# Chapter 10

# Other Models of Turing Machines

Our definition of a standard Turing machine is not the only possible one; there are alternative definitions that could serve equally well. The conclusions we can draw about the power of a Turing machine are largely independent of the specific structure chosen for it. In this chapter we look at several variations, showing that the standard Turing machine is equivalent, in a sense we will define, to other, more complicated models.

If we accept Turing's thesis, we expect that complicating the standard Turing machine by giving it a more complex storage device will not have any effect on the power of the automaton. Any computation that can be performed on such a new arrangement will still fall under the category of a mechanical computation and, therefore, can be done by a standard model. It is nevertheless instructive to study more complex models, if for no other reason than that an explicit demonstration of the expected result will demonstrate the power of the Turing machine and thereby increase our confidence in Turing's thesis. Many variations on the basic model of Definition 9.1 are possible. For example, we can consider Turing machines with more than one tape or with tapes that extend in several dimensions.

We will consider variants that will be useful in subsequent discussions.

We also look at nondeterministic Turing machines and show that they are no more powerful than deterministic ones. This is unexpected, since Turing's thesis covers only mechanical computations and does not address the clever guessing implicit in nondeterminism. Another issue that is not immediately resolved by Turing's thesis is that of one machine executing different programs at different times. This leads to the idea of a "reprogrammable" or "universal" Turing machine.

Finally, in preparation for later chapters, we look at linear bounded automata. These are Turing machines that have an infinite tape, but that can make use of the tape only in a restricted way.

## 10.1 Minor Variations on the Turing Machine Theme

We first consider some relatively minor changes in Definition 9.1 and investigate whether these changes make any difference in the general concept. Whenever we change a definition, we introduce a new type of automata and raise the question whether these new automata are in any real sense different from those we have already encountered. What do we mean by an essential difference between one class of automata and another? Although there may be clear differences in their definitions, these differences may not have any interesting consequences. We have seen an example of this in the case of deterministic and nondeterministic finite automata. These have quite different definitions, but they are equivalent in the sense that they both are identified exactly with the family of regular languages. Extrapolating from this, we can define equivalence or nonequivalence for classes of automata in general.

### Equivalence of Classes of Automata

Whenever we define equivalence for two automata or classes of automata, we must carefully state what is to be understood by this equivalence. For the rest of this chapter, we follow the precedence established for nfa's and dfa's and define equivalence with respect to the ability to accept languages.

**Definition 10.1**

Two automata are equivalent if they accept the same language. Consider two classes of automata $C_1$ and $C_2$. If for every automaton $M_1$ in $C_1$ there

is an automaton $M_2$ in $C_2$ such that

$$L(M_1) = L(M_2),$$

we say that $C_2$ is at least as powerful as $C_1$. If the converse also holds and for every $M_2$ in $C_2$ there is an $M_1$ in $C_1$ such that $L(M_1) = L(M_2)$, we say that $C_1$ and $C_2$ are equivalent.

---

There are many ways to establish the equivalence of automata. The construction of Theorem 2.2 does this for dfa's and nfa's. For demonstrating the equivalence in connection with Turing's machines, we often use the important technique of **simulation.**

Let $M$ be an automaton. We say that another automaton $\widehat{M}$ can simulate a computation of $M$, if $\widehat{M}$ can mimic the computation of $M$ in the following manner. Let $d_0, d_1, \ldots$ be the sequence of instantaneous descriptions of the computation of $M$, that is

$$d_0 \vdash_M d_1 \vdash_M \cdots \vdash_M d_n \cdots .$$

Then $\widehat{M}$ simulates this computation if it carries out a computation analogous to that of $M$,

$$\widehat{d}_0 \overset{*}{\vdash}_{\widehat{M}} \widehat{d}_1 \overset{*}{\vdash}_{\widehat{M}} \cdots \overset{*}{\vdash}_{\widehat{M}} \widehat{d}_n \cdots ,$$

where $\widehat{d}_0, \widehat{d}_1, \ldots$ are instantaneous descriptions, such that each of them is associated with a unique configuration of $M$. In other words, if we know the computation carried out by $\widehat{M}$, we can determine from it exactly what computations $M$ would have done, given the corresponding starting configuration.

Note that the simulation of a single move $d_i \vdash_M d_{i+1}$ of $M$ may involve several moves of $\widehat{M}$. The intermediate configurations in $\widehat{d}_i \overset{*}{\vdash}_{\widehat{M}} \widehat{d}_{i+1}$ may not correspond to any configuration of $M$, but this does not affect anything if we can tell which configurations of $\widehat{M}$ are relevant. As long as we can determine from the computation of $\widehat{M}$ what $M$ would have done, the simulation is proper. If $\widehat{M}$ can simulate every computation of $M$, we say that $\widehat{M}$ can simulate $M$. It should be clear that if $\widehat{M}$ can simulate $M$, then matters can be arranged so that $M$ and $\widehat{M}$ accept the same language, and the two automata are equivalent. To demonstrate the equivalence of two classes of automata, we show that for every machine in one class, there is a machine in the second class capable of simulating it.

## Turing Machines with a Stay-Option

In our definition of a standard Turing machine, the read-write head must move either to the right or to the left. Sometimes it is convenient to provide

a third option, to have the read-write head stay in place after rewriting the cell content. Thus, we can define a Turing machine with stay-option by replacing $\delta$ in Definition 9.1 by

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

with the interpretation that $S$ signifies no movement of the read-write head. This option does not extend the power of the automaton.

**Theorem 10.1** The class of Turing machines with stay-option is equivalent to the class of standard Turing machines.

**Proof:** Since a Turing machine with stay-option is clearly an extension of the standard model, it is obvious that any standard Turing machine can be simulated by one with a stay-option.

