Notes on Turing Machines

Foundations of Computer Science

February 18, 2016

A Turing machine is essentially a (deterministic) finite automaton equipped with an infinite tape that can be written and read by moving the tape head to any cell on the tape. Moreover, instead of an input string that gets scanned from left to right automatically as the machine progresses, the input string for a Turing machine is initially provided on the tape, and the machine must read it explicitly.

Definition. Formally, a *(deterministic) Turing machine* is a tuple

$$M = (Q, \Sigma, \Gamma, \vdash, \llcorner\!\lrcorner, \delta, s, acc, rej)$$

where

- Q is a finite set of *states*
- Σ is a finite *input alphabet*
- Γ is a finite *tape alphabet* ($\Sigma \subseteq \Gamma$)
- $\vdash$ is the *leftmost marker symbol* ($\vdash \in \Gamma - \Sigma$)
- $\llcorner\!\lrcorner$ is the *blank symbol* indicating an empty tape cell ($\llcorner\!\lrcorner \in \Gamma - \Sigma$)
- δ is the *transition function*, $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$
- s is the *start state*
- acc is the *accept state*
- rej is the *reject state*

The transition function takes a state in Q and a symbol in tape alphabet Γ (intuitively, the symbol written in the tape cell where the tape head is), and tells you the new state to which the machine transitions, the symbol to write in the cell where the tape head is, and then whether to move the tape head to the left (L) or to the right (R).

The tape is infinite to the right. In its first (leftmost) cell, we assume that the leftmost marker $\vdash$ is written. Empty cells are assumed to contain the symbol $\llcorner\!\lrcorner$. We assume that the machine can never move the tape head to the left of the leftmost marker. (This is enforced by having the transition function always make the tape head move right upon reading the leftmost marker symbol.)

Computation by Turing machine. Here is how a Turing machine M computes when trying to accept string $w = a_1 \dots a_k \in \Sigma^*$:

1. Put $\vdash \; a_1 \; \dots \; a_k$ on the leftmost cells of the tape.
2. The tape head is initially on the leftmost cell containing $\vdash$.
3. The state is initially s, the start state.
4. If the machine is in state p and the symbol in the cell where the tape head is is a, and $\delta(p, a) = (q, b, d)$, then the machine writes b in the cell where the tape head is, moves the tape head in direction d, and moves to state q.
5. Repeat step (4) until either the state is *acc* (and the machine accepts) or the state is *rej* (and the machine rejects).

Acceptance. We say machine M *accepts* w if M reaches the accept state starting with $\vdash w$ on its input tape.

The *language accepted by a Turing machine* M is defined by

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

Enumerability. We say a language A is *Turing-enumerable*[1] if there exists a Turing machine M such that $L(M) = A$.

Examples. Figure 1 gives a simple Turing machine accepting the language described by regular expression $\mathtt{a}^*\mathtt{b}^*$, which is of course a regular language. As usual, we simply draw the diagram of state transitions, where a transition of the form $\delta(p, a) = (q, b, d)$ is drawn as an arrow between states p and q labelled by $a/b, d$. The tape alphabet is given when unclear.

Figure 2 gives an already more complex Turing machine accepting $\{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$, which is not regular. The machine works in two phases. First, it scans the input string from left to right to ensure it is of the form $\mathtt{a}^m\mathtt{b}^n$ for some m, n. If not, then it goes to a sink state. If it is of that form, then the tape head is rewound to the left, and finds first an $\mathtt{a}$ and then a $\mathtt{b}$, replacing both by a new tape symbol X. The tape head is rewound to the left again, and another $\mathtt{a}$ and

[1] also called *Turing-recognizable*, or *semi-decidable*, or *recursively enumerable*

another $\mathtt{b}$ are crossed out, skipping over previous crossed out symbols. If a $\mathtt{b}$ is found when an $\mathtt{a}$ was looked for, or no $\mathtt{b}$ found after an $\mathtt{a}$ is found, then the machine transitions to a sink state, otherwise, it accepts. The tape alphabet is $\{\vdash, \llcorner\!\lrcorner, \mathtt{a}, \mathtt{b}, \mathtt{X}\}$.

Two things to note about these Turing machines. First off, since the transition function is a function, it needs to be defined for every combination of state and tape symbol. This means, in particular, that the accept state needs to have transitions out of it for every tape symbol. The convention is that those transitions just loop back to the accept state itself, making the accept state a sink state. (The same thing holds true for the reject state as well.)

Second off, I did not give a reject state for those machines. The description of a Turing machine says that there has to a reject state. When one is not explicitly given, say, when describing a Turing machine via a diagram, I will just assume that there is a reject state whose transitions simply loop back to itself, and that any transition that is not explicitly given in the diagram just takes the machine to the reject state. This will keep our diagram neat and tidy.[2]

It is an easy exercise to modify the Turing machine in Figure 2 to accept the language $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$. The resulting Turing machine is in Figure 3.

