symbol on each track to indicate that this is the symbol currently being scanned on the corresponding tape of M. The configuration of M illustrated above might be simulated by the following configuration of N.

$\vdash$	$\vdash$	a	a	$\widehat{b}$	b	b	a	⊔	⊔	⊔	⊔	···
	$\vdash$	b	b	a	$\widehat{b}$	b	a	a	⊔	⊔	⊔	
	$\vdash$	a	b	b	a	b	$\widehat{a}$	a	⊔	⊔	⊔	

A tape symbol of N is either $\vdash$, an element of Σ, or a triple

c
d
e

where c, d, e are tape symbols of M, each either marked or unmarked. Formally, we might take the tape alphabet of N to be

$$\Sigma \cup \{\vdash\} \cup (\Gamma \cup \Gamma')^3,$$

where

$$\Gamma' \stackrel{\text{def}}{=} \{\widehat{c} \mid c \in \Gamma\}.$$

The three elements of $\Gamma \cup \Gamma'$ stand for the symbols in corresponding positions on the three tapes of M, either marked or unmarked, and

⊔
⊔
⊔

is the blank symbol of N.

On input $x = a_1 a_2 \cdots a_n$, N starts with tape contents

$$\vdash a_1\, a_2\, a_3\, \cdots\, a_n \sqcup \sqcup \sqcup \cdots.$$

It first copies the input to its top track and fills in the bottom two tracks with blanks. It also shifts everything right one cell so that it can fill in the leftmost cells on the three tracks with the simulated left endmarker of M, which it marks with $\widehat{\ }$ to indicate the position of the heads in the starting configuration of M.

$\vdash$	$\widehat{\vdash}$	a_1	a_2	a_3	a_4	···	a_n	⊔	⊔	···
	$\widehat{\vdash}$	⊔	⊔	⊔	⊔		⊔	⊔	⊔	
	$\widehat{\vdash}$	⊔	⊔	⊔	⊔		⊔	⊔	⊔	

Each step of M is simulated by several steps of N. To simulate one step of M, N starts at the left of the tape, then scans out until it sees all three marks, remembering the marked symbols in its finite control. When it has seen all three, it determines what to do according to M's transition function δ, which it has encoded in its finite control. Based on this information, it goes back to all three marks, rewriting the symbols on each track and moving the marks appropriately. It then returns to the left end of the tape to simulate the next step of M.

Two-Way Infinite Tapes

Two-way infinite tapes do not add any power. Just fold the tape someplace and simulate it on two tracks of a one-way infinite tape:

···	a	a	b	a	a	b	b	a	b	a	a	b	b	b	b	b	···

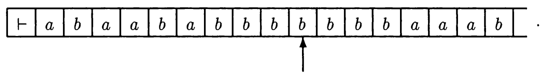
fold here

$\vdash$	a	b	b	a	a	b	a	a	···
	b	a	a	b	b	b	b	b	

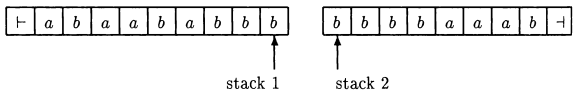
The bottom track is used to simulate the original machine when its head is to the right of the fold, and the top track is used to simulate the machine when its head is to the left of the fold, moving in the opposite direction.

Two Stacks

A machine with a two-way, *read-only* input head and two stacks is as powerful as a Turing machine. Intuitively, the computation of a one-tape TM can be simulated with two stacks by storing the tape contents to the left of the head on one stack and the tape contents to the right of the head on the other stack. The motion of the head is simulated by popping a symbol off one stack and pushing it onto the other. For example,



is simulated by



Counter Automata

A *k-counter automaton* is a machine equipped with a two-way read-only input head and k integer counters. Each counter can store an arbitrary nonnegative integer. In each step, the automaton can independently increment or decrement its counters and test them for 0 and can move its input head one cell in either direction. It cannot write on the tape.

A stack can be simulated with two counters as follows. We can assume without loss of generality that the stack alphabet of the stack to be simulated contains only two symbols, say 0 and 1. This is because we can encode finitely many stack symbols as binary numbers of fixed length, say m; then pushing or popping one stack symbol is simulated by pushing or popping m binary digits. Then the contents of the stack can be regarded as a binary number whose least significant bit is on top of the stack. The simulation maintains this number in the first of the two counters and uses the second to effect the stack operations. To simulate pushing a 0 onto the stack, we need to double the value in the first counter. This is done by entering a loop that repeatedly subtracts one from the first counter and adds two to the second until the first counter is 0. The value in the second counter is then twice the original value in the first counter. We can then transfer that value back to the first counter, or just switch the roles of the two counters. To push 1, the operation is the same, except the value of the second counter is incremented once at the end. To simulate popping, we need to divide the counter value by two; this is done by decrementing one counter while incrementing the other counter every second step. Testing the parity of the original counter contents tells whether a simulated 1 or 0 was popped.

Since a two-stack machine can simulate an arbitrary TM, and since two counters can simulate a stack, it follows that a four-counter automaton can simulate an arbitrary TM.

However, we can do even better: a two-counter automaton can simulate a four-counter automaton. When the four-counter automaton has the values i, j, k, ℓ in its counters, the two-counter automaton will have the value $2^i 3^j 5^k 7^\ell$ in its first counter. It uses its second counter to effect the counter operations of the four-counter automaton. For example, if the four-counter automaton wanted to add one to k (the value of the third counter), then the two-counter automaton would have to multiply the value in its first counter by 5. This is done in the same way as above, adding 5 to the second counter for every 1 we subtract from the first counter. To simulate a test for zero, the two-counter automaton has to determine whether the value in its first counter is divisible by 2, 3, 5, or 7, respectively, depending on which counter of the four-counter automaton is being tested.

Combining these simulations, we see that two-counter automata are as powerful as arbitrary Turing machines. However, as you can imagine, it takes an enormous number of steps of the two-counter automaton to simulate one step of the Turing machine.

One-counter automata are not as powerful as arbitrary TMs, although they can accept non-CFLs. For example, the set $\{a^n b^n c^n \mid n \geq 0\}$ can be accepted by a one-counter automaton.

Enumeration Machines

We defined the recursively enumerable (r.e.) sets to be those sets accepted by Turing machines. The term *recursively enumerable* comes from a different but equivalent formalism embodying the idea that the elements of an r.e. set can be enumerated one at a time in a mechanical fashion.

Define an *enumeration machine* as follows. It has a finite control and two tapes, a read/write *work tape* and a write-only *output tape*. The work tape head can move in either direction and can read and write any element of Γ. The output tape head moves right one cell when it writes a symbol, and it can only write symbols in Σ. There is no input and no accept or reject state. The machine starts in its start state with both tapes blank. It moves according to its transition function like a TM, occasionally writing symbols on the output tape as determined by the transition function. At some point it may enter a special *enumeration state*, which is just a distinguished state of its finite control. When that happens, the string currently written on the output tape is said to be *enumerated*. The output tape is then automatically erased and the output head moved back to the beginning of the tape (the work tape is left intact), and the machine continues from that point. The machine runs forever. The set $L(E)$ is defined to be the set of all strings in Σ^* that are ever enumerated by the enumeration machine E. The machine might never enter its enumeration state, in which case $L(E) = \varnothing$, or it might enumerate infinitely many strings. The same string may be enumerated more than once.

