

CSCI3390-Lecture 1:What is a Computation?

The Turing machine model.

August 30, 2018

1 Summary

- A sort of informal introduction to Turing's solution to the 'What is an algorithm?' problem, before the formal development.
- Two kinds of computational problems: decision problems (yes-no answer), and search or function-evaluation problems (longer answer).
- Inputs to problems are encoded as strings of symbols over a finite alphabet of symbols. (Notation: Σ^* denotes the set of all strings over the alphabet Σ .)
- A decision problem is identified with the set of input strings that give a 'yes' answer (a *language* $L \subseteq \Sigma^*$). A search problem is identified with a function $f : \Sigma^* \rightarrow \Gamma^*$, where Γ is the output alphabet. Solving an instance of a decision problem means determining if $w \in L$ for the given input w . Solving an instance of a search problem means evaluating $f(w)$.
- A general solution is an algorithm for determining $w \in L$, or calculating $f(w)$ for any given input w .
- Turing machine: mathematical model of a machine executing an algorithm. It is claimed that *any* algorithm can be implemented with a Turing machine, and thus this provides a rigorous mathematical definition of what a computation is. This claim is called the *Church-Turing thesis*.

$$\begin{array}{rcccc}
 & 1 & 0 & 1 & 1 \\
 + & 1 & 1 & 0 & \\
 \hline
 1 & 0 & 0 & 0 & 1
 \end{array}$$

Table 1: *Addition in binary. The inputs are written in rows 1 and 2, the outputs in row 3.*

2 What is an algorithm? An informal introduction.

One of the main goals in the first part of this course is the proof that there are certain computational problems that are *undecidable*, in the sense that there is no algorithm for their solution. Now if I tell you that there *is* an algorithm for a problem, it is usually sufficient to describe it and to prove that it does what I claim it does—there is rarely any question of whether the procedure presented really constitutes an algorithm. Here we typically take the attitude ‘I know one when I see one’.

But if I want to prove that there is *no* algorithm to solve a problem, then I need a precise definition that perfectly captures this intuitive concept. This definitional problem was solved in the 1930’s by three different mathematicians—Alonso Church, Kurt Gödel, and Alan Turing—in three very different-looking ways. Remarkably, all three approaches turned out to be equivalent, although this is far from obvious when you read the descriptions. We will begin with Turing’s solution, which is the easiest to grasp and the most philosophically satisfying.

We begin with one of the first algorithms you learned—how to add two multi-digit integers. You learned this using decimal integers, of course, but we will work in binary. There’s no fundamental difference, but the binary version has a shorter description.

Let’s do a little thought experiment: We’ll solve this problem by the conventional method, but imagine that you can only see one digit at a time—maybe the summands are written in the sand on a very long stretch of beach (it’s still summertime here!) You begin on the top row of the rightmost column and head downward, keeping track of a running sum of the bits that you have seen so far. When you arrive at the bottom row, you write the low-order bit of the sum in the blank space. If that sum was greater than 1, then you reset the running sum you’re remembering to 1, otherwise to 0. You now go back up two rows, turn left to the next column, and repeat the procedure with the second column from the right. This continues until you reach a column in which there is no bit in the first

two rows; in this case you either write 1 into the third row of this column (if the running sum is 1), or nothing at all, and then your work is done.

You only have to keep a few things in your head: The running sum, which can be either 0, 1, 2(=10 in binary) or 3(=11); where you are in the procedure (which row, whether you are heading up or down); and to keep track of whether you have seen a blank entry in a column. Significantly, you do not need to keep track of which *column* you are on—the algorithm is the same whether the summands each have only two bits or a million.

The entire procedure is described in the decision diagram shown below. Each node represents what is in your memory at a given step. This is the ‘state’ of the computation. The state, together with the input symbol you are standing on, determine what the next state will be, and what action (move up, down, or left; write 0 or 1) you will take.

Just so you understand how this diagram works: The computation is in the state indicated by the node labeled ‘row 2; down;1’ if you are standing on the second row, heading downward, and the running sum in your head is 1. If the space you are standing on is blank, or contains a 0, then the instruction marked on the edge says to head downward and adjust the state accordingly, so that you are now in row 3, with a running sum that is still 1. An edge marked with a ‘write’ instruction, such as ‘1:write 0,up’, means, ‘if you’re reading a 1, write a 0 in the space you’re currently on, then move up’. This only occurs in row 3. It is a very good idea to walk through a small example with this diagram and carry out the complete computation.

