Chapter 2
Introduction to Computability

This subject is primarily concerned with the limitations of computing. As one of the highlights of this study, we will learn several specific problems that computers cannot solve.

A robust theory of computability dates back to the work of Church [Chu36] and Turing [Tur36] and provides models of computation and sophisticated methods that will be useful in our study of complexity theory as well. Although much of that work predated digital computers and was without forethought of modern technology, we know that von Neumann was influenced by Turing's invention of a universal, general-purpose, stored-program computer.

The basic model of computation for our study is the Turing machine, and for complexity theory, is the multitape Turing machine. However, these subjects should not depend too much on the choice of computational model. To this end, we will discuss Church's thesis as well as an expanded version of Church's thesis. Church's thesis states that every computational device can be simulated by a Turing machine. Evidence for this thesis comes from the fact that it has withstood the test of time, and it has been proven for all known reasonable models of sequential computation, including random access machines (RAMs). One of the topics in this chapter is the simulation of RAMs by Turing machines.

We view the study of computability to be an important prelude to the study of complexity theory. First, the models and methods that we learn in these chapters will be important later as well. Second, before concerning ourselves with the question of what problems can be efficiently computed, it is good to appreciate that there is a vast world of problems that we can easily formulate as computational problems but that computers cannot solve.

2.1 Turing Machines

Even before the invention of modern computers, Alan Turing (1936) described a theoretical model of a computing machine. Although very simple in structure, the

Turing machine possesses remarkable properties. In particular, a Turing machine is as powerful as any other computing device.

Computer memory is organized linearly as the cells of an infinite tape. Each cell of the tape may hold exactly one of a finite number of symbols chosen from a finite tape alphabet, Γ. Initially, an input word is placed on the otherwise empty tape as a word w in the input alphabet, Σ. This is shown in Fig. 2.1.

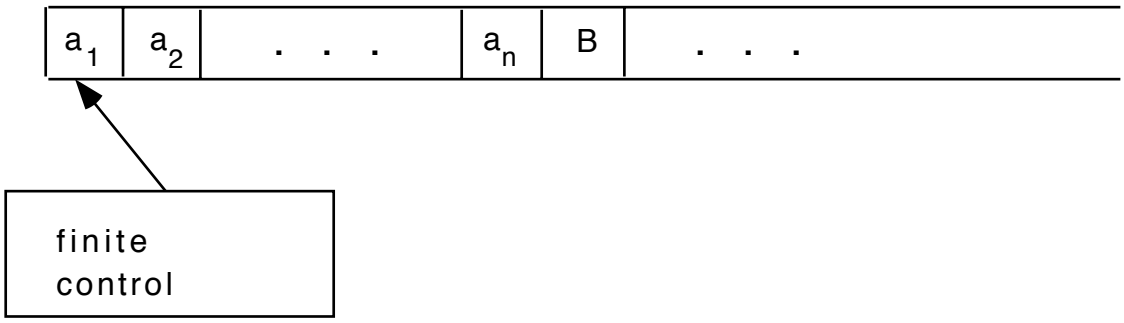


Fig. 2.1 Diagram of a Turing machine.

A *finite control* is attached to a head that scans one cell of the tape at a time. What the control does next, whether it scans a different symbol or prints a new symbol on the square, depends on its "internal state" as well as the symbol currently being scanned.

One can think of this as a machine, made of transistors or toothpicks and rubber bands. It doesn't matter. The combination of whatever the insides look like (the state) coupled with what symbol is being scanned determines what happens next.

In a move, the machine

1. prints a new symbol,
2. shifts its head one cell left or right,
3. changes state.

The number of states is finite, and at any time only a finite amount of memory is being used. A machine must have a finite description. A Turing machine has two designated special states, q_{accept} and q_{reject}. The machine halts (stops operating) when it enters either of these states. Turing machines may run forever on some inputs. This is entirely consistent with the experience of most programmers.

We let Q be the finite set of states. We permit symbols to be written on cells of the tape from a finite alphabet $\Gamma \supseteq \Sigma$.

The next move is determined by a *transition* function

$$\delta : (Q - \{q_{accept}, q_{reject}\}) \times \Gamma \to Q \times \Gamma \times \{L, R\}.$$

Hence, δ maps a (current state, current symbol scanned) pair into a triple consisting of (the next state, the new symbol to be printed, indication to move the head one cell to the left or right). There is no next move from either q_{accept} or q_{reject}.

