Turing Machines

A Turing Machine is a very simple model of a computing device:

- It has a *control unit*, which at any given time is in one of a finite set of *states*.
- It has a one-dimensional *tape* with an unbounded number of cells into which a single character from a finite *tape alphabet* can be stored.
- The tape is accessed via a *read/write head* that can be moved left or right one step at a time to *scan* any individual cell.

- It obeys a finite set of *instructions*, each of which is of the following form:
  - "If the control is in state q with the head scanning tape symbol a, then replace a by b, move the head one step either left or right, and change the control state to r."

Mathematical Formalization

As we intend to construct mathematical proofs about Turing Machines, we need a fully precise and unambiguous definition.

Definition 3.3: A *Turing Machine* is a 7-tuple,

$$(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$$

where Q, Σ, Γ are all finite sets and

1. Q is the set of *states*.
2. Σ is the *input alphabet*, not containing the *blank symbol* $\sqcup$.
3. Γ is the *tape alphabet*, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$.
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathsf{L}, \mathsf{R}\}$ is the *transition function*.
5. $q_0 \in Q$ is the *start state*.
6. $q_{\text{accept}} \in Q$ is the *accept state*.
7. $q_{\text{reject}} \in Q$ is the *reject state*, where $q_{\text{reject}} \neq q_{\text{accept}}$.

Configurations

The state of a Turing Machine at any given instant is represented by a *configuration* (sometimes called an *instantaneous description*).

Definition: A *configuration* is a triple (u, q, v), where:

1. $q \in Q$ is the current state.
2. $u \in \Gamma^*$ is the contents of the tape to the left of the head.
3. $v \in \Gamma^+$ is the contents of the tape to the right of the head, where the first symbol of v is the *scanned symbol*.

Note: We regard configurations $(u, q, v\sqcup^i)$ and $(u, q, v\sqcup^j)$ as equivalent for all $i, j \geq 0$. That is, we assume that the portion of the tape to the right of v is blank.

The "Yields" Relation

Configuration C_1 *yields* configuration C_2 if the machine can go from C_1 to C_2 in a single step:

1.(a) $q_i\, bv$ *yields* $q_j\, cv$, if $\delta(q_i, b) = (q_j, c, L)$

(b) $ua\, q_i\, bv$ *yields* $u\, q_j\, acv$, if $\delta(q_i, b) = (q_j, c, L)$

2.(a) $ua\, q_i\, b$ *yields* $uac\, q_j\, \sqcup$, if $\delta(q_i, b) = (q_j, c, R)$

(b) $ua\, q_i\, bv$ *yields* $uac\, q_j\, v$, if $\delta(q_i, b) = (q_j, c, R)$ and v is not empty

Note: Cases (1a) and (2a) are special cases for when the head is at the left- or right-hand end of the configuration. In case (1a), the head does not move. In case (2a), a blank is added.

Variations

Different authors use slightly different formalizations of the Turing Machine model.

- Sometimes the existence of a separate "reject state" is not assumed.
- Sometimes it is assumed that there is a special "left end marker".
- Sometimes it is assumed that the machine "hangs" if it attempts to move left off the end of the tape.

Sipser's definition is chosen so that every configuration (u, q, v) with $q \notin \{q_{\text{accept}}, q_{\text{reject}}\}$ yields a unique successor configuration.

These are a lot like differences in machine languages between real computers: they affect the programming details, but in the end they don't affect what you can do with the computer.

Of course, we have to be convinced of this . . .

Special Configurations

- The *start configuration* of M on input $w \in \Sigma^*$ is the configuration $q_0\, w$.
- An *accepting configuration* has the form $u\, q_{\text{accept}}\, v$.
- A *rejecting configuration* has the form $u\, q_{\text{reject}}\, v$.

Accepting an Input String

A Turing Machine M *accepts* string $w \in \Sigma^*$ if there exists a sequence of configurations $C_1, C_2, \ldots C_k$, where:

- C_1 is the start configuration $q_0\, w$.
- C_i is neither an accepting nor a rejecting configuration, for $1 \leq i < k$.
- Each C_i yields C_{i+1}.
- C_k is an accepting configuration.

The set $L(M) \subseteq \Sigma^*$ of all strings that M accepts is called the *language recognized by* M.

Sipser's treatment of accepting and rejecting is confusing/sloppy.