Rejection. We say machine M *rejects* w if M reaches the reject state starting with $\vdash w$ on its input tape.

Machine M *halts on input* w if M either accepts w or rejects w.

We say machine M is *total* if it halts on every input string in Σ^*.

Decidability. We say a language A is *Turing-decidable* (or simply *decidable*) if there exists a *total* Turing machine M such that $L(M) = A$.

Clearly, every decidable language is also enumerable, since total Turing machines are just a special class of Turing machines.

Recasting the definition, a language is decidable if and only if there a Turing machine M that accepts every string $w \in L(M)$ and that rejects every string $w \notin L(M)$.

We say a Turing machine M *decides* language A if M is total and M accepts A.

It is easy to see that every regular language is decidable; it suffices to show that DFAs can be simulated by Turing machines that always halt.

Figure 4 gives a total Turing machine deciding $\{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$. It basically amounts to taking the Turing machine in Figure 2 and replacing the sink states by a single reject state. We can similarly exhibit a total Turing machine deciding

[2]If you pay attention, you see that the Turing machines I gave as example have all of their transitions defined. So there is no transition that actually goes to the reject state. A Turing machine doesn't have to reject to be useful. But see below.

$\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$.

As we shall see, there are languages that are enumerable but not decidable. And similarly, languages that are not even enumerable. Given how general and expressive Turing machines are, this is somewhat surprising.

Pseudocode Descriptions of Turing Machines. Giving complete descriptions of Turing machines becomes painful for anything but the simplest of machines. Therefore, we will generally resort to a pseudocode description of the behavior of Turing machines, focusing on tape head movement, and symbol replacement. For example, here is a reasonable description of a total version of the Turing machine accepting $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$ from Figure 3.

On input w:

1. Scan tape from left to right, checking that `a`s follow `b`s follow `c`s. Reject if not.
2. Move tape head back to leftmost position.
3. Scan from left to right, replacing the first `a` encountered with `X`,
   then the first `b` encountered with `X`, then the first `c` encountered with `X`.
4. If no `a`, `b`, or `c` encountered, accept.
5. If any of `a`, `b`, or `c` is not encountered, reject.
6. Go back to step 2.

It must be the case that every step in the pseudocode description of a Turing machine should be easily translatable into a set of transitions between states.

Configurations. A configuration is a snapshot of the execution of a Turing machine. To describe the Turing machine at any point in its execution, we need to give: the content of the tape, the position of the tape head, and the current state of the machine.

This information can be represented by a triple (u, q, v), where the content of the tape is the string uv,[3] the tape head is currently pointing to the tape cell at position $|u|$, and thus the first symbol of v is currently in the cell where the tape head is, and the current state of the machine is q. We do not allow v to be ϵ; if ever we have a configuration where v is ϵ, we rewrite it as a configuration where $v = \sqcup$. (This captures the fact that there is an infinite supply of $\sqcup$s on the tape.)

Thus, a configuration is a tuple in $\Gamma^* \times Q \times \Gamma^+$.

Fix a Turing machine $M = (Q, \Sigma, \Gamma, \vdash, \sqcup, \delta, s, acc, rej)$.

A *starting configuration* for M is a configuration of the form $(\epsilon, s, \vdash w)$ for some input string w.

An *accepting configuration* for M is a configuration of the form (u, acc, v) for some u, v.

[3]We only need to describe the content of the tape up to the first blank space to the right of which there are only blank spaces, and there can only be finitely many cells that have been rewritten by the Turing machine during any execution, so uv is a finite string.

A *rejecting configuration* for M is a configuration of the form (u, rej, v) for some u, v.

A *halting configuration* is a configuration that is either accepting or rejecting.

We define a step relation between configurations, written $C \xrightarrow[M]{1} C'$, that describes how configuration evolve as the Turing machine computes and transitions between states. The step relation is defined by the following rules:

$$
\begin{aligned}
(u, p, av) &\xrightarrow[M]{1} (ub, q, v) && \text{if } \delta(p, a) = (q, b, R) \\
(uc, p, av) &\xrightarrow[M]{1} (u, q, cbv) && \text{if } \delta(p, a) = (q, b, L)
\end{aligned}
$$

(It seems that the step relation is not defined for a configuration of the form (u, p, v) with u empty or v empty. If v is empty, then we saw above that we replace v by ␣, so we're good to go. If u is empty, then the tape head is at the way beginning of the tape, with $\vdash$ written in the cell pointed to by the tape head, and we saw that we always assumed that the Turing machine always moves right when $\vdash$ is written in the cell pointed to by the tape head.)