Formally, a Turing machine is a system

$$M = \langle Q, \Sigma, \Gamma, \delta, q_0, B, q_{accept}, q_{reject} \rangle,$$

where

Q is the finite set of states,
Γ is the finite tape alphabet,
$B \in \Gamma$, the *blank*,
Σ is the input alphabet, $\Sigma \subseteq \Gamma - \{B\}$,
δ is the transition function,
$q_0 \in Q$ is the *initial state*,
q_{accept} is the accepting state, and
q_{reject} is the rejecting state.

To avoid confusion, we usually take $Q \cap \Gamma = \emptyset$.

If M tries to move left when in the leftmost square, the head stays in the same place and the computation continues. The first instance of the blank symbol B denotes the end of the input word.

Example 2.1. A parity counter. The input is a word w in $\{0,1\}^*$. The Turing machine M is to halt in state q_{accept} if the number of 1's on its tape is odd and to halt in state q_{reject} if the number of 1's is even.

$\Sigma = \{0,1\}$, $\Gamma = \{0,1,B\}$, and $Q = \{q_0, q_1, q_{accept}, q_{reject}\}$ (q_0 for even and q_1 for odd). The transition function δ is often given as a matrix with rows of the form

$$q \; a \;\big|\; q' \; b \; D$$

where q is the current state, a is the symbol currently stored in the cell being scanned, q' is the next state, b is the symbol to be written in the cell, and the direction D is either left or right. Hence, we describe the transition function for the parity counter as follows:

$$\begin{array}{cc|ccc}
q_0 & 0 & q_0 & 0 & R \\
q_0 & 1 & q_1 & 1 & R \\
q_0 & B & q_{reject} & - & - \\
q_1 & 0 & q_1 & 0 & R \\
q_1 & 1 & q_0 & 1 & R \\
q_1 & B & q_{accept} & - & -
\end{array}$$

A Turing machine continues until it reaches the accept or reject state. If it never reaches one of these states, then the computation continues forever.

As a Turing machine computes, changes occur to the state, tape contents, and current head position. A setting of this information is called a configuration. Formally, we define an *instantaneous description* (ID) or *configuration* of a Turing machine M to be a word $\alpha_1 q \alpha_2$, where $q \in Q$ is the current state and $\alpha_1\alpha_2$ is the contents of the tape up to the rightmost nonblank or up to the symbol to the left of the head, whichever is rightmost. The tape head is scanning the first symbol of α_2 or B, in case $\alpha_2 = \lambda$.

We define the *next move* relation, denoted by a turnstile, $\vdash_M$. Let

$$X_1 X_2 \ldots X_{i-1} q X_i \ldots X_n$$

be an ID.

1. Suppose $\delta(q, X_i) = (p, Y, L)$. If $i > 1$, then

$$X_1 X_2 \ldots X_{i-1} q X_i \ldots X_n \vdash_M X_1 X_2 \ldots X_{i-2} p X_{i-1} Y X_{i+1} \ldots X_n.$$

   If $i = 1$, then

$$q X_1 \ldots X_n \vdash_M pY X_2 \ldots X_n.$$

2. Suppose $\delta(q, X_i) = (p, Y, R)$. Then

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

   unless $i - 1 = n$, in which case

$$X_1 \ldots X_n q \vdash_M X_1 \ldots X_n Y p.$$

The relation $\vdash_M^*$ is the reflexive, transitive closure of $\vdash_M$. This means that $C \vdash_M^* D$, where C and D are configurations, if and only if $C = D$ or there is a sequence of configurations $C_1, \ldots, C_k$ such that

$$C = C_1 \vdash C_2 \vdash \cdots \vdash C_k = D.$$

A configuration is *accepting* if the state of the configuration is q_{accept} and is *rejecting* if the state is q_{reject}. Accepting and rejecting configurations are the only *halting* configurations. The Turing machine M *accepts* an input word w if $q_0 w \vdash_M^* I$, where I is an accepting configuration. Similarly, the Turing machine M *rejects* a word w if $q_0 w \vdash_M^* I$, where I is rejecting.

2.2 Turing Machine Concepts

Definition 2.1. Let M be a Turing machine. The language *accepted* by M is

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

A language L, $L \subseteq \Sigma^*$, is *Turing-machine-acceptable* if there is a Turing machine that accepts L.

Note that M might not halt on words w that belong to $\overline{L}$.

Definition 2.2. A language L is *Turing-machine-decidable* if L is accepted by some Turing machine that halts on every input, and a Turing machine that halts on every input and accepts L is called a *decider* for L.