Exercises (Sipser Ex. 3.5)

Examine the formal definition of a Turing machine to answer the following questions:

1. Can a Turing machine ever write the blank symbol $\sqcup$ on its tape?
2. Can the tape alphabet Γ be the same as the input alphabet Σ?
3. Can a Turing machine's head *ever* be in the same location in two successive steps?
4. Can a Turing machine contain just a single state?

Deciders

A Turing Machine M is called a *decider* if it is not possible for it to run forever from a start configuration without eventually reaching either an accepting or rejecting configuration.

More formally:

- For all $w \in \Sigma^*$ and all sequences $C_1, C_2, \ldots$, where C_1 is the start configuration $q_0\ w$ and each C_i yields C_{i+1}, there exists $k \geq 1$ such that C_k is either an accepting or rejecting configuration.

Note: Not every Turing Machine is a decider.

Turing-Recognizable and Turing-Decidable Languages

- A language L is *Turing-recognizable* (also called *semi-decidable*) if $L = L(M)$ for some Turing Machine M.
- A language L is *Turing-decidable* (also called *decidable*) if $L = L(M)$ for some Turing Machine M that is a decider.

Note: Every Turing-decidable language is clearly Turing-recognizable, but it turns out the the converse is not true.

Regular Languages are Turing-Decidable

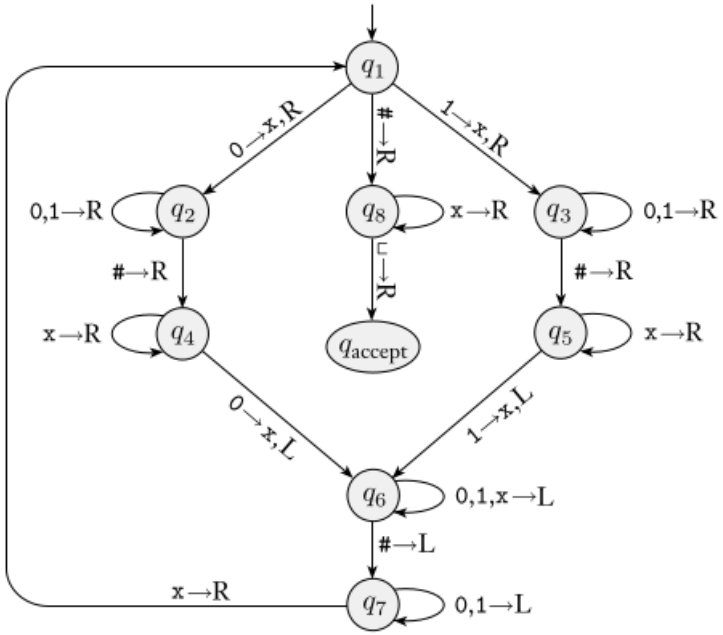
Claim: Every regular language is Turing-decidable.

Proof: Suppose language L is regular. Then $L = L(A)$, where A is a DFA. It is very easy to transform A into a TM M that simulates it: from a start configuration $q_0\, w$ it scans from left to right, changing state as A would. When the right-hand end of w is reached, M enters q_{accept} or q_{reject} depending on whether or not it is in the accept state of A.

Issue: How to identify the right-hand end of w?

Example (Sipser 3.9)

The following is the state diagram for a Turing Machine that decides the language $B = \{w\#w \mid w \in \{0,1\}^*\}$.



Exercises (Sipser Ex. 3.8)

Give implementation-level descriptions of Turing machines that decide the following languages over the alphabet $\{0, 1\}$:

1. $\{w \mid w$ contains an equal number of 0's and 1's$\}$.
2. $\{w \mid w$ contains twice as many 0's as 1's$\}$.
3. $\{w \mid w$ does not contain twice as many 0's as 1's$\}$.

Turing Machine Hacking

Like any programming language, it takes some experience to learn tricks and idioms for constructing Turing Machines. Here are some examples:

- *Identifying the right-hand end of the input:* Look for the first blank (note: $\sqcup$ is not in the input alphabet).
- *Returning to the left-hand end of the tape:* On the first step, remember the scanned symbol in the finite control, then replace that symbol by a special marker. Later, we can go left until we find that marker.