Enumeration machines and Turing machines are equivalent in computational power:

Theorem 30.1 *The family of sets enumerated by enumeration machines is exactly the family of r.e. sets. In other words, a set is $L(E)$ for some enumeration machine E if and only if it is $L(M)$ for some Turing machine M.*

Proof. We show first that given an enumeration machine E, we can construct a Turing machine M such that $L(M) = L(E)$. Let M on input x copy x to one of three tracks on its tape, then simulate E, using the other two tracks to record the contents of E's work tape and output tape. For every string enumerated by E, M compares this string to x and accepts if they match. Then M accepts its input x iff x is ever enumerated by E, so the set of strings accepted by M is exactly the set of strings enumerated by E.

Conversely, given a TM M, we can construct an enumeration machine E such that $L(E) = L(M)$. We would like E somehow to simulate M on all possible strings in Σ^* and enumerate those that are accepted.

Here is an approach that doesn't quite work. The enumeration machine E writes down the strings in Σ^* one by one on the bottom track of its work tape in some order. For every input string x, it simulates M on input x, using the top track of its work tape to do the simulation. If M accepts x, E copies x to its output tape and enters its enumeration state. It then goes on to the next string.

The problem with this procedure is that M might not halt on some input x, and then E would be stuck simulating M on x forever and would never move on to strings later in the list (and it is impossible to determine in general whether M will ever halt on x, as we will see in Lecture 31). Thus E should not just list the strings in Σ^* in some order and simulate M on

those inputs one at a time, waiting for each simulation to halt before going on to the next, because the simulation might never halt.

The solution to this problem is *timesharing*. Instead of simulating M on the input strings one at a time, the enumeration machine E should run several simulations at once, working a few steps on each simulation and then moving on to the next. The work tape of E can be divided into segments separated by a special marker $\# \in \Gamma$, with a simulation of M on a different input string running in each segment. Between passes, E can move way out to the right, create a new segment, and start up a new simulation in that segment on the next input string. For example, we might have E simulate M on the first input for one step, then the first and second inputs for one step each, then the first, second, and third inputs for one step each, and so on. If any simulation needs more space than initially allocated in its segment, the entire contents of the tape to its right can be shifted to the right one cell. In this way M is eventually simulated on all input strings, even if some of the simulations never halt. $\square$

Historical Notes

Turing machines were invented by Alan Turing [120]. Originally they were presented in the form of enumeration machines, since Turing was interested in enumerating the decimal expansions of computable real numbers and values of real-valued functions. Turing also introduced the concept of nondeterminism in his original paper, although he did not develop the idea.

The basic properties of the r.e. sets were developed by Kleene [68] and Post [100, 101].

Counter automata were studied by Fischer [38], Fischer et al. [39], and Minsky [88].

Lecture 31

Universal Machines and Diagonalization

A Universal Turing Machine

Now we come to a crucial observation about the power of Turing machines: there exist Turing machines that can simulate other Turing machines whose descriptions are presented as part of the input. There is nothing mysterious about this; it is the same as writing a LISP interpreter in LISP.

First we need to fix a reasonable encoding scheme for Turing machines over the alphabet $\{0,1\}$. This encoding scheme should be simple enough that all the data associated with a machine M—the set of states, the transition function, the input and tape alphabets, the endmarker, the blank symbol, and the start, accept, and reject states—can be determined easily by another machine reading the encoded description of M. For example, if the string begins with the prefix

$$0^n10^m10^k10^s10^t10^r10^u10^v1,$$

this might indicate that the machine has n states represented by the numbers 0 to $n-1$; it has m tape symbols represented by the numbers 0 to $m-1$, of which the first k represent input symbols; the start, accept, and reject states are s, t, and r, respectively; and the endmarker and blank symbol are u and v, respectively. The remainder of the string can consist

of a sequence of substrings specifying the transitions in δ. For example, the substring

$$0^p10^a10^q10^b10$$

might indicate that δ contains the transition

$$((p,a),\ (q,b,L)),$$

the direction to move the head encoded by the final digit. The exact details of the encoding scheme are not important. The only requirements are that it should be easy to interpret and able to encode all Turing machines up to isomorphism.

Once we have a suitable encoding of Turing machines, we can construct a *universal Turing machine* U such that

$$L(U) \stackrel{\text{def}}{=} \{M\#x \mid x \in L(M)\}.$$

In other words, presented with (an encoding over $\{0,1\}$ of) a Turing machine M and (an encoding over $\{0,1\}$ of) a string x over M's input alphabet, the machine U accepts $M\#x$ iff M accepts x.[1] The symbol $\#$ is just a symbol in U's input alphabet other than 0 or 1 used to delimit M and x.

The machine U first checks its input $M\#x$ to make sure that M is a valid encoding of a Turing machine and x is a valid encoding of a string over M's input alphabet. If not, it immediately rejects.

If the encodings of M and x are valid, the machine U does a step-by-step simulation of M. This might work as follows. The tape of U is partitioned into three tracks. The description of M is copied to the top track and the string x to the middle track. The middle track will be used to hold the simulated contents of M's tape. The bottom track will be used to remember the current state of M and the current position of M's read/write head. The machine U then simulates M on input x one step at a time, shuttling back and forth between the description of M on its top track and the simulated contents of M's tape on the middle track. In each step, it updates M's state and simulated tape contents as dictated by M's transition function, which U can read from the description of M. If ever M halts and accepts or halts and rejects, then U does the same.

As we have observed, the string x over the input alphabet of M and its encoding over the input alphabet of U are two different things, since the two machines may have different input alphabets. If the input alphabet of

[1]Note that we are using the metasymbol M for both a Turing machine and its encoding over $\{0,1\}$ and the metasymbol x for both a string over M's input alphabet and its encoding over $\{0,1\}$. This is for notational convenience.

M is bigger than that of U, then each symbol of x must be encoded as a string of symbols over U's input alphabet. Also, the tape alphabet of M may be bigger than that of U, in which case each symbol of M's tape alphabet must be encoded as a string of symbols over U's tape alphabet. In general, each step of M may require many steps of U to simulate.

Diagonalization

We now show how to use a universal Turing machine in conjunction with a technique called *diagonalization* to prove that the halting and membership problems for Turing machines are undecidable. In other words, the sets

$$\mathrm{HP} \stackrel{\text{def}}{=} \{M\#x \mid M \text{ halts on } x\},$$

$$\mathrm{MP} \stackrel{\text{def}}{=} \{M\#x \mid x \in L(M)\}$$

are not recursive.

The technique of diagonalization was first used by Cantor at the end of the nineteenth century to show that there does not exist a one-to-one correspondence between the natural numbers $\mathbb{N}$ and its *power set*

$$2^{\mathbb{N}} = \{A \mid A \subseteq \mathbb{N}\},$$

the set of all subsets of $\mathbb{N}$. In fact, there does not even exist a function

$$f : \mathbb{N} \to 2^{\mathbb{N}}$$

that is onto. Here is how Cantor's argument went.

Suppose (for a contradiction) that such an onto function f did exist. Consider an infinite two-dimensional matrix indexed along the top by the natural numbers $0, 1, 2, \ldots$ and down the left by the sets $f(0), f(1), f(2), \ldots$. Fill in the matrix by placing a 1 in position i, j if j is in the set $f(i)$ and 0 if $j \notin f(i)$.

$$\begin{array}{c|cccccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & \cdots \\
\hline
f(0) & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & \\
f(1) & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & \\
f(2) & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & \\
f(3) & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & \\
f(4) & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & \cdots \\
f(5) & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & \\
f(6) & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & \\
f(7) & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & \\
f(8) & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & \\
f(9) & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & \\
\vdots & & & & & \vdots & & & & & & \ddots
\end{array}$$

The ith row of the matrix is a bit string describing the set $f(i)$. For example, in the above picture, $f(0) = \{0, 3, 4, 6, 8, 9, \ldots\}$ and $f(1) = \{2, 3, 5, 6, 9, \ldots\}$. By our (soon to be proved fallacious) assumption that f is onto, every subset of $\mathbb{N}$ appears as a row of this matrix.