Usually we will write "acceptable" instead of "Turing-machine-acceptable," and write "decidable" instead of "Turing-machine-decidable." Note the distinction between these two definitions. If M accepts L, then, for all words $x \in \Sigma^*$,

$$\begin{aligned} x \in L \Rightarrow\ & M \text{ eventually enters state } q_{accept}, \text{ and} \\ x \notin L \Rightarrow\ & \text{either } M \text{ eventually enters state } q_{reject} \text{ or} \\ & M \text{ runs forever.} \end{aligned}$$

However, if M decides L, then

$$\begin{aligned} x \in L \Rightarrow\ & M \text{ eventually enters state } q_{accept}, \text{ and} \\ x \notin L \Rightarrow\ & M \text{ eventually enters state } q_{reject}. \end{aligned}$$

Every Turing machine M accepts some language, but a Turing machine might not be a decider for any language at all, simply because it does not halt on all input strings.

Proposition 2.1. *If L is decidable, then $\overline{L}$ is decidable.*

Homework 2.1 *Design Turing machines to decide the following languages:*[1]

1. $\{0^n 1^n 0^n \mid n \geq 1\}$;
2. $\{ww \mid w \in \{0,1\}^*\}$;
3. $\{ww^R \mid w \in \{0,1\}^*\}$. *($w^R$ denotes the "reversal" of w, so if $w = a_1 a_2 \cdots a_k$, then $w^R = a_k a_{k-1} \cdots a_1$.)*

Primarily we will be concerned with questions about whether certain languages are either acceptable or decidable, and eventually about whether certain languages have efficient deciders. For this reason, we formulated Turing machines as acceptors of languages. However, it is important to realize that Turing machines can also compute functions. After all, ordinary computing involves both input and output,

[1] You should not give the formal descriptions of Turing machines that solve homework problems such as this one. Since Turing-machine programs are unreadable, as is true in general of machine-language programs, it is not desirable to give formal descriptions. Instead, describe in relative detail how the Turing machine moves its head, stores data on its tape, changes states, and so forth. In this manner describe a Turing machine that implements your algorithm for solving the problems.

and output is usually more complex than merely a bit indication of the machine's final state. For this reason, when computing a partial function the final state is no longer relevant. We will continue to assume that a Turing machine that computes a partial function contains the state q_{accept}, but we no longer assume that M contains the state q_{reject}. The state q_{accept} is the only halting state. We arrive at the following definition:

A Turing machine M *computes* the partial function $\phi : (\Sigma^*)^n \to \Sigma^*$ if, when the initial ID is $q_0 w_1 B w_2 B \ldots w_n$, then

1. M eventually enters an accepting configuration if and only if $\phi(w_1, \ldots, w_n) \downarrow$, and
2. if and when M does so, then the accepting configuration is of the form

$$\phi(w_1, \ldots, w_n) q_{accept}.$$

That is, the tape is is empty except for the value of $\phi(w_1, \ldots, w_n)$, and the Turing machine halts with the head behind this value.

When M executes on an arbitrary input word x, only two possibilities exist: Either M accepts x, in which case $x \in dom(\phi)$ and M outputs the value $\phi(x)$, or M executes forever. Observe that $L(M)$ is the domain of ϕ.

Every Turing machine M and $n \geq 1$ determines the partial function ϕ_M^n of n arguments such that M halts behind $\phi_M^n(w_1, \ldots, w_n)$ if and when M halts after being run on initial ID $q_0 w_1 B w_2 B \ldots w_n$.

A partial function ϕ is *partial computable* if there is some Turing machine that computes it. If $\phi(w_1, \ldots, w_n) \downarrow$ for all $w_1, \ldots, w_n$, then ϕ is *total computable*. Observe that a Turing machine that computes a total computable function accepts and halts on every input. Therefore, if M computes a total computable function, then $L(M) = \Sigma^*$.

It is more traditional to think of the partial computable functions as mappings on the natural numbers. Fixing 2-adic as the representation gives us a unique class of partial computable functions over the natural numbers. Many texts represent the positive integer n by the string 1^{n+1}, but, thinking ahead, from a complexity theoretician's point of view, the more succinct representation is preferable.

2.3 Variations of Turing Machines

Students of computer science should have little difficulty appreciating that Turing machines have as much power as any other computing device. That is, the Turing machine accepts or decides the exact same languages and computes the exact same partial functions as any other computing device. The Turing machine is designed to perform symbol manipulation in a simple, straightforward manner. This is one common view of computing. What might be the differences between a Turing machine and computers we use every day? Both store symbols in storage locations. The latter might store 32 or 64 symbols in one storage location, whereas the Turing machine

stores only one symbol in each storage location, but that difference is not essential; rather, it makes the machines that we use more efficient. Both types of machines can change these symbols in one move and change state. The machines that we use have "random access," meaning, for example, that they can be reading the contents of memory location 100 and then, by executing one move, read memory location 1000. The Turing machine can easily simulate that, but must make 900 moves in order to move its head 900 cells to the right. Again, the difference is in efficiency rather than in fundamental computing power.