Turing had three important insights about algorithms that can be described in this way.

The first (and this is really the critical one) is that *every algorithm can be described this way!* We will return to this point later, but for the time being, think of any algorithm that you have learned, either in a grade-school or high-school math class, or a computer science class, or in some other context. It is usually first described to you with a little pencil-and-paper, or blackboard example; with a little bit of reflection (and some rather tedious attention to detail) you can present it in the form given above. The claim that any computation can be described in this form is not something that we can prove, because it depends entirely on an intuitive notion of what constitutes an algorithmic procedure.

The second insight is more modest, and is really there only to simplify the theory: It is that the two-dimensional nature of the workspace is not essential. The whole procedure could be carried out by writing the summands on a one-

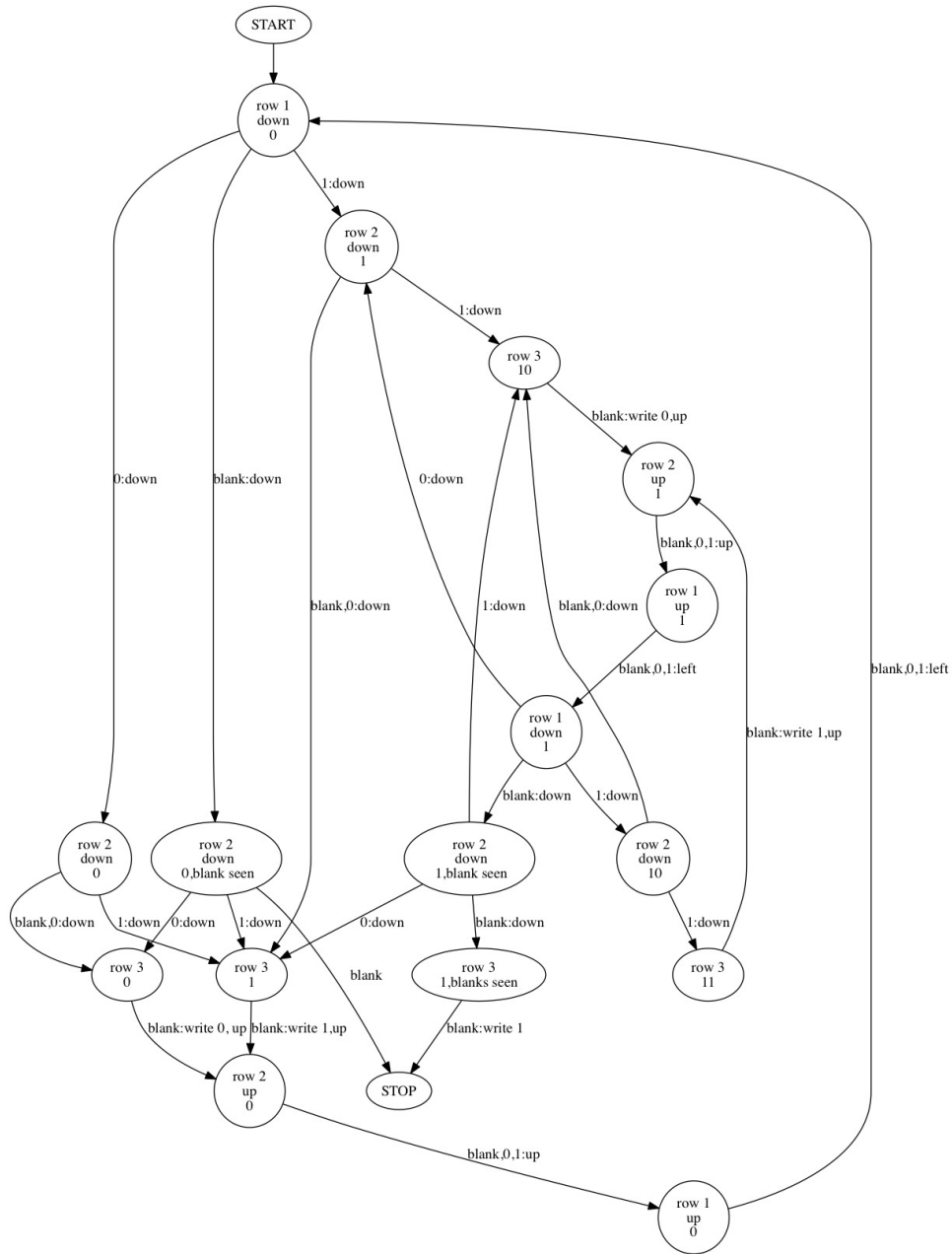


Figure 1: State-transition diagram for the binary addition problem. This is a two-dimensional version of the Turing machine model.

