5 REDUCIBILITY

In Chapter 4 we established the Turing machine as our model of a general purpose computer. We presented several examples of problems that are solvable on a Turing machine and gave one example of a problem, A_{TM}, that is computationally unsolvable. In this chapter we examine several additional unsolvable problems. In doing so we introduce the primary method for proving that problems are computationally unsolvable. It is called ***reducibility***.

A ***reduction*** is a way of converting one problem to another problem in such a way that a solution to the second problem can be used to solve the first problem. Such reducibilities come up often in everyday life, even if we don't usually refer to them in this way.

For example, suppose that you want to find your way around a new city. You know that doing so would be easy if you had a map. Thus you can reduce the problem of finding your way around the city to the problem of obtaining a map of the city.

Reducibility always involves two problems, which we call A and B. If A reduces to B, we can use a solution to B to solve A. So in our example, A is the problem of finding your way around the city and B is the problem of obtaining a map. Note that reducibility says nothing about solving A or B alone, but only about the solvability of A in the presence of a solution to B.

The following are further examples of reducibilities. The problem of traveling from Boston to Paris reduces to the problem of buying a plane ticket between the two cities. That problem in turn reduces to the problem of earning the money for the ticket. And that problem reduces to the problem of finding a job.

Reducibility also occurs in mathematical problems. For example, the problem of measuring the area of a rectangle reduces to the problem of measuring its length and width. The problem of solving a system of linear equations reduces to the problem of inverting a matrix.

Reducibility plays an important role in classifying problems by decidability and later in complexity theory as well. When A is reducible to B, solving A cannot be harder than solving B because a solution to B gives a solution to A. In terms of computability theory, if A is reducible to B and B is decidable, A also is decidable. Equivalently, if A is undecidable and reducible to B, B is undecidable. This last version is key to proving that various problems are undecidable.

In short, our method for proving that a problem is undecidable will be to show that some other problem already known to be undecidable reduces to it.

5.1 UNDECIDABLE PROBLEMS FROM LANGUAGE THEORY

We have already established the undecidability of A_{TM}, the problem of determining whether a Turing machine accepts a given input. Let's consider a related problem, $HALT_{\mathsf{TM}}$, the problem of determining whether a Turing machine halts (by accepting or rejecting) on a given input.[1] We use the undecidability of A_{TM} to prove the undecidability of $HALT_{\mathsf{TM}}$ by reducing A_{TM} to $HALT_{\mathsf{TM}}$. Let

$$HALT_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}.$$

THEOREM 5.1

$HALT_{\mathsf{TM}}$ is undecidable.

PROOF IDEA This proof is by contradiction. We assume that $HALT_{\mathsf{TM}}$ is decidable and use that assumption to show that A_{TM} is decidable, contradicting Theorem 4.11. The key idea is to show that A_{TM} is reducible to $HALT_{\mathsf{TM}}$.

Let's assume that we have a TM R that decides $HALT_{\mathsf{TM}}$. Then we use R to construct S, a TM that decides A_{TM}. To get a feel for the way to construct S, pretend that you are S. Your task is to decide A_{TM}. You are given an input of the form $\langle M, w\rangle$. You must output *accept* if M accepts w, and you must output

[1]In Section 4.2, we used the term *halting problem* for the language A_{TM} even though $HALT_{\mathsf{TM}}$ is the real halting problem. From here on we distinguish between the two by calling A_{TM} the *acceptance problem*.

reject if M loops or rejects on w. Try simulating M on w. If it accepts or rejects, do the same. But you may not be able to determine whether M is looping, and in that case your simulation will not terminate. That's bad, because you are a decider and thus never permitted to loop. So this idea, by itself, does not work.

Instead, use the assumption that you have TM R that decides $HALT_{\mathsf{TM}}$. With R, you can test whether M halts on w. If R indicates that M doesn't halt on w, reject because $\langle M, w\rangle$ isn't in A_{TM}. However, if R indicates that M does halt on w, you can do the simulation without any danger of looping.

Thus, if TM R exists, we can decide A_{TM}, but we know that A_{TM} is undecidable. By virtue of this contradiction we can conclude that R does not exist. Therefore $HALT_{\mathsf{TM}}$ is undecidable.

PROOF Let's assume for the purposes of obtaining a contradiction that TM R decides $HALT_{\mathsf{TM}}$. We construct TM S to decide A_{TM}, with S operating as follows.

```
S = "On input ⟨M, w⟩, an encoding of a TM M and a string w:
    1. Run TM R on input ⟨M, w⟩.
    2. If R rejects, reject.
    3. If R accepts, simulate M on w until it halts.
    4. If M has accepted, accept; if M has rejected, reject."
```

Clearly, if R decides $HALT_{\mathsf{TM}}$, then S decides A_{TM}. Because A_{TM} is undecidable, $HALT_{\mathsf{TM}}$ also must be undecidable.

Theorem 5.1 illustrates our strategy for proving that a problem is undecidable. This strategy is common to most proofs of undecidability, except for the undecidability of A_{TM} itself, which is proved directly via the diagonalization method.

We now present several other theorems and their proofs as further examples of the reducibility method for proving undecidability. Let

$$E_{\mathsf{TM}} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) = \emptyset\}.$$

THEOREM 5.2

E_{TM} is undecidable.

PROOF IDEA We follow the pattern adopted in Theorem 5.1. We assume for the purposes of obtaining a contradiction that E_{TM} is decidable and then show that A_{TM} is decidable—a contradiction. Let R be a TM that decides E_{TM}. We use R to construct TM S that decides A_{TM}. How will S work when it receives input $\langle M, w\rangle$?

One idea is for S to run R on input $\langle M \rangle$ and see whether it accepts. If it does, we know that $L(M)$ is empty and therefore that M does not accept w. But, if R rejects $\langle M \rangle$, all we know is that $L(M)$ is not empty and therefore that M accepts some string, but we still do not know whether M accepts the particular string w. So we need to use a different idea.

Instead of running R on $\langle M \rangle$ we run R on a modification of $\langle M \rangle$. We modify $\langle M \rangle$ to guarantee that M rejects all strings except w, but on input w it works as usual. Then we use R to determine whether the modified machine recognizes the empty language. The only string the machine can now accept is w, so its language will be nonempty iff it accepts w. If R accepts when it is fed a description of the modified machine, we know that the modified machine doesn't accept anything and that M doesn't accept w.

PROOF Let's write the modified machine described in the proof idea using our standard notation. We call it M_1.

```
$M_1$ = "On input $x$:
    1. If $x \neq w$, *reject*.
    2. If $x = w$, run $M$ on input $w$ and *accept* if $M$ does."
```

This machine has the string w as part of its description. It conducts the test of whether $x = w$ in the obvious way, by scanning the input and comparing it character by character with w to determine whether they are the same.