In part, the rest of this chapter is dedicated to expanding on this brief argument, to provide evidence that the Turing machine is as powerful as any other computer. This hypothesis is known as "Church's thesis." In this section we will introduce two important variations of the Turing-machine model and prove that they are equivalent to Turing machines. The next section discusses Church's thesis in more detail, and then we will examine random access machines more technically than in the previous paragraph.

2.3.1 Multitape Turing Machines

A k-tape Turing machine M, as pictured in Fig. 2.2, has k tapes, each with its own read/write head. When M is in a state q reading the scanned symbol in each tape, M writes over each of these symbols, moves left or right on each tape, and enters a new state.

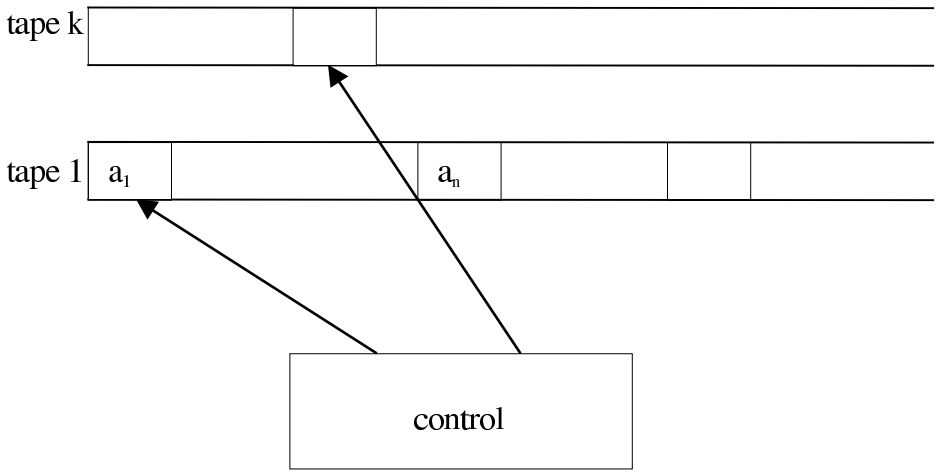
Formally, the transition function is of the form

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

Thus, given state q and scanned symbols $a_1,\ldots,a_k$, on tapes $1,\ldots,k$, respectively, $\delta(q,a_1,\ldots,a_k) = (p,b_1,\ldots,b_k,D_1,\ldots,D_k)$, where $b_1,\ldots,b_k$ are the new symbols to be written, $D_1,\ldots,D_k$ are the directions (left or right) in which the heads should move, and p is the next state.

Definition 2.3. Two Turing machines are *equivalent* if they accept the same language.

Two Turing machines might accept the same language L, and the first might be a decider for L, while the second does not even halt on all inputs. We will prove that every multitape Turing machine is equivalent to some ordinary (one-tape) Turing machine. Our interest in this result is twofold. First, showing that adding features does not increase a Turing machine's computational power helps us to convince ourselves that the Turing machine is a natural model of computing. Second, the multitape Turing machine is in fact the model that we will accept as our standard, and as such is the one to be used in the remainder of the course. This is a matter of efficiency. The multitape Turing machine is more efficient and easier to program than single-tape Turing machines. Also, there are efficient simulations between the multitape Turing machine and all other known models of computation.


tape k
tape 1
a_1
a_n
control

Example 2.2. This example illustrates the efficiency of multitape Turing machines over single-tape Turing machines. We can easily design a two-tape Turing machine to accept the language $L = \{ww^R \mid w \in \{0,1\}^*\}$. Namely, given an input word that is placed on tape 1, the machine copies the input word onto the second tape. This requires one sweep of both reading heads from left to right. Next, move the reading head on tape 1 back to the left end. Then, compare the word claimed to be w^R with the word claimed to be w by moving the two heads in opposite directions, moving the head on tape 1 from left to right and on tape 2 from right to left. At each step, if the symbols on the scanned cells are identical, then continue, else halt without accepting. Finally, after the input word passes these tests, determine whether the length of the input word is even and if it is, then accept.

Observe that this machine requires no more that a few scans of its heads from one end of the input word to the other. Hence, the number of steps is $O(n)$, where $n = |w|$. In general, we will measure the complexity of computational problems as a function of the length of the input word.