We define $C \xrightarrow[M]{*} C'$ if either $C = C'$ or there exists $C_1, \ldots, C_k$ such that $C \xrightarrow[M]{*} C_1 \xrightarrow[M]{*} \cdots \xrightarrow[M]{*} C_k \xrightarrow[M]{*} C'$.

We can now define acceptance formally: M accepts w if $(\epsilon, s, \vdash w) \xrightarrow[M]{*} C_{acc}$ for some accepting configuration C_{acc}.

Similarly, M rejects w if $(\epsilon, s, \vdash w) \xrightarrow[M]{*} C_{rej}$ for some rejecting configuration C_{rej}.

Example. As an example, consider the total Turing machine in Figure 4, which is a total variant of the Turing machine in Figure 2.

Here is the sequence of configurations showing how the machine accepts `aabb`:

$$
\begin{aligned}
(\epsilon, s, \vdash \texttt{aabb}) &\xrightarrow[M]{1} (\vdash, s, \texttt{aabb}) \\
&\xrightarrow[M]{1} (\vdash \texttt{a}, s, \texttt{abb}) \\
&\xrightarrow[M]{1} (\vdash \texttt{aa}, s, \texttt{bb}) \\
&\xrightarrow[M]{1} (\vdash \texttt{aab}, q_1, \texttt{b}) \\
&\xrightarrow[M]{1} (\vdash \texttt{aabb}, q_1, \text{␣}) \\
&\xrightarrow[M]{1} (\vdash \texttt{aabb␣}, q_2, \text{␣}) \\
&\xrightarrow[M]{1} (\vdash \texttt{aabb}, q_2, \text{␣␣})
\end{aligned}
$$

$$
\begin{aligned}
&\xrightarrow[M]{1} && (\vdash \texttt{aab}, q_2, \texttt{b}\text{␣␣}) \\
&\xrightarrow[M]{1} && (\vdash \texttt{aa}, q_2, \texttt{bb}\text{␣␣}) \\
&\xrightarrow[M]{1} && (\vdash \texttt{a}, q_2, \texttt{abb}\text{␣␣}) \\
&\xrightarrow[M]{1} && (\vdash, q_2, \texttt{abb}\text{␣␣}) \\
&\xrightarrow[M]{1} && (\vdash, q_2, \texttt{aabb}\text{␣␣}) \\
&\xrightarrow[M]{1} && (\epsilon, q_2, \vdash \texttt{aabb}\text{␣␣}) \\
&\xrightarrow[M]{1} && (\vdash, q_3, \texttt{aabb}\text{␣␣}) \\
&\xrightarrow[M]{1} && (\vdash \texttt{X}, q_4, \texttt{abb}\text{␣␣}) \\
&\xrightarrow[M]{1} && (\vdash \texttt{Xa}, q_4, \texttt{bb}\text{␣␣}) \\
&\xrightarrow[M]{1} && (\vdash \texttt{XaX}, q_2, \texttt{b}\text{␣␣}) \\
&\xrightarrow[M]{1} && (\vdash \texttt{Xa}, q_2, \texttt{Xb}\text{␣␣}) \\
&\xrightarrow[M]{1} && (\vdash \texttt{X}, q_2, \texttt{aXb}\text{␣␣}) \\
&\xrightarrow[M]{1} && (\vdash, q_2, \texttt{XaXb}\text{␣␣}) \\
&\xrightarrow[M]{1} && (\epsilon, q_2, \vdash \texttt{XaXb}\text{␣␣}) \\
&\xrightarrow[M]{1} && (\vdash, q_3, \texttt{XaXb}\text{␣␣}) \\
&\xrightarrow[M]{1} && (\vdash \texttt{X}, q_3, \texttt{aXb}\text{␣␣}) \\
&\xrightarrow[M]{1} && (\vdash \texttt{XX}, q_4, \texttt{Xb}\text{␣␣}) \\
&\xrightarrow[M]{1} && (\vdash \texttt{XXX}, q_4, \texttt{b}\text{␣␣}) \\
&\xrightarrow[M]{1} && (\vdash \texttt{XXXX}, q_2, \text{␣␣}) \\
&\xrightarrow[M]{1} && (\vdash \texttt{XXX}, q_2, \texttt{X}\text{␣␣}) \\
&\xrightarrow[M]{1} && (\vdash \texttt{XX}, q_2, \texttt{XX}\text{␣␣}) \\
&\xrightarrow[M]{1} && (\vdash \texttt{X}, q_2, \texttt{XXX}\text{␣␣}) \\
&\xrightarrow[M]{1} && (\vdash, q_2, \texttt{XXXX}\text{␣␣}) \\
&\xrightarrow[M]{1} && (\epsilon, q_2, \vdash \texttt{XXXX}\text{␣␣})
\end{aligned}
$$