Putting all this together, we assume that TM R decides E_{TM} and construct TM S that decides A_{TM} as follows.

```
$S$ = "On input $\langle M, w \rangle$, an encoding of a TM $M$ and a string $w$:
    1. Use the description of $M$ and $w$ to construct the TM $M_1$ just
       described.
    2. Run $R$ on input $\langle M_1 \rangle$.
    3. If $R$ accepts, *reject*; if $R$ rejects, *accept*."
```

Note that S must actually be able to compute a description of M_1 from a description of M and w. It is able to do so because it needs only add extra states to M that perform the $x = w$ test.

If R were a decider for E_{TM}, S would be a decider for A_{TM}. A decider for A_{TM} cannot exist, so we know that E_{TM} must be undecidable.

Another interesting computational problem regarding Turing machines concerns determining whether a given Turing machine recognizes a language that also can be recognized by a simpler computational model. For example, we let $REGULAR_{\mathsf{TM}}$ be the problem of determining whether a given Turing machine has an equivalent finite automaton. This problem is the same as determining

whether the Turing machine recognizes a regular language. Let

$$REGULAR_{\mathsf{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is a regular language}\}.$$

THEOREM 5.3

$REGULAR_{\mathsf{TM}}$ is undecidable.

PROOF IDEA As usual for undecidability theorems, this proof is by reduction from A_{TM}. We assume that $REGULAR_{\mathsf{TM}}$ is decidable by a TM R and use this assumption to construct a TM S that decides A_{TM}. Less obvious now is how to use R's ability to assist S in its task. Nonetheless we can do so.

The idea is for S to take its input $\langle M, w \rangle$ and modify M so that the resulting TM recognizes a regular language if and only if M accepts w. We call the modified machine M_2. We design M_2 to recognize the nonregular language $\{0^n1^n \mid n \geq 0\}$ if M does not accept w, and to recognize the regular language Σ^* if M accepts w. We must specify how S can construct such an M_2 from M and w. Here, M_2 works by automatically accepting all strings in $\{0^n1^n \mid n \geq 0\}$. In addition, if M accepts w, M_2 accepts all other strings.

PROOF We let R be a TM that decides $REGULAR_{\mathsf{TM}}$ and construct TM S to decide A_{TM}. Then S works in the following manner.

```
$S$ = "On input $\langle M, w \rangle$, where $M$ is a TM and $w$ is a string:
    1. Construct the following TM $M_2$.
       $M_2$ = "On input $x$:
           1. If $x$ has the form $0^n1^n$, *accept*.
           2. If $x$ does not have this form, run $M$ on input $w$ and
              *accept* if $M$ accepts $w$."
    2. Run $R$ on input $\langle M_2 \rangle$.
    3. If $R$ accepts, *accept*; if $R$ rejects, *reject*."
```

Similarly, the problems of testing whether the language of a Turing machine is a context-free language, a decidable language, or even a finite language, can be shown to be undecidable with similar proofs. In fact, a general result, called Rice's theorem, states that testing *any property* of the languages recognized by Turing machines is undecidable. We give Rice's theorem in Problem 5.28.

So far, our strategy for proving languages undecidable involves a reduction from A_{TM}. Sometimes reducing from some other undecidable language, such as E_{TM}, is more convenient when we are showing that certain languages are undecidable. The following theorem shows that testing the equivalence of two Turing machines is an undecidable problem. We could prove it by a reduction from A_{TM}, but we use this opportunity to give an example of an undecidability proof by reduction from E_{TM}. Let

$$EQ_{\mathsf{TM}} = \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}.$$

THEOREM 5.4

EQ_{TM} is undecidable.

PROOF IDEA Show that, if EQ_{TM} were decidable, E_{TM} also would be decidable, by giving a reduction from E_{TM} to EQ_{TM}. The idea is simple. E_{TM} is the problem of determining whether the language of a TM is empty. EQ_{TM} is the problem of determining whether the languages of two TMs are the same. If one of these languages happens to be $\emptyset$, we end up with the problem of determining whether the language of the other machine is empty—that is, the E_{TM} problem. So in a sense, the E_{TM} problem is a special case of the EQ_{TM} problem wherein one of the machines is fixed to recognize the empty language. This idea makes giving the reduction easy.

PROOF We let TM R decide EQ_{TM} and construct TM S to decide E_{TM} as follows.

```
S = "On input ⟨M⟩, where M is a TM:
    1. Run R on input ⟨M, M_1⟩, where M_1 is a TM that rejects all inputs.
    2. If R accepts, accept; if R rejects, reject."
```

If R decides EQ_{TM}, S decides E_{TM}. But E_{TM} is undecidable by Theorem 5.2, so EQ_{TM} also must be undecidable.

REDUCTIONS VIA COMPUTATION HISTORIES

The computation history method is an important technique for proving that A_{TM} is reducible to certain languages. This method is often useful when the problem to be shown undecidable involves testing for the existence of something. For example, this method is used to show the undecidability of Hilbert's tenth problem, testing for the existence of integral roots in a polynomial.

The computation history for a Turing machine on an input is simply the sequence of configurations that the machine goes through as it processes the input. It is a complete record of the computation of this machine.

DEFINITION 5.5

Let M be a Turing machine and w an input string. An ***accepting computation history*** for M on w is a sequence of configurations, $C_1, C_2, \ldots, C_l$, where C_1 is the start configuration of M on w, C_l is an accepting configuration of M, and each C_i legally follows from C_{i-1} according to the rules of M. A ***rejecting computation history*** for M on w is defined similarly, except that C_l is a rejecting configuration.

Computation histories are finite sequences. If M doesn't halt on w, no accepting or rejecting computation history exists for M on w. Deterministic machines have at most one computation history on any given input. Nondeterministic machines may have many computation histories on a single input, corresponding to the various computation branches. For now, we continue to focus on deterministic machines. Our first undecidability proof using the computation history method concerns a type of machine called a linear bounded automaton.

DEFINITION 5.6

A ***linear bounded automaton*** is a restricted type of Turing machine wherein the tape head isn't permitted to move off the portion of the tape containing the input. If the machine tries to move its head off either end of the input, the head stays where it is, in the same way that the head will not move off the left-hand end of an ordinary Turing machine's tape.

A linear bounded automaton is a Turing machine with a limited amount of memory, as shown schematically in the following figure. It can only solve problems requiring memory that can fit within the tape used for the input. Using a tape alphabet larger than the input alphabet allows the available memory to be increased up to a constant factor. Hence we say that for an input of length n, the amount of memory available is linear in n—thus the name of this model.