But we can construct a new set that does not appear in the list by complementing the main diagonal of the matrix (hence the term *diagonalization*). Look at the infinite bit string down the main diagonal (in this example, $1011010010\cdots$) and take its Boolean complement (in this example, $0100101101\cdots$). This new bit string represents a set B (in this example, $B = \{1, 4, 6, 7, 9, \ldots\}$). But the set B does not appear anywhere in the list down the left side of the matrix, since it differs from every $f(i)$ on at least one element, namely i. This is a contradiction, since every subset of $\mathbb{N}$ was supposed to occur as a row of the matrix, by our assumption that f was onto.

This argument works not only for the natural numbers $\mathbb{N}$, but for any set A whatsoever. Suppose (for a contradiction) there existed an onto function from A to its power set:

$$f : A \to 2^A.$$

Let

$$B = \{x \in A \mid x \notin f(x)\}$$

(this is the formal way of *complementing the diagonal*). Then $B \subseteq A$. Since f is onto, there must exist $y \in A$ such that $f(y) = B$. Now we ask whether $y \in f(y)$ and discover a contradiction:

$$\begin{aligned} y \in f(y) &\iff y \in B && \text{since } B = f(y) \\ &\iff y \notin f(y) && \text{definition of } B. \end{aligned}$$

Thus no such f can exist.

Undecidability of the Halting Problem

We have discussed how to encode descriptions of Turing machines as strings in $\{0, 1\}^*$ so that these descriptions can be read and simulated by a universal Turing machine U. The machine U takes as input an encoding of a Turing machine M and a string x and simulates M on input x, and

- halts and accepts if M halts and accepts x,
- halts and rejects if M halts and rejects x, and
- loops if M loops on x.

The machine U doesn't do any fancy analysis on the machine M to try to determine whether or not it will halt. It just blindly simulates M step by step. If M doesn't halt on x, then U will just go on happily simulating M forever.

It is natural to ask whether we can do better than just a blind simulation. Might there be some way to analyze M to determine in advance, before doing the simulation, whether M would eventually halt on x? If U could say for sure in advance that M would not halt on x, then it could skip the simulation and save itself a lot of useless work. On the other hand, if U could ascertain that M *would* eventually halt on x, then it could go ahead with the simulation to determine whether M accepts or rejects. We could then build a machine U' that takes as input an encoding of a Turing machine M and a string x, and

- halts and accepts if M halts and accepts x,
- halts and rejects if M halts and rejects x, and
- halts and rejects if M loops on x.

This would say that $L(U') = L(U) = \mathrm{MP}$ is a recursive set.

Unfortunately, this is not possible in general. There are certainly machines for which it is possible to determine halting by some heuristic or other: machines for which the start state is the accept state, for example. However, there is no general method that gives the right answer for all machines.

We can prove this using Cantor's diagonalization technique. For $x \in \{0,1\}^*$, let M_x be the Turing machine with input alphabet $\{0,1\}$ whose encoding over $\{0,1\}^*$ is x. (If x is not a legal description of a TM with input alphabet $\{0,1\}^*$ according to our encoding scheme, we take M_x to be some arbitrary but fixed TM with input alphabet $\{0,1\}$, say a trivial TM with one state that immediately halts.) In this way we get a list

$$M_\epsilon, M_0, M_1, M_{00}, M_{01}, M_{10}, M_{11}, M_{100}, M_{101}, \ldots \qquad (31.1)$$

containing all possible Turing machines with input alphabet $\{0,1\}$ indexed by strings in $\{0,1\}^*$. We make sure that the encoding scheme is simple enough that a universal machine can determine M_x from x for the purpose of simulation.

Now consider an infinite two-dimensional matrix indexed along the top by strings in $\{0,1\}^*$ and down the left by TMs in the list (31.1). The matrix contains an H in position x, y if M_x halts on input y and an L if M_x loops on input y.

	ϵ	0	1	00	01	10	11	000	001	010	$\cdots$
M_ϵ	H	L	L	H	H	L	H	L	H	H	
M_0	L	L	H	H	L	H	H	L	L	H	
M_1	L	H	H	L	L	L	H	H	L	H	
M_{00}	L	H	L	H	H	L	H	H	L	L	
M_{01}	H	L	H	L	L	H	L	L	H	H	$\cdots$
M_{10}	H	L	H	H	L	H	H	H	L	H	
M_{11}	L	L	H	L	H	H	L	L	H	H	
M_{000}	H	H	H	L	H	H	H	L	H	L	
M_{001}	L	L	H	L	L	L	L	H	H	L	
M_{010}	H	H	L	L	H	L	L	H	L	L	
$\vdots$					$\vdots$						$\ddots$

The xth row of the matrix describes for each input string y whether or not M_x halts on y. For example, in the above picture, M_ϵ halts on inputs $\epsilon, 00, 01, 11, 001, 010, \ldots$ and does not halt on inputs $0, 1, 10, 000, \ldots$.

Suppose (for a contradiction) that there existed a *total* machine K accepting the set HP; that is, a machine that for any given x and y could determine the x,yth entry of the above table in finite time. Thus on input $M\#x$,

- K halts and accepts if M halts on x, and
- K halts and rejects if M loops on x.

Consider a machine N that on input $x \in \{0,1\}^*$

(i) constructs M_x from x and writes $M_x\#x$ on its tape;

(ii) runs K on input $M_x\#x$, accepting if K rejects and going into a trivial loop if K accepts.

Note that N is essentially complementing the diagonal of the above matrix. Then for any $x \in \{0,1\}^*$,

$$\begin{aligned} N \text{ halts on } x &\iff K \text{ rejects } M_x\#x && \text{definition of } N \\ &\iff M_x \text{ loops on } x && \text{assumption about } K. \end{aligned}$$

This says that N's behavior is different from every M_x on at least one string, namely x. But the list (31.1) was supposed to contain all Turing machines over the input alphabet $\{0,1\}$, including N. This is a contradiction. $\square$

The fallacious assumption that led to the contradiction was that it was possible to determine the entries of the matrix effectively; in other words, that there existed a Turing machine K that given M and x could determine in a finite time whether or not M halts on x.

One can always simulate a given machine on a given input. If the machine ever halts, then we will know this eventually, and we can stop the simulation and say that it halted; but if not, there is no way in general to stop after a finite time and say for certain that it will never halt.

Undecidability of the Membership Problem

The membership problem is also undecidable. We can show this by *reducing* the halting problem to it. In other words, we show that if there were a way to decide membership in general, we could use this as a subroutine to decide halting in general. But we just showed above that halting is undecidable, so membership must be undecidable too.

Here is how we would use a total TM that decides membership as a subroutine to decide halting. Given a machine M and input x, suppose we wanted to find out whether M halts on x. Build a new machine N that is exactly like M, except that it accepts whenever M would either accept or reject. The machine N can be constructed from M simply by adding a new accept state and making the old accept and reject states transfer to this new accept state. Then for all x, N accepts x iff M halts on x. The membership problem for N and x (asking whether $x \in L(N)$) is therefore the same as the halting problem for M and x (asking whether M halts on x). If the membership problem were decidable, then we could decide whether M halts on x by constructing N and asking whether $x \in L(N)$. But we have shown above that the halting problem is undecidable, therefore the membership problem must also be undecidable.

