Chapter 12

Limits of Algorithmic Computation

Having talked about what Turing machines can do, we now look at what they cannot do. Although Turing's thesis leads us to believe that there are few limitations to the power of a Turing machine, we have claimed on several occasions that there could not exist any algorithms for the solution of certain problems. Now we make more explicit what we mean by this claim. Some of the results came about quite simply; if a language is nonrecursive, then by definition there is no membership algorithm for it. If this were all there was to this issue, it would not be very interesting; nonrecursive languages have little practical value. But the problem goes deeper. For example, we have stated (but not yet proved) that there exists no algorithm to determine whether a context-free grammar is unambiguous. This question is clearly of practical significance in the study of programming languages.

We first define the concept of **decidability** and **computability** to pin down what we mean when we say that something cannot be done by a Turing machine. We then look at several classical problems of this type, among them the well-known halting problem for Turing machines. From this follow a number of related problems for Turing machines and recursively enumerable languages. After this, we look at some questions relating to context-free languages. Here we find quite a few important problems for which, unfortunately, there are no algorithms.

12.1 Some Problems That Cannot Be Solved by Turing Machines

The argument that the power of mechanical computations is limited is not surprising. Intuitively we know that many vague and speculative questions require special insight and reasoning well beyond the capacity of any computer that we can now construct or even plausibly foresee. What is more interesting to computer scientists is that there are questions that can be clearly and simply stated, with an apparent possibility of an algorithmic solution, but which are known to be unsolvable by any computer.

Computability and Decidability

In Definition 9.4, we stated that a function f on a certain domain is said to be computable if there exists a Turing machine that computes the value of f for all arguments in its domain. A function is uncomputable if no such Turing machine exists. There may be a Turing machine that can compute f on part of its domain, but we call the function computable only if there is a Turing machine that computes the function on the whole of its domain. We see from this that, when we classify a function as computable or not computable, we must be clear on what its domain is.

Our concern here will be the somewhat simplified setting where the result of a computation is a simple "yes" or "no." In this case, we talk about a problem being **decidable** or **undecidable**. By a *problem* we will understand a set of related statements, each of which must be either true or false. For example, we consider the statement "For a context-free grammar G, the language $L(G)$ is ambiguous." For some G this is true, for others it is false, but clearly we must have one or the other. The problem is to decide whether the statement is true for any G we are given. Again, there is an underlying domain, the set of all context-free grammars. We say that a problem is decidable if there exists a Turing machine that gives the correct answer for every statement in the domain of the problem.

When we state decidability or undecidability results, we must always know what the domain is, because this may affect the conclusion. The problem may be decidable on some domain but not on another. Specifically, a single instance of a problem is always decidable, since the answer is either true or false. In the first case, a Turing machine that always answers "true" gives the correct answer, while in the second case one that always answers "false" is appropriate. This may seem like a facetious answer, but it emphasizes an important point. The fact that we do not know what the correct answer is makes no difference, what matters is that there exists some Turing machine that does give the correct response.

The Turing Machine Halting Problem

We begin with some problems that have some historical significance and that at the same time give us a starting point for developing later results. The best-known of these is the Turing machine **halting problem**. Simply stated, the problem is: given the description of a Turing machine M and an input w, does M, when started in the initial configuration q_0w, perform a computation that eventually halts? Using an abbreviated way of talking about the problem, we ask whether M applied to w, or simply (M, w), halts or does not halt. The domain of this problem is to be taken as the set of all Turing machines and all w; that is, we are looking for a single Turing machine that, given the description of an arbitrary M and w, will predict whether or not the computation of M applied to w will halt.

We cannot find the answer by simulating the action of M on w, say by performing it on a universal Turing machine, because there is no limit on the length of the computation. If M enters an infinite loop, then no matter how long we wait, we can never be sure that M is in fact in a loop. It may simply be a case of a very long computation. What we need is an algorithm that can determine the correct answer for any M and w by performing some analysis on the machine's description and the input. But as we now show, no such algorithm exists.

For subsequent discussion, it is convenient to have a precise idea of what we mean by the halting problem; for this reason, we make a specific definition of what we stated somewhat loosely above.

Definition 12.1

Let w_M be a string that describes a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$, and let w be a string in M's alphabet. We will assume that w_M and w are encoded as a string of 0's and 1's, as suggested in Section 10.4. A solution of the halting problem is a Turing machine H, which for any w_M and w performs the computation

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

if M applied to w halts, and

$$q_0 w_M w \overset{*}{\vdash} y_1 q_n y_2,$$

if M applied to w does not halt. Here q_y and q_n are both final states of H.

Theorem 12.1 There does not exist any Turing machine H that behaves as required by Definition 12.1. The halting problem is therefore undecidable.

Proof: We assume the contrary, namely, that there exists an algorithm, and consequently some Turing machine H, that solves the halting problem. The input to H will be the string $w_M w$. The requirement is then that, given any $w_M w$, the Turing machine H will halt with either a yes or no answer. We achieve this by asking that H halt in one of two corresponding final states, say, q_y or q_n. The situation can be visualized by a block diagram like Figure 12.1. The intent of this diagram is to indicate that, if M is started in state q_0 with input $w_M w$, it will eventually halt in state q_y or q_n. As required by Definition 12.1, we want H to operate according to the following rules:

$$q_0 w_M w \overset{*}{\vdash}_H x_1 q_y x_2,$$

if M applied to w halts, and

$$q_0 w_M w \overset{*}{\vdash}_H y_1 q_n y_2,$$

if M applied to w does not halt.

Next, we modify H to produce a Turing machine H' with the structure shown in Figure 12.2. With the added states in Figure 12.2 we want to convey that the transitions between state q_y and the new states q_a and q_b are to be made, regardless of the tape symbol, in such a way that the tape remains unchanged. The way this is done is straightforward. Comparing H and H' we see that, in situations where H reaches q_y and halts, the modified machine H' will enter an infinite loop. Formally, the action of H' is described by

$$q_0 w_M w \overset{*}{\vdash}_{H'} \infty,$$

if M applied to w halts, and

$$q_0 w_M w \overset{*}{\vdash}_{H'} y_1 q_n y_2,$$

if M applied to w does not halt.