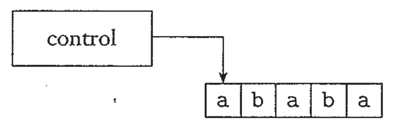


FIGURE 5.7
Schematic of a linear bounded automaton

Despite their memory constraint, linear bounded automata (LBAs) are quite powerful. For example, the deciders for A_{DFA}, A_{CFG}, E_{DFA}, and E_{CFG} all are LBAs. Every CFL can be decided by an LBA. In fact, coming up with a decidable language that can't be decided by an LBA takes some work. We develop the techniques to do so in Chapter 9.

Here, A_{LBA} is the problem of determining whether an LBA accepts its input. Even though A_{LBA} is the same as the undecidable problem A_{TM} where the Turing machine is restricted to be an LBA, we can show that A_{LBA} is decidable. Let

$$A_{\text{LBA}} = \{\langle M, w\rangle \mid M \text{ is an LBA that accepts string } w\}.$$

Before proving the decidability of A_{LBA}, we find the following lemma useful. It says that an LBA can have only a limited number of configurations when a string of length n is the input.

LEMMA 5.8

Let M be an LBA with q states and g symbols in the tape alphabet. There are exactly qng^n distinct configurations of M for a tape of length n.

PROOF Recall that a configuration of M is like a snapshot in the middle of its computation. A configuration consists of the state of the control, position of the head, and contents of the tape. Here, M has q states. The length of its tape is n, so the head can be in one of n positions, and g^n possible strings of tape symbols appear on the tape. The product of these three quantities is the total number of different configurations of M with a tape of length n.

THEOREM 5.9

A_{LBA} is decidable.

PROOF IDEA In order to decide whether LBA M accepts input w, we simulate M on w. During the course of the simulation, if M halts and accepts or rejects, we accept or reject accordingly. The difficulty occurs if M loops on w. We need to be able to detect looping so that we can halt and reject.

The idea for detecting when M is looping is that, as M computes on w, it goes from configuration to configuration. If M ever repeats a configuration it would go on to repeat this configuration over and over again and thus be in a loop. Because M is an LBA, the amount of tape available to it is limited. By Lemma 5.8, M can be in only a limited number of configurations on this amount of tape. Therefore only a limited amount of time is available to M before it will enter some configuration that it has previously entered. Detecting that M is looping is possible by simulating M for the number of steps given by Lemma 5.8. If M has not halted by then, it must be looping.

PROOF The algorithm that decides A_{LBA} is as follows.

```
L = "On input ⟨M, w⟩, where M is an LBA and w is a string:
    1. Simulate M on w for qng^n steps or until it halts.
    2. If M has halted, accept if it has accepted and reject if it has
       rejected. If it has not halted, reject."
```

If M on w has not halted within qng^n steps, it must be repeating a configuration according to Lemma 5.8 and therefore looping. That is why our algorithm rejects in this instance.

Theorem 5.9 shows that LBAs and TMs differ in one essential way: For LBAs the acceptance problem is decidable, but for TMs it isn't. However, certain other problems involving LBAs remain undecidable. One is the emptiness problem $E_{\text{LBA}} = \{\langle M\rangle \mid M \text{ is an LBA where } L(M) = \emptyset\}$. To prove that E_{LBA} is undecidable, we give a reduction that uses the computation history method.

THEOREM 5.10

E_{LBA} is undecidable.

PROOF IDEA This proof is by reduction from A_{TM}. We show that, if E_{LBA} were decidable, A_{TM} would also be. Suppose that E_{LBA} is decidable. How can we use this supposition to decide A_{TM}?

For a TM M and an input w we can determine whether M accepts w by constructing a certain LBA B and then testing whether $L(B)$ is empty. The language that B recognizes comprises all accepting computation histories for M on w. If M accepts w, this language contains one string and so is nonempty. If M does not accept w, this language is empty. If we can determine whether B's language is empty, clearly we can determine whether M accepts w.

Now we describe how to construct B from M and w. Note that we need to show more than the mere existence of B. We have to show how a Turing machine can obtain a description of B, given descriptions of M and w.

We construct B to accept its input x if x is an accepting computation history for M on w. Recall that an accepting computation history is the sequence of configurations, $C_1, C_2, \ldots, C_l$ that M goes through as it accepts some string w. For the purposes of this proof we assume that the accepting computation history is presented as a single string, with the configurations separated from each other by the # symbol, as shown in Figure 5.11.

$$\#\underbrace{\qquad\qquad}_{C_1}\#\underbrace{\qquad\qquad}_{C_2}\#\underbrace{\qquad\qquad}_{C_3}\#\ \cdots\ \#\underbrace{\qquad\qquad}_{C_l}\#$$

FIGURE 5.11
A possible input to B

The LBA B works as follows. When it receives an input x, B is supposed to accept if x is an accepting computation for M on w. First, B breaks up x according to the delimiters into strings $C_1, C_2, \ldots, C_l$. Then B determines whether the C_i satisfy the three conditions of an accepting computation history.

1. C_1 is the start configuration for M on w.
2. Each C_{i+1} legally follows from C_i.
3. C_l is an accepting configuration for M.

The start configuration C_1 for M on w is the string $q_0w_1w_2\cdots w_n$, where q_0 is the start state for M on w. Here, B has this string directly built in, so it is able to check the first condition. An accepting configuration is one that contains the q_{accept} state, so B can check the third condition by scanning C_l for q_{accept}. The second condition is the hardest to check. For each pair of adjacent configurations, B checks on whether C_{i+1} legally follows from C_i. This step involves verifying that C_i and C_{i+1} are identical except for the positions under and adjacent to the head in C_i. These positions must be updated according to the transition function of M. Then B verifies that the updating was done properly by zig-zagging between corresponding positions of C_i and C_{i+1}. To keep track of the current positions while zig-zagging, B marks the current position with dots on the tape. Finally, if conditions 1, 2, and 3 are satisfied, B accepts its input.

Note that the LBA B is *not* constructed for the purposes of actually running it on some input—a common confusion. We construct B only for the purpose of feeding a description of B into the decider for E_{LBA} that we have assumed to exist. Once this decider returns its answer we can invert it to obtain the answer to whether M accepts w. Thus we can decide A_{TM}, a contradiction.