Lecture 32

Decidable and Undecidable Problems

Here are some examples of decision problems involving Turing machines. Is it decidable whether a given Turing machine

(a) has at least 481 states?

(b) takes more than 481 steps on input ϵ?

(c) takes more than 481 steps on *some* input?

(d) takes more than 481 steps on *all* inputs?

(e) ever moves its head more than 481 tape cells away from the left endmarker on input ϵ?

(f) accepts the null string ϵ?

(g) accepts any string at all?

(h) accepts every string?

(i) accepts a finite set?

(j) accepts a regular set?

(k) accepts a CFL?

(l) accepts a recursive set?

(m) is equivalent to a Turing machine with a shorter description?

Problems (a) through (e) are decidable and problems (f) through (m) are undecidable (proofs below). We will show that problems (f) through (l) are undecidable by showing that a decision procedure for one of these problems could be used to construct a decision procedure for the halting problem, which we know is impossible. Problem (m) is a little more difficult, and we will leave that as an exercise (Miscellaneous Exercise 131). Translated into modern terms, problem (m) is the same as determining whether there exists a shorter PASCAL program equivalent to a given one.

The best way to show that a problem is decidable is to give a total Turing machine that accepts exactly the "yes" instances. Because it must be total, it must also reject the "no" instances; in other words, it must not loop on any input.

Problem (a) is easily decidable, since the number of states of M can be read off from the encoding of M. We can build a Turing machine that, given the encoding of M written on its input tape, counts the number of states of M and accepts or rejects depending on whether the number is at least 481.

Problem (b) is decidable, since we can simulate M on input ϵ with a universal machine for 481 steps (counting up to 481 on a separate track) and accept or reject depending on whether M has halted by that time.

Problem (c) is decidable: we can just simulate M on all inputs of length at most 481 for 481 steps. If M takes more than 481 steps on some input, then it will take more than 481 steps on some input of length at most 481, since in 481 steps it can read at most the first 481 symbols of the input.

The argument for problem (d) is similar. If M takes more than 481 steps on all inputs of length at most 481, then it will take more than 481 steps on all inputs.

For problem (e), if M never moves more than 481 tape cells away from the left endmarker, then it will either halt or loop in such a way that we can detect the looping after a finite time. This is because if M has k states and m tape symbols, and never moves more than 481 tape cells away from the left endmarker, then there are only $482km^{481}$ configurations it could possibly ever be in, one for each choice of head position, state, and tape contents that fit within 481 tape cells. If it runs for any longer than that without moving more than 481 tape cells away from the left endmarker, then it must be in a loop, because it must have repeated a configuration. This can be detected by a machine that simulates M, counting the number of steps M takes on a separate track and declaring M to be in a loop if the bound of $482km^{481}$ steps is ever exceeded.

Problems (f) through (l) are undecidable. To show this, we show that the ability to decide any one of these problems could be used to decide the halting problem. Since we know that the halting problem is undecidable, these problems must be undecidable too. This is called a *reduction*.

Let's consider (f) first (although the same construction will take care of (g) through (i) as well). We will show that it is undecidable whether a given machine accepts ϵ, because the ability to decide this question would give the ability to decide the halting problem, which we know is impossible.

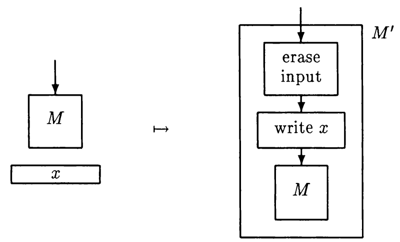
Suppose we could decide whether a given machine accepts ϵ. We could then decide the halting problem as follows. Say we are given a Turing machine M and string x, and we wish to determine whether M halts on x. Construct from M and x a new machine M' that does the following on input y:

(i) erases its input y;

(ii) writes x on its tape (M' has x hard-wired in its finite control);

(iii) runs M on input x (M' also has a description of M hard-wired in its finite control);

(iv) accepts if M halts on x.



Note that M' does the same thing on all inputs y: if M halts on x, then M' accepts its input y; and if M does not halt on x, then M' does not halt on y, therefore does not accept y. Moreover, this is true for every y. Thus

$$L(M') = \begin{cases} \Sigma^* & \text{if } M \text{ halts on } x, \\ \varnothing & \text{if } M \text{ does not halt on } x. \end{cases}$$

Now if we could decide whether a given machine accepts the null string ϵ, we could apply this decision procedure to the M' just constructed, and this would tell whether M halts on x. In other words, we could obtain a decision procedure for halting as follows: given M and x, construct M', then ask whether M' accepts ϵ. The answer to the latter question is "yes" iff M halts on x. Since we know the halting problem is undecidable, it must also be undecidable whether a given machine accepts ϵ.

Similarly, if we could decide whether a given machine accepts any string at all, or whether it accepts every string, or whether the set of strings it accepts is finite, we could apply any of these decision procedures to M' and this would tell whether M halts on x. Since we know that the halting problem is undecidable, all of these problems must be undecidable too.

To show that (j), (k), and (l) are undecidable, pick your favorite r.e. but nonrecursive set A (HP or MP will do) and modify the above construction as follows. Given M and x, build a new machine M'' that does the following on input y:

(i) saves y on a separate track of its tape;

(ii) writes x on a different track (x is hard-wired in the finite control of M'');

(iii) runs M on input x (M is also hard-wired in the finite control of M'');

(iv) if M halts on x, then M'' runs a machine accepting A on its original input y, and accepts if that machine accepts.

Either M does not halt on x, in which case the simulation in step (iii) never halts and M'' never accepts any string; or M does halt on x, in which case M'' accepts its input y iff $y \in A$. Thus

$$L(M'') = \begin{cases} A & \text{if } M \text{ halts on } x, \\ \varnothing & \text{if } M \text{ does not halt on } x. \end{cases}$$

Since A is neither recursive, CFL, nor regular, and $\varnothing$ is all three of these things, if one could decide whether a given TM accepts a recursive, context-free, or regular set, then one could apply this decision procedure to M'' and this would tell whether M halts on x.

Lecture 33

Reduction

There are two main techniques for showing that problems are undecidable: *diagonalization* and *reduction*. We saw examples of diagonalization in Lecture 31 and reduction in Lecture 32.

Once we have established that a problem such as HP is undecidable, we can show that another problem B is undecidable by *reducing* HP to B. Intuitively, this means we can manipulate instances of HP to make them look like instances of the problem B in such a way that "yes" instances of HP become "yes" instances of B and "no" instances of HP become "no" instances of B. Although we cannot tell effectively whether a given instance of HP is a "yes" instance, the manipulation preserves "yes"-ness and "no"-ness. If there existed a decision procedure for B, then we could apply it to the disguised instances of HP to decide membership in HP. In other words, combining a decision procedure for B with the manipulation procedure would give a decision procedure for HP. Since we have already shown that no such decision procedure for HP can exist, we can conclude that no decision procedure for B can exist.

We can give an abstract definition of reduction and prove a general theorem that will save us a lot of work in undecidability proofs from now on.