To show the converse, let $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ be a Turing machine with stay-option to be simulated by a standard Turing machine $\widehat{M} = \left(\widehat{Q}, \Sigma, \Gamma, \widehat{\delta}, \widehat{q}_0, \square, \widehat{F}\right)$. For each move of $M$, the simulating machine $\widehat{M}$ does the following. If the move of $M$ does not involve the stay-option, the simulating machine performs one move, essentially identical to the move to be simulated. If $S$ is involved in the move of $M$, then $\widehat{M}$ will make two moves: the first rewrites the symbol and moves the read-write head right; the second moves the read-write head left, leaving the tape contents unaltered. The simulating machine can be constructed by $M$ by defining $\widehat{\delta}$ as follows: For each transition

$$\delta(q_i, a) = (q_j, b, L \text{ or } R),$$

we put into $\widehat{\delta}$

$$\widehat{\delta}(\widehat{q}_i, a) = (\widehat{q}_j, b, L \text{ or } R).$$

For each $S$-transition

$$\delta(q_i, a) = (q_j, b, S),$$

we put into $\widehat{\delta}$ the corresponding transitions

$$\widehat{\delta}(\widehat{q}_i, a) = (\widehat{q}_{js}, b, R),$$

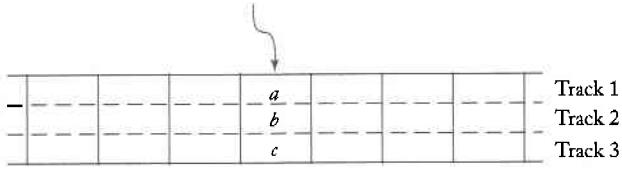
and

$$\widehat{\delta}(\widehat{q}_{js}, c) = (\widehat{q}_j, c, L),$$

for all $c \in \Gamma$.

It is reasonably obvious that every computation of $M$ has a corresponding computation of $\widehat{M}$, so that $\widehat{M}$ can simulate $M$. ■

Figure 10.1




Simulation is a standard technique for showing the equivalence of automata, and the formalism we have described makes it possible, as shown in the above theorem, to talk about the process precisely and prove theorems about equivalence. In our subsequent discussion, we use the notion of simulation frequently, but we generally make no attempt to describe everything in a rigorous and detailed way. Complete simulations with Turing machines are often cumbersome. To avoid this, we keep our discussion descriptive, rather than in theorem-proof form. The simulations are given only in broad outline, but it should not be hard to see how they can be made rigorous. The reader will find it instructive to sketch each simulation in some higher level language or in pseudocode.

Before introducing other models, we make one remark on the standard Turing machine. It is implicit in Definition 9.1 that each tape symbol can be a composite of characters rather than just a single one. This can be made more explicit by drawing an expanded version of Figure 9.1 (Figure 10.1), in which the tape symbols are triplets from some simpler alphabet.

In the picture, we have divided each cell of the tape into three parts, called **tracks**, each containing one member of the triplet. Based on this visualization, such an automaton is sometimes called a Turing machine with **multiple tracks**, but such a view in no way extends Definition 9.1, since all we need to do is make $\Gamma$ an alphabet in which each symbol is composed of several parts.

However, other Turing machine models involve a change of definition, so the equivalence with the standard machine has to be demonstrated. Here we look at two such models, which are sometimes used as the standard definition. Some variants that are less common are explored in the exercises at the end of this section.

## Turing Machines with Semi-Infinite Tape

Many authors do not consider the model in Figure 9.1 as standard, but use one with a tape that is unbounded only in one direction. We can visualize this as a tape that has a left boundary (Figure 10.2). This Turing machine is otherwise identical to our standard model, except that no left move is permitted when the read-write head is at the boundary.

It is not difficult to see that this restriction does not affect the power of the machine. To simulate a standard Turing machine $M$ by a machine $\widehat{M}$ with a semi-infinite tape, we use the arrangement shown in Figure 10.3.

**Figure 10.2**

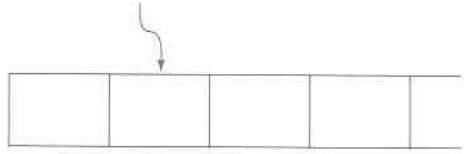


**Figure 10.3**


The simulating machine $\widehat{M}$ has a tape with two tracks. On the upper one, we keep the information to the right of some reference point on $M$'s tape. The reference point could be, for example, the position of the read-write head at the start of the computation. The lower track contains the left part of $M$'s tape in reverse order. $\widehat{M}$ is programmed so that it will use information on the upper track only as long as $M$'s read-write head is to the right of the reference point, and work on the lower track as $M$ moves into the left part of its tape. The distinction can be made by partitioning the state set of $\widehat{M}$ into two parts, say $Q_U$ and $Q_L$: the first to be used when working on the upper track, the second to be used on the lower one. Special end markers # are put on the left boundary of the tape to facilitate switching from one track to the other. For example, assume that the machine to be simulated and the simulating machine are in the respective configurations shown in Figure 10.4 and that the move to be simulated is generated by

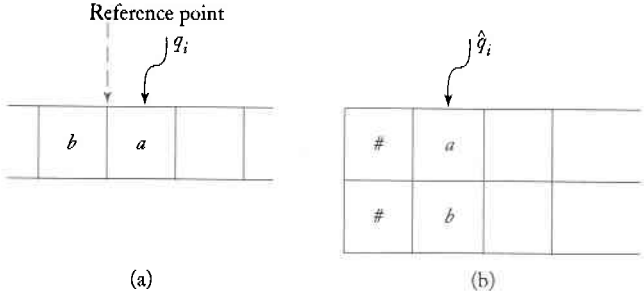
$$\delta(q_i, a) = (q_j, c, L).$$

The simulating machine will first move via the transition

$$\widehat{\delta}(\widehat{q}_i, (a, b)) = (\widehat{q}_j, (c, b), L),$$

where $\widehat{q}_i \in Q_U$. Because $\widehat{q}_i$ belongs to $Q_U$, only information in the upper track is considered at this point. Now, the simulating machine sees $(\#, \#)$

**Figure 10.4**
(a) Machine to be simulated.
(b) Simulating machine.



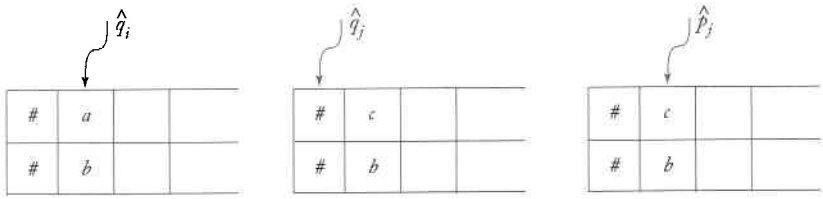



**Figure 10.5** Sequence of configurations in simulating $\delta(q_i, a) = (q_j, c, L)$.

in state $\hat{q}_j \in Q_U$. It next uses a transition

$$\hat{\delta}(\hat{q}_j, (\#, \#)) = (\hat{p}_j, (\#, \#), R),$$

with $\hat{p}_j \in Q_L$, putting it into the configuration shown in Figure 10.5. Now the machine is in a state from $Q_L$ and will work on the lower track. Further details of the simulation are straightforward.