Figure 12.1

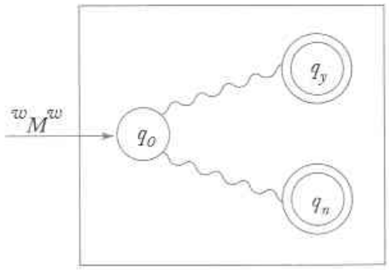
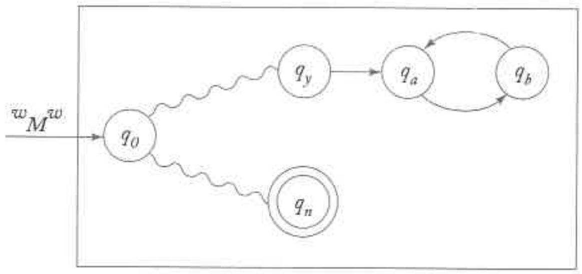


Figure 12.2



From H' we construct another Turing machine $\widehat{H}$. This new machine takes as input w_M and copies it, ending in its initial state q_0. After that, it behaves exactly like H'. Then the action of $\widehat{H}$ is such that

$$q_0 w_M \overset{*}{\vdash}_{\widehat{H}} q_0 w_M w_M \overset{*}{\vdash}_{\widehat{H}} \infty,$$

if M applied to w_M halts, and

$$q_0 w_M \overset{*}{\vdash}_{\widehat{H}} q_0 w_M w_M \overset{*}{\vdash}_{\widehat{H}} y_1 q_n y_2,$$

if M applied to w_M does not halt.

Now $\widehat{H}$ is a Turing machine, so it has a description in $\{0,1\}^*$, say, $\widehat{w}$. This string, in addition to being the description of $\widehat{H}$, also can be used as input string. We can therefore legitimately ask what would happen if $\widehat{H}$ is applied to $\widehat{w}$. From the above, identifying M with $\widehat{H}$, we get

$$q_0 \widehat{w} \overset{*}{\vdash}_{\widehat{H}} \infty,$$

if $\widehat{H}$ applied to $\widehat{w}$ halts, and

$$q_0 \widehat{w} \overset{*}{\vdash}_{\widehat{H}} y_1 q_n y_2,$$

if $\widehat{H}$ applied to $\widehat{w}$ does not halt. This is clearly nonsense. The contradiction tells us that our assumption of the existence of H, and hence the assumption of the decidability of the halting problem, must be false. ■

One may object to Definition 12.1, since we required that, to solve the halting problem, H had to start and end in very specific configurations. It is, however, not hard to see that these somewhat arbitrarily chosen conditions play only a minor role in the argument, and that essentially the same reasoning could be used with any other starting and ending configurations. We have tied the problem to a specific definition for the sake of the discussion, but this does not affect the conclusion.

It is important to keep in mind what Theorem 12.1 says. It does not preclude solving the halting problem for specific cases; often we can tell by an analysis of M and w whether or not the Turing machine will halt. What the theorem says is that this cannot always be done; there is no algorithm that can make a correct decision for all w_M and w.

The arguments for proving Theorem 12.1 were given because they are classical and of historical interest. The conclusion of the theorem is actually implied in previous results as the following argument shows.

Theorem 12.2 If the halting problem were decidable, then every recursively enumerable language would be recursive. Consequently, the halting problem is undecidable.

Proof: To see this, let L be a recursively enumerable language on Σ, and let M be a Turing machine that accepts L. Let H be the Turing machine that solves the halting problem. We construct from this the following procedure:

1. Apply H to $w_M w$. If H says "no," then by definition w is not in L.
2. If H says "yes," then apply M to w. But M must halt, so it will eventually tell us whether w is in L or not.

This constitutes a membership algorithm, making L recursive. But we already know that there are recursively enumerable languages that are not recursive. The contradiction implies that H cannot exist, that is, that the halting problem is undecidable. ■

The simplicity with which the halting problem can be obtained from Theorem 11.5 is a consequence of the fact that the halting problem and the membership problem for recursively enumerable languages are nearly identical. The only difference is that in the halting problem we do not distinguish between halting in a final and nonfinal state, whereas in the membership problem we do. The proofs of Theorems 11.5 (via Theorem 11.3) and 12.1 are closely related, both being a version of diagonalization.

Reducing One Undecidable Problem to Another

The above argument, connecting the halting problem to the membership problem, illustrates the very important technique of reduction. We say that a problem A is **reduced** to a problem B if the decidability of A follows from the decidability of B. Then, if we know that A is undecidable, we can conclude that B is also undecidable. Let us do a few examples to illustrate this idea.

Example 12.1 The **state-entry problem** is as follows. Given any Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ and any $q \in Q$, $w \in \Sigma^+$, decide whether or not the state q is ever entered when M is applied to w. This problem is undecidable.

To reduce the halting problem to the state-entry problem, suppose that we have an algorithm A that solves the state-entry problem. We could then use it to solve the halting problem. For example, given any M and w, we first modify M to get $\widehat{M}$ in such a way that $\widehat{M}$ halts in state q if and only if M halts. We can do this simply by looking at the transition function δ of M. If M halts, it does so because some $\delta(q_i, a)$ is undefined. To get $\widehat{M}$, we change every such undefined δ to

$$\delta(q_i, a) = (q, a, R),$$

where q is a final state. We apply the state-entry algorithm A to $\left(\widehat{M}, q, w\right)$. If A answers yes, that is, the state q is entered, then (M, w) halts. If A says no, then (M, w) does not halt.