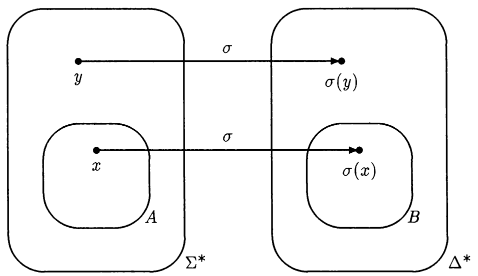
Given sets $A \subseteq \Sigma^*$ and $B \subseteq \Delta^*$, a (many-one) *reduction* of A to B is a computable function

$$\sigma : \Sigma^* \to \Delta^*$$

such that for all $x \in \Sigma^*$,

$$x \in A \Longleftrightarrow \sigma(x) \in B. \tag{33.1}$$

In other words, strings in A must go to strings in B under σ, and strings not in A must go to strings not in B under σ.



The function σ need not be one-to-one or onto. It must, however, be *total* and *effectively computable.* This means σ must be computable by a total Turing machine that on any input x halts with $\sigma(x)$ written on its tape. When such a reduction exists, we say that A is *reducible* to B via the map σ, and we write $A \leq_{\mathrm{m}} B$. The subscript m, which stands for "many-one," is used to distinguish this relation from other types of reducibility relations.

The relation $\leq_{\mathrm{m}}$ of reducibility between languages is transitive: if $A \leq_{\mathrm{m}} B$ and $B \leq_{\mathrm{m}} C$, then $A \leq_{\mathrm{m}} C$. This is because if σ reduces A to B and τ reduces B to C, then $\tau \circ \sigma$, the composition of σ and τ, is computable and reduces A to C.

Although we have not mentioned it explicitly, we have used reductions in the last few lectures to show that various problems are undecidable.

Example 33.1 In showing that it is undecidable whether a given TM accepts the null string, we constructed from a given TM M and string x a TM M' that accepted the null string iff M halts on x. In this example,

$$A = \{M\#x \mid M \text{ halts on } x\} = \mathrm{HP},$$
$$B = \{M \mid \epsilon \in L(M)\},$$

and σ is the computable map $M\#x \mapsto M'$. □

Example 33.2 In showing that it is undecidable whether a given TM accepts a regular set, we constructed from a given TM M and string x a TM M'' such that

$L(M'')$ is a nonregular set if M halts on x and $\varnothing$ otherwise. In this example,

$$A = \{M\#x \mid M \text{ halts on } x\} = \mathrm{HP},$$
$$B = \{M \mid L(M) \text{ is regular}\},$$

and σ is the computable map $M\#x \mapsto M''$. □

Here is a general theorem that will save us some work.

Theorem 33.3 *(i) If $A \leq_{\mathrm{m}} B$ and B is r.e., then so is A. Equivalently, if $A \leq_{\mathrm{m}} B$ and A is not r.e., then neither is B.*

(ii) If $A \leq_{\mathrm{m}} B$ and B is recursive, then so is A. Equivalently, if $A \leq_{\mathrm{m}} B$ and A is not recursive, then neither is B.

Proof. (i) Suppose $A \leq_{\mathrm{m}} B$ via the map σ and B is r.e. Let M be a TM such that $B = L(M)$. Build a machine N for A as follows: on input x, first compute $\sigma(x)$, then run M on input $\sigma(x)$, accepting if M accepts. Then

$$\begin{aligned} N \text{ accepts } x &\Longleftrightarrow M \text{ accepts } \sigma(x) && \text{definition of } N \\ &\Longleftrightarrow \sigma(x) \in B && \text{definition of } M \\ &\Longleftrightarrow x \in A && \text{by (33.1)}. \end{aligned}$$

(ii) Recall from Lecture 29 that a set is recursive iff both it and its complement are r.e. Suppose $A \leq_{\mathrm{m}} B$ via the map σ and B is recursive. Note that $\sim A \leq_{\mathrm{m}} \sim B$ via the same σ (Check the definition!). If B is recursive, then both B and $\sim B$ are r.e. By (i), both A and $\sim A$ are r.e., thus A is recursive. □

We can use Theorem 33.3(i) to show that certain sets are not r.e. and Theorem 33.3(ii) to show that certain sets are not recursive. To show that a set B is not r.e., we need only give a reduction from a set A we already know is not r.e. (such as $\sim$HP) to B. By Theorem 33.3(i), B cannot be r.e.

Example 33.4 Let's illustrate by showing that neither the set

$$\mathrm{FIN} = \{M \mid L(M) \text{ is finite}\}$$

nor its complement is r.e. We show that neither of these sets is r.e. by reducing $\sim$HP to each of them, where

$$\sim\mathrm{HP} = \{M\#x \mid M \text{ does not halt on } x\}:$$

(a) $\sim\mathrm{HP} \leq_{\mathrm{m}} \mathrm{FIN}$,

(b) $\sim\mathrm{HP} \leq_{\mathrm{m}} \sim\mathrm{FIN}$.

Since we already know that $\sim$HP is not r.e., it follows from Theorem 33.3(i) that neither FIN nor $\sim$FIN is r.e.

For (a), we want to give a computable map σ such that

$$M\#x \in {\sim}\mathrm{HP} \iff \sigma(M\#x) \in \mathrm{FIN}.$$

In other words, from $M\#x$ we want to construct a Turing machine $M' = \sigma(M\#x)$ such that

$$M \text{ does not halt on } x \iff L(M') \text{ is finite.} \tag{33.2}$$

Note that the description of M' can depend on M and x. In particular, M' can have a description of M and the string x hard-wired in its finite control if desired.

We have actually already given a construction satisfying (33.2). Given $M\#x$, construct M' such that on all inputs y, M' takes the following actions:

(i) erases its input y;

(ii) writes x on its tape (M' has x hard-wired in its finite control);

(iii) runs M on input x (M' also has a description of M hard-wired in its finite control);

(iv) accepts if M halts on x.

If M does not halt on input x, then the simulation in step (iii) never halts, and M' never reaches step (iv). In this case M' does not accept its input y. This happens the same way for all inputs y, therefore in this case, $L(M) = \varnothing$. On the other hand, if M does halt on x, then the simulation in step (iii) halts, and y is accepted in step (iv). Moreover, this is true for all y. In this case, $L(M) = \Sigma^*$. Thus

$$\begin{array}{rcccl} M \text{ halts on } x & \Rightarrow & L(M') = \Sigma^* & \Rightarrow & L(M') \text{ is infinite,} \\ M \text{ does not halt on } x & \Rightarrow & L(M') = \varnothing & \Rightarrow & L(M') \text{ is finite.} \end{array}$$

Thus (33.2) is satisfied. Note that this is all we have to do to show that FIN is not r.e.: we have given the reduction (a), so by Theorem 33.3(i) we are done.

There is a common pitfall here that we should be careful to avoid. It is important to observe that the computable map σ that produces a description of M' from M and x does not need to execute the program (i) through (iv). It only produces the description of a machine M' that does so. The computation of σ is quite simple—it does not involve the simulation of any other machines or anything complicated at all. It merely takes a description of a Turing machine M and string x and plugs them into a general description of a machine that executes (i) through (iv). This can be done quite easily by a total TM, so σ is total and effectively computable.