## The Off-Line Turing Machine

The general definition of an automaton in Chapter 1 contained an input file as well as temporary storage. In Definition 9.1 we discarded the input file for reasons of simplicity, claiming that this made no difference to the Turing machine concept. We now expand on this claim.

If we put the input file back into the picture, we get what is known as an **off-line Turing machine**. In such a machine, each move is governed by the internal state, what is currently read from the input file, and what is seen by the read-write head. A schematic representation of an off-line machine is shown in Figure 10.6. A formal definition of an off-line Turing machine is easily made, but we will leave this as an exercise. What we

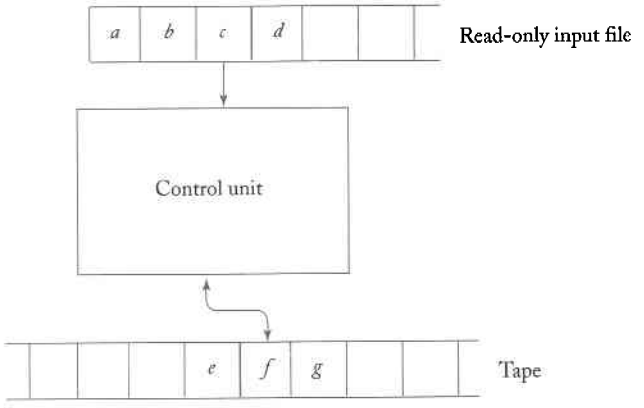
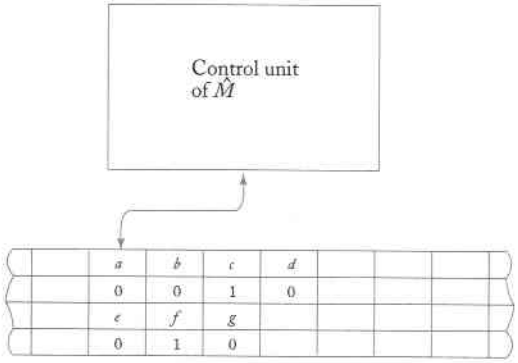



**Figure 10.6**

Figure 10.7




want to do briefly is to indicate why the class of off-line Turing machines is equivalent to the class of standard machines.

First, the behavior of any standard Turing machine can be simulated by some off-line model. All that needs to be done by the simulating machine is to copy the input from the input file to the tape. Then it can proceed in the same way as the standard machine.

The simulation of an off-line machine $M$ by a standard machine $\widehat{M}$ requires a lengthier description. A standard machine can simulate the computation of an off-line machine by using the four-track arrangement shown in Figure 10.7. In that picture, the tape contents shown represent the specific configuration of Figure 10.6. Each of the four tracks of $\widehat{M}$ plays a specific role in the simulation. The first track has the input, the second marks the position at which the input is read, the third represents the tape of $M$, and the fourth shows the position of $M$'s read-write head.

The simulation of each move of $M$ requires a number of moves of $\widehat{M}$. Starting from some standard position, say the left end, and with the relevant information marked by special end markers, $\widehat{M}$ searches track 2 to locate the position at which the input file of $M$ is read. The symbol found in the corresponding cell on track 1 is remembered by putting the control unit of $\widehat{M}$ into a state chosen for this purpose. Next, track 4 is searched for the position of the read-write head of $M$. With the remembered input and the symbol on track 3, we now know that $M$ is to do. This information is again remembered by $\widehat{M}$ with an appropriate internal state. Next, all four tracks of $\widehat{M}$'s tape are modified to reflect the move of $M$. Finally, the read-write head of $\widehat{M}$ returns to the standard position for the simulation of the next move.

## EXERCISES

1. Give a formal definition of a Turing machine with a semi-infinite tape. Then

prove that the class of Turing machines with semi-infinite tape is equivalent to the class of standard Turing machines.

2. Give a formal definition of an off-line Turing machine.

3. Give convincing arguments that any language accepted by an off-line Turing machine is also accepted by some standard machine.

4. Consider a Turing machine that, on any particular move, can either change the tape symbol or move the read-write head, but not both.

   (a) Give a formal definition of such a machine.

   (b) Show that the class of such machines is equivalent to the class of standard Turing machines.

5. Consider a model of a Turing machine in which each move permits the read-write head to travel more than one cell to the left or right, the distance and direction of travel being one of the arguments of $\delta$. Give a precise definition of such an automaton and sketch a simulation of it by a standard Turing machine.

6. A nonerasing Turing machine is one that cannot change a nonblank symbol to a blank. This can be achieved by the restriction that if

$$\delta(q_i, a) = (q_j, \square, L \text{ or } R),$$

   then $a$ must be $\square$. Show that no generality is lost by making such a restriction.

7. Consider a Turing machine that cannot write blanks; that is, for all $\delta(q_i, a) = (q_j, b, L \text{ or } R)$, $b$ must be in $\Gamma - \{\square\}$. Show how such a machine can simulate a standard Turing machine.

8. Suppose we make the requirement that a Turing machine can halt only in a final state, that is, we ask that $\delta(q, a)$ be defined for all pairs $(q, a)$ with $a \in \Gamma$ and $q \notin F$. Does this restrict the power of the Turing machine?

9. Suppose we make the restriction that a Turing machine must always write a symbol different from the one it reads, that is, if

$$\delta(q_i, a) = (q_j, b, L \text{ or } R),$$

   then $a$ and $b$ must be different. Does this limitation reduce the power of the automaton?

10. Consider a version of the standard Turing machine in which transitions can depend not only on the cell directly under the read-write head, but also on the cells to the immediate right and left. Make a formal definition of such a machine, then sketch its simulation by a standard Turing machine.

11. Consider a Turing machine with a different decision process in which transitions are made if the current tape symbol is not one of a specified set. For example

$$\delta(q_i, \{a, b\}) = (q_j, c, R)$$

will allow the indicated move if the current tape symbol is neither $a$ nor $b$. Formalize this concept and show that this modification is equivalent to a standard Turing machine.