$$
\begin{aligned}
&\xrightarrow[M]{1} \quad (\vdash, q_3, \mathtt{XXXX}\llcorner\lrcorner\llcorner\lrcorner) \\
&\xrightarrow[M]{1} \quad (\vdash \mathtt{X}, q_3, \mathtt{XXX}\llcorner\lrcorner\llcorner\lrcorner) \\
&\xrightarrow[M]{1} \quad (\vdash \mathtt{XX}, q_3, \mathtt{XX}\llcorner\lrcorner\llcorner\lrcorner) \\
&\xrightarrow[M]{1} \quad (\vdash \mathtt{XXX}, q_3, \mathtt{X}\llcorner\lrcorner\llcorner\lrcorner) \\
&\xrightarrow[M]{1} \quad (\vdash \mathtt{XXXX}, q_3, \llcorner\lrcorner\llcorner\lrcorner) \\
&\xrightarrow[M]{1} \quad (\vdash \mathtt{XXXX}\llcorner\lrcorner, acc, \llcorner\lrcorner)
\end{aligned}
$$

and this last configuration $(\vdash \mathtt{XXXX}\llcorner\lrcorner, acc, \llcorner\lrcorner)$ is an accepting configuration, and thus w is accepted.

Multitape Turing Machines. Turing machines are simple. Despite their simplicity, they are powerful. But while their simplicity is useful for proving things about them, it is painful when the time comes to design Turing machines to recognize or decide specific languages.

Variants of Turing machines exist that are simpler to use, but that still lead to the same class of enumerable (and decidable) languages.

For instance, while a Turing machine has a single tape, we can imagine working with a Turing machine with multiple tapes. Having multiple tapes (with independent tape heads) means that we can use some of those tapes as temporary storage, or scratch pad to perform calculations, and so on. Having more than one tape is handy.

We can define multitape Turing machines easily enough. A multitape Turing machine (with k tapes) is a tuple $M = (Q, \Sigma, \Gamma, \vdash, \llcorner\lrcorner, \delta, s, acc, rej)$ defined exactly as a one-tape Turing machine, except that the transition relation δ has the form

$$\delta : Q \times \Gamma^k \longrightarrow Q \times \Gamma^k \times \{L, R, S\}^k.$$

(We also allow a tape head to remain in place, indicated by a direction S.) Intuitively, $\delta(p, \langle a_1, \dots, a_k\rangle) = (q, \langle b_1, \dots, b_k\rangle, \langle d_1, \dots, d_k\rangle)$ says that when in state p and when a_i is on tape i under tape $i's$ tape head, then M can transition to state q, writing b_i on tape i under its tape head, and moving each tape head in direction d_i. As for Turing machines, we assume that every tape has a leftmost marker in its leftmost cell, and that the machine cannot move the tape head to the left when on the leftmost marker.

This can be formalized using the notion of configuration, as with standard Turing machines. A *k-tape configuration* is a tuple $(u_1, q, v_1, u_2, v_2, u_3, v_3, \dots, u_k, v_k)$ where q is a state to the multitape Turing machine and u_i, v_i are all strings over the tape alphabet of the machine. As usual, when any of the v_i is ϵ, we replace it with a $\llcorner\lrcorner$ to capture the fact that tapes are infinite to the right and filled with blanks. The starting configuration of the machine with input w is simply

$(, s, \vdash\ w, , \vdash, , \vdash, \ldots, , \vdash)$, where s is the start state of the machine; a configuration is accepting (resp., rejecting) if the state is the accept (resp., reject) state of the machine. The step relation $C \xrightarrow[M]{1} C'$ is defined exactly as expected:

$$(u_1, p, a_1v_1, u_2, a_2v_2, \ldots, u_k, a_kv_k) \xrightarrow[M]{1} (u_1', q, v_1', u_2', v_2', \ldots, u_k', v_k')$$

$$\begin{aligned}
&\text{if } \delta(p, \langle a_1, \ldots, a_k\rangle) = (q, \langle b_1, \ldots, b_k\rangle, \langle d_1, \ldots, d_k\rangle)\\
&\text{and for all } i\text{: if } d_i = R \text{ then } u_i' = u_ib_i \text{ and } v_i' = v_i\\
&\qquad\qquad \text{and if } d_i = L \text{ and } u_i = uc\text{, then } u_i' = u \text{ and } v_i' = cb_iv_i\\
&\qquad\qquad \text{and if } d_i = S \text{ then } u_i' = u_i \text{ and } v_i' = b_iv_i
\end{aligned}$$

$C \xrightarrow[M]{*} C'$ is defined as for Turing machines, and M accepting and rejecting a string w is defined as for Turing machines using the $\xrightarrow[M]{*}$ relation. The language of a multitape Turing machine M is just the set of strings accepted by M. A multitape Turing machine is total if it halts on all inputs.