Now (b). By definition of reduction, a map reducing ${\sim}$HP to ${\sim}$FIN also reduces HP to FIN, so it suffices to give a computable map τ such that

$$M\#x \in \mathrm{HP} \iff \tau(M\#x) \in \mathrm{FIN}.$$

In other words, from M and x we want to construct a Turing machine $M'' = \tau(M\#x)$ such that

$$M \text{ halts on } x \iff L(M'') \text{ is finite.} \tag{33.3}$$

Given $M\#x$, construct a machine M'' that on input y

(i) saves y on a separate track;

(ii) writes x on the tape;

(iii) simulates M on x for $|y|$ steps (it erases one symbol of y for each step of M on x that it simulates);

(iv) accepts if M has *not* halted within that time, otherwise rejects.

Now if M never halts on x, then M'' halts and accepts y in step (iv) after $|y|$ steps of the simulation, and this is true for all y. In this case $L(M'') = \Sigma^*$. On the other hand, if M does halt on x, then it does so after some finite number of steps, say n. Then M'' accepts y in (iv) if $|y| < n$ (since the simulation in (iii) has not finished by $|y|$ steps) and rejects y in (iv) if $|y| \geq n$ (since the simulation in (iii) does have time to complete). In this case M'' accepts all strings of length less than n and rejects all strings of length n or greater, so $L(M'')$ is a finite set. Thus

$$\begin{aligned} M \text{ halts on } x &\Rightarrow L(M'') = \{y \mid |y| < \text{running time of } M \text{ on } x\} \\ &\Rightarrow L(M'') \text{ is finite,} \end{aligned}$$

$$\begin{aligned} M \text{ does not halt on } x &\Rightarrow L(M'') = \Sigma^* \\ &\Rightarrow L(M'') \text{ is infinite.} \end{aligned}$$

Then (33.3) is satisfied.

It is important that the functions σ and τ in these two reductions can be computed by Turing machines that always halt. □

Historical Notes

The technique of diagonalization was first used by Cantor [16] to show that there were fewer real algebraic numbers than real numbers.

Universal Turing machines and the application of Cantor's diagonalization technique to prove the undecidability of the halting problem appear in Turing's original paper [120].

Reducibility relations are discussed by Post [101]; see [106, 116].

Lecture 34

Rice's Theorem

Rice's theorem says that undecidability is the rule, not the exception. It is a very powerful theorem, subsuming many undecidability results that we have seen as special cases.

Theorem 34.1 **(Rice's theorem)** *Every nontrivial property of the r.e. sets is undecidable.*

Yes, you heard right: that's *every* nontrivial property of the r.e. sets. So as not to misinterpret this, let us clarify a few things.

First, fix a finite alphabet Σ. A *property of the r.e. sets* is a map

$$P : \{\text{r.e. subsets of } \Sigma^*\} \to \{\top, \bot\},$$

where $\top$ and $\bot$ represent truth and falsity, respectively. For example, the property of emptiness is represented by the map

$$P(A) = \begin{cases} \top & \text{if } A = \varnothing, \\ \bot & \text{if } A \neq \varnothing. \end{cases}$$

To ask whether such a property P is decidable, the set has to be presented in a finite form suitable for input to a TM. We assume that r.e. sets are presented by TMs that accept them. But keep in mind that the property is a property of *sets*, not of Turing machines; thus it must be true or false independent of the particular TM chosen to represent the set.

Here are some other examples of properties of r.e. sets: $L(M)$ is finite; $L(M)$ is regular; $L(M)$ is a CFL; M accepts 101001 (i.e., $101001 \in L(M)$); $L(M) = \Sigma^*$. Each of these properties is a property of the set accepted by the Turing machine.

Here are some examples of properties of Turing machines that are *not* properties of r.e. sets: M has at least 481 states; M halts on all inputs; M rejects 101001; there exists a smaller machine equivalent to M. These are not properties of sets, because in each case one can give two TMs that accept the same set, one of which satisfies the property and the other of which doesn't.

For Rice's theorem to apply, the property also has to be *nontrivial*. This just means that the property is neither universally true nor universally false; that is, there must be at least one r.e. set that satisfies the property and at least one that does not. There are only two trivial properties, and they are both trivially decidable.

Proof of Rice's theorem. Let P be a nontrivial property of the r.e. sets. Assume without loss of generality that $P(\varnothing) = \bot$ (the argument is symmetric if $P(\varnothing) = \top$). Since P is nontrivial, there must exist an r.e. set A such that $P(A) = \top$. Let K be a TM accepting A.

We reduce HP to the set $\{M \mid P(L(M)) = \top\}$, thereby showing that the latter is undecidable (Theorem 33.3(ii)). Given $M\#x$, construct a machine $M' = \sigma(M\#x)$ that on input y

(i) saves y on a separate track someplace;

(ii) writes x on its tape (x is hard-wired in the finite control of M');

(iii) runs M on input x (a description of M is also hard-wired in the finite control of M');

(iv) if M halts on x, M' runs K on y and accepts if K accepts.

Now either M halts on x or not. If M does not halt on x, then the simulation in (iii) will never halt, and the input y of M' will not be accepted. This is true for every y, so in this case $L(M') = \varnothing$. On the other hand, if M does halt on x, then M' always reaches step (iv), and the original input y of M' is accepted iff y is accepted by K; that is, if $y \in A$. Thus

$$\begin{array}{rcccl} M \text{ halts on } x & \Rightarrow & L(M') = A & \Rightarrow & P(L(M')) = P(A) = \top, \\ M \text{ does not halt on } x & \Rightarrow & L(M') = \varnothing & \Rightarrow & P(L(M')) = P(\varnothing) = \bot. \end{array}$$

This constitutes a reduction from HP to the set $\{M \mid P(L(M)) = \top\}$. Since HP is not recursive, by Theorem 33.3, neither is the latter set; that is, it is undecidable whether $L(M)$ satisfies P. □

Rice's Theorem, Part II

A property $P : \{\text{r.e. sets}\} \to \{\top, \bot\}$ of the r.e. sets is called *monotone* if for all r.e. sets A and B, if $A \subseteq B$, then $P(A) \le P(B)$. Here $\le$ means less than or equal to in the order $\bot \le \top$. In other words, P is *monotone* if whenever a set has the property, then all supersets of that set have it as well. For example, the properties "$L(M)$ is infinite" and "$L(M) = \Sigma^*$" are monotone but "$L(M)$ is finite" and "$L(M) = \varnothing$" are not.

Theorem 34.2 **(Rice's theorem, part II)** *No nonmonotone property of the r.e. sets is semidecidable. In other words, if P is a nonmonotone property of the r.e. sets, then the set $T_P = \{M \mid P(L(M)) = \top\}$ is not r.e.*

Proof. Since P is nonmonotone, there exist TMs M_0 and M_1 such that $L(M_0) \subseteq L(M_1)$, $P(M_0) = \top$, and $P(M_1) = \bot$.

We want to reduce $\sim$HP to T_P, or equivalently, HP to $\sim T_P = \{M \mid P(L(M)) = \bot\}$. Since $\sim$HP is not r.e., neither will be T_P. Given $M\#x$, we want to show how to construct a machine M' such that $P(M') = \bot$ iff M halts on x. Let M' be a machine that does the following on input y:

(i) writes its input y on the top and middle tracks of its tape;

(ii) writes x on the bottom track (it has x hard-wired in its finite control);

(iii) simulates M_0 on input y on the top track, M_1 on input y on the middle track, and M on input x on the bottom track in a round-robin fashion; that is, it simulates one step of each of the three machines, then another step, and so on (descriptions of M_0, M_1, and M are all hard-wired in the finite control of M');

(iv) accepts its input y if either of the following two events occurs:

(a) M_0 accepts y, or

(b) M_1 accepts y and M halts on x.

