

# Lecture 7: Turing Machines

David Dill

Department of Computer Science

# Outline

---

- (finish up) Decision problems on regular languages.
- Turing machines and their languages.
- Programming tricks.

## Some decision problems

**Decision problem:** A question with a yes/no answer.

Is string  $x \in L(N)$ ? (The membership problem.)

Solution: Compute  $\hat{\delta}(q_0, x)$ , check for final states.

Given a finite automaton,  $N$ , is  $L(N) = \emptyset$ ? (The emptiness problem.)

Given a FA, how do you check for emptiness reasonably quickly?

*Answer:* Do depth-first or breadth-first search from  $q_0$  for a reachable final state.

Is  $L(N) = \Sigma^*$ ? (The universality problem.)

*Answer:* Check  $\overline{L(N)} = \emptyset$ . This is easy if  $N$  is a DFA. Otherwise, your best bet is to convert it to a DFA and complement.

Is  $L(N) \subseteq L(M)$ ? (The subset problem)

*Answer:* Check for  $L(N) \cap \overline{L(M)} = \emptyset$

Equivalence?

*Answer:* Check for subset in both directions, OR minimize both DFAs and check if they are the same.

# Turing Machines

Models of computation so far have been highly restricted.

Next model is really the most general useful of computation.

This material is full of surprises. The proof that some problems are undecidable is one of the greatest intellectual accomplishments of humanity.

First surprise: There IS a most general model of computation.

It seems that all general models of computation have the same power. This is called the “Church-Turing thesis.” It’s not really provable, but it seems to be true.

# Turing Machines

Finite control with a *tape* which is used for input and for unbounded storage.

There is a “head” that can read/write the tape.

⟨⟨ TM picture ⟩⟩

- The tape is infinite in both directions.
- There is a special “blank” symbol (B).
- All but a finite number of positions are blank at any given time.

**Def** A *move* of a TM is based on the current state and the symbol currently being scanned, and it changes state, writes a new symbol, and moves left or right.

## Def. of Turing Machine

All sets are finite:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

$Q$  – states

$\Sigma$  – input symbols

$\Gamma$  – tape symbols.  $\Sigma \subseteq \Gamma$

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

$q_0$  – start symbol

$B$  – “blank”  $B \in \Gamma - \Sigma$ .

$F$  – Final states.

## TM Instantaneous Descriptions

An ID for a TM is a string  $X_1X_2 \dots X_{i-1}qX_i \dots X_n$  where

- $q$  is the current state.
- The tape head is at position  $i$ .
- $X_1X_2 \dots X_n$  are the other symbols. The infinite string of blanks to the left and right of the nonblanks and state are suppressed.

## Moves of a TM

This definition is messy, but the main idea is simple enough.

Suppose  $\delta(q, X_i) = (p, Y, L)$ :

$$\begin{array}{l} X_1 X_2 \dots X_{i-2} X_{i-1} q X_i X_{i+1} \dots X_n \vdash \\ X_1 X_2 \dots X_{i-2} p X_{i-1} Y X_{i+1} \dots X_n \end{array}$$

If  $\delta(q, X_i) = (p, Y, R)$ :

$$X_1 X_2 \dots X_{i-1} q X_i X_{i+1} \dots X_n \vdash X_1 X_2 \dots X_{i-1} Y p X_{i+1} \dots X_n$$

The messy bits have to do with adding and removing the invisible blanks at each end of the tape when the head moves:

If  $\delta(q, X_i) = (p, Y, L)$  and  $q$  is already at the left end:

$$q X_1 X_2 \dots X_i \dots X_n \vdash p B Y X_2 \dots X_i \dots X_n$$

Or, if  $q$  is at the right end and  $Y = B$ :

$$X_1 X_2 \dots X_{n-1} q X_n \vdash X_1 X_2 \dots p X_{n-1}$$

etc. See the text.

## Language of a Turing Machine

$\alpha \vdash^* \beta$  means we can get from ID  $\alpha$  to ID  $\beta$  in 0 or more  $\vdash$  steps (i.e.,  $\vdash^*$  is the reflexive transitive closure of  $\vdash$ ).

A string  $w \in \Sigma^*$  is accepted by a TM if  $q_0 w \vdash^* \alpha p \beta$  and  $p \in F$ , and  $\alpha \in \Gamma^*$  and  $\beta \in \Gamma^*$ .

The languages accepted by Turing machines are called *Recursively Enumerable (RE)* (for historical reasons).

The recursively enumerable languages are those that can be recognized by any reasonable computer if resources are unbounded.

## Acceptance by Halting

A TM *halts* if the current state is  $q$ , current tape symbol is  $X$ , and  $\delta(q, X)$  is undefined.

We can make sure a TM always halts when it enters a final state by removing all exit transitions (obviously, this does not change the language).

**From now on, we assume that this has been done. Turing machines will halt when they accept.**

**Very important note:** Rejecting  $w$  means *not accepting*  $w$ . A Turing machine can reject by running forever without entering a final state.

This creates an interesting and important problem: You can't tell whether the TM has rejected the input, or just hasn't accepted it yet!

If we require the TM to halt always, it greatly restricts the class of languages that can be accepted.

The class of languages accepted by TMs that always halt is a proper subset of the RE languages. These are the *recursive* languages.

Alternative definition of language of a TM: The TM accepts iff it halts. It is easy to modify a TM with one type of acceptance into an equivalent machine with the other.