We say a language is *enumerable by a multitape Turing machine* if there is a multitape Turing machine that accepts it. It is *decidable by a multitape Turing machine* if there is a total multitape Turing machine that accepts it.

Multitape Turing machines, while convenient, do not give us more recognizable or decidable languages.

Theorem: A language is enumerable (resp., decidable) by a multitape Turing machine if and only if it is Turing-enumerable (resp., Turing-decidable).

We first prove the reverse direction: if a language is (Turing-)enumerable, it is enumerable by a multitape Turing machine. If a language is enumerable, there is a Turing machine that accepts it. A Turing machine is just a multitape Turing machine with a single tape (you can check the definitions above agree in that case), and so there is a multitape Turing machine (with one tape) that accepts it. Same things if the language is decidable.

The forward direction is more interesting. Suppose a language A is enumerable by a multitape Turing machine, say M_{mt}, with k tapes. To show it is Turing-enumerable, we need to show that there is a Turing machine M_A that accepts A. We build this Turing machine M_A by essentially simulating what M_{mt} is doing, but using a single tape. The idea is to put all the tapes that M_{mt} would use on a single tape, separated by a special tape symbol `#` that marks where each simulated tape ends and a new simulated tape begins. We indicate where each tape head is by using marked tape symbols of the form $\widehat{\mathtt{a}}$, adding those marked symbols to the tape alphabet.

When M_A runs, it simulates every step of Turing machine M_{mt}: whenever M_{mt} would take a transition in state q, M_A takes a sequence of transitions from a state $\overline{q}$ corresponding to q that first determine what transition M_{mt} would make, updates the tapes accordingly, and then transitions to a new state $\overline{q'}$ that corresponds to the state q' that M_{mt} transitions to.

Given M_{mt}, M_A is defined as follows:

On input w:

1. Rewrite $\vdash\ a_1\ \ldots\ a_n$ on the tape into $\vdash\ \widehat{\texttt{\#}}\, a_1\ \ldots\ a_n\, \widehat{\texttt{\#}}\, \llcorner\, \widehat{\texttt{\#}}\, \llcorner\, \widehat{\texttt{\#}}\ \ldots\ \texttt{\#}$ ($k+1$ #s in total)
2. Simulate a move of M_{mt}:
   a. Scan tape from left to right, noting the marked symbols until the $(k+1)^{\text{th}}$ #
   b. scan tape from left to right, replacing marked symbols, and moving the mark left or right according to M_{mt} (treat # as $\vdash$)
   c. if the mark moves right onto a #, shift whole tape right from that position and write $\widehat{\llcorner}$ in the cell
3. If M_{mt} accepts, accept; if M_{mt} rejects, reject
4. Go to step 2

Observe that M_A is total exactly when M_{mt} is total; this means that the construction in fact shows that if A is enumerable (resp., decidable) by a multitape Turing machine, it is Turing-enumerable (resp., Turing-decidable).

Church-Turing Thesis. What the multitape Turing machine example shows (and the nondeterministic Turing machine you will see in Tutorial Section) is that the definition of decidable languages is quite robust: extensions to Turing machines do not add any expressive power to Turing machines. Intuitively, standard Turing machines are powerful enough to be able to simulate any variant of Turing machines.

In fact, Turing machines are powerful enough, as far as we know, to simulate any model of computation, or any programming language. This thesis is known as the *Church-Turing thesis*:

The Church-Turing Thesis: Any feasible model of computation can be simulated by Turing machines.

The notion *feasible* is kept vague, partly because it cannot really be defined. But it captures the idea of a model of computation that is, for instance, physically realizable—it does not depend on the ability to perform an infinite amount of computation in a finite amount of time, or requiring an infinite amount of space.

Because *feasible* cannot be defined, the Church-Turing thesis is not a theorem. Nevertheless, we have a lot of evidence that the thesis is true.

While it may seem surprising that a Turing machine can in fact simulate any model of computation, it should not be surprising to us that have grown up in the computer age that computers can simulate any model of computation. It remains to show that Turing machines can simulate computers. But a computer is really just a CPU with a bunch of attendant hardware to make it interact with the world, and a CPU is really just a sort of Turing machines. Putting it the other way around, it is not difficult to imagine simulating a CPU with a Turing machine: the alphabet of the Turing machine includes `0` and `1`, which are the basic values that a CPU works with; the registers of the CPU can live on the tape; the finite number of gates of the CPU can be modeled by the finite control of the Turing machine; and the memory that the CPU has access to, which is finite, can be against stored on the tape. While a CPU addresses memory by indexing into it, the Turing machine would have to move the tape head to the appropriate cell corresponding to the memory location, but the difference is merely one of efficiency.[4]