Either M halts on x or not, independent of the input y to M'. If M does not halt on x, then event (b) in step (iv) will never occur, so M' will accept y iff event (a) occurs, thus in this case $L(M') = L(M_0)$. On the other hand, if M does halt on x, then y will be accepted iff it is accepted by either M_0 or M_1; that is, if $y \in L(M_0) \cup L(M_1)$. Since $L(M_0) \subseteq L(M_1)$, this is equivalent to saying that $y \in L(M_1)$, thus in this case $L(M') = L(M_1)$. We have shown

$$\begin{aligned} M \text{ halts on } x &\Rightarrow L(M') = L(M_1) \\ &\Rightarrow P(L(M')) = P(L(M_1)) = \bot, \end{aligned}$$

$$\begin{aligned} M \text{ does not halt on } x &\Rightarrow L(M') = L(M_0) \\ &\Rightarrow P(L(M')) = P(L(M_0)) = \top. \end{aligned}$$

The construction of M' from M and x constitutes a reduction from $\sim$HP to the set $T_P = \{M \mid P(L(M)) = \top\}$. By Theorem 33.3(i), the latter set is not r.e. $\square$

Historical Notes

Rice's theorem was proved by H. G. Rice [104, 105].

Lecture 35

Undecidable Problems About CFLs

In this lecture we show that a very simple problem about CFLs is undecidable, namely the problem of deciding whether a given CFG generates all strings.

It is decidable whether a given CFG generates any string at all, since we know by the pumping lemma that a CFG G that generates any string at all must generate a short string; and we can determine for all short strings x whether $x \in L(G)$ by the CKY algorithm.

This decision procedure is rather inefficient. Here is a better one. Let $G = (N, \Sigma, P, S)$ be the given CFG. To decide whether $L(G)$ is nonempty, we will execute an inductive procedure that marks a nonterminal when it is determined that that nonterminal generates some string in Σ^*—any string at all—and when we are done, ask whether the start symbol S is marked.

At stage 0, mark all the symbols of Σ. At each successive stage, mark a nonterminal $A \in N$ if there is a production $A \to \beta \in P$ and all symbols of β are marked. Quit when there are no more changes; that is, when for each production $A \to \beta$, either A is marked or there is an unmarked symbol of β. This must happen after a finite time, since there are only finitely many symbols to mark.

It can be shown that A is marked by this procedure if and only if there is a string $x \in \Sigma^*$ such that $A \xrightarrow[G]{*} x$. This can be proved by induction, the implication $\Rightarrow$ by induction on the stage that A is marked, and the implication $\Leftarrow$ by induction on the length of the derivation $A \xrightarrow[G]{*} x$.

Then $L(G)$ is nonempty iff there exists an $x \in \Sigma^*$ such that $S \xrightarrow[G]{*} x$ iff S is marked.

Believe it or not, this procedure can be implemented in linear time, so it is in fact quite easy to decide whether $L(G) = \varnothing$. See also Miscellaneous Exercise 134 for another approach.

The finiteness problem for CFLs is also decidable (Miscellaneous Exercise 135).

Valid Computation Histories

In contrast to the efficient algorithm just given, it is impossible to decide in general for a given CFG G whether $L(G) = \Sigma^*$. We will show this by a reduction from the halting problem.

The reduction will involve the set VALCOMPS(M, x) of *valid computation histories* of a Turing machine M on input x, defined below. This set is also useful in showing the undecidability of other problems involving CFLs, such as whether the intersection of two given CFLs is nonempty or whether the complement of a given CFL is a CFL.

Recall that a *configuration* α of a Turing machine M is a triple (q, y, n) where q is a state, y is a semi-infinite string describing the contents of the tape, and n is a nonnegative integer describing the head position.

We can encode configurations as finite strings over the alphabet

$$\Gamma \times (Q \cup \{-\}),$$

where Q is the set of states of M, Γ is the tape alphabet of M, and $-$ is a new symbol. A pair in $\Gamma \times (Q \cup \{-\})$ is written vertically with the element of Γ on top. A typical configuration (q, y, k) might be encoded as the string

$$\begin{array}{ccccccccc} \vdash & b_1 & b_2 & b_3 & \cdots & b_k & \cdots & b_m \\ - & - & - & - & \cdots & q & \cdots & - \end{array}$$

which shows the nonblank symbols of y on the top and indicates that the machine is in state q scanning the kth tape cell. Recall that the *start configuration* of M on input x is

$$\begin{array}{ccccc} \vdash & a_1 & a_2 & \cdots & a_n \\ s & - & - & \cdots & - \end{array}$$

where s is the start state of M and $x = a_1 a_2 \cdots a_n$.

A *valid computation history* of M on x is a list of such encodings of configurations of M separated by a special marker $\# \notin \Gamma \times (Q \cup \{-\})$; that is, a string

$$\#\alpha_0\#\alpha_1\#\alpha_2\#\cdots\#\alpha_N\#$$

such that

- α_0 is the start configuration of M on x;
- α_N is a halting configuration; that is, the state appearing in α_N is either the accept state t or the reject state r; and
- α_{i+1} follows in one step from α_i according to the transition function δ of M, for $0 \le i \le N-1$; that is,

  $$\alpha_i \xrightarrow[M]{1} \alpha_{i+1}, \quad 0 \le i \le N-1,$$

  where $\xrightarrow[M]{1}$ is the next configuration relation of M.

In other words, the valid computation history describes a halting computation of the machine M on input x, if M does indeed halt. If M does not halt on x, then no such valid computation history exists.

Let $\Delta = \{\#\} \cup (\Gamma \times (Q \cup \{-\}))$. Then a valid computation history of M on x, if it exists, is a string in Δ^*. Define

$$\text{VALCOMPS}(M, x) \stackrel{\text{def}}{=} \{\text{valid computation histories of } M \text{ on } x\}.$$

Then VALCOMPS$(M, x) \subseteq \Delta^*$, and

$$\text{VALCOMPS}(M, x) = \varnothing \iff M \text{ does not halt on } x. \tag{35.1}$$

Thus the complement of VALCOMPS(M, x), namely

$$\sim\text{VALCOMPS}(M, x) = \Delta^* - \text{VALCOMPS}(M, x),$$

is equal to Δ^* iff M does not halt on x.

The key claim now is that $\sim$VALCOMPS(M, x) is a CFL. Moreover, without knowing whether or not M halts on x, we can construct a CFG G for $\sim$VALCOMPS(M, x) from a description of M and x. By (35.1), we will have

$$L(G) = \Delta^* \iff M \text{ does not halt on } x.$$

Since we can construct G effectively from M and x, this will constitute a reduction

$$\sim\text{HP} \le_m \{G \mid G \text{ is a CFG and } L(G) = \Delta^*\}.$$

By Theorem 33.3(i), the latter set is not r.e., which is what we want to show.

To show that $\sim$VALCOMPS(M,x) is a CFL, let us carefully write down all the conditions for a string $z \in \Delta^*$ to be a valid computation history of M on x:

(1) z must begin and end with a $\#$; that is, it must be of the form

$$\#\alpha_0\#\alpha_1\#\cdots\#\alpha_N\#,$$

where each α_i is in $(\Delta - \#)^*$;

(2) each α_i is a string of symbols of the form

$$\begin{array}{c} a \\ - \end{array} \quad \text{or} \quad \begin{array}{c} a \\ q \end{array}$$

where exactly one symbol of α_i has an element of Q on the bottom and the others have $-$, and only the leftmost has a $\vdash$ on top;

(3) α_0 represents the start configuration of M on x;

(4) a halt state, either t or r, appears somewhere in z (by our convention that Turing machines always remain in a halt state once they enter it, this is equivalent to saying that α_N is a halt configuration); and

(5) $\alpha_i \xrightarrow[M]{1} \alpha_{i+1}$ for $0 \le i \le N-1$.