dimensional tape, say as

$$1011 + 110 =$$

There will be a lot of shuttling back and forth, and making little scratches in the sand to mark digits that we have already visited. The whole thing will take longer, but it can still be done.

The third, which we will take up later, is that we don't need a different diagram for each different algorithm—in a sense, *there is a single algorithm that includes them all*.

3 What is a computational problem?

3.1 Some computational problems. A more formal look.

These are like the example problems in Lecture 0.

1. Given two non-negative integers m, n , find the sum $m + n$.

$$77 + 231 \mapsto 308.$$

2. Given an integer $n > 1$, determine if n is prime.

$$91 \mapsto \text{no}$$

$$97 \mapsto \text{yes}$$

3. Given a graph G , determine if there is a path that visits every vertex exactly once. (a *Hamiltonian path*.) The pictured graph has a Hamiltonian path (find it).
4. Given a graph G , find a Hamiltonian path, if one exists.

Problems 2 and 3 are *decision problems*: the answer is 'yes' or 'no'. Problems 1 and 4 are *search problems*, or *function-evaluation problems*.

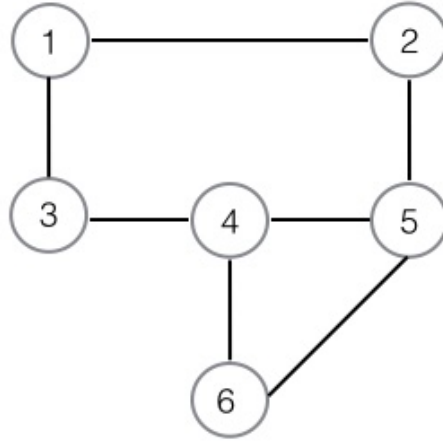


Figure 2: Find a Hamiltonian path—a path that visits every vertex exactly once.

3.2 Encoding problem instances as strings

If we want to *prove* something about computational problems, we will need a formal definition of such problems. Problem *instances* (that is, the input—the pair of summands to add, the graph to test, the polynomial in several variables, *etc.*) are always encoded as *strings*, or *words* over a finite *alphabet* of *letters*, or *symbols*.

If the alphabet is $\Sigma = \{a, b\}$, then $aaba$, $bbaabab$, a are all words. So is ϵ , which we use to denote the *empty* word.

Some notation: We write $|aaba| = 4$, $|\epsilon| = 0$, *etc.* for the length of a word. We write Σ^* to denote the set of all words over Σ . Also we abbreviate using exponential notation, for example:

$$aaba = a^2ba, bbaabab = b^2a(ab)^2.$$

What do the encodings themselves look like? For primality testing, the problem instance is a single integer, which we can encode by its decimal representation, so that ninety-seven is encoded by the string 97. Thus a problem instance is a word over the alphabet $\{0, 1, 2, 3, 4, 5, 7, 8, 9\}$. Of course we have some leeway here—we can encode a problem instance in binary, or some other base, or some different number system.

For the addition problem, a problem instance is a pair of integers, which we might encode as follows:

$$77 + 231,$$

that is, as a word over the alphabet $\{+, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

What about graphs? There are a number of ways we might encode a graph. We could simply list all the edges: Each edge is a pair of vertices, and if we number the vertices in decimal, we get a word over $\{\#, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, where $\#$ is used to separate vertices in our list.

For example, the graph depicted in the illustration in these notes would be encoded by

$$1\#3\#2\#1\#5\#2\#5\#4\#3\#4\#4\#6\#5\#6,$$

representing the sequence

$$(1, 3), (2, 1), (5, 2), (5, 4), (3, 4), (4, 6), (5, 6).$$

There is a subtle drawback in this encoding scheme: It does not correctly encode graphs in which there is a vertex with no neighbors.