Thus, the assumption that the state-entry problem is decidable gives us an algorithm for the halting problem. Because the halting problem is undecidable, the state-entry problem must also be undecidable. ■

Example 12.2 The **blank-tape halting problem** is another problem to which the halting problem can be reduced. Given a Turing machine M, determine whether or not M halts if started with a blank tape. This is undecidable.

To show how this reduction is accomplished, assume that we are given some M and some w. We first construct from M a new machine M_w that starts with a blank tape, writes w on it, then positions itself in a configuration q_0w. After that, M_w acts like M. Clearly M_w will halt on a blank tape if and only if M halts on w.

Suppose now that the blank-tape halting problem were decidable. Given any (M, w), we first construct M_w, then apply the blank-tape halting problem algorithm to it. The conclusion tells us whether M applied to w will halt. Since this can be done for any M and w, an algorithm for the blank-tape halting problem can be converted into an algorithm for the halting problem. Since the latter is known to be undecidable, the same must be true for the blank-tape halting problem. ■

The construction in the arguments of these two examples illustrates an approach common in establishing undecidability results. A block diagram often helps us visualize the process. The construction in Example 12.2 is summarized in Figure 12.3. In that diagram, we first use an algorithm that transforms (M, w) into M_w; such an algorithm clearly exists. Next, we use the algorithm for solving the blank-tape halting problem, which we assume exists. Putting the two together yields an algorithm for the halting problem. But this is impossible, and we can conclude that A cannot exist.

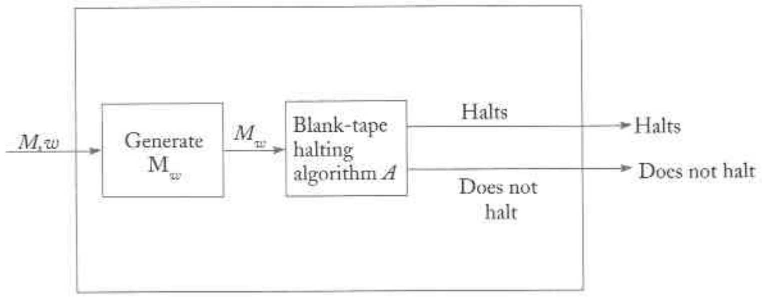


Figure 12.3 Algorithm for halting problem.

A decision problem is effectively a function with a range $\{0, 1\}$, that is, a true or false answer. We can look also at more general functions to see if they are computable; to do so, we follow the established method and reduce the halting problem (or any other problem known to be undecidable) to the problem of computing the function in question. Because of Turing's thesis, we expect that functions encountered in practical circumstances will be computable, so for examples of uncomputable functions we must look a little further. Most examples of uncomputable functions are associated with attempts to predict the behavior of Turing machines.

Example 12.3 Let $\Gamma = \{0, 1, \square\}$. Consider the function $f(n)$ whose value is the maximum number of moves that can be made by any n-state Turing machine that halts when started with a blank tape. This function, as it turns out, is not computable.

Before we set out to demonstrate this, let us make sure that $f(n)$ is defined for all n. Notice first that there are only a finite number of Turing machines with n states. This is because Q and Γ are finite, so δ has a finite domain and range. This in turn implies that there are only a finite number of different δ's and therefore a finite number of different n-state Turing machines.

Of all of the n-state machines, there are some that always halt, for example machines that have only final states and therefore make no moves. Some of the n-state machines will not halt when started with a blank tape, but they do not enter the definition of f. Every machine that does halt will execute a certain number of moves; of these, we take the largest to give $f(n)$.

Take any Turing machine M and positive number m. It is easy to modify M to produce $\widehat{M}$ in such a way that the latter will always halt with one of two answers: M applied to a blank tape halts in no more than m moves, or M applied to a blank tape makes more than m moves. All we have to do for this is to have $\widehat{M}$ count its moves and terminate when this count exceeds m. Assume now that $f(n)$ is computable by some Turing

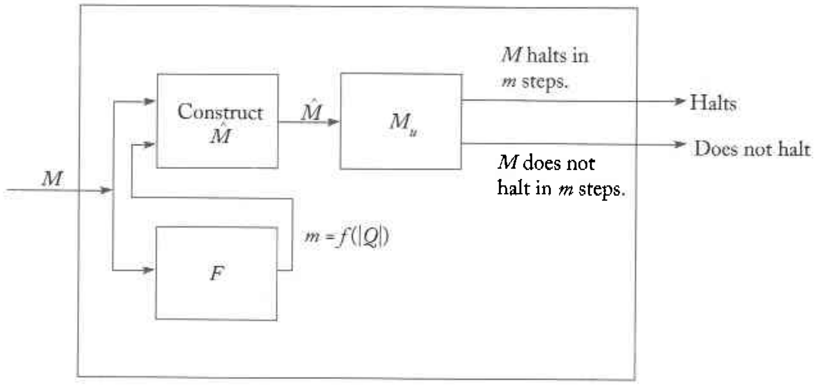


Figure 12.4 Algorithm for blank-tape halting problem.

machine F. We can then put $\widehat{M}$ and F together as shown in Figure 12.4. First we compute $f(|Q|)$, where Q is the state set of M. This tells us the maximum number of moves that M can make if it is to halt. The value we get is then used as m to construct $\widehat{M}$ as outlined, and a description of $\widehat{M}$ is given to a universal Turing machine for execution. This tells us whether M applied to a blank tape halts or does not halt in less than $f(|Q|)$ steps. If we find that M applied to a blank tape makes more than $f(|Q|)$ moves, then because of the definition of f, the implication is that M never halts. Thus we have a solution to the blank tape halting problem. The impossibility of the conclusion forces us to accept that f is not computable. ■

Exercises

1. Describe in detail how H in Theorem 12.1 can be modified to produce H'.

2. Suppose we change Definition 12.1 to require that $q_0 w_M w \overset{*}{\vdash} q_y w$ or $q_0 w_M w \overset{*}{\vdash} q_n w$, depending on whether M applied to w halts or not. Reexamine the proof of Theorem 12.1 to show that this difference in the definition does not affect the proof in any significant way.