Let

$$A_i = \{x \in \Delta^* \mid x \text{ satisfies condition } (i)\}, \quad 1 \le i \le 5.$$

A string in Δ^* is in VALCOMPS(M,x) iff it satisfies all five conditions listed above; that is,

$$\text{VALCOMPS}(M,x) = \bigcap_{1 \le i \le 5} A_i.$$

A string is in $\sim$VALCOMPS(M,x) iff it fails to satisfy at least one of conditions (1) through (5); that is, if it is in at least one of the $\sim A_i$, $1 \le i \le 5$. We show that each of the sets $\sim A_i$ is a CFL and show how to obtain a CFG G_i for it. Then $\sim$VALCOMPS(M,x) is the union of the $\sim A_i$, and we know how to construct a grammar G for this union from the G_i.

The sets A_1, A_2, A_3, and A_4 are all regular sets, and we can easily construct right-linear CFGs for their complements from finite automata or regular expressions. The only difficult case will be A_5.

The set A_1 is the set of strings beginning and ending with a $\#$. This is the regular set

$$\#\Delta^*\#.$$

To check that a string is in A_2, we need only check that between every two $\#$'s there is exactly one symbol with a state q on the bottom, and $\vdash$ occurs on the top immediately after each $\#$ (except the last) and nowhere else. This can easily be checked with a finite automaton.

The set A_3 is the regular set

$$\begin{array}{cccccccc} \# & \vdash & a_1 & a_2 & \cdots & a_n & \# & \Delta^* \\ & s & - & - & \cdots & - & & \end{array}$$

To check that a string is in A_4, we need only check that t or r appears someplace in the string. Again, this is easily checked by a finite automaton.

Finally, we are left with the task of showing that $\sim A_5$ is a CFL. Consider a substring $\cdots\#\alpha\#\beta\#\cdots$ of a string in Δ^* satisfying conditions (1) through (4). Note that if $\alpha \xrightarrow[M]{1} \beta$, then the two configurations must agree in most symbols except for a few near the position of the head; and the differences that can occur near the position of the head must be consistent with the action of δ. For example, the substring might look like

$$\begin{array}{ccccccccccccccccccccccc} \cdots & \# & \vdash & a & b & a & a & b & a & b & b & \# & \vdash & a & b & a & b & b & a & b & b & \# & \cdots \\ & & - & - & - & - & q & - & - & - & - & & - & - & - & p & - & - & - & - & - & & \end{array}$$

This would occur if $\delta(q,a) = (p,b,L)$. We can check that $\alpha \xrightarrow[M]{1} \beta$ by checking for all three-element substrings u of α that the corresponding three-element substring v of β differs from u in a way that is consistent with the operation of δ. *Corresponding* means occurring at the same distance from the closest $\#$ to its left. For example, the pair

$$\begin{array}{ccc} a & a & b \\ - & q & - \end{array} \qquad \begin{array}{ccc} a & b & b \\ p & - & - \end{array}$$

occurring at a distance 4 from the closest $\#$ to their left in α and β, respectively, are consistent with δ, since $\delta(q,a) = (p,b,L)$. The pair

$$\begin{array}{ccc} a & b & b \\ - & - & - \end{array} \qquad \begin{array}{ccc} a & b & b \\ - & - & - \end{array}$$

occurring at distance 7 are consistent (any two identical length-three substrings are consistent, since this would occur if the tape head were far away). The pair

$$\begin{array}{ccc} a & b & a \\ - & - & - \end{array} \qquad \begin{array}{ccc} a & b & a \\ - & - & p \end{array}$$

occurring at distance 2 are consistent, because there exists a transition moving left and entering state p.

We can write down all consistent pairs of strings of length three over Δ. For any configurations α and β, if $\alpha \xrightarrow[M]{1} \beta$, then all corresponding substrings of length three of α and β are consistent. Conversely, if all corresponding substrings of length three of α and β are consistent, then $\alpha \xrightarrow[M]{1} \beta$. Thus, to check that $\alpha \xrightarrow[M]{1} \beta$ does *not* hold, we need only check that there exists a substring of α of length three such that the corresponding substring of β of length three is not consistent with the action of δ.

We now describe a nondeterministic PDA that accepts $\sim A_5$. We need to check that there exists i such that α_{i+1} does *not* follow from α_i according to δ. The PDA will scan across z and guess α_i nondeterministically. It then checks that α_{i+1} does not follow from α_i by guessing some length-three substring u of α_i, remembering it in its finite control, and checking that the corresponding length-three substring v of α_{i+1} is not consistent with u under the action of δ. It uses its stack to check that the distance of u from the last # is the same as the distance of v from the last #. It does this by pushing the prefix of α_i in front of u onto the stack and then popping as it scans the prefix of α_{i+1} in front of v, checking that these two prefixes are the same length.

For example, suppose $\delta(q, a) = (p, b, R)$ and z contains the following substring:

$$\begin{array}{cccccccccccccccccccccccc} \cdots & \# & \vdash & a & b & a & a & b & a & b & b & \# & \vdash & a & b & a & a & b & b & b & b & \# & \cdots \\ & & - & - & - & - & - & - & q & - & - & & - & - & - & - & - & p & - & - & - & & \end{array}$$

Then z does not satisfy condition (5), because δ said to go right but z went left. We can check with a PDA that this condition is violated by guessing where the error is and checking that the corresponding length-three subsequences are not consistent with the action of δ. Scan right, pushing symbols from the # up to the substring

$$\begin{array}{ccc} b & a & b \\ - & q & - \end{array}$$

(we nondeterministically guess where this is). Scan these three symbols, remembering them in the finite control. Scan to the next # without altering the stack, then scan and pop the stack. When the stack is empty, we are about to scan the symbols

$$\begin{array}{ccc} b & b & b \\ p & - & - \end{array}$$

We scan these and compare them to the symbols from the first configuration we remembered in the finite control, and then we discover the error.

We have given a nondeterministic PDA accepting $\sim A_5$. From this and the finite automata for $\sim A_i$, $1 \le i \le 4$, we can construct a CFG G for their union $\sim$VALCOMPS(M, x), and

$$L(G) = \Delta^* \iff M \text{ does not halt on } x.$$

If we could decide whether G generates all strings over its terminal alphabet, it would answer the question of whether M halts on x. We have thus reduced the halting problem to the question of whether a given grammar generates all strings. Since the halting problem is undecidable, we have shown:

Theorem 35.1 *It is undecidable for a given CFG G whether or not $L(G) = \Sigma^*$.*

Many other simple problems involving CFLs are undecidable: whether a given CFL is a DCFL, whether the intersection of two given CFLs is a CFL, whether the complement of a given CFL is a CFL, and so on. These problems can all be shown to be undecidable using valid computation histories. We leave these as exercises (Miscellaneous Exercise 121).

Historical Notes

Undecidable properties of context-free languages were established by Bar-Hillel et al. [8], Ginsburg and Rose [47], and Hartmanis and Hopcroft [56]. The idea of valid computation histories is essentially from Kleene [67, 68], where it is called the *T-predicate*.