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 their limitations are few, 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 Limits of Algorithmic Computation

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 foreseeable computer. 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 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 $w_M$ describe a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, \#, F)$, and let $w$ be any element of $\Sigma^+$. A solution of the halting problem is a Turing machine $H$, which for any $w_M$ and $w$, performs the computation

$$q_0 w_M w \quad \vdash \quad x_1 q_y x_2,$$
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 description (encoded in some form) of $M$, say $w_M$, as well as the input $w$. 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:

$$q_0w_Mw \xrightarrow{H} x_1q_yx_2,$$

if $M$ applied to $w$ halts, and

$$q_0w_Mw \xrightarrow{H} y_1q_nx_2,$$

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

$$q_0w_Mw \xrightarrow{w} H.\infty,$$

if $M$ applied to $w$ halts, and

$$q_0w_Mw \xrightarrow{w} H'y_1q_ny_2,$$

if $M$ applied to $w$ does not halt.

From $H'$ we construct another Turing machine $\hat{H}$. This new machine takes as input $w_M$, copies it, and then behaves exactly like $H'$. Then the action of $\hat{H}$ is such that

$$q_0w_M \xrightarrow{\hat{w}} \hat{h}q_0w_Mw_M \xrightarrow{\hat{w}} \hat{h}\infty,$$

if $M$ applied to $w_M$ halts, and

$$q_0w_M \xrightarrow{\hat{w}} \hat{h}q_0w_Mw_M \xrightarrow{\hat{w}} \hat{h}y_1q_ny_2,$$

if $M$ applied to $w_M$ does not halt.
Now $\hat{H}$ is a Turing machine, so that it will have some description in $\Sigma^*$, say $\hat{w}$. This string, in addition to being the description of $\hat{H}$ can also be used as input string. We can therefore legitimately ask what would happen if $\hat{H}$ is applied to $\hat{w}$. From the above, identifying $M$ with $\hat{H}$, we get

$$q_0\hat{w} \vdash^* \hat{H}\infty,$$

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

$$q_0\hat{w} \vdash^* \hat{H}y_1q_ny_2,$$

if $\hat{H}$ applied to $\hat{w}$ 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 $\Sigma$, and let $M$ be a Turing machine that accepts $L$. Let $H$ be the Turing machine