The Church-Turing thesis makes the distinctions between decidable languages and languages that are *not* decidable relevant for the working programmer. Recall that languages over an alphabet Σ are just functions from Σ^* to the Booleans. A language A is decidable if it can be accepted by a total Turing machine; in other words, a function A from Σ^* to the Booleans is decidable if you can come up with a total Turing machines that says yes to a string w exactly when the function A maps w to `true`. A language is not decidable (we'll call those undecidable) if there is no total Turing machine that accepts it. Because of the Church-Turing thesis, if a language is undecidable, then not only is there no total Turing machine that accepts it, there is also no program in any of your favorite programming languages that implements the function corresponding to the language: if there was, then you could also write a Turing machine that simulates that program, and this would contradict undecidability. Thus, if a language is undecidable, there can be no program that implements the corresponding function, period. The notion of decidability, even though it seems like a technical notion having to do with Turing machines only, in fact impacts programming in general. Being able to determine what languages are decidable and which aren't determines which problems can be solved using computer programs, and which can't.

[4]The Church-Turing thesis never claimed that Turing machines can simulate any model of computation *efficiently*, just that they can simulate it.

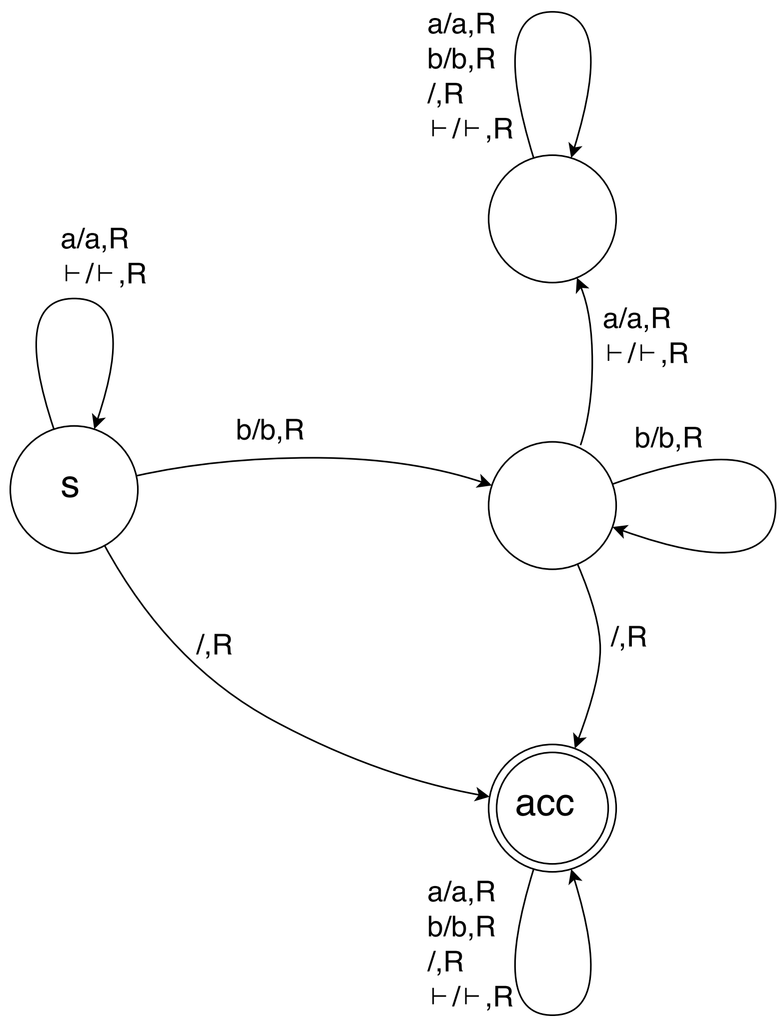

Figure 1: Turing machine accepting $\{\mathtt{a}^m\mathtt{b}^n \mid m, n \geq 0\}$

