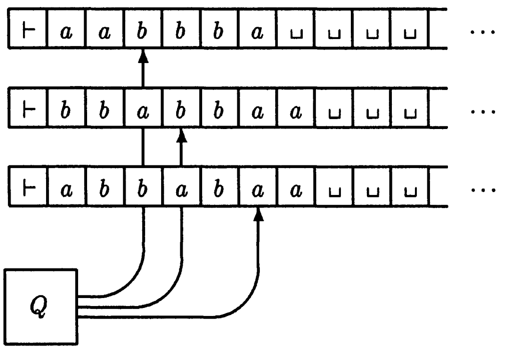
Lecture 30

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

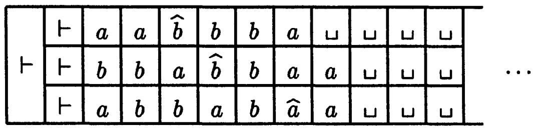
First, we show how to simulate multitape Turing machines with single-tape Turing machines. Thus extra tapes don't add any power. A three-tape machine is similar to a one-tape TM except that it has three semi-infinite tapes and three independent read/write heads. Initially, the input occupies the first 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.


⊢ a a b b b a ⊔ ⊔ ⊔ ⊔ ···
⊢ b b a b b a a ⊔ ⊔ ⊔ ···
⊢ a b b a b a a ⊔ ⊔ ⊔ ···
Q

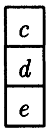
Its transition function is of type

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

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.


⊢
⊢ a a $\hat{b}$ b b a ⊔ ⊔ ⊔ ⊔
⊢ b b a $\hat{b}$ b a a ⊔ ⊔ ⊔ ···
⊢ a b b a b $\hat{a}$ a ⊔ ⊔ ⊔

A tape symbol of N is either ⊢, an element of Σ, or a triple



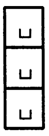
where c, d, e are tape symbols of M, each either marked or unmarked. Formally, we might take the tape alphabet of N to be

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

where

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

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

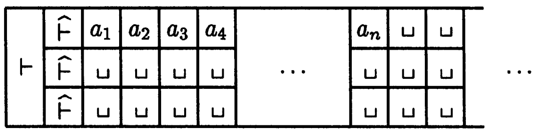


is the blank symbol of N.

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 $\hat{}$ to indicate the position of the heads in the starting configuration of M.

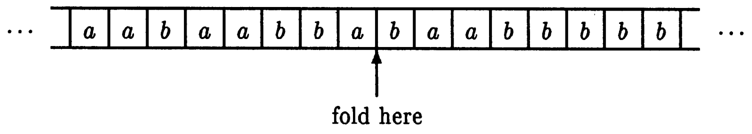


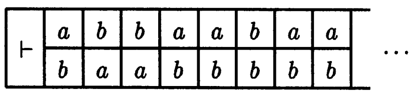
$\vdash$
$\hat{\vdash}$ a_1 a_2 a_3 a_4 $\cdots$ a_n $\sqcup$ $\sqcup$
$\hat{\vdash}$ $\sqcup$ $\sqcup$ $\sqcup$ $\sqcup$ $\cdots$ $\sqcup$ $\sqcup$ $\sqcup$ $\cdots$
$\hat{\vdash}$ $\sqcup$ $\sqcup$ $\sqcup$ $\sqcup$ $\sqcup$ $\sqcup$ $\sqcup$

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:

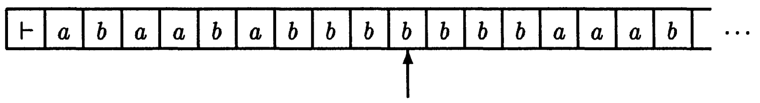




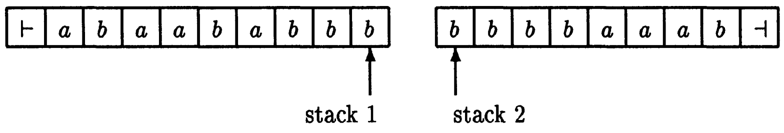
The bottom track is used to simulate the original machine when its head is to the right of the fold, and the top track is used to simulate the machine when its head is to the left of the fold, moving in the opposite direction.

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. □

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].