In contrast, consider your solution to Homework 2.1. First, the above solution is probably simpler. Second, your solution probably compares the first symbol of the input word with the last, placing markers to detect that they were visited. Then the machine compares the second symbol with the next-to-last symbol, and so on. The first comparison requires n steps, the second comparison requires $n-1$ steps, and so forth, until we reach the last comparison, which requires only one step. Thus, the total number of comparisons is $O(n^2)$. Indeed, one can prove that every single-tape Turing machine that accepts L must take at least $O(n^2)$ steps.

Homework 2.2 *In the above example, why do you need to check whether the input word is even before accepting?*

Theorem 2.1. *Every multitape Turing machine has an equivalent one-tape Turing machine.*

Proof. Let M be a k-tape Turing machine. The single tape of N is viewed as consisting of k "tracks," one track for each work tape of M. This is accomplished by enlarging the tape alphabet of N so that each cell of N contains a symbol that represents k symbols of M and possibly a marker $\uparrow$ to indicate M's head position on the ith tape head. The tape alphabet of N is sufficiently large so that each tape symbol of N uniquely denotes such an array of M's tape symbols and head indicator. (For example, in the illustration given in Fig. 2.3, the second cell of N contains a symbol that uniquely denotes the array $(a_{12}, a_{22}, \ldots, \uparrow\ a_{k2})$.)

The one-tape Turing machine N simulates a move of M as follows: Initially N's head is at the leftmost cell containing a head marker. N sweeps right, visiting each of the cells with head markers, and stores in its control the symbol scanned by each of the k heads of M.[2] When N crosses a head marker, it updates the count of head markers to its right. When no more head markers are to the right, N has enough

[2] By increasing the number of states, a Turing machine can always store a fixed number of symbols in its finite control. The technique is to let each new state uniquely represent a combination of stored symbols.

information to determine the move of M. Next, N sweeps left until it reaches the leftmost head marker. While doing so, it updates the tape symbols of M scanned by the head markers and moves the head markers one cell left or right as required. Finally, N changes M's current state, which is stored in N's control. N accepts its input word if and only if the new state of M is accepting. □

In this proof, if M is a decider for a language L, then so is N, for we need only to stipulate that N rejects if and only if the new state of M is rejecting. Also, note that N must make many moves in order to simulate one move of M. This is frequently true of simulations. Since one head of M might be moving to the left while another is moving to the right, it can take about $\Sigma_{i=1}^{n} 2i = O(n^2)$ moves of N to simulate n moves of M.

By definition, two Turing machines are equivalent if they accept the same language, and this definition will continue to be useful for us. You might think it more natural to have defined "equivalent" to mean that the two machines compute the same partial function. Clearly, this is the stronger notion, for it is possible for two different partial computable functions to have the same domain. The simulation given in the proof of Theorem 2.1 holds for the stronger notion as well. Thus, we state the following corollary:

Corollary 2.1. *For every multitape Turing machine there is a one-tape Turing machine that computes the same partial computable function.*

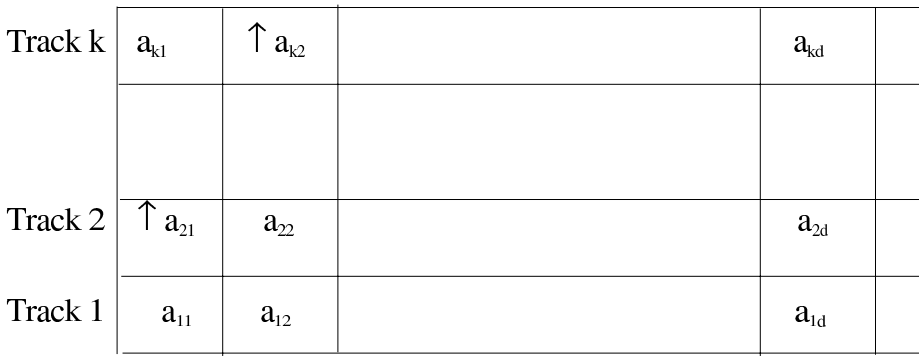
The following theorem is an important characterization of the decidable sets and pinpoints the distinction between decidable and acceptable. Its proof is a good illustration of the usefulness of multitape Turing machines.