PROOF Now we are ready to state the reduction of A_{TM} to E_{LBA}. Suppose that TM R decides E_{LBA}. Construct TM S that decides A_{TM} as follows.

```
S = "On input ⟨M, w⟩, where M is a TM and w is a string:
    1. Construct LBA B from M and w as described in the proof idea.
    2. Run R on input ⟨B⟩.
    3. If R rejects, accept; if R accepts, reject."
```

If R accepts $\langle B\rangle$, then $L(B) = \emptyset$. Thus M has no accepting computation history on w and M doesn't accept w. Consequently S rejects $\langle M, w\rangle$. Similarly, if R rejects $\langle B\rangle$, the language of B is nonempty. The only string that B can accept is an accepting computation history for M on w. Thus M must accept w. Consequently S accepts $\langle M, w\rangle$. Figure 5.12 illustrates LBA B.

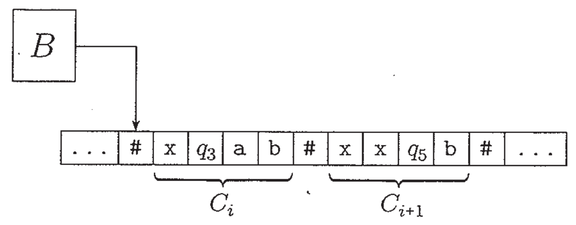


FIGURE 5.12
LBA B checking a TM computation history

We can also use the technique of reduction via computation histories to establish the undecidability of certain problems related to context-free grammars and pushdown automata. Recall that in Theorem 4.8 we presented an algorithm to decide whether a context-free grammar generates any strings—that is, whether $L(G) = \emptyset$. Now we show that a related problem is undecidable. It is the problem of determining whether a context-free grammar generates all possible strings. Proving that this problem is undecidable is the main step in showing that the equivalence problem for context-free grammars is undecidable. Let

$$ALL_{\text{CFG}} = \{\langle G\rangle \mid G \text{ is a CFG and } L(G) = \Sigma^*\}.$$

THEOREM 5.13

ALL_{CFG} is undecidable.

PROOF This proof is by contradiction. To get the contradiction we assume that ALL_{CFG} is decidable and use this assumption to show that A_{TM} is decidable. This proof is similar to that of Theorem 5.10 but with a small extra twist: It is a reduction from A_{TM} via computation histories, but we have to modify the representation of the computation histories slightly for a technical reason that we will explain later.

We now describe how to use a decision procedure for ALL_{CFG} to decide A_{TM}. For a TM M and an input w we construct a CFG G that generates all strings if and only if M does not accept w. So, if M does accept w, G does *not* generate some particular string. This string is—guess what—the accepting computation history for M on w. That is, G is designed to generate all strings that are *not* accepting computation histories for M on w.

To make the CFG G generate all strings that fail to be an accepting computation history for M on w, we utilize the following strategy. A string may fail to be an accepting computation history for several reasons. An accepting computation history for M on w appears as $\#C_1\#C_2\#\cdots\#C_l\#$, where C_i is the configuration of M on the ith step of the computation on w. Then, G generates all strings that

1. *do not* start with C_1,
2. *do not* end with an accepting configuration, or
3. where some C_i *does not* properly yield C_{i+1} under the rules of M.

If M does not accept w, no accepting computation history exists, so *all* strings fail in one way or another. Therefore G would generate all strings, as desired.

Now we get down to the actual construction of G. Instead of constructing G, we construct a PDA D. We know that we can use the construction given in Theorem 2.20 (page 117) to convert D to a CFG. We do so because, for our purposes, designing a PDA is easier than designing a CFG. In this instance, D will start by nondeterministically branching to guess which of the preceding three conditions to check. One branch checks on whether the beginning of the input string is C_1 and accepts if it isn't. Another branch checks on whether the input string ends with a configuration containing the accept state, q_{accept}, and accepts if it isn't.

The third branch is supposed to accept if some C_i does not properly yield C_{i+1}. It works by scanning the input until it nondeterministically decides that it has come to C_i. Next, it pushes C_i onto the stack until it comes to the end as marked by the # symbol. Then D pops the stack to compare with C_{i+1}. They are supposed to match except around the head position where the difference is dictated by the transition function of M. Finally, D accepts if it is a mismatch or an improper update.

The problem with this idea is that, when D pops C_i off the stack, it is in reverse order and not suitable for comparison with C_{i+1}. At this point the twist in the proof appears: We write the accepting computation history differently. Every other configuration appears in reverse order. The odd positions remain written in the forward order, but the even positions are written backward. Thus an accepting computation history would appear as shown in the following figure.

$$\#\underbrace{\longrightarrow}_{C_1}\#\underbrace{\longleftarrow}_{C_2^{\mathcal{R}}}\#\underbrace{\longrightarrow}_{C_3}\#\underbrace{\longleftarrow}_{C_4^{\mathcal{R}}}\#\ \cdots\ \#\underbrace{\qquad\quad}_{C_l}\#$$

FIGURE 5.14
Every other configuration written in reverse order

In this modified form, the PDA is able to push a configuration so that when it is popped, the order is suitable for comparison with the next one. We design D to accept any string that is not an accepting computation history in the modified form.

....

In Exercise 5.1 you can use Theorem 5.13 to show that EQ_{CFG} is undecidable.

5.2 A SIMPLE UNDECIDABLE PROBLEM

In this section we show that the phenomenon of undecidability is not confined to problems concerning automata. We give an example of an undecidable problem concerning simple manipulations of strings. It is called the ***Post correspondence problem***, or ***PCP***.

We can describe this problem easily as a type of puzzle. We begin with a collection of dominos, each containing two strings, one on each side. An individual domino looks like

$$\left[\frac{\mathtt{a}}{\mathtt{ab}}\right]$$

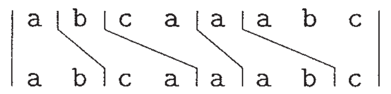
and a collection of dominos looks like

$$\left\{\left[\frac{\mathtt{b}}{\mathtt{ca}}\right], \left[\frac{\mathtt{a}}{\mathtt{ab}}\right], \left[\frac{\mathtt{ca}}{\mathtt{a}}\right], \left[\frac{\mathtt{abc}}{\mathtt{c}}\right]\right\}$$

The task is to make a list of these dominos (repetitions permitted) so that the string we get by reading off the symbols on the top is the same as the string of symbols on the bottom. This list is called a ***match***. For example, the following list is a match for this puzzle.

$$\left[\frac{\mathtt{a}}{\mathtt{ab}}\right]\left[\frac{\mathtt{b}}{\mathtt{ca}}\right]\left[\frac{\mathtt{ca}}{\mathtt{a}}\right]\left[\frac{\mathtt{a}}{\mathtt{ab}}\right]\left[\frac{\mathtt{abc}}{\mathtt{c}}\right].$$

Reading off the top string we get abcaaabc, which is the same as reading off the bottom. We can also depict this match by deforming the dominos so that the corresponding symbols from top and bottom line up.