## 10.2 Turing Machines with More Complex Storage

The storage device of a standard Turing machine is so simple that one might think it possible to gain power by using more complicated storage devices. But this is not the case as we now illustrate with two examples.

### Multitape Turing Machines

A multitape Turing machine is a Turing machine with several tapes, each with its own independently controlled read-write head (Figure 10.8).

The formal definition of a multitape Turing machine goes beyond Definition 9.1, since it requires a modified transition function. Typically, we define an $n$-tape machine by $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$, where $Q, \Sigma, \Gamma, q_0, F$ are as in Definition 9.1, but where

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

specifies what happens on all the tapes. For example, if $n = 2$, with a current configuration shown in Figure 10.8, then

$$\delta(q_0, a, e) = (q_1, x, y, L, R)$$

is interpreted as follows. The transition rule can be applied only if the machine is in state $q_0$ and the first read-write head sees an $a$ and the second an $e$. The symbol on the first tape will then be replaced with an $x$ and its read-write head will move to the left. At the same time, the symbol on

Figure 10.8

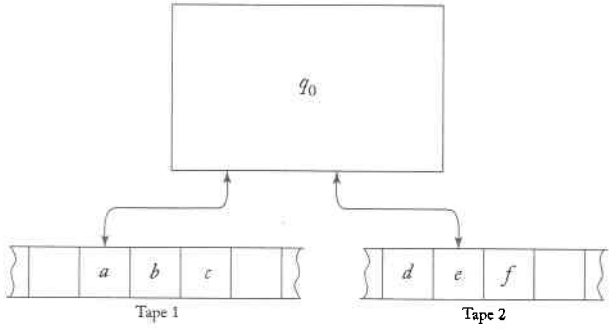
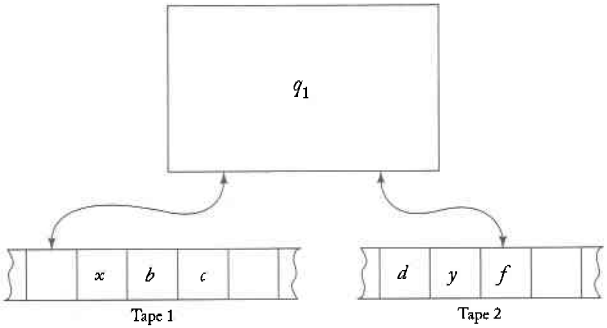


Figure 10.9




the second tape is rewritten as $y$ and the read-write head moves right. The control unit then changes its state to $q_1$ and the machine goes into the new configuration shown in Figure 10.9.

To show the equivalence between multitape and standard Turing machines, we argue that any given multitape Turing machine $M$ can be simulated by a standard Turing machine $\widehat{M}$ and, conversely, that any standard Turing machine can be simulated by a multitape one. The second part of this claim needs no elaboration, since we can always elect to run a multitape machine with only one of its tapes doing useful work. The simulation of a multitape machine by one with a single tape is a little more complicated, but conceptually straightforward.

Consider, for example, the two-tape machine in the configuration depicted in Figure 10.10. The simulating single-tape machine will have four

Figure 10.10

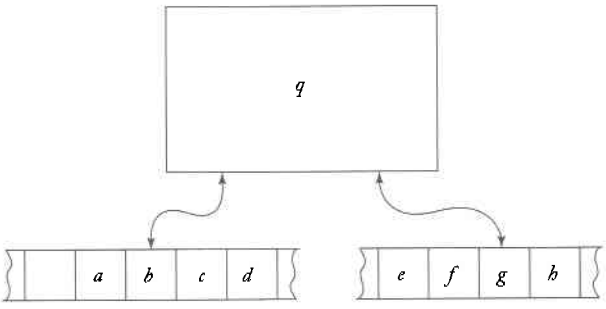
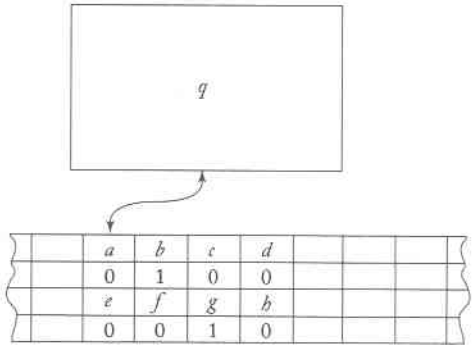


Figure 10.11




tracks (Figure 10.11). The first track represents the contents of tape 1 of $M$. The nonblank part of the second track has all zeros, except for a single 1 marking the position of $M$'s read-write head. Tracks 3 and 4 play a similar role for tape 2 of $M$. Figure 10.11 makes it clear that, for the relevant configurations of $\widehat{M}$ (that is, the ones that have the indicated form), there is a unique corresponding configuration of $M$.

The representation of a multitape machine by a single-tape machine is similar to that used in the simulation of an off-line machine. The actual steps in the simulation are also much the same, the only difference being that there are more tapes to consider. The outline given for the simulation of off-line machines carries over to this case with minor modifications and suggests a procedure by which the transition function $\widehat{\delta}$ of $\widehat{M}$ can be constructed from the transition function $\delta$ of $M$. While it is not difficult to make the construction precise, it takes a lot of writing. Certainly, the computations of $\widehat{M}$ given the appearance of being lengthy and elaborate, but this has no bearing on the conclusion. Whatever can be done of $M$ can also be done on $\widehat{M}$.

It is important to keep in mind the following point. When we claim that a Turing machine with multiple tapes is no more powerful than a standard one, we are making a statement only about what can be done by these machines, particularly, what languages can be accepted.

**Example 10.1** Consider the language $\{a^n b^n\}$. In Example 9.7, we described a laborious method by which this language can be accepted by a Turing machine with one tape. Using a two-tape machine makes the job much easier. Assume that an initial string $a^n b^m$ is written on tape 1 at the beginning of the computation. We then read all the $a$'s, copying them onto tape 2. When we reach the end of the $a$'s, we match the $b$'s on tape 1 against the copied $a$'s on tape 2. This way, we can determine whether there are an equal number of $a$'s and $b$'s without repeated back-and-forth movement of the read-write head.