Theorem 2.2. *A language L is decidable if and only if both L and $\overline{L}$ are acceptable.*

Proof. If L is decidable, then it is acceptable, by definition. Let M be a decider for L. A Turing machine that is exactly like M, but with the states q_{accept} and q_{reject} reversed, is a decider for $\overline{L}$ (which, in passing, proves Proposition 2.1) and so $\overline{L}$ is acceptable. For the proof of the converse, let M_L and $M_{\overline{L}}$ be Turing machines that accept L and $\overline{L}$, respectively. Design N so that on an input word w, N copies the input to a second tape and then simulates M_L on some of its tapes while simultaneously simulating $M_{\overline{L}}$ on others of its tapes. N is to accept w if the simulation of M_L accepts, and is to reject w if the simulation of $M_{\overline{L}}$ accepts. Clearly, N accepts L. Since every word w belongs to either L or $\overline{L}$, either the simulation of M_L eventually accepts or the simulation of $M_{\overline{L}}$ eventually accepts. Thus, N halts on every input, which proves that L is decidable. □

Homework 2.3 *Using either a 2-adic or binary representation for numbers, describe multitape Turing machines that compute the following functions:*

1. $\lceil \log_2 n \rceil$;
2. $n!$;
3. n^2.


Track k
a_{k1}
$\uparrow a_{k2}$
a_{kd}
Track 2
$\uparrow a_{21}$
a_{22}
a_{2d}
Track 1
a_{11}
a_{12}
a_{1d}

2.3.2 Nondeterministic Turing Machines

A nondeterministic Turing machine allows for the possibility of more than one next move from a given configuration. If there is more than one next move, we do not specify which next move the machine makes, only that it chooses one such move. This is a crucial concept for our study of complexity. Unlike deterministic computing machines, we do not design nondeterministic machines to be executed. Rather, one should understand nondeterministic Turing machines to be a useful device for describing languages, and later, when we study complexity theory, for classifying computational problems.

Formally, a nondeterministic Turing machine M is the same as a multitape Turing machine, except that the transition function has the form

$$\delta : Q \times \Gamma^k \to \mathscr{P}(Q \times \Gamma^k \times \{L,R\}^k).$$

Recall that for any set A, $\mathscr{P}(A)$ denotes the power set of A. To keep the notation manageable, let us assume for the moment that $k = 1$, so that M is a nondeterministic single-tape Turing machine. Then, given state q and scanned symbol a, $\delta(q,a) = \{(p_1,b_1,D_1),\ldots,(p_n,b_n,D_n)\}$, for some $n \geq 1$. We interpret this to mean that in state q reading the symbol a, M may make any of the possible moves indicated by one of the triples (p_i,b_i,D_i), $1 \leq i \leq n$. Thus, for some $1 \leq i \leq n$, M will write the symbol b_i in the cell that is currently scanned, move in the direction D_i, and change to state p_i.

There are two additional subtle but important distinctions between deterministic and nondeterministic Turing machines: First, it is possible that $\delta(q,a) = \emptyset$, in which case there is no next move and the machine halts without accepting. Second, we do not include the state q_{reject}, so there are no *rejecting* computations.

A nondeterministic Turing machine M and input word w specify a *computation tree* as follows: The *root* of the tree is the initial configuration of M. The *children* of a node are the configurations that follow in one move. Note that a node is a *leaf* if there is no next move. A path from the root to a leaf is an accepting computation if and only if the leaf is an accepting configuration. In general, some computation paths might be infinite, some might be accepting, and others might halt in nonaccepting states. By definition, M accepts w if and only if there is an accepting computation of M on w (in which case, the other possibilities are irrelevant). Recall that M accepts the language $L(M) = \{w \mid M \text{ accepts } w\}$. Thus, $w \in L(M)$ if and only if there is an accepting computation of M on w. This is the important point to remember. It does not matter whether certain computations run forever or whether there is a computation that halts without accepting. All that matters is that at least one computation of M on w accepts, in which case $w \in L(M)$. Conversely, w does *not* belong to $L(M)$ if and only if every computation of M on w is *not* an accepting computation. Presumably, one cannot know whether this is so without executing all possible computations of M on w.

Now we prove that nondeterministic Turing machines are not more powerful than deterministic ones after all.

Theorem 2.3. *Every nondeterministic Turing machine has an equivalent deterministic Turing machine.*

The idea of the proof is a familiar one: Given a nondeterministic Turing machine N, design a deterministic Turing machine M that on an input word w builds the computation tree of N on input w and performs a standard tree-search algorithm that halts and accepts if and only if it finds a leaf that is accepting. Implementation is not difficult, but notice one possible pitfall. Suppose that M implements a depth-first search of the computation tree. Suppose that M has an infinite computation path that is to the left of some accepting computation. Then M will descend the infinite computation path, running forever, without ever finding the accepting computation. The solution to this difficulty is to implement a breadth-first search. Then the computation tree is searched one level at a time, so, if there is a leaf that is accepting, the simulation will find it. The proof to follow gives the details.

Proof. We assume that N is a single-tape nondeterministic Turing machine. The Turing machine M will have three tapes. Tape 1 contains the input word and is never changed. Tape 2 contains a copy of N's tape on some branch of its nondeterministic computation. Tape 3 records M's location in the computation tree.