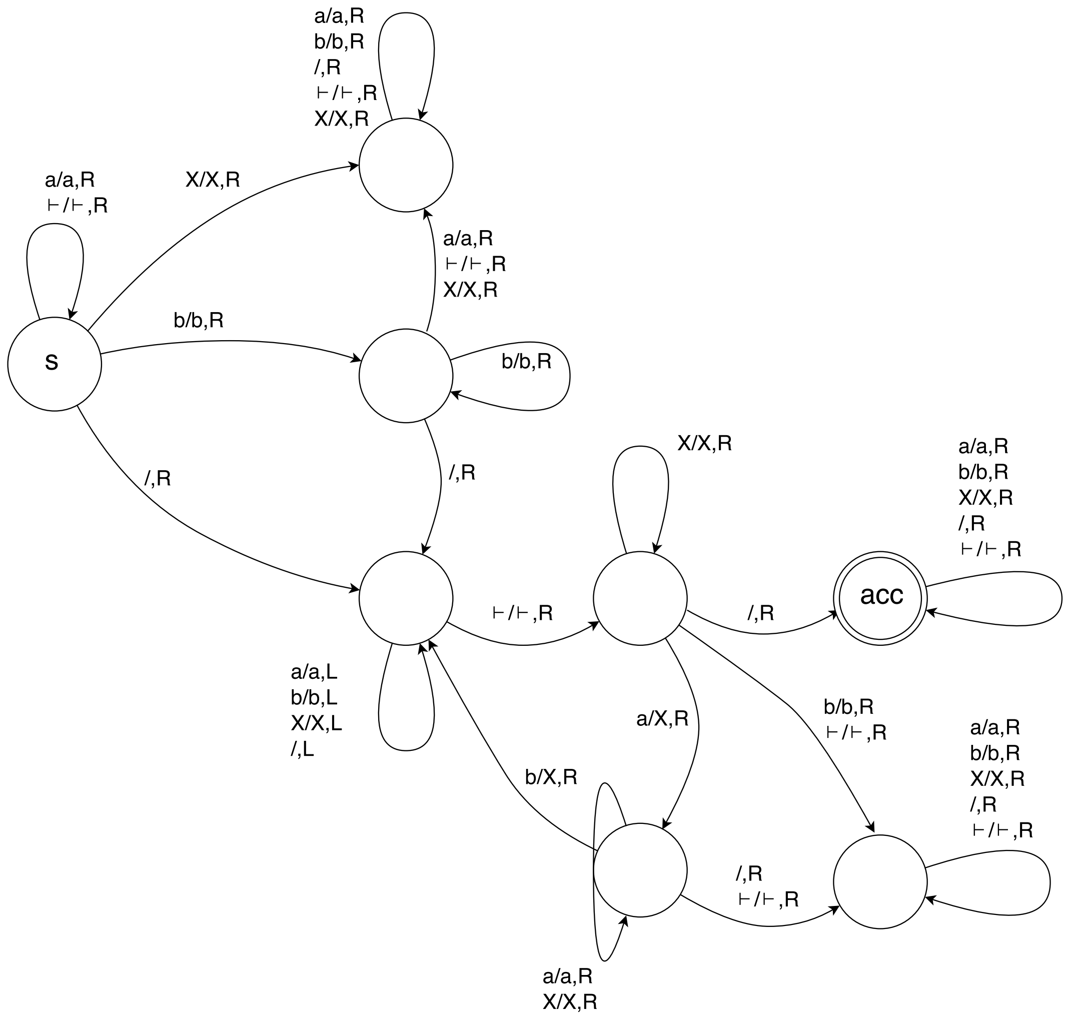

Figure 2: Turing machine accepting $\{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$