- *Marking tape squares:* Use a tape alphabet that contains "marked" and "unmarked" variants of each symbol. To mark a cell, replace its contents by the marked version.
- *Setting up a "scratch area":* Initially, scan right to end of input. Write a special marker where the first blank is found. Later, return to that area by scanning right, looking for the marker.
- *Subroutines:* A scratch area can be used to a "call stack" of control states. A subroutine call can be performed by pushing the desired return state onto the call stack and jumping to the entry state for the subroutine.
- *"Registers":* Markers can be used to set aside space in the scratch area for any desired purpose. Additional space

can be inserted anywhere by shifting tape contents to the right.

- *Counting:* It is straightforward to implement increment, decrement, and test-for-zero operations on numbers (represented in unary or binary) stored in registers.
- *"Memory":* A "memory unit" can be implemented using an address register, a data register, and read/write subroutines. Counting can be used to locate the register with a given address.

etc.

Turing Machine Variants

Various extensions and generalizations can be made, without affecting the classes of recognizable or decidable languages:

- "Multi-track" tape (single head)
- Two-way infinite tape
- Multiple tapes with independent heads
- Nondeterminism

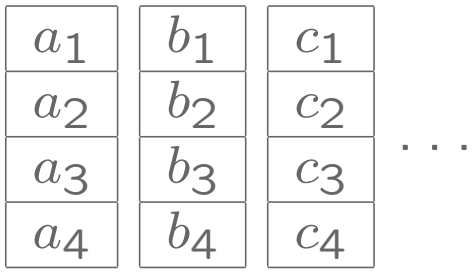
"Multi-track" Tape

We can allow the Turing machine to have a "multi-track" tape (with a single head).

Claim: Language L is recognized (resp. decided) by a TM with a two-way infinite tape iff it is recognized (resp. decided) by a standard TM.

Proof: Suppose L is recognized by a standard TM M. A standard TM is a multi-track TM with just one track.

Conversely, suppose L is recognized by a multi-track TM. We can construct an ordinary TM to recognize M if we use an expanded tape alphabet that incorporates "multi-track" symbols, each of which gives the contents of all tracks at a single head position.



Technical point: Input initially present on the TM tape has to be "converted to multitrack form", before multitrack simulation can begin.

Two-way Infinite Tape

Suppose we allow the Turing machine tape to be unbounded to the left as well as to the right. Assume the tape head is at the left end of the input in a start configuration.

Claim: Language L is recognized (resp. decided) by a TM with a two-way infinite tape iff it is recognized (resp. decided) by a standard TM.

Proof: Suppose L recognized by a standard TM M. Then we can easily construct a two-way infinite TM M' that simulates M (involves minor adjustments) to handle situations in which M might try to move left at the left-hand end of its tape.

Conversely, suppose L recognized by a TM M' with a two-way infinite tape. Construct standard TM M with a "two-track" tape alphabet: one track representing the data to the left of the start position and the other representing the data to the right.

- Use a special marker to identify the "origin" position.
- Use the finite control to keep track of which track we are currently scanning.

Initially, input presented to M in the standard way has to be converted to "two-track form" before beginning the simulation of M'.

Multiple Tapes with Independent Heads

Claim: Language L is recognized (resp. decided) by a multi-tape TM (with independent heads) iff it is recognized (resp. decided) by a standard TM.

Proof: Suppose L recognized by a standard TM M. Then L is recognized by a multitape TM M' that simulates M (using only one of its tracks).

Conversely, suppose L recognized by a k-tape TM M'. Construct a standard TM M using an expanded tape alphabet that allows each cell of M's tape to store k symbols of M', plus k markers that indicate the current positions of M''s heads.

Initially, convert input to "multi-track form" before beginning the simulation of M'.

Nondeterminism

Sipser defines a nondeterministic TM to have a transition function that gives a set of possible alternative moves, rather than a single one.

Alternatively, we can define a nondeterministic TM to be a TM with *two* standard transition functions: δ_0 and δ_1. The "yields" relation between configurations now allows a choice of using δ_0 or δ_1.

Definition: A nondeterministic TM is a *decider* if *every* computation sequence from a start configuration eventually reaches either an accepting or a rejecting configuration.

Claim: Language L is *recognized* (resp. *decided*) by a nondeterministic TM iff it is recognized (resp. decided) by a standard TM.

Proof: A standard TM M amounts to a nondeterministic TM M' with $\delta_0 = \delta_1$ (so the choice does not matter).

Conversely, given a nondeterministic TM M', we can construct a 3-tape deterministic TM that simulates all possible computations of M.