■

Remember that the various models of Turing machines are considered equivalent only with respect to their ability to do things, not with respect to ease of programming or any other efficiency measure we might consider. We will return to this important point in Chapter 14.

## Multidimensional Turing Machines

A multidimensional Turing machine is one in which the tape can be viewed as extending infinitely in more than one dimension. A diagram of a two-dimensional Turing machine is shown in Figure 10.12.

The formal definition of a two-dimensional Turing machine involves a transition function $\delta$ of the form

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

where $U$ and $D$ specify movement of the read-write head up and down, respectively.

To simulate this machine on a standard Turing machine, we can use the two-track model depicted in Figure 10.13. First, we associate an ordering or address with the cells of the two-dimensional tape. This can be done in a number of ways, for example, in the two-dimensional fashion indicated in Figure 10.12. The two-track tape of the simulating machine will use one track to store cell contents and the other one to keep the associated address. In the scheme of Figure 10.12, the configuration in which cell $(1, 2)$ contains $a$ and cell $(10, -3)$ contains $b$ is shown in Figure 10.13. Note one complication: the cell address can involve arbitrarily large integers, so the address track cannot use a fixed-size field to store addresses. Instead, we must use a variable field-size arrangement, using some special symbols to delimit the fields, as shown in the picture.

Let us assume that, at the start of the simulation of each move, the read-write head of the two-dimensional machine $M$ and the read-write head of the simulating machine $\widehat{M}$ are always on corresponding cells. To simulate a move, the simulating machine $\widehat{M}$ first computes the address of the cell to

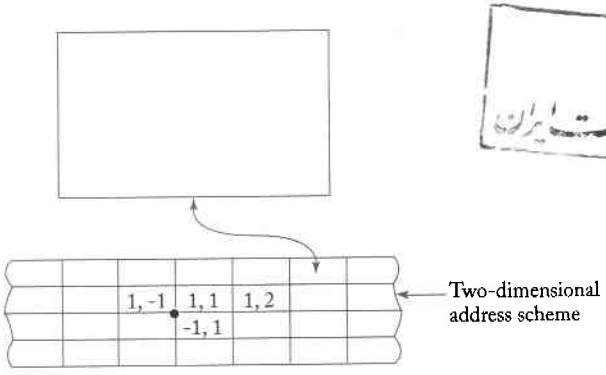



Figure 10.12

Figure 10.13

| | a | | | | b | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | # | 2 | # | 1 | 0 | # | - | 3 | # | |

which $M$ is to move. Using the two-dimensional address scheme, this is a simple computation. Once the address is computed, $\widehat{M}$ finds the cell with this address on track 2 and then changes the cell contents to account for the move of $M$. Again, given $M$, there is a straightforward construction for $\widehat{M}$.

## EXERCISES

The purpose of much of our discussion of Turing machines is to lend credence to Turing's thesis by showing how seemingly more complex situations can be simulated on a standard Turing machine. Unfortunately, detailed simulations are very tedious and conceptually uninteresting. In the exercises below, describe the simulations in just enough depth to show that the details can be worked out.

1. A multihead Turing machine can be visualized as a Turing machine with a single tape and a single control unit but with multiple, independent read-write heads. Give a formal definition of a multihead Turing machine, and then show how such a machine can be simulated with a standard Turing machine. ●

2. Give a formal definition of a multihead-multitape Turing machine. Then show how such a machine can be simulated by a standard Turing machine.

3. Give a formal definition of a Turing machine with a single tape but multiple control units, each with a single read-write head. Show how such a machine can be simulated with a multitape machine.

★ 4. A **queue automaton** is an automaton in which the temporary storage is a queue. Assume that such a machine is an on-line machine, that is, it has no input file, with the string to be processed placed in the queue prior to the start of the computation. Give a formal definition of such an automaton, then investigate its power in relation to Turing machines. ●

★ 5. Show that for every Turing machine there exists an equivalent standard Turing machine with no more than six states.

6. Reduce the number of required states in Exercise 5 as far as you can (Hint: the smallest possible number is three).

★ 7. A counter is a stack with an alphabet of exactly two symbols, a stack start symbol and a counter symbol. Only the counter symbol can be put on the stack or removed from it. A **counter automaton** is a deterministic automaton with one or more counters as storage. Show that any Turing machine can be simulated using a counter automaton with four counters.

8. Show that every computation that can be done by a standard Turing machine can be done by a multitape machine with a stay-option and at most two states. ●

9. Write out a detailed program for the computation in Example 10.1.

## 10.3 Nondeterministic Turing Machines

While Turing's thesis makes it plausible that the specific tape structure is immaterial to the power of the Turing machine, the same cannot be said of nondeterminism. Since nondeterminism involves an element of choice and so has a nonmechanistic flavor, an appeal to Turing's thesis is inappropriate. We must look at the effect of nondeterminism in more detail if we want to argue that nondeterminism adds nothing to the power of a Turing machine. Again we resort to simulation, showing that nondeterministic behavior can be handled deterministically.

**Definition 10.2**

A nondeterministic Turing machine is an automaton as given by Definition 9.1, except that $\delta$ is now a function

$$\delta : Q \times \Gamma \to 2^{Q \times \Gamma \times \{L,R\}}.$$

As always when nondeterminism is involved, the range of $\delta$ is a set of possible transitions, any of which can be chosen by the machine.

**Example 10.2** If a Turing machine has transitions specified by

$$\delta(q_0, a) = \{(q_1, b, R), (q_2, c, L)\}$$

it is nondeterministic. The moves

$$q_0aaa \vdash bq_1aa$$

and

$$q_0aaa \vdash q_2\Box caa$$

are both possible. ■

Since it is not clear what role nondeterminism plays in computing functions, nondeterministic automata are usually viewed as accepters. A nondeterministic Turing machine is said to accept $w$ if there is any possible sequence of moves such that

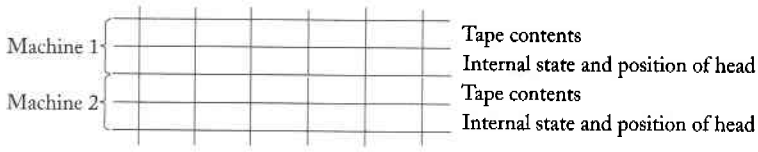
$$q_0 w \overset{*}{\vdash} x_1 q_f x_2,$$

with $q_f \in F$. A nondeterministic machine may have moves available that lead to a nonfinal state or to an infinite loop. But, as always with nondeterminism, these alternatives are irrelevant; all we are interested in is the existence of some sequence of moves leading to acceptance.