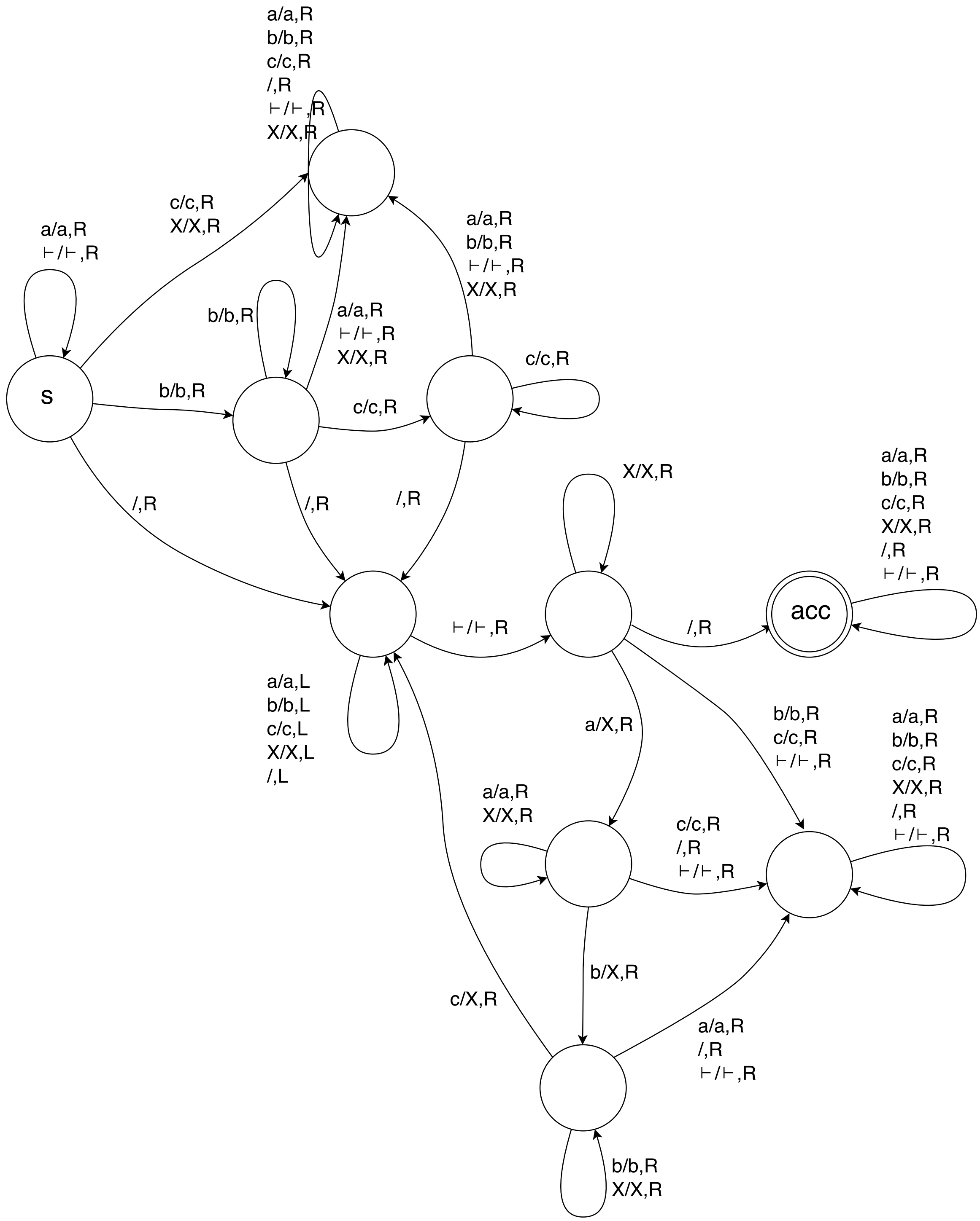

Figure 3: Turing machine accepting $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$

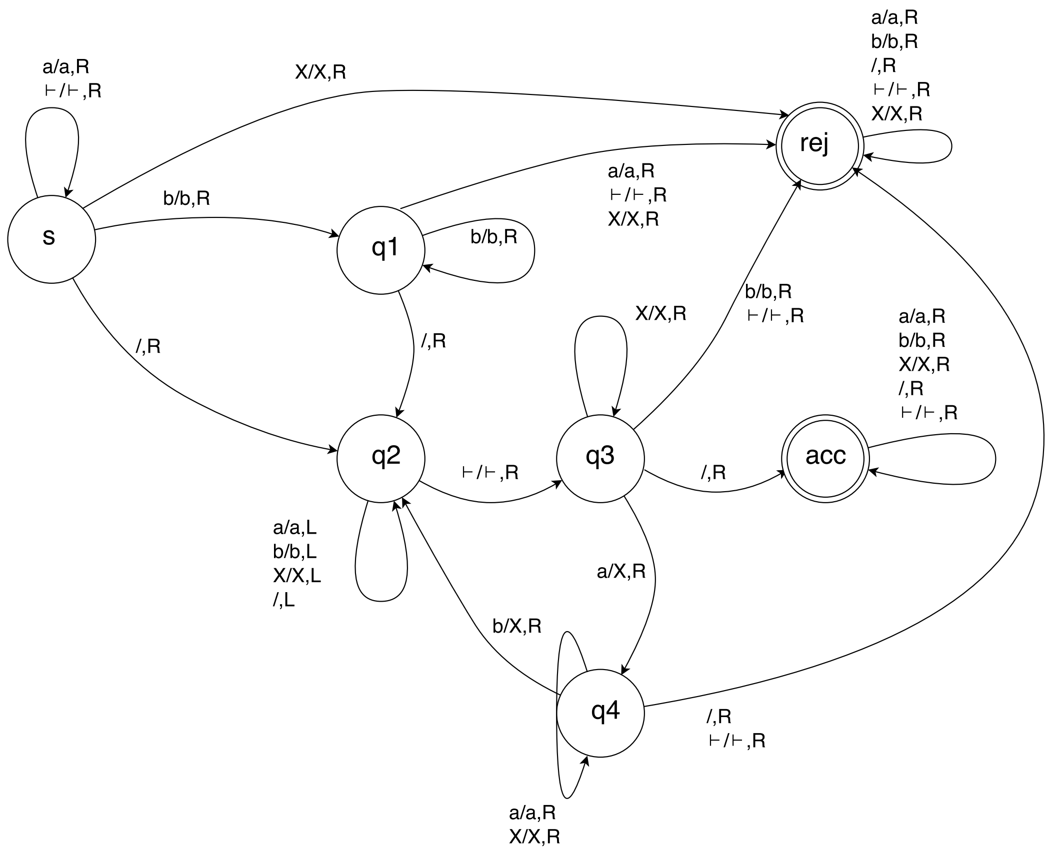

Figure 4: Total Turing machine deciding $\{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$