Let b be the largest number of choices given by N's transition function. Assign each node an address that is a string in $\{1,2,\ldots,b\}$. A node has address $a_1 \ldots a_k$ if the node is at level $k+1$, a_1 is the a_1th child of the root, and for $i = 2,\ldots,k$, $a_1 \ldots a_i$ is the a_ith child of the node with address $a_1 \ldots a_{i-1}$. The address of the root is the empty word λ. Tape 3 will contain addresses.

The computation of M proceeds as follows:

1. Initially, tape 1 contains the input word w and tapes 2 and 3 are empty.
2. M copies tape 1 to tape 2.
3. On tape 2, M simulates N on w using the string on tape 3 to determine which choices to make. If tape 3 does not contain symbols for a choice or the symbol gives an invalid choice, then M aborts this branch and goes to step 4. If this simulation reaches an accepting configuration, then M accepts, but if it reaches a halting configuration that is nonaccepting, then M aborts this branch and goes to step 4.
4. M replaces the string on tape 3 with the lexicographically next string. Then M returns to step 2.

It is self-evident that M correctly simulates N. □

Corollary 2.2. *If every computation path of a nondeterministic Turing machine N halts on every input word, then there is a deterministic Turing machine M that decides the language $L(N)$.*

For the proof, notice that M must be able to determine when an input word w does not belong to $L(N)$. This occurs if and only if every computation path is nonaccepting. Since, by hypothesis, no computation paths are infinite, the breadth-first search will eventually search the entire computation tree and visit every leaf node, thereby gaining the information that it needs.

Example 2.3. Given a graph G, a *clique* H is a complete subgraph of G, meaning that every two vertices in H are connected by an edge. Consider the following nondeterministic algorithm that accepts the set

$$C = \{(G,k) \mid G \text{ is a graph, } k \text{ is a positive integer, and } G \text{ has a clique with } k \text{ vertices}\}:$$

The procedure on an input pair (G,k) nondeterministically chooses a subset of k vertices. Then the procedure tests whether there is an edge between every two vertices in the subset. If so, then the procedure accepts.

If the procedure accepts, then it has chosen a clique with k vertices, so the pair (G,k) belongs to C. Conversely, if $(G,k) \in C$, then G contains a clique H of size k. The computation path of the nondeterministic procedure that selects H is an accepting computation. Hence, the procedure has an accepting computation on input (G,k) if and only if (G,k) has a clique with k vertices.

We can easily implement this procedure on a nondeterministic Turing machine. However, in order to do so we would have to make decisions about how to represent graphs as words over a finite alphabet (because an input to a Turing machine is a word and not a graph). We postpone this discussion of representation to a later chapter. Then we would need to tend to the tedious details of Turing-machine programming. Without exercising this distraction, let us simply note that some nondeterministic Turing machine accepts a suitable representation of the set C.

A deterministic Turing machine to decide the same language, following the idea of the proof of Theorem 2.3, systematically searches all subsets of k vertices of G and accepts if and only if one of these is a complete subgraph.

2.4 Church's Thesis

Church's thesis states that every "effective computation," or "algorithm," can be programmed to run on a Turing machine. Every "computational device" can be simulated by some Turing machine. In 1936, the same year as Turing introduced the Turing machine [Tur36], Emil Post created the Post machine [Pos65], which he hoped would prove to be the "universal algorithm machine" sought after. Also that year, Alonzo Church [Chu36] developed the *lambda calculus*. Slightly *before* Turing invented his machine, Church proposed the thesis that every function that can be computed by an algorithm can be defined using his lambda calculus. That is, he identified effective computability, a heretofore imprecise notion, with a specific mathematical formulation. Then, independently, Turing posited the thesis that every algorithm can be programmed on a Turing machine. The Church–Post–Turing formulations are provably equivalent, so these theses express the same belief. Several factors contributed to the general acceptance of Church's thesis. Turing's paper contains a convincing analysis of the basic steps of calculation and he demonstrated how this analysis led to the definition of the Turing machine. That is one important factor. Another important factor is the simplicity and naturalness of the Turing-

machine model. A third factor is that the formulations of Church, Post, and Turing have been proven to be equivalent. This is no small matter, for each was independently invented from different considerations and perspectives. We should rightfully write the "Church–Turing" thesis, or even the "Church–Post–Turing" thesis, but for brevity we will continue to refer to "Church's" thesis.