If we wanted, we could do everything over an alphabet of *two* letters 0 and 1, and encode every problem instance as a sequence of *bits*. This is exactly how problem instances are represented in conventional computers.

Now that we know how to encode problem instances as strings, how do we define the problem itself? We can think of the decision problem as the set of all problem instances for which the answer is ‘yes’. Thus the problem is a set of strings, or a *language*. For instance, Problem 3 above is treated as identical with the set of all strings that are encodings of graphs with a Hamiltonian path.

We identify a search problem not with a *set* of strings, but with a *function* from the set of strings over the input alphabet to the set of strings over some output alphabet. For example, the addition problem is a function

$$f : \{+, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^* \rightarrow \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^*.$$

If we let $\text{enc}(m)$ denote the decimal encoding of the positive integer m , then the function f is defined by

$$f((\text{enc}(m)) + (\text{enc}(n))) = \text{enc}(m + n).$$

We stress that the argument of f on the left-hand side of the equation is a *string*, and the right-hand side of the equation is a *string*. The ‘+’ on the left-hand side is a letter in a string. However the expressions m , n , $m + n$ represent *integers*.

There is a little problem here. As described, the domain of f should be the set of all strings over the input alphabet, but the formula defining f does not tell us what to do on ‘bad’ inputs like $++23++9+$. There are simple ways to patch this up: We could have the value of f on such inputs be the empty string, or we could add a special error symbol to the output alphabet.

Solving an instance of a decision problem L amounts to determining if a given string w over the input alphabet belongs to L .

Solving an instance of a search problem f amounts to finding the value of $f(w)$ for a given input string w .

A *general solution* of the decision problem is an *algorithm* for determining, given any w , if $w \in L$. A *general solution* of the search problem is an *algorithm* for computing $f(w)$, given any $w \in L$.

So what is an algorithm?

4 Turing Machines

Here we give a formal definition of the Turing machine model.

4.1 High-level description

- The workspace is a tape that extends infinitely in both directions, each cell of which holds one symbol.
- At each step, there is a *current position*. Think of this as a ‘read-write head’ that can both read the symbol at the current position and write a symbol in that square. Figure 3 is an impressionistic picture of a Turing machine in operation.
- At each step, the machine is in one of a finite number of *states*—these correspond to the nodes in the diagram representation of the algorithm that we drew above.
- Tape symbols include both the input symbols and a finite collection of auxillary symbols, including a special *blank* symbol \square . At each step, all but finitely many of the tape cells hold the blank symbol. In our two-dimensional addition example, we only needed the original input symbols and the blank, but if you try to do this problem in a single dimension, you

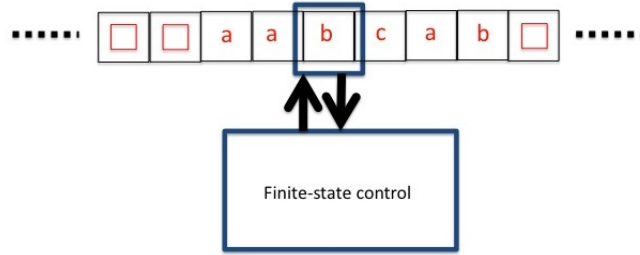


Figure 3: A Turing machine in action

will see that you need to use new symbols to mark when a digit of a summand has already been processed.

- At each step, the machine updates as follows: Depending on the current state and the currently scanned input symbol, the machine erases the symbol and writes a new symbol into that cell. (The new symbol could be the same as the one that was erased.) It moves the current position one cell to the left or right, and changes to a new state.
- There is a special *halt* state. When the machine enters this state, it stops updating.

4.2 Turing machine–formal description-version 1

$$\mathcal{M} = (Q, \Sigma, \Gamma, q_0, \mathbf{h}, \delta).$$

Q is a finite set (set of states).

Σ is a finite alphabet (the input alphabet)

Γ is a finite alphabet, with $\Sigma \subseteq \Gamma$, $\square \in \Gamma - \Sigma$. That is, the tape alphabet contains the input alphabet, and at least one additional symbol, denoting a blank cell.

$q_0 \in Q$ (initial state)

$\mathbf{h} \in Q$ (halt state)

δ is a function

$$\delta : (Q - \{\mathbf{h}\}) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}.$$

Interpretation:

$$\delta(q, \gamma) = (q', \gamma', R)$$

means: if current state is q and currently-scanned symbol is γ , the machine writes γ' in this cell (erasing original γ), changes to state q' , and changes the currently-scanned symbol to the next tape square to the right. We could have $\gamma = \gamma'$, which we typically interpret to mean ‘don’t write anything’, since the symbol in the cell doesn’t change.