- Tape 1: read-only input
- Tape 2: work tape
- Tape 3: "oracle" tape

TM M uses the oracle tape to "count" through all possible sequences of 0's and 1's (shorter sequences first).

For each sequence, the input is copied to the work tape and a simulation of M' is begun, using the bits on the oracle tape to determine whether to use δ_0 or δ_1 at each step.

- If an accepting configuration is reached, M accepts.
- If a rejecting configuration is reached, or if M runs out of bits on the oracle tape, then abort the current simulation, increment the oracle tape, and restart.

If M' has an accepting computation, then M will also accept, once the correct oracle value is reached.

If M' has no accepting computation, then M will never accept, but it might not reject either (it might run simulations forever). However, if M' is a decider, then every computation path eventually accepts or rejects.

It follows from *König's Lemma* that the computation tree for M' is finite, so M will exhaust all these computations in finite time (how can M tell when this has occurred?).

Prop. *(König's Lemma):* If a finitely branching tree is infinite, then it has an infinite path.

TMs as Language Enumerators

Another way to use a TM to describe a language is as an *enumerator*. This can be done using two tapes:

- Use one tape as a "write-once" output tape, whose head can only move to the right; the other as a work tape.
- The TM starts with both tapes blank.
- Each time it prints (special symbol #) on the output tape, we regard it as having enumerated an additional string.

The (possibly infinite) set of all strings printed on the output tape is the language enumerated by the TM.

Definition: A language is *Turing-enumerable* if it is the language enumerated by some Turing Machine.

Theorem 3.21: A language is Turing-recognizable iff it is Turing-enumerable.

Proof: Suppose L is recognized by TM M. We can construct an enumerator E for L that works as follows:

- For $i = 0, 1, 2, 3, \ldots$,
  For each string w with $|w| \leq i$:
  Simulate M for i steps on input w. If M accepts, print w.

Then E will eventually print w iff M accepts w.

Conversely, suppose L is enumerated by E. Construct a recognizer M for L that works as follows:

- On input w, simulate E. Every time E prints a string, compare it with w. If equal, accept, otherwise continue the simulation.

Then M accepts a string iff it is enumerated by E.

Turing-Decidable *vs.* Turing-Enumerable Languages

- Every Turing-decidable language is Turing-recognizable, hence also Turing-enumerable.
- There are Turing-enumerable languages that are *not* Turing-decidable (we will show this later).

Using TMs to Compute Functions

TMs can also be used to compute functions. However, since TM computation need not terminate, the function computed by a TM may be *partial*.

Definition: A *partial function* $f : \Sigma^* \to \Sigma^*$ is a subset of $\Sigma^* \times \Sigma^*$ such that if (w, x) and (w, y) are two pairs in f, then $x = y$. A partial function is *total* if for every $w \in \Sigma^*$ there exists an x such that (w, x) is in f.

Definition: The *function computed by a TM* M is the following partial function:

- $f(M)$ is the set of all pairs $(w, x) \in \Sigma^* \times \Sigma^*$ such that M accepts w and x is the contents of the tape in the accepting configuration.

(If M loops on w, or the tape contents are not in Σ^*, then $f(M)$ contains no pair (w, x).)

Definition: A partial function $f : \Sigma^* \to \Sigma^*$ is a *computable partial function* (or *partial computable function*) if $f = f(M)$ for some TM M. If f happens to be total, then it is a *total computable function* (or just a *computable function*).

The Church-Turing Thesis

Claim: Any "reasonable" formalization of the notions:

- Language L being decided by some algorithm
- Language L being recognized by some algorithm
- Partial function f being computed by some algorithm
- Total function f being computed by some algorithm

are equivalent to the corresponding Turing-machine-based versions.

"Reasonable" means, roughly:

- The computation can be carried out by a mechanical computing device equipped with an unbounded amount of storage.
- The algorithm has a finite description in terms of a finite set of basic operations.
- Each basic operation can be carried out in finite time, using a finite amount of storage.
- Each basic operation has a predictable effect.

Note: The C-T thesis is not a mathematical statement that can be proved.

It is empirically justified by the fact that many different "reasonable" formalizations of "algorithm" have been devised, and they have all been shown equivalent.

Some important computing models that yield equivalent notions of "algorithm": *Turing machines*, *recursive functions*, *λ-calculus*.