3. Show that the following problem is undecidable. Given any Turing machine M, $a \in \Gamma$, and $w \in \Sigma^+$, determine whether or not the symbol a is ever written when M is applied to w. ✹

4. In the general halting problem, we ask for an algorithm that gives the correct answer for any M and w. We can relax this generality, for example, by looking for an algorithm that works for all M but only a single w. We say that such a problem is decidable if for every w there exists a (possibly different) algorithm that determines whether or not (M, w) halts. Show that even in this restricted setting the problem is undecidable.

5. Show that there is no algorithm to decide whether or not an arbitrary Turing machine halts on all input.

6. Consider the question: "Does a Turing machine in the course of a computation revisit the starting cell (i.e. the cell under the read-write head at the beginning of the computation)?" Is this a decidable question?

7. Show that there is no algorithm for deciding if any two Turing machines M_1 and M_2 accept the same language. ●

8. How is the conclusion of Exercise 7 affected if M_2 is a finite automaton?

9. Is the halting problem solvable for deterministic pushdown automata; that is, given a pda as in Definition 7.2, can we always predict whether or not the automaton will halt on input w?

10. Let M be any Turing machine and x and y two possible instantaneous descriptions of it. Show that the problem of determining whether or not

$$x \overset{*}{\vdash}_M y$$

is undecidable. ●

11. In Example 12.3, give the values of $f(1)$ and $f(2)$.

12. Show that the problem of determining whether a Turing machine halts on any input is undecidable.

13. Let B be the set of all Turing machines that halt when started with a blank tape. Show that this set is recursively enumerable, but not recursive. ●

14. Consider the set of all n-state Turing machines with tape alphabet $\Gamma = \{0, 1, \square\}$. Give an expression for $m(n)$, the number of distinct Turing machines with this Γ.

15. Let $\Gamma = \{0, 1, \square\}$ and let $b(n)$ be the maximum number of tape cells examined by any n-state Turing machine that halts when started with a blank tape. Show that $b(n)$ is not computable.

16. Determine whether or not the following statement is true: Any problem whose domain is finite is decidable. ●

12.2 Undecidable Problems for Recursively Enumerable Languages

We have determined that there is no membership algorithm for recursively enumerable languages. The lack of an algorithm to decide on some property is not an exceptional state of affairs for recursively enumerable languages, but rather is the general rule. As we now show, there is little we can say about these languages. Recursively enumerable languages are so general that, in essence, any question we ask about them is undecidable. Invariably, when we ask a question about recursively enumerable languages, we find that there is some way of reducing the halting problem to this question. We give here some examples to show how this is done and from these examples derive an indication of the general situation.

Theorem 12.3 Let G be an unrestricted grammar. Then the problem of determining whether or not

$$L(G) = \varnothing$$

is undecidable.

Proof: We will reduce the membership problem for recursively enumerable languages to this problem. Suppose we are given a Turing machine M and some string w. We can modify M as follows. M first saves its input on some special part of its tape. Then, whenever it enters a final state, it checks its saved input and accepts it if and only if it is w. We can do this by changing δ in a simple way, creating for each w a machine M_w such that

$$L(M_w) = L(M) \cap \{w\}.$$

Using Theorem 11.7, we then construct a corresponding grammar G_w. Clearly, the construction leading from M and w to G_w can always be done. Equally clear is that $L(G_w)$ is nonempty if and only if $w \in L(M)$.

Assume now that there exists an algorithm A for deciding whether or not $L(G) = \varnothing$. If we let T denote an algorithm by which we generate G_w, then we can put T and A together as shown in Figure 12.5. Figure 12.5 is a Turing machine which for any M and w tells us whether or not w is in $L(M)$. If such a Turing machine existed, we would have a membership algorithm for any recursively enumerable language, in direct contradiction to a previously established result. We conclude therefore that the stated problem "$L(G) = \varnothing$" is not decidable. ■

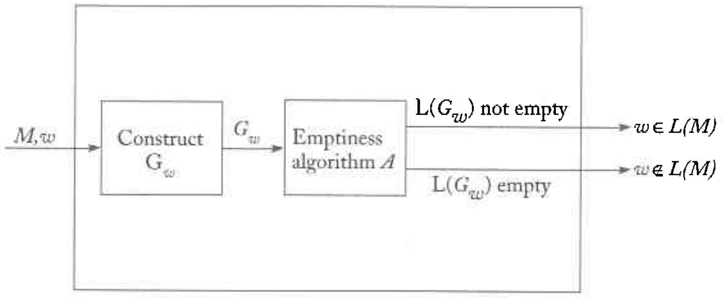


Figure 12.5 Membership algorithm.

Theorem 12.4 Let M be any Turing machine. Then the question of whether or not $L(M)$ is finite is undecidable.

Proof: Consider the halting problem (M, w). From M we construct another Turing machine $\widehat{M}$ that does the following. First, the halting states of M are changed so that if any one is reached, all input is accepted by $\widehat{M}$. This can be done by having any halting configuration go to a final state. Second, the original machine is modified so that the new machine $\widehat{M}$ first generates w on its tape, then performs the same computations as M, using the newly created w and some otherwise unused space. In other words, the moves made by $\widehat{M}$ after it has written w on its tape are the same as would have been made by M had it started in the original configuration $q_0 w$. If M halts in any configuration, then $\widehat{M}$ will halt in a final state.

Therefore, if (M, w) halts, $\widehat{M}$ will reach a final state for all input. If (M, w) does not halt, then $\widehat{M}$ will not halt either and so will accept nothing. In other words, $\widehat{M}$ either accepts the infinite language Σ^+ or the finite language $\varnothing$.