For some collections of dominos finding a match may not be possible. For example, the collection

$$\left\{\left[\frac{\mathtt{abc}}{\mathtt{ab}}\right], \left[\frac{\mathtt{ca}}{\mathtt{a}}\right], \left[\frac{\mathtt{acc}}{\mathtt{ba}}\right]\right\}$$

cannot contain a match because every top string is longer than the corresponding bottom string.

The Post correspondence problem is to determine whether a collection of dominos has a match. This problem is unsolvable by algorithms.

Before getting to the formal statement of this theorem and its proof, let's state the problem precisely and then express it as a language. An instance of the PCP is a collection P of dominos:

$$P = \left\{ \left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \ldots, \left[\frac{t_k}{b_k}\right] \right\},$$

and a match is a sequence $i_1, i_2, \ldots, i_l$, where $t_{i_1}t_{i_2}\cdots t_{i_l} = b_{i_1}b_{i_2}\cdots b_{i_l}$. The problem is to determine whether P has a match. Let

$$PCP = \{\langle P\rangle |\ P \text{ is an instance of the Post correspondence problem with a match}\}.$$

THEOREM 5.15

PCP is undecidable.

PROOF IDEA Conceptually this proof is simple, though it involves many technical details. The main technique is reduction from A_{TM} via accepting computation histories. We show that from any TM M and input w we can construct an instance P where a match is an accepting computation history for M on w. If we could determine whether the instance has a match, we would be able to determine whether M accepts w.

How can we construct P so that a match is an accepting computation history for M on w? We choose the dominos in P so that making a match forces a simulation of M to occur. In the match, each domino links a position or positions in one configuration with the corresponding one(s) in the next configuration.

Before getting to the construction we handle three small technical points. (Don't worry about them too much on your initial reading through this construction.) First, for convenience in constructing P, we assume that M on w never attempts to move its head off the left-hand end of the tape. That requires first altering M to prevent this behavior. Second, if $w = \varepsilon$, we use the string ␣ in place of w in the construction. Third, we modify the PCP to require that a match starts with the first domino,

$$\left[\frac{t_1}{b_1}\right].$$

Later we show how to eliminate this requirement. We call this problem the modified Post correspondence problem (MPCP). Let

$$MPCP = \{\langle P\rangle |\ P \text{ is an instance of the Post correspondence problem with a match that starts with the first domino}\}.$$

Now let's move into the details of the proof and design P to simulate M on w.

PROOF We let TM R decide the PCP and construct S deciding A_{TM}. Let

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}),$$

where Q, Σ, Γ, and δ, are the state set, input alphabet, tape alphabet, and transition function of M, respectively.

In this case S constructs an instance of the PCP P that has a match iff M accepts w. To do that S first constructs an instance P' of the MPCP. We describe the construction in seven parts, each of which accomplishes a particular aspect of simulating M on w. To explain what we are doing we interleave the construction with an example of the construction in action.

Part 1. The construction begins in the following manner.

$$\text{Put } \left[\frac{\#}{\#q_0w_1w_2\cdots w_n\#}\right] \text{ into } P' \text{ as the first domino } \left[\frac{t_1}{b_1}\right].$$

Because P' is an instance of the MPCP, the match must begin with this domino. Thus the bottom string begins correctly with $C_1 = q_0w_1w_2\cdots w_n$, the first configuration in the accepting computation history for M on w, as shown in the following figure.

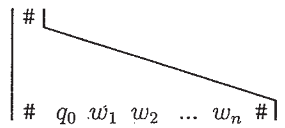


FIGURE 5.16
Beginning of the MPCP match

In this depiction of the partial match achieved so far, the bottom string consists of $\#q_0w_1w_2\cdots w_n\#$ and the top string consists only of #. To get a match we need to extend the top string to match the bottom string. We provide additional dominos to allow this extension. The additional dominos cause M's next configuration to appear at the extension of the bottom string by forcing a single-step simulation of M.

In parts 2, 3, and 4, we add to P' dominos that perform the main part of the simulation. Part 2 handles head motions to the right, part 3 handles head motions to the left, and part 4 handles the tape cells not adjacent to the head.

Part 2. For every $a, b \in \Gamma$ and every $q, r \in Q$ where $q \neq q_{\text{reject}}$,

$$\text{if } \delta(q,a) = (r,b,\mathrm{R}), \text{ put } \left[\frac{qa}{br}\right] \text{ into } P'.$$

Part 3. For every $a, b, c \in \Gamma$ and every $q, r \in Q$ where $q \neq q_{\text{reject}}$,

$$\text{if } \delta(q,a) = (r,b,\mathrm{L}), \text{ put } \left[\frac{cqa}{rcb}\right] \text{ into } P'.$$

Part 4. For every $a \in \Gamma$,

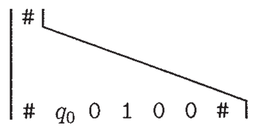
$$\text{put } \left[\frac{a}{a}\right] \text{ into } P'.$$

Now we make up a hypothetical example to illustrate what we have built so far. Let $\Gamma = \{0, 1, 2, \sqcup\}$. Say that w is the string 0100 and that the start state of M is q_0. In state q_0, upon reading a 0, let's say that the transition function dictates that M enters state q_7, writes a 2 on the tape, and moves its head to the right. In other words, $\delta(q_0, 0) = (q_7, 2, \mathrm{R})$.

Part 1 places the domino

$$\left[\frac{\#}{\#q_0 0100\#}\right] = \left[\frac{t_1}{b_1}\right]$$

in P', and the match begins:



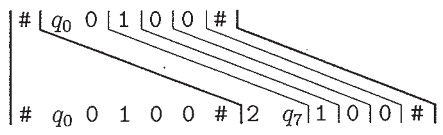
In addition, part 2 places the domino

$$\left[\frac{q_0 0}{2q_7}\right]$$

as $\delta(q_0, 0) = (q_7, 2, \mathrm{R})$ and part 4 places the dominos

$$\left[\frac{0}{0}\right], \left[\frac{1}{1}\right], \left[\frac{2}{2}\right], \text{ and } \left[\frac{\sqcup}{\sqcup}\right]$$

in P', as 0, 1, 2, and $\sqcup$ are the members of Γ. That, together with part 5, allows us to extend the match to



Thus the dominos of parts 2, 3, and 4 let us extend the match by adding the second configuration after the first one. We want this process to continue, adding the third configuration, then the fourth, and so on. For it to happen we need to add one more domino for copying the # symbol.

Part 5.