A *configuration*, or *instantaneous description* of a TM is given by the tape contents, the current position, and the state. We could denote the instantaneous description by a drawing like the one in the figure, provided we name the state, but we like to have a notation that is easier to write. So if the machine is in the pictured configuration in state \mathbf{q} , we denote the configuration by

$$aa\mathbf{q}bcab.$$

When we start the machine, the tape holds the input string $w \in \Sigma^*$, the current position is the leftmost letter of w , and the machine is in state q_0 . So the configuration is:

$$\mathbf{q}_0aaba.$$

Suppose $\delta(q_0, a) = (q_1, X, R)$, then the next configuration is

$$X\mathbf{q}_1aba.$$

We would then write

$$\mathbf{q}_0aaba \Rightarrow X\mathbf{q}_1aba.$$

We write

$$c \xRightarrow{*} c'$$

if configuration c' results from configuration c after 0 or more steps.

If c is the initial configuration \mathbf{q}_0w , and

$$c \xRightarrow{*} \mathbf{h},$$

the halt state, then we say

$$f_{\mathcal{M}}(w) = w',$$

where $w' \in \Gamma^*$ is contents of the tape when the machine enters state \mathbf{h} . The function $f_{\mathcal{M}}$ is what \mathcal{M} computes. Observe though (and this is important) \mathcal{M} might not halt on every possible input, so $f_{\mathcal{M}}$ is really only a partial function from Σ^* to Γ^* .

4.3 Example.

We will describe a Turing machine \mathcal{M} that reads an input bit string, and halts with the *reverse* of that string on the input tape. That is, for example,

$$f_{\mathcal{M}}(01001) = 10010.$$

The idea is this: the machine moves right until it finds the end of the input, and marks this with the symbol $\#$:

$$01001\#.$$

A. It then moves left to the next 0 or 1 it finds, crosses it out, and ‘remembers’ the crossed-out symbol in its state.

$$0100X\#.$$

B. It moves right to the next blank cell, and writes the remembered symbol, then moves left until it finds the $\#$.

$$0100X\#1.$$

The machine now repeats steps A and B.

$$010XX\#1$$

$$010XX\#10$$

$$01XXX\#10$$

$$01XXX\#100$$

$$0XXXX\#100$$

$$XXXXX\#1001$$

$$XXXXX\#10010$$

If, in step A, a blank cell is found, the machine moves to the right, erasing everything up to and including the $\#$.

$$10010$$

Let’s implement this by figuring out what states we need and what the state-transition function δ should do. In the first phase the machine moves to the right until it finds a blank, upon which it writes the mark $\#$, then moves left for the next phase:

$$\delta(q_0, 0) = (q_0, 0, R), \delta(q_0, 1) = (q_0, 1, R), \delta(q_0, \square) = (q_1, \#, L).$$

In the next phase, it moves left past the crossed-out symbols until it finds 0, 1, or a blank, and gets ready to move right.

$$\delta(q_1, X) = (q_1, X, L), \delta(q_1, 0) = (q_2, X, R), \delta(q_1, 1) = (q_3, 1, R), \delta(q_1, \square) = (q_4, \square, R).$$

If it's in state q_2 or q_3 , the machine moves right past all the other symbols until it finds a blank, and writes 0 or 1, according to the state. So if $\gamma \neq \square$:

$$\delta(q_2, \gamma) = (q_2, \gamma, R),$$

$$\delta(q_3, \gamma) = (q_3, \gamma, R),$$

But if $\gamma = \square$,

$$\delta(q_2, \square) = (q_5, 0, L), \delta(q_3, \square) = (q_5, 1, L).$$

Then it moves back to the $\#$, and repeats the computation beginning in q_1 .

$$\delta(q_5, \gamma) = (q_5, \gamma, L),$$

if $\gamma = 0$ or 1 .

$$\delta(q_5, \#) = (q_1, \#, L).$$

If the machine is in state q_4 , it cleans up, erasing symbols up to and including the $\#$.