To show that a nondeterministic Turing machine is no more powerful than a deterministic one, we need to provide a deterministic equivalent for the nondeterminism. We have already alluded to one. Nondeterminism can be viewed as a deterministic backtracking algorithm, and a deterministic machine can simulate a nondeterministic one as long as it can handle the bookkeeping involved in the backtracking. To see how this can be done simply, let us consider an alternative view of nondeterminism, one which is useful in many arguments: a nondeterministic machine can be seen as one that has the ability to replicate itself whenever necessary. When more than one move is possible, the machine produces as many replicas as needed and gives each replica the task of carrying out one of the alternatives.

This view of nondeterminism may seem particularly nonmechanistic. Unlimited replication is certainly not within the power of present-day computers. Nevertheless, the process can be simulated by a deterministic Turing machine. Consider a Turing machine with a two-dimensional tape (Figure 10.14). Each pair of horizontal tracks represents one machine; the top track containing the machine's tape, the bottom one for indicating its internal state and the position of the read-write head. Whenever a new machine is to be created, two new tracks are started with the appropriate information. Figure 10.15 represents the initial configuration of the machine in Example 10.2 and its successor configurations. The simulating machine searches all active tracks; they are bracketed with special markers and so can always be found. It then carries out the indicated moves, activating new machines as needed. Quite a few details have to be resolved before we can claim to have a reasonable outline of the simulation, but we will leave this to the reader.

Figure 10.14



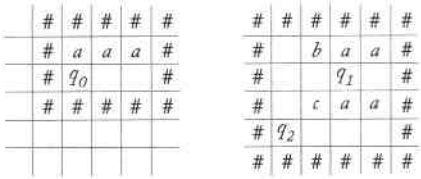



**Figure 10.15** Simulation of a nondeterministic move.

Based on this simulation, our conclusion is that for every nondeterministic Turing machine there exists an equivalent deterministic one. Because of its importance, we state this formally.

**Theorem 10.2** The class of deterministic Turing machines and the class of nondeterministic Turing machines are equivalent.

**Proof:** Use the construction suggested above to show that any nondeterministic Turing machine can be simulated by a deterministic one. ■

## EXERCISES

1. Discuss in detail the simulation of a nondeterministic Turing machine by a deterministic one. Indicate explicitly how new machines are created, how active machines are identified, and how machines that halt are removed from further consideration.
2. Show how a two-dimensional nondeterministic Turing machine can be simulated by a deterministic machine.
3. Write a program for a nondeterministic Turing machine that accepts the language

$$L = \{ww : w \in \{a, b\}^+\}.$$

Contrast this with a deterministic solution.

4. Outline how one would write a program for a nondeterministic Turing machine to accept the language

$$L = \left\{ww^R w : w \in \{a, b\}^+\right\}.$$

5. Write a simple program for a nondeterministic Turing machine that accepts the language

$$L = \left\{xww^R y : x, y, w \in \{a, b\}^+, |x| \geq |y|\right\}.$$

How would you solve this problem deterministically?

6. Design a nondeterministic Turing machine that accepts the language

$$L = \{a^n : n \text{ is not a prime number}\}.$$

★ 7. A two-stack automaton is a nondeterministic pushdown automaton with two independent stacks. To define such an automaton, we modify Definition 7.1 so that

$$\delta : Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \times \Gamma \rightarrow \text{ finite subsets of } Q \times \Gamma^* \times \Gamma^*.$$

A move depends on the tops of the two stacks and results in new values being pushed on these two stacks. Show that the class of two-stack automata is equivalent to the class of Turing machines.

## 10.4 A Universal Turing Machine

Consider the following argument against Turing's thesis: "A Turing machine as presented in Definition 9.1 is a special purpose computer. Once $\delta$ is defined, the machine is restricted to carrying out one particular type of computation. Digital computers, on the other hand, are general purpose machines that can be programmed to do different jobs at different times. Consequently, Turing machines cannot be considered equivalent to general purpose digital computers."

This objection can be overcome by designing a reprogrammable Turing machine, called a **universal Turing machine**. A universal Turing machine $M_u$ is an automaton that, given as input the description of any Turing machine $M$ and a string $w$, can simulate the computation of $M$ on $w$. To construct such an $M_u$, we first choose a standard way of describing Turing machines. We may, without loss of generality, assume that

$$Q = \{q_1, q_2, ..., q_n\},$$

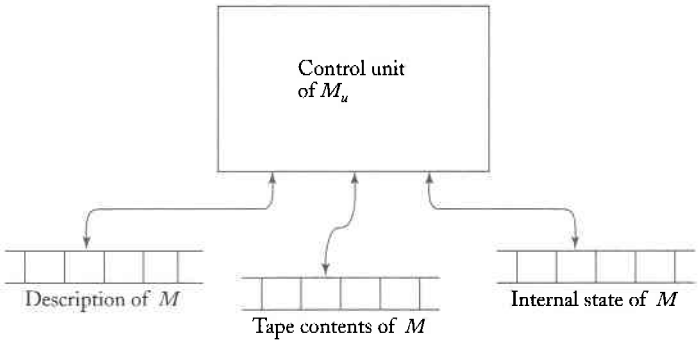
with $q_1$ the initial state, $q_2$ the single final state, and

$$\Gamma = \{a_1, a_2, ..., a_m\},$$

where $a_1$ represents the blank. We then select an encoding in which $q_1$ is represented by 1, $q_2$ is represented by 11, and so on. Similarly, $a_1$ is encoded as 1, $a_2$ as 11, etc. The symbol 0 will be used as a separator between the 1's. With the initial and final state and the blank defined by this convention, any Turing machine can be described completely with $\delta$ only. The transition function is encoded according to this scheme, with the arguments and result in some prescribed sequence. For example, $\delta(q_1, a_2) = (q_2, a_3, L)$ might appear as

$$\cdots 1\,0\,1\,1\,0\,1\,1\,0\,1\,1\,1\,0\,1\,0 \cdots .$$

Figure 10.16




It follows from this that any Turing machine has a finite encoding as a string on $\{0,1\}^+$, and that, given any encoding of $M$, we can decode it uniquely. Some strings will not represent any Turing machine (e.g., the strong 00011), but we can easily spot these, so they are of no concern.