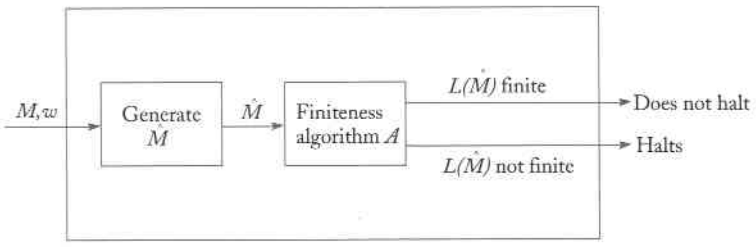
If we now assume the existence of an algorithm A that tells us whether or not $L\left(\widehat{M}\right)$ is finite, we can construct a solution to the halting problem as shown in Figure 12.6. Therefore no algorithm for deciding whether or not $L(M)$ is finite can exist. ■

Notice that in the proof of Theorem 12.4, the specific nature of the question asked, namely "Is $L(M)$ finite?", is immaterial. We can change the nature of the problem without significantly affecting the argument.

Example 12.4 Show that for an arbitrary Turing machine M with $\Sigma = \{a, b\}$, the problem "$L(M)$ contains two different strings of the same length" is undecidable.

To show this, we use exactly the same approach as in Theorem 12.4, except that when $\widehat{M}$ reaches a halting configuration, it will be modified to accept the two strings a and b. For this, the initial input is saved and at the

Figure 12.6



end of the computation compared with a and b, accepting only these two strings. Thus, if (M, w) halts, $\widehat{M}$ will accept two strings of equal length, otherwise $\widehat{M}$ will accept nothing. The rest of the argument then proceeds as in Theorem 12.4.

■

In exactly the same manner, we can substitute other questions such as "Does $L(M)$ contain any string of length five?" or "Is $L(M)$ regular?" without affecting the argument essentially. These questions, as well as similar questions, are all undecidable. A general result formalizing this is known as **Rice's theorem**. This theorem states that any nontrivial property of a recursively enumerable language is undecidable. The adjective "nontrivial" refers to a property possessed by some but not all recursively enumerable languages. A precise statement and a proof of Rice's theorem can be found in Hopcroft and Ullman (1979).

EXERCISES

1. Show in detail how the machine $\widehat{M}$ in Theorem 12.4 is constructed.
2. Show that the two problems mentioned at the end of the preceding section, namely

   (a) $L(M)$ contains any string of length five,

   (b) $L(M)$ is regular,

   are undecidable.
3. Let M_1 and M_2 be arbitrary Turing machines. Show that the problem "$L(M_1) \subseteq L(M_2)$" is undecidable. ●
4. Let G be any unrestricted grammar. Does there exist an algorithm for determining whether or not $L(G)^R$ is recursively enumerable?
5. Let G be any unrestricted grammar. Does there exist an algorithm for determining whether or not $L(G) = L(G)^R$?
6. Let G_1 be any unrestricted grammar, and G_2 any regular grammar. Show that the problem

   $$L(G_1) \cap L(G_2) = \varnothing$$

   is undecidable. ●
7. Show that the question in Exercise 6 is undecidable for any fixed G_2, as long as $L(G_2)$ is not empty.
8. For an unrestricted grammar G, show that the question "Is $L(G) = L(G)^*$?" is undecidable. Argue (a) from Rice's Theorem and (b) from first principles. ●

12.3 The Post Correspondence Problem

The undecidability of the halting problem has many consequences of practical interest, particularly in the area of context-free languages. But in many instances it is cumbersome to work with the halting problem directly, and it is convenient to establish some intermediate results that bridge the gap between the halting problem and other problems. These intermediate results follow from the undecidability of the halting problem, but are more closely related to the problems we want to study and therefore make the arguments easier. One such intermediate result is the **Post correspondence problem**.

The Post correspondence problem can be stated as follows. Given two sequences of n strings on some alphabet Σ, say

$$A = w_1, w_2, ..., w_n$$

and

$$B = v_1, v_2, ..., v_n$$

we say that there exists a Post correspondence solution (PC-solution) for pair (A, B) if there is a nonempty sequence of integers $i, j, ..., k$, such that

$$w_i w_j \cdots w_k = v_i v_j \cdots v_k.$$

The Post correspondence problem is to devise an algorithm that will tell us, for any (A, B), whether or not there exists a PC-solution.

Example 12.5 Let $\Sigma = \{0, 1\}$ and take A and B as

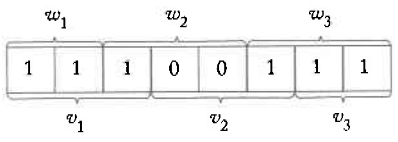
$$w_1 = 11, w_2 = 100, w_3 = 111$$
$$v_1 = 111, v_2 = 001, v_3 = 11$$

For this case, there exists a PC-solution as Figure 12.7 shows.

If we take

$$w_1 = 00, w_2 = 001, w_3 = 1000$$
$$v_1 = 0, v_2 = 11, v_3 = 011$$

Figure 12.7



there cannot be any PC-solution simply because any string composed of elements of A will be longer than the corresponding string from B.

■

In specific instances we may be able to show by explicit construction that a pair (A, B) permits a PC-solution, or we may be able to argue, as we did above, that no such solution can exist. But in general, there is no algorithm for deciding this question under all circumstances. The Post correspondence problem is therefore undecidable.

To show this is a somewhat lengthy process. For the sake of clarity, we break it into two parts. In the first part, we introduce the **modified Post correspondence problem**. We say that the pair (A, B) has a modified Post correspondence solution (MPC-solution) if there exists a sequence of integers $i, j, ..., k$, such that

$$w_1 w_i w_j \cdots w_k = v_1 v_i v_j \cdots v_k.$$

In the modified Post correspondence problem, the first elements of the sequences A and B play a special role. An MPC solution must start with w_1 on the left side and with v_1 on the right side. Note that if there exists an MPC-solution, then there is also a PC-solution, but the converse is not true.