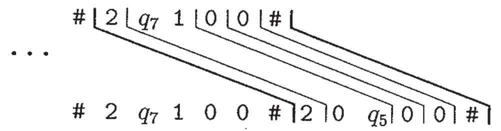
$$\text{Put } \left[\frac{\#}{\#}\right] \text{ and } \left[\frac{\#}{\sqcup\#}\right] \text{ into } P'.$$

The first of these dominos allows us to copy the # symbol that marks the separation of the configurations. In addition to that, the second domino allows us to add a blank symbol $\sqcup$ at the end of the configuration to simulate the infinitely many blanks to the right that are suppressed when we write the configuration.

Continuing with the example, let's say that in state q_7, upon reading a 1, M goes to state q_5, writes a 0, and moves the head to the right. That is, $\delta(q_7, 1) = (q_5, 0, \mathrm{R})$. Then we have the domino

$$\left[\frac{q_7 1}{0q_5}\right] \text{ in } P'.$$

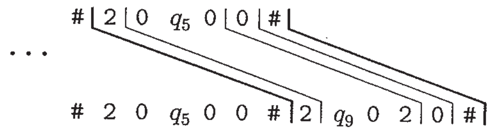
So the latest partial match extends to



Then, suppose that in state q_5, upon reading a 0, M goes to state q_9, writes a 2, and moves its head to the left. So $\delta(q_5, 0) = (q_9, 2, \mathrm{L})$. Then we have the dominos

$$\left[\frac{0q_5 0}{q_9 02}\right], \left[\frac{1q_5 0}{q_9 12}\right], \left[\frac{2q_5 0}{q_9 22}\right], \text{ and } \left[\frac{\sqcup q_5 0}{q_9 \sqcup 2}\right].$$

The first one is relevant because the symbol to the left of the head is a 0. The preceding partial match extends to

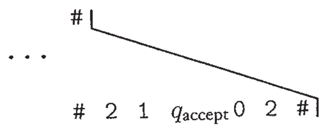


Note that, as we construct a match, we are forced to simulate M on input w. This process continues until M reaches a halting state. If an accept state occurs, we want to let the top of the partial match "catch up" with the bottom so that the match is complete. We can arrange for that to happen by adding additional dominos.

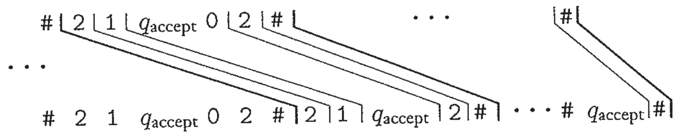
Part 6. For every $a \in \Gamma$,

$$\text{put } \left[\frac{a\, q_{\text{accept}}}{q_{\text{accept}}}\right] \text{ and } \left[\frac{q_{\text{accept}}\, a}{q_{\text{accept}}}\right] \text{ into } P'.$$

This step has the effect of adding "pseudo-steps" of the Turing machine after it has halted, where the head "eats" adjacent symbols until none are left. Continuing with the example, if the partial match up to the point when the machine halts in an accept state is



The dominos we have just added allow the match to continue:

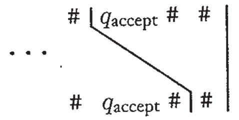


# 2 1 q_accept 0 2 # ... #
...
# 2 1 q_accept 0 2 # 2 1 q_accept 2 # ... # q_accept #

Part 7. Finally we add the domino

$$\left[\frac{q_{\text{accept}}\#\#}{\#}\right]$$

and complete the match:



That concludes the construction of P'. Recall that P' is an instance of the MPCP whereby the match simulates the computation of M on w. To finish the proof, we recall that the MPCP differs from the PCP in that the match is required to start with the first domino in the list. If we view P' as an instance of

the PCP instead of the MPCP, it obviously has a match, regardless of whether M halts on w. Can you find it? (Hint: It is very short.)

We now show how to convert P' to P, an instance of the PCP that still simulates M on w. We do so with a somewhat technical trick. The idea is to build the requirement of starting with the first domino directly into the problem so that stating the explicit requirement becomes unnecessary. We need to introduce some notation for this purpose.

Let $u = u_1u_2 \cdots u_n$ be any string of length n. Define $\star u$, $u\star$, and $\star u\star$ to be the three strings

$$\begin{aligned} \star u &= * u_1 * u_2 * u_3 * \;\cdots\; * u_n \\ u\star &= u_1 * u_2 * u_3 * \;\cdots\; * u_n * \\ \star u\star &= * u_1 * u_2 * u_3 * \;\cdots\; * u_n * . \end{aligned}$$

Here, $\star u$ adds the symbol $*$ before every character in u, $u\star$ adds one after each character in u, and $\star u\star$ adds one both before and after each character in u.

To convert P' to P, an instance of the PCP, we do the following. If P' were the collection

$$\left\{ \left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \left[\frac{t_3}{b_3}\right], \ldots, \left[\frac{t_k}{b_k}\right] \right\},$$

we let P be the collection

$$\left\{ \left[\frac{\star t_1}{\star b_1\star}\right], \left[\frac{\star t_1}{b_1\star}\right], \left[\frac{\star t_2}{b_2\star}\right], \left[\frac{\star t_3}{b_3\star}\right], \ldots, \left[\frac{\star t_k}{b_k\star}\right], \left[\frac{*\diamond}{\diamond}\right] \right\}.$$

Considering P as an instance of the PCP, we see that the only domino that could possibly start a match is the first one,

$$\left[\frac{\star t_1}{\star b_1\star}\right],$$

because it is the only one where both the top and the bottom start with the same symbol—namely, $*$. Besides forcing the match to start with the first domino, the presence of the $*$s doesn't affect possible matches because they simply interleave with the original symbols. The original symbols now occur in the even positions of the match. The domino

$$\left[\frac{*\diamond}{\diamond}\right]$$

is there to allow the top to add the extra $*$ at the end of the match.

5.3 MAPPING REDUCIBILITY

We have shown how to use the reducibility technique to prove that various problems are undecidable. In this section we formalize the notion of reducibility. Doing so allows us to use reducibility in more refined ways, such as for proving that certain languages are not Turing-recognizable and for applications in complexity theory.

The notion of reducing one problem to another may be defined formally in one of several ways. The choice of which one to use depends on the application. Our choice is a simple type of reducibility called ***mapping reducibility***.[2]

Roughly speaking, being able to reduce problem A to problem B by using a mapping reducibility means that a computable function exists that converts instances of problem A to instances of problem B. If we have such a conversion function, called a *reduction*, we can solve A with a solver for B. The reason is that any instance of A can be solved by first using the reduction to convert it to an instance of B and then applying the solver for B. A precise definition of mapping reducibility follows shortly.

COMPUTABLE FUNCTIONS

A Turing machine computes a function by starting with the input to the function on the tape and halting with the output of the function on the tape.