# Turing Machine Programming Tricks

These make it easier to describe TM computations.

*Storing finite data in the states.*

⟨⟨ Picture of control with state and other stuff. ⟩⟩

Suppose you want to store the values 1..10 and a Boolean variable (like local variables in a program, or a register in assembly language). Make the states more complicated. Instead of  $q$ , the state becomes  $[q, 3, T]$ .

⟨⟨ Picture of tape with multiple tracks ⟩⟩

*Multiple tracks.* This allows you to split the tape into multiple tracks. You can think of this as storing records in the tape cells. Easy: Just expand the tape alphabet. If you have three tracks, the tape symbols look like  $[A, B, C]$ .

*Subroutines.* To perform a task, like copying a section of the tape, create new states for that task with a special entry and exit state.

To “call” the subroutine, enter the entry state, and make the exit state go to the correct “return” state when it is done.

This simple concept does not have a stack of return addresses, but it is sufficient for what we want to do.

## Extensions to Turing Machines

Since Turing machines are already as powerful as any real model of computation can be, adding stuff to them doesn't increase their power.

It rarely even speeds them up all that much.

### *Multi-tape Turing Machine*

It has  $k$  tapes, each with a separate head.

The input is on the first tape with the head on the first non-blank. All other tapes are blank, and the heads are in arbitrary positions (doesn't matter).

The transition function depends on the state and currently scanned symbol on each tape, and updates the state and writes a new symbol on each tape and moves each head.

Each head can also remain stationary instead of moving left or right.

Everything else is the same.

## Equivalence of Single and Multi-Tape Turing Machines

(I'll present the basic idea. Details are in the book.)

Obviously, an MTTM can do anything a single-tape TM can do.

Construction in the other direction: Create a TM  $N$  that simulates a MTTM  $M$  as follows:

$N$  is a multi-track (but single tape) machine.

Use 2 tracks for each simulated tape. One track has the tape contents, the other has a marker for the head position.

Simulating a move of  $M$  on  $N$  requires numerous moves to find the individual heads, update the tapes, and move the heads.

To simulate a move of  $M$ ,  $N$  must find all of the markers and collect the tape symbols in its finite control.

## Running time

*Running time* of  $M$  on  $w$  is the number of steps  $M$  takes before halting. It is infinite if  $M$  fails to halt on  $w$ .

The *time complexity* of  $M$  is  $T(n)$ , the maximum running time of  $M$  over all inputs of length  $n$ .

$T(n)$  is finite iff  $M$  halts on all inputs of length  $n$ .

Abstractly, we consider  $M$  to be “fast enough” if  $T(n)$  is polynomial. (This is a pretty crude approximation of reality, but no worse than the idea of a “language.”)

“Polynomial” is a robust concept. (Abbreviation: PTIME = Polynomial Time)

## Slowdown of Multi-tape Simulation

Let  $T_M(n)$  be the running time on a multi-tape TM,  $T_S(n)$  be the running time of a single-tape TM simulating the MTTM on an input of length  $w$ .

While the MTTM construction may seem grossly inefficient, the running time  $T_S(n)$  is  $O(T_M(n)^2)$ . So, if there is a  $T_M(n)$  algorithm to solve a particular problem on an MTTM, there is an  $O(T_M(n)^2)$  algorithm on a STTM.

Consequence: If you can work out a PTIME algorithm on an MTTM, there is a PTIME algorithm on an STTM.

MTTMs are easier to program, so you might as well use them.

## Running time of ordinary computer vs. Turing machine

An “ordinary computer” is really a finite-state automaton.

However, it is usually more useful to think of them as having unbounded memory (but only a finite amount gets used in a finite computation).

Under reasonable assumptions, an MTTM can simulate an ordinary computer in  $O(n^3)$  time, so an STTM can simulate one in  $O(n^6)$  time.

That’s “fast enough,” because we’re only going to be concerned with polynomial vs. worse-than-polynomial complexity here.

# Non-deterministic Turing Machines

As usual,  $\delta$  gives a *set* of possible next moves.

As usual, if any computation enters a final state, the input is accepted.

## Equivalence of NTMs and DTMs

**Thm** Every language accepted by a DTM is also accepted by an NTM.

**Sketch** Obviously, NTMs are no less powerful than DTMs.

We can simulate an NTM on a multi-tape DTM using *breadth-first search* on the tree of IDs generated by the possibilities of the NTM. If the NTM accepts an input (either by final state or by halting), there is at least one finite path in the tree.

Why not *depth-first search*? Because the machine can go down an infinite possible computation forever, and never get around to doing one of the finite (halting) ones.

The DTM accepts iff the simulated NTM enters a final state.

(Rejection? No path of the NTM has an accepting state. Paths may be finite or infinite.)

**Simulation overhead:** Suppose the maximum number of alternative moves in the NTM is  $m$ , and that the NTM takes  $n$  steps to accept its input. Then the number of nodes in the tree that must be explored is  $nm^n$ .

The cost is “single exponential”  $O(2^{p(n)})$ , where  $p(n)$  is a polynomial function.

# $P = NP$

*Can we simulate an NTM in polynomial time on a DTM?*

This is the famous problem of whether  $P = NP$ .

$P$  – problems that can be solved in polynomial time on a DTM.

$NP$  – problems that can be solved in polynomial time on an NTM.

At this point, no one knows, but “no” might be a good bet.