The modified Post correspondence problem is to devise an algorithm for deciding if an arbitrary pair (A, B) admits an MPC-solution. This problem is also undecidable. We will demonstrate the undecidability of the modified Post correspondence problem by reducing a known undecidable problem, the membership problem for recursively enumerable languages, to it. To this end, we introduce the following construction. Suppose we are given an unrestricted grammar $G = (V, T, S, P)$ and a target string w. With these, we create the pair (A, B) as shown in Figure 12.8. In Figure 12.8, the string $FS \Rightarrow$ is to be taken as w_1 and the string F as v_1. The order of the rest of the strings is immaterial.

We want to claim eventually that $w \in L(G)$ if and only if the sets A and B constructed in this way have an MPC-solution. Since this is perhaps not immediately obvious, let us illustrate it with a simple example.

Example 12.6 Let $G = (\{A, B, C\}, \{a, b, c, \}, S, P)$ with productions

$$S \to aABb|Bbb,$$
$$Bb \to C,$$
$$AC \to aac,$$

Figure 12.8

A	B	
$FS \Rightarrow$	F	F is a symbol not in $V \cup T$
a	a	for every $a \in T$
V_i	V_i	for every $V_i \in V$
E	$\Rightarrow wE$	E is a symbol not in $V \cup T$
y_i	x_i	for every $x_i \to y_i$ in P
$\Rightarrow$	$\Rightarrow$	

and take $w = aaac$. The sequences A and B obtained from the suggested construction are given in Figure 12.9. The string $w = aaac$ is in $L(G)$ and has a derivation

$$S \Rightarrow aABb \Rightarrow aAC \Rightarrow aaac.$$

How this derivation is paralleled by an MPC-solution with the constructed sets can be seen in Figure 12.10, where the first two steps in the derivation are shown. The integers above and below the derivation string show the indices for w and v, respectively, used to create the string.

Figure 12.9

i	w_i	v_i
1	$FS \Rightarrow$	F
2	a	a
3	b	b
4	c	c
5	A	A
6	B	B
7	C	C
8	S	S
9	E	$\Rightarrow aaacE$
10	$aABb$	S
11	Bbb	S
12	C	Bb
13	aac	AC
14	$\Rightarrow$	$\Rightarrow$

Figure 12.10

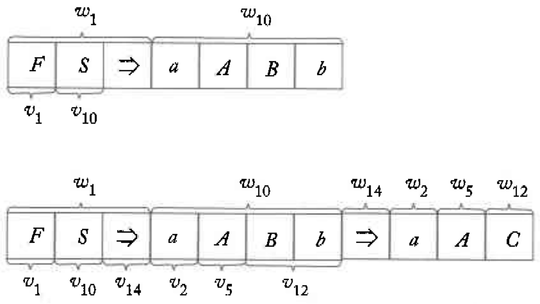
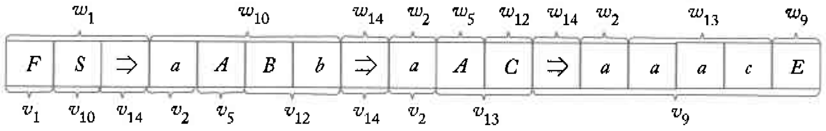

w_1
w_{10}
F S ⇒ a A B b
v_1 v_{10}
w_1 w_{10} w_{14} w_2 w_5 w_{12}
F S ⇒ a A B b ⇒ a A C
v_1 v_{10} v_{14} v_2 v_5 v_{12}

Figure 12.11


w_1 w_{10} w_{14} w_2 w_5 w_{12} w_{14} w_2 w_{13} w_9
F S ⇒ a A B b ⇒ a A C ⇒ a a a c E
v_1 v_{10} v_{14} v_2 v_5 v_{12} v_{14} v_2 v_{13} v_9

Examine Figure 12.10 carefully to see what is happening. We want to construct an MPC-solution, so we must start with w_1, that is, $FS \Rightarrow$. This string contains S, so to match it we have to use v_{10} or v_{11}. In this instance, we use v_{10}; this brings in w_{10}, leading us to the second string in the partial derivation. Looking at several more steps, we see that the string $w_1 w_i w_j \ldots$ is always longer than the corresponding string $v_1 v_i v_j \ldots$, and that the first is exactly one step ahead in the derivation. The only exception is the last step, where w_9 must be applied to let the v-string catch up. The complete MPC-solution is shown in Figure 12.11. The construction, together with the example, indicate the lines along which the next result is established. ■

Theorem 12.5

Let $G = (V, T, S, P)$ be any unrestricted grammar, with w any string in T^+. Let (A, B) be the correspondence pair constructed from G and w be the process exhibited in Figure 12.8. Then the pair (A, B) permits an MPC-solution if and only if $w \in L(G)$.

Proof: The proof involves a formal inductive argument based on the outlined reasoning. We will omit the details. ■

With this result, we can reduce the membership problem for recursively enumerable languages to the modified Post correspondence problem and thereby demonstrate the undecidability of the latter.

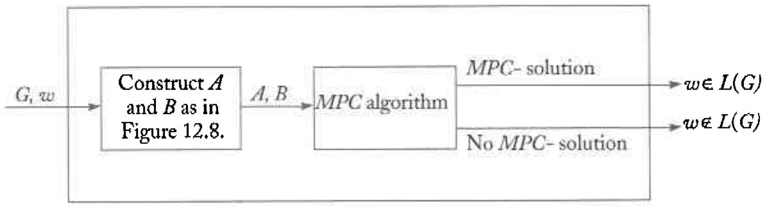


Figure 12.12 Membership algorithm.

Theorem 12.6 The modified Post correspondence problem is undecidable.

Proof: Given any unrestricted grammar $G = (V, T, S, P)$ and $w \in T^+$, we construct the sets A and B as suggested above. By Theorem 12.5, the pair (A, B) has an MPC-solution if and only if $w \in L(G)$.

Suppose now we assume that the modified Post correspondence problem is decidable. We can then construct an algorithm for the membership problem of G as sketched in Figure 12.12. An algorithm for constructing A from B from G and w clearly exists, but a membership algorithm for G and w does not. We must therefore conclude that there cannot be any algorithm for deciding the modified Post correspondence problem. ■

With this preliminary work, we are now ready to prove the Post correspondence problem in its original form.