DEFINITION 5.17

A function $f\colon \Sigma^* \longrightarrow \Sigma^*$ is a ***computable function*** if some Turing machine M, on every input w, halts with just $f(w)$ on its tape.

EXAMPLE 5.18

All usual arithmetic operations on integers are computable functions. For example, we can make a machine that takes input $\langle m, n\rangle$ and returns $m + n$, the sum of m and n. We don't give any details here, leaving them as exercises.

EXAMPLE 5.19

Computable functions may be transformations of machine descriptions. For example, one computable function f takes input w and returns the description of a Turing machine $\langle M'\rangle$ if $w = \langle M\rangle$ is an encoding of a Turing machine M.

[2]It is called ***many–one reducibility*** in some other textbooks.

The machine M' is a machine that recognizes the same language as M, but never attempts to move its head off the left-hand end of its tape. The function f accomplishes this task by adding several states to the description of M. The function returns ε if w is not a legal encoding of a Turing machine.

FORMAL DEFINITION OF MAPPING REDUCIBILITY

Now we define mapping reducibility. As usual we represent computational problems by languages.

DEFINITION 5.20

Language A is ***mapping reducible*** to language B, written $A \leq_{\mathrm{m}} B$, if there is a computable function $f\colon \Sigma^* \longrightarrow \Sigma^*$, where for every w,

$$w \in A \iff f(w) \in B.$$

The function f is called the ***reduction*** of A to B.

The following figure illustrates mapping reducibility.

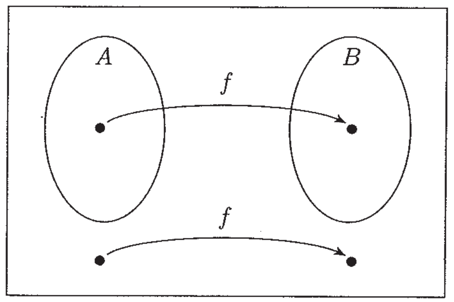


FIGURE 5.21
Function f reducing A to B

A mapping reduction of A to B provides a way to convert questions about membership testing in A to membership testing in B. To test whether $w \in A$, we use the reduction f to map w to $f(w)$ and test whether $f(w) \in B$. The term *mapping reduction* comes from the function or mapping that provides the means of doing the reduction.

If one problem is mapping reducible to a second, previously solved problem, we can thereby obtain a solution to the original problem. We capture this idea in the following theorem.

THEOREM 5.22

If $A \leq_m B$ and B is decidable, then A is decidable.

PROOF We let M be the decider for B and f be the reduction from A to B. We describe a decider N for A as follows.

```
N = "On input w:
      1. Compute f(w).
      2. Run M on input f(w) and output whatever M outputs."
```

Clearly, if $w \in A$, then $f(w) \in B$ because f is a reduction from A to B. Thus M accepts $f(w)$ whenever $w \in A$. Therefore N works as desired.

The following corollary of Theorem 5.22 has been our main tool for proving undecidability.

COROLLARY 5.23

If $A \leq_m B$ and A is undecidable, then B is undecidable.

Now we revisit some of our earlier proofs that used the reducibility method to get examples of mapping reducibilities.

EXAMPLE 5.24

In Theorem 5.1 we used a reduction from A_{TM} to prove that $HALT_{TM}$ is undecidable. This reduction showed how a decider for $HALT_{TM}$ could be used to give a decider for A_{TM}. We can demonstrate a mapping reducibility from A_{TM} to $HALT_{TM}$ as follows. To do so we must present a computable function f that takes input of the form $\langle M, w\rangle$ and returns output of the form $\langle M', w'\rangle$, where

$$\langle M, w\rangle \in A_{TM} \text{ if and only if } \langle M', w'\rangle \in HALT_{TM}.$$

The following machine F computes a reduction f.

```
F = "On input ⟨M, w⟩:
      1. Construct the following machine M'.
         M' = "On input x:
               1. Run M on x.
               2. If M accepts, accept.
               3. If M rejects, enter a loop."
      2. Output ⟨M', w⟩."
```

A minor issue arises here concerning improperly formed input strings. If TM F determines that its input is not of the correct form as specified in the input line "On input $\langle M, w\rangle$:" and hence that the input is not in A_{TM}, the TM outputs a

string not in $HALT_{TM}$. Any string not in $HALT_{TM}$ will do. In general, when we describe a Turing machine that computes a reduction from A to B, improperly formed inputs are assumed to map to strings outside of B.

EXAMPLE 5.25

The proof of the undecidability of the Post correspondence problem in Theorem 5.15 contains two mapping reductions. First, it shows that $A_{TM} \leq_m MPCP$ and then it shows that $MPCP \leq_m PCP$. In both cases we can easily obtain the actual reduction function and show that it is a mapping reduction. As Exercise 5.6 shows, mapping reducibility is transitive, so these two reductions together imply that $A_{TM} \leq_m PCP$.

EXAMPLE 5.26

A mapping reduction from E_{TM} to EQ_{TM} lies in the proof of Theorem 5.4. In this case the reduction f maps the input $\langle M\rangle$ to the output $\langle M, M_1\rangle$, where M_1 is the machine that rejects all inputs.

EXAMPLE 5.27

The proof of Theorem 5.2 showing that E_{TM} is undecidable illustrates the difference between the formal notion of mapping reducibility that we have defined in this section and the informal notion of reducibility that we used earlier in this chapter. The proof shows that E_{TM} is undecidable by reducing A_{TM} to it. Let's see whether we can convert this reduction to a mapping reduction.

From the original reduction we may easily construct a function f that takes input $\langle M, w\rangle$ and produces output $\langle M_1\rangle$, where M_1 is the Turing machine described in that proof. But M accepts w iff $L(M_1)$ is *not* empty so f is a mapping reduction from A_{TM} to $\overline{E_{TM}}$. It still shows that E_{TM} is undecidable because decidability is not affected by complementation, but it doesn't give a mapping reduction from A_{TM} to E_{TM}. In fact, no such reduction exists, as you are asked to show in Exercise 5.5.

The sensitivity of mapping reducibility to complementation is important in the use of reducibility to prove nonrecognizability of certain languages. We can also use mapping reducibility to show that problems are not Turing-recognizable. The following theorem is analogous to Theorem 5.22.

THEOREM 5.28

If $A \leq_m B$ and B is Turing-recognizable, then A is Turing-recognizable.

The proof is the same as that of Theorem 5.22, except that M and N are recognizers instead of deciders.

COROLLARY 5.29

If $A \leq_{\mathrm{m}} B$ and A is not Turing-recognizable, then B is not Turing-recognizable.