Church's thesis cannot be "proven" because concepts such as "effective process" and "algorithms" are not part of any branch of mathematics. Yet evidence for the correctness of Church's thesis abounds. Two points are important to understand with regard to the notion of "algorithm" or "machine" as used here. The first is that every machine must have a "finite description." For example, even though there is no bound on the size of a Turing-machine tape, the description of the Turing machine as a tuple, including the transition function δ, has a finite length. The second is the notion of *determinancy*. For example, once a Turing machine is defined, for every input word, the transition function *determines* uniquely what sequence of IDs will occur. This never changes. Run the machine once or one hundred times, and the same sequence of IDs will occur.

In Chapter 3 we will learn about languages L that are Turing-machine-undecidable. (There exists no Turing machine that halts on every input and accepts L.) Using Church's thesis, we can understand results of this kind more broadly: There is no computational procedure (of any kind) that halts on every input and that for every input word w correctly determines whether w belongs to L.

In this course we are studying both computability theory and complexity theory. The basic model of computation for complexity theory is the multitape Turing machine. However, complexity measures should not depend too much on the choice of computational model. To this end, we introduce an *expanded version of Church's thesis.* Church's thesis states that every computational device can be simulated by a Turing machine. Our expanded Church's thesis is even stronger. It asserts that every computational device can be simulated by a multitape Turing machine with the simulation taking at most polynomial time. (That is, if M is some computational device, then there is a Turing machine T_M that simulates M such that programs run at most a polynomial number of additional steps on T_M than on M.)

This thesis is particularly fortunate because of another assertion known as Cobham's thesis (1964). Cobham's thesis asserts that computational problems can be feasibly computed on some computational device only if they can be computed in polynomial time. Truth be told, an n^{100}-time algorithm is not a useful algorithm. It is a remarkable phenomenon, though, that problems for which polynomial algorithms are found have such algorithms with small exponents and with small coefficients. Thus, combining the two theses, a problem can be feasibly computed only if it can be computed in polynomial time on some multitape Turing machine.

We are neglecting to describe the intellectual fervor that existed at the turn of the last century. What was in the air to cause brilliant scientists to converge on equivalent formulations of universal computing in 1936? This story is told extremely well in a collection of articles edited by Herken [Her94]. Alan Turing, if not the father of computer science, is frequently credited for being the father of theoretical computer science and artificial intelligence. His work in cryptography has been credited

in recent years for being instrumental in enabling the Allied forces to win the Second World War. The extraordinary story of his life and work is described in the biography by Hodges [Hod83].

The next section will provide further evidence for the correctness of Church's thesis.

2.5 RAMs

A *random access machine* (RAM) is a conceptual model of a digitial computer. A RAM contains registers that serve as the computer's memory and there is random access to the registers. The basic model, as a general model of computation, is due to Shepherdson and Sturgis [SS63]. The variation that we will describe here is due to Machtey and Young [MY78]. Random access machines are important models for the analysis of concrete algorithms [CR73].

For each finite alphabet Σ, there is a RAM for that alphabet. Thus, let us fix a finite alphabet $\Sigma = \{a_1, \ldots, a_k\}$ with $k > 1$ letters. The RAM consists of a potentially infinite set of registers $R1, R2, \ldots$, each of which can store any word of Σ^*. Any given program uses only the finite set of registers that it specifically names, and any given computation that halts uses only a finite set of words of Σ^*. (Thus, any such computation needs only a finite amount of "hardware.") RAM instructions have available an infinite set of *line names* $N0, N1, \ldots$.

RAM instructions are of the following seven types:

1_j X **add**$_j$ Y,
2 X **del** Y,
3 X **clr** Y,
4 X $Y \leftarrow Z$,
5 X **jmp** X',
6_j X Y **jmp**$_j$ X',
7 **continue**,

where X is either a line name or nothing, Y and Z are register names, X' is a line name followed by an "a" or a "b" (e.g., $N6a$), and $1 \leq j \leq k$.

Instructions of types 1 through 4 affect the contents of registers in obvious ways: Type 1_j adds a_j to the right end of the word in register Y. Type 2 deletes the leftmost letter of the word in Y, if there is one. Type 3 changes the word in Y to λ. Type 4 copies the word in Z into Y and leaves Z unchanged.

Types 5 and 6 are jump instructions. Normally instructions are executed in the order in which they are written. When a **jmp** Nia is executed, the next instruction to be executed is the closest instruction above bearing the line name Ni; **jmp** Nib goes to the closest instruction below bearing line name Ni. Several different instructions in a program may have the same line name. Type 6_j are conditional jumps that are performed only if the first letter of the word in Y is a_j. Type 7 are "no-ops" instructions. Table 2.1 summarizes the actions of some of these instructions.