A universal Turing machine $M_u$ then has an input alphabet that includes $\{0,1\}$ and the structure of a multitape machine, as shown in Figure 10.16.

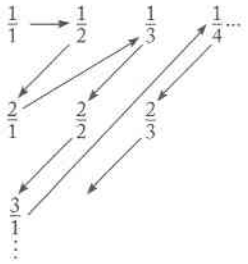
For any input $M$ and $w$, tape 1 will keep an encoded definition of $M$. Tape 2 will contain the tape contents of $M$, and tape 3 the internal state of $M$. $M_u$ looks first at the contents of tapes 2 and 3 to determine the configuration of $M$. It then consults tape 1 to see what $M$ would do in this configuration. Finally, tapes 2 and 3 will be modified to reflect the result of the move.

It is within reason to construct an actual universal Turing machine (see, for example, Denning, Dennis, and Qualitz 1978), but the process is uninteresting. We prefer instead to appeal to Turing's hypothesis. The implementation clearly can be done using some programming language; in fact, the program suggested in Exercise 1, Section 9.1 is a realization of a universal Turing machine in a higher level language. Therefore, we expect that it can also be done by a standard Turing machine. We are then justified in claiming the existence of a Turing machine that, given any program, can carry out the computations specified by that program and that is therefore a proper model for a general purpose computer.

The observation that every Turing machine can be represented by a string of 0's and 1's has important implications. But before we explore these implications, we need to review some results from set theory.

Some sets are finite, but most of the interesting sets (and languages) are infinite. For infinite sets, we distinguish between sets that are **countable** and sets that are **uncountable**. A set is said to be countable if its elements can be put into a one-to-one correspondence with the positive integers. By this we mean that the elements of the set can be written in some order, say, $x_1, x_2, x_3, \ldots$, so that every element of the set has some finite index. For example, the set of all even integers can be written in the order $0, 2, 4, \ldots$.

Figure 10.17



Since any positive integer $2n$ occurs in position $n+1$, the set is countable. This should not be too surprising, but there are more complicated examples, some of which may seem counterintuitive. Take the set of all quotients of the form $p/q$, where $p$ and $q$ are positive integers. How should we order this set to show that it is countable? We cannot write

$$\frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \cdots$$

because then $\frac{2}{3}$ would never appear. This does not imply that the set is uncountable; in this case, there is a clever way of ordering the set to show that it is in fact countable. Look at the scheme depicted in Figure 10.17, and write down the element in the order encountered following the arrows. This gives us

$$\frac{1}{1}, \frac{1}{2}, \frac{2}{1}, \frac{1}{3}, \frac{2}{2}, \cdots$$

Here the element $\frac{2}{3}$ occurs in the eighth place, and every element has some place in the sequence. The set is therefore countable.

We see from this example that we can prove that a set is countable if we can produce a method by which its elements can be written in some sequence. We call such a method an **enumeration procedure**. Since an enumeration procedure is some kind of mechanical process, we can use a Turing machine model to define it formally.

**Definition 10.3**

Let $S$ be a set of strings on some alphabet $\Sigma$. Then an enumeration procedure for $S$ is a Turing machine that can carry out the sequence of steps

$$q_0\Box \overset{*}{\vdash} q_s x_1 \# s_1 \overset{*}{\vdash} q_s x_2 \# s_2 \ldots,$$

with $x_i \in \Gamma^* - \{\#\}$, $s_i \in S$, in such a way that any $s$ in $S$ is produced in a finite number of steps. The state $q_s$ is a state signifying membership in $S$; that is, whenever $q_s$ is entered, the string following $\#$ must be in $S$.

Not every set is countable. As we will see in the next chapter, there are some uncountable sets. But any set for which an enumeration procedure exists is countable because the enumeration gives the required sequence.

Strictly speaking, an enumeration procedure cannot be called an algorithm, since it will not terminate when $S$ is infinite. Nevertheless, it can be considered a meaningful process, because it produces well-defined and predictable results.

**Example 10.3** Let $\Sigma = \{a, b, c\}$. We can show that the $S = \Sigma^+$ is countable if we can find an enumeration procedure that produces its elements in some order, say in the order in which they would appear in a dictionary. However, the order used in dictionaries is not suitable without modification. In a dictionary, all words beginning with $a$ are listed before the string $b$. But when there are an infinite number of $a$ words, we will never reach $b$, thus violating the condition of Definition 10.3 that any given string be listed after a finite number of steps.

Instead, we can use a modified order, in which we take the length of the string as the first criterion, followed by an alphabetic ordering of all equal-length strings. This is an enumeration procedure that produces the sequence

$$a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, \ldots.$$

As we will have several uses for such an ordering we will call it the **proper order**. ■

An important consequence of the above discussion is that Turing machines are countable.

**Theorem 10.3** The set of all Turing machines, although infinite, is countable.

**Proof:** We can encode each Turing machine using 0 and 1. With this encoding, we then construct the following enumeration procedure.

1. Generate the next string in $\{0, 1\}^+$ in proper order.
2. Check the generated string to see if it defines a Turing machine. If so, write it on the tape in the form required by Definition 10.3. If not, ignore the string.
3. Return to Step 1.

Since every Turing machine has a finite description, any specific machine will eventually be generated by this process. ■

The particular ordering of Turing machines depends on the encoding we use; if we use a different encoding, we must expect a different ordering. This is of no consequence, however, and shows that the ordering itself is unimportant. What matters is the existence of some ordering.