A complete run of the machine on the length 3 input 100 takes about 40 steps to reach the halt state. The first few configurations are

$$\mathbf{q_0}001 \Rightarrow 0\mathbf{q_0}01 \Rightarrow 00\mathbf{q_0}1 \Rightarrow 001\mathbf{q_0}\square \Rightarrow 00\mathbf{q_1}1\# \Rightarrow 00X\mathbf{q_3}\# \Rightarrow 00x\#\mathbf{q_3}\square \Rightarrow 00x\mathbf{q_5}\#1$$

Note that δ has not been completely specified: for instance, we have not defined $\delta(q_5, X)$. We can set its value arbitrarily, since we can never be looking at the symbol X in state q_5 . (Better yet, we might have the machine halt if it ever has an unexpected symbol, for example a $\#$ in state 0. This could mean that the input was prepared incorrectly. The resulting tape contents won't make any sense, of course—a case of 'garbage-in, garbage-out'.)

We can depict the behavior of the TM compactly in a *state-transition diagram*, depicted below. If $\delta(q_i, \gamma) = \delta(q_j, \beta, R)$, then we write

$$\gamma \rightarrow \beta, R$$

as the label of the arrow from q_i to q_j . Some conventions: We can leave off the β if $\gamma = \beta$ (in other words if the new symbol on the scanned square is the same as the old one) and list several different γ before the arrow if the behavior of the machine is the same for each of these symbols.

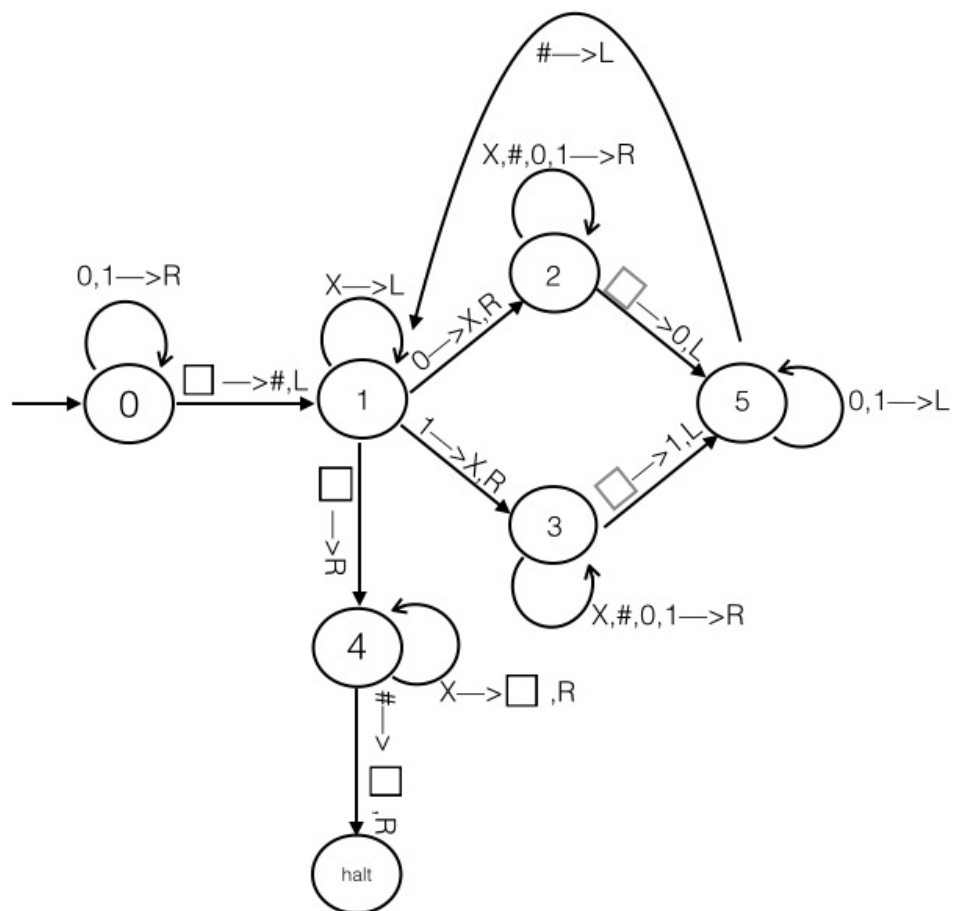


Figure 4: State-transition diagram of the Turing machine example in this section.