Theorem 12.7 The Post correspondence problem is undecidable.

Proof: We argue that if the Post correspondence problem were decidable, the modified Post correspondence problem would be decidable.

Suppose we are given sequences $A = w_1, w_2, ..., w_n$ and $B = v_1, v_2, ..., v_n$ on some alphabet Σ. We then introduce new symbols ¢ and § and the new sequences

$$C = y_0, y_1, ..., y_{n+1},$$
$$D = z_0, z_1, ..., z_{n+1},$$

defined as follows. For $i = 1, 2, ..., n$

$$y_i = w_{i1}¢w_{i2}¢ \cdots w_{im_i}¢,$$
$$z_i = ¢v_{i1}¢v_{i2}¢ \cdots v_{ir_i},$$

where w_{ij} and v_{ij} denote the j^{th} letter of w_i and v_i, respectively, and $m_i = |w_i|, r_i = |v_i|$. In words, y_i is created from w_i by appending ¢ to each

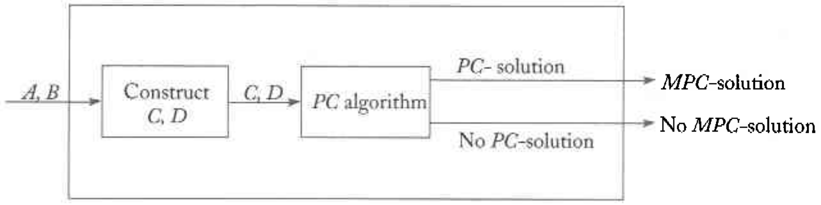


Figure 12.13
MPC algorithm.

character, while z_i is obtained by prefixing each character of v_i with ¢. To complete the definition of C and D, we take

$$\begin{aligned} y_0 &= ¢y_1, \\ y_{n+1} &= §, \\ z_0 &= z_1, \\ z_{n+1} &= ¢§. \end{aligned}$$

Consider now the pair (C, D), and suppose it has a PC-solution. Because of the placement of ¢ and §, such a solution must have y_0 on the left and y_{n+1} on the right and so must look like

$$¢w_{11}¢w_{12}\cdots¢w_{j1}¢\cdots¢w_{k1}\cdots¢§ = ¢v_{11}¢v_{12}\cdots¢v_{j1}¢\cdots¢v_{k1}\cdots¢§.$$

Ignoring the characters ¢ and §, we see that this implies

$$w_1w_j\cdots w_k = v_1v_j\cdots v_k,$$

so that the pair (A, B) permits an MPC-solution.

We can turn the argument around to show that if there is an MPC-solution for (A, B) then there is a PC-solution for the pair (C, D).

Assume now that the Post correspondence problem is decidable. We can then construct the machine shown in Figure 12.13. This machine clearly decides the modified Post correspondence problem. But the modified Post correspondence problem is undecidable, consequently, we cannot have an algorithm for deciding the Post correspondence problem. ∎

Exercises

1. Let $A = \{001, 0011, 11, 101\}$ and $B = \{01, 111, 111, 010\}$. Does the pair (A, B) have a PC-solution? Does it have an MPC-solution?

2. Provide the details of the proof of Theorem 12.5.

3. Show that for $|\Sigma| = 1$, the Post correspondence problem is decidable, that is, there is an algorithm that can decide whether or not (A, B) has a PC-solution for any given (A, B) on a single-letter alphabet. ●

4. Suppose we restrict the domain of the Post correspondence problem to include only alphabets with exactly two symbols. Is the resulting correspondence problem decidable?

5. Show that the following modifications of the Post correspondence problem are undecidable.

   (a) There is an MPC-solution if there is a sequence of integers such that $w_i w_j \cdots w_k w_1 = v_i v_j \cdots v_k v_1$. ●

   (b) There is an MPC-solution if there is a sequence of integers such that $w_1 w_2 w_i w_j \cdots w_k = v_1 v_2 v_i v_j \cdots v_k$.

6. The correspondence pair (A, B) is said to have an *even* PC-solution if and only if there exists a nonempty sequence of even integers $i, j, ...k$ such that $w_i w_j \cdots w_k = v_i v_j \cdots v_k$. Show that the problem of deciding whether or not an arbitrary pair (A, B) has an even PC-solution is undecidable.

12.4 Undecidable Problems for Context-Free Languages

The Post correspondence problem is a convenient tool for studying undecidable questions for context-free languages. We illustrate this with a few selected results.

Theorem 12.8 There exists no algorithm for deciding whether any given context-free grammar is ambiguous.

Proof: Consider two sequences of strings $A = (w_1, w_2, ..., w_n)$ and $B = (v_1, v_2, ...v_n)$ over some alphabet Σ. Choose a new set of distinct symbols $a_1, a_2, ..., a_n$, such that

$$\{a_1, a_2, ..., a_n\} \cap \Sigma = \varnothing,$$

and consider the two languages

$$L_A = \{w_i w_j \cdots w_l w_k a_k a_l \cdots a_j a_i\}$$

and

$$L_B = \{v_i v_j \cdots v_l v_k a_k a_l \cdots a_j a_i\}.$$

Now look at the context-free grammar

$$G = (\{S, S_A, S_B\}, \Sigma \cup \{a_1, a_2, ...a_n\}, P, S)$$

where the set of productions P is the union of the two subsets: the first set P_A consists of

$$\begin{aligned} &S \to S_A, \\ &S_A \to w_i S_A a_i | w_i a_i, \qquad i = 1, 2, ..., n, \end{aligned}$$

the second set P_B has the productions

$$\begin{aligned} &S \to S_B, \\ &S_B \to v_i S_B a_i | v_i a_i, \qquad i = 1, 2, ..., n. \end{aligned}$$