## EXERCISES

1. Sketch an algorithm that examines a string in $\{0,1\}^+$ to determine whether or not it represents an encoded Turing machine.
2. Give a complete encoding, using the suggested method, for the Turing machine with

$$\begin{aligned}\delta(q_1, a_1) &= (q_1, a_1, R),\\ \delta(q_1, a_2) &= (q_3, a_1, L),\\ \delta(q_3, a_1) &= (q_2, a_2, L).\end{aligned}$$

3. Sketch a Turing machine program that enumerates the set $\{0,1\}^+$ in proper order.
4. What is the index of $0^i1^j$ in Exercise 3?
5. Design a Turing machine that enumerates the following set in proper order.

$$L = \{a^n b^n : n \geq 1\}$$

6. For Example 10.3, find a function $f(w)$ that gives for each $w$ its index in the proper ordering.
7. Show that the set of all triplets, $(i, j, k)$ with $i, j, k$ positive integers, is countable.
8. Suppose that $S_1$ and $S_2$ are countable sets. Show that then $S_1 \cup S_2$ and $S_1 \times S_2$ are also countable.
9. Show that the Cartesian product of a finite number of countable sets is countable.

## 10.5 Linear Bounded Automata

While it is not possible to extend the power of the standard Turing machine by complicating the tape structure, it is possible to limit it by restricting the way in which the tape can be used. We have already seen an example of this with pushdown automata. A pushdown automaton can be regarded as a nondeterministic Turing machine with a tape that is restricted to being used like a stack. We can also restrict the tape usage in other ways; for example, we might permit only a finite part of the tape to be used as work space. It can be shown that this leads us back to finite automata (see

Exercise 3 at the end of this section), so we need not pursue this. But there is a way of limiting tape use that leads to a more interesting situation: we allow the machine to use only that part of the tape occupied by the input. Thus, more space is available for long input strings than for short ones, generating another class of machines, the **linear bounded automata** (or **lba**).

A linear bounded automaton, like a standard Turing machine, has an unbounded tape, but how much of the tape can be used is a function of the input. In particular, we restrict the usable part of the tape to exactly the cells taken by the input. To enforce this, we can envision the input as bracketed by two special symbols, the **left-end marker** ([) and the **right-end marker** (]). For an input $w$, the initial configuration of the Turing machine is given by the instantaneous description $q_0\,[w]$. The end markers cannot be rewritten, and the read-write head cannot move to the left of [ or to the right of ]. We sometimes say that the read-write head "bounces" off the end markers.

**Definition 10.4**

A linear bounded automaton is a nondeterministic Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$, as in Definition 10.2, subject to the restriction that $\Sigma$ must contain two special symbols [ and ], such that $\delta(q_i, [)$ can contain only elements of the form $(q_j, [, R)$, and $\delta(q_i, ])$ can contain only elements of the form $(q_j, ], L)$.

**Definition 10.5**

A string $w$ is accepted by a linear bounded automaton if there is a possible sequence of moves

$$q_0\,[w] \overset{*}{\vdash} [x_1 q_f x_2]$$

for some $q_f \in F, x_1, x_2 \in \Gamma^*$. The language accepted by the lba is the set of all such accepted strings.

Note that in this definition a linear bounded automaton is assumed to be nondeterministic. This is not just a matter of convenience but essential to the discussion of lba's. While one can define deterministic lba's, it is not known whether they are equivalent to the nondeterministic version. For some exploration of this, see Exercise 8 at the end of this section.

**Example 10.4** The language

$$L = \{a^n b^n c^n : n \geq 1\}$$

is accepted by some linear bounded automaton. This follows from the discussion in Example 9.8. The computation outlined there does not require space outside the original input, so it can be carried out by a linear bounded automaton. ■

**Example 10.5** Find a linear bounded automaton that accepts the language

$$L = \{a^{n!} : n \geq 0\}.$$

One way to solve the problem is to divide the number of $a$'s successively by $2, 3, 4, \ldots$, until we can either accept or reject the string. If the input is in $L$, eventually there will be a single $a$ left; if not, at some point a nonzero remainder will arise. We sketch the solution to point out one tacit implication of Definition 10.4. The tape of a linear bounded automaton may be multitrack, and the extra tracks can be used as scratch space. For this problem, we can use a two-track tape. The first track contains the number of $a$'s left during the process of division, and the second track contains the current divisor (Figure 10.18). The actual solution is fairly simple. Using the divisor on the second track, we divide the number of $a$'s on the first track, say by removing all symbols except those at multiples of the divisor. After this, we increment the divisor by one, and continue until we either find a nonzero remainder or are left with a single $a$. ■

The last two examples suggest that linear bounded automata are more powerful than pushdown automata, since neither of the languages are context-free. To prove such a conjecture, we still have to show that any context-free language can be accepted by a linear bounded automaton. We will do this later in a somewhat roundabout way; a more direct approach is suggested in Exercises 5 and 6 at the end of this section. It is not so easy to make a conjecture on the relation between Turing machines and linear bounded automata. Problems like Example 10.5 are invariably solvable by a linear bounded automaton, since an amount of scratch space proportional

Figure 10.18

to the length of the input is available. In fact, it is quite difficult to come up with a concrete and explicitly defined language that cannot be accepted by any linear bounded automaton. In Chapter 11 we will show that the class of linear bounded automata is less powerful than the class of unrestricted Turing machines, but a demonstration of this requires a lot more work.

## EXERCISES

1. Give details for the solution of Example 10.5.
2. Find a solution for Example 10.5 that does not require a second track as scratch space.
3. Consider an off-line Turing machine in which the input can be read only once, moving left to right, and not rewritten. On its work tape, it can use at most $n$ extra cells for work space, where $n$ is fixed for all inputs. Show that such a machine is equivalent to a finite automaton.
4. Find linear bounded automata for the following languages.
   (a) $L = \{a^n : n = m^2, m \geq 1\}$
   (b) $L = \{a^n : n \text{ is a prime number}\}$
   (c) $L = \{a^n : n \text{ is not a prime number}\}$
   (d) $L = \{ww : w \in \{a,b\}^+\}$
   (e) $L = \{w^n : w \in \{a,b\}^+, n \geq 1\}$
   (f) $L = \{www^R : w \in \{a,b\}^+\}$
5. Find an lba for the complement of the language in Example 10.5, assuming that $\Sigma = \{a,b\}$.
6. Show that for every context-free language there exists an accepting pda, such that the number of symbols in the stack never exceeds the length of the input string by more than one.
7. Use the observation in the above exercise to show that any context-free language not containing $\lambda$ is accepted by some linear bounded automaton.
8. To define a deterministic linear bounded automaton, we can use Definition 10.4, but require that the Turing machine be deterministic. Examine your solutions to Exercise 4. Are the solutions all deterministic linear bounded automata? If not, try to find solutions that are.