Now take

$$G_A = (\{S, S_A\}, \Sigma \cup \{a_1, a_2, ..., a_n\}, P_A, S)$$

and

$$G_B = (\{S, S_B\}, \Sigma \cup \{a_1, a_2, ..., a_n\}, P_B, S).$$

Then clearly

$$\begin{aligned} L_A &= L(G_A), \\ L_B &= L(G_B), \end{aligned}$$

and

$$L(G) = L_A \cup L_B.$$

It is easy to see that G_A and G_B by themselves are unambiguous. If a given string in $L(G)$ ends with a_i, then its derivation with grammar G_A must have started with $S \Rightarrow w_i S a_i$. Similarly, we can tell at any later stage which rule has to be applied. Thus, if G is ambiguous it must be because there is a w for which there are two derivations

$$S \Rightarrow S_A \Rightarrow w_i S_A a_i \overset{*}{\Rightarrow} w_i w_j \cdots w_k a_k \cdots a_j a_i = w$$

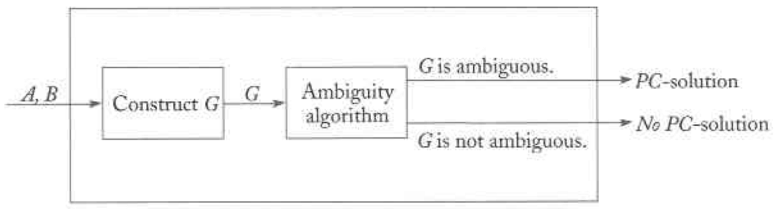
and

$$S \Rightarrow S_B \Rightarrow v_i S_B a_i \overset{*}{\Rightarrow} v_i v_j \cdots v_k a_k \cdots a_j a_i = w.$$

Consequently, if G is ambiguous, then the Post correspondence problem with the pair (A, B) has a solution. Conversely, if G is unambiguous, then the Post correspondence problem cannot have a solution.

If there existed an algorithm for solving the ambiguity problem, we could adapt it to solve the Post correspondence problem as shown in Figure 12.14. But since there is no algorithm for the Post correspondence problem, we conclude that the ambiguity problem is undecidable. ■

Figure 12.14
PC algorithm.



Theorem 12.9 There exists no algorithm for deciding whether or not

$$L(G_1) \cap L(G_2) = \varnothing$$

for arbitrary context-free grammars G_1 and G_2.

Proof: Take as G_1 the grammar G_A and as G_2 the grammar G_B as defined in the proof of Theorem 12.8. Suppose that $L(G_A)$ and $L(G_B)$ have a common element, that is

$$S_A \overset{*}{\Rightarrow} w_i w_j \cdots w_k a_k \cdots a_j a_i$$

and

$$S_B \overset{*}{\Rightarrow} v_i v_j \cdots v_k a_k \cdots a_j a_i.$$

Then the pair (A, B) has a PC-solution. Conversely, if the pair does not have a PC-solution, then $L(G_A)$ and $L(G_B)$ cannot have a common element. We conclude that $L(G_A) \cap L(G_B)$ is nonempty if and only if (A, B) has a PC-solution. This reduction proves the theorem. ■

There is a variety of other known results along these lines. Some of them can be reduced to the Post correspondence problem, while others are more easily solved by establishing different intermediate results first (see for example Exercises 6 and 7 at the end of this section). We will not give the arguments here, but point to some additional results in the exercises.

That there are many undecidable problems connected with context-free languages seems surprising at first and shows that there are limitations to computations in an area in which we might be tempted to try an algorithmic approach. For example, it would be helpful if we could tell if a programming language defined in BNF is ambiguous, or if two different specifications of a language are in fact equivalent. But the results that have been established tell us that this is not possible, and it would be a waste of time to look for an algorithm for either of these tasks. Keep in mind that this does not rule out the possibility that there may be ways of getting the answer for specific cases or perhaps even most interesting ones. What the undecidability results tell us is that there is no completely general algorithm and that no matter how many different cases a method can handle, there are invariably some situations for which it will break down.

EXERCISES

1. Prove the claim made in Theorem 12.8 that G_A and G_B by themselves are unambiguous.

★ 2. Show that the problem of determining whether or not

$$L(G_1) \subseteq L(G_2)$$

is undecidable for context-free grammars G_1, G_2.

★ 3. Show that, for arbitrary context-free grammars G_1 and G_2, the problem "$L(G_1) \cap L(G_2)$ is context-free" is undecidable.

★ 4. Show that if the language $L(G_A) \cap L(G_B)$ in Theorem 12.8 is regular, then it must be empty. Use this to show that the problem "$L(G)$ is regular" is undecidable for context-free G.

★ 5. Let L_1 be a regular language and G a context-free grammar. Show that the problem "$L_1 \subseteq L(G)$" is undecidable.

★ 6. Let M be any Turing machine. We can assume without loss of generality that every computation involves an even number of moves. For any such computation

$$q_0w \vdash x_1 \vdash x_2 \vdash \cdots \vdash x_n,$$

we can then construct the string

$$q_0w \vdash x_1^R \vdash x_2 \vdash x_3^R \vdash \cdots \vdash x_n.$$

This is called a **valid computation**.

Show that for every M we can construct three context-free grammars G_1, G_2, G_3, such that

(a) the set of all valid computations is $L(G_1) \cap L(G_2)$,

(b) the set of all invalid computations (that is, the complement of the set of valid computations) is $L(G_3)$.

★ 7. Use the results of the above exercise to show that "$L(G) = \Sigma^*$" is undecidable over the domain of all context-free grammars G.

★ 8. Let G_1 be a context-free grammar and G_2 a regular grammar. Is the problem

$$L(G_1) \cap L(G_2) = \varnothing$$

decidable?

★ 9. Let G_1 and G_2 be grammars with G_1 regular. Is the problem

$$L(G_1) = L(G_2)$$

decidable when

(a) G_2 is unrestricted,

(b) when G_2 is context-free,

(c) when G_2 is regular?