In a typical application of this corollary, we let A be $\overline{A_{\mathsf{TM}}}$, the complement of A_{TM}. We know that $\overline{A_{\mathsf{TM}}}$ is not Turing-recognizable from Corollary 4.23. The definition of mapping reducibility implies that $A \leq_{\mathrm{m}} B$ means the same as $\overline{A} \leq_{\mathrm{m}} \overline{B}$. To prove that B isn't recognizable we may show that $A_{\mathsf{TM}} \leq_{\mathrm{m}} \overline{B}$. We can also use mapping reducibility to show that certain problems are neither Turing-recognizable nor co-Turing-recognizable, as in the following theorem.

THEOREM 5.30

EQ_{TM} is neither Turing-recognizable nor co-Turing-recognizable.

PROOF First we show that EQ_{TM} is not Turing-recognizable. We do so by showing that A_{TM} is reducible to $\overline{EQ_{\mathsf{TM}}}$. The reducing function f works as follows.

```
F = "On input ⟨M, w⟩ where M is a TM and w a string:
      1. Construct the following two machines $M_1$ and $M_2$.
           $M_1$ = "On any input:
                1. Reject."
           $M_2$ = "On any input:
                1. Run M on w. If it accepts, accept."
      2. Output ⟨$M_1$, $M_2$⟩."
```

Here, M_1 accepts nothing. If M accepts w, M_2 accepts everything, and so the two machines are not equivalent. Conversely, if M doesn't accept w, M_2 accepts nothing, and they are equivalent. Thus f reduces A_{TM} to $\overline{EQ_{\mathsf{TM}}}$, as desired.

To show that $\overline{EQ_{\mathsf{TM}}}$ is not Turing-recognizable we give a reduction from A_{TM} to the complement of $\overline{EQ_{\mathsf{TM}}}$—namely, EQ_{TM}. Hence we show that $A_{\mathsf{TM}} \leq_{\mathrm{m}} EQ_{\mathsf{TM}}$. The following TM G computes the reducing function g.

```
G = "The input is ⟨M, w⟩ where M is a TM and w a string:
      1. Construct the following two machines $M_1$ and $M_2$.
           $M_1$ = "On any input:
                1. Accept."
           $M_2$ = "On any input:
                1. Run M on w.
                2. If it accepts, accept."
      2. Output ⟨$M_1$, $M_2$⟩."
```

The only difference between f and g is in machine M_1. In f, machine M_1 always rejects, whereas in g it always accepts. In both f and g, M accepts w iff M_2 always accepts. In g, M accepts w iff M_1 and M_2 are equivalent. That is why g is a reduction from A_{TM} to EQ_{TM}.

EXERCISES

5.1 Show that EQ_{CFG} is undecidable.

5.2 Show that EQ_{CFG} is co-Turing-recognizable.

5.3 Find a match in the following instance of the Post Correspondence Problem.

$$\left\{ \left[\frac{\mathtt{ab}}{\mathtt{abab}}\right], \left[\frac{\mathtt{b}}{\mathtt{a}}\right], \left[\frac{\mathtt{aba}}{\mathtt{b}}\right], \left[\frac{\mathtt{aa}}{\mathtt{a}}\right] \right\}$$

5.4 If $A \leq_{\mathrm{m}} B$ and B is a regular language, does that imply that A is a regular language? Why or why not?

$^{\mathrm{A}}$5.5 Show that A_{TM} is not mapping reducible to E_{TM}. In other words, show that no computable function reduces A_{TM} to E_{TM}. (Hint: Use a proof by contradiction, and facts you already know about A_{TM} and E_{TM}.)

$^{\mathrm{A}}$5.6 Show that $\leq_{\mathrm{m}}$ is a transitive relation.

$^{\mathrm{A}}$5.7 Show that if A is Turing-recognizable and $A \leq_{\mathrm{m}} \overline{A}$, then A is decidable.

$^{\mathrm{A}}$5.8 In the proof of Theorem 5.15 we modified the Turing machine M so that it never tries to move its head off the left-hand end of the tape. Suppose that we did not make this modification to M. Modify the PCP construction to handle this case.

PROBLEMS

5.9 Let $T = \{\langle M \rangle \mid M$ is a TM that accepts $w^{\mathcal{R}}$ whenever it accepts $w\}$. Show that T is undecidable.

$^{\mathrm{A}}$5.10 Consider the problem of determining whether a two-tape Turing machine ever writes a nonblank symbol on its second tape when it is run on input w. Formulate this problem as a language, and show that it is undecidable.

$^{\mathrm{A}}$5.11 Consider the problem of determining whether a two-tape Turing machine ever writes a nonblank symbol on its second tape during the course of its computation on any input string. Formulate this problem as a language, and show that it is undecidable.

5.12 Consider the problem of determining whether a single-tape Turing machine ever writes a blank symbol over a nonblank symbol during the course of its computation on any input string. Formulate this problem as a language, and show that it is undecidable.

5.13 A ***useless state*** in a Turing machine is one that is never entered on any input string. Consider the problem of determining whether a Turing machine has any useless states. Formulate this problem as a language and show that it is undecidable.

5.14 Consider the problem of determining whether a Turing machine M on an input w ever attempts to move its head left when its head is on the left-most tape cell. Formulate this problem as a language and show that it is undecidable.

5.15 Consider the problem of determining whether a Turing machine M on an input w ever attempts to move its head left at any point during its computation on w. Formulate this problem as a language and show that it *is* decidable.

5.16 Let $\Gamma = \{0, 1, \sqcup\}$ be the tape alphabet for all TMs in this problem. Define the ***busy beaver function*** $BB\colon \mathcal{N} \longrightarrow \mathcal{N}$ as follows. For each value of k, consider all k-state TMs that halt when started with a blank tape. Let $BB(k)$ be the maximum number of 1s that remain on the tape among all of these machines. Show that BB is not a computable function.

5.17 Show that the Post Correspondence Problem is decidable over the unary alphabet $\Sigma = \{1\}$.

5.18 Show that the Post Correspondence Problem is undecidable over the binary alphabet $\Sigma = \{0,1\}$.

5.19 In the *silly Post Correspondence Problem*, *SPCP*, in each pair the top string has the same length as the bottom string. Show that the *SPCP* is decidable.

5.20 Prove that there exists an undecidable subset of $\{1\}^*$.

5.21 Let $AMBIG_{\mathrm{CFG}} = \{\langle G\rangle \mid G \text{ is an ambiguous CFG}\}$. Show that $AMBIG_{\mathrm{CFG}}$ is undecidable. (Hint: Use a reduction from *PCP*. Given an instance

$$P = \left\{ \left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \ldots, \left[\frac{t_k}{b_k}\right] \right\},$$

of the Post Correspondence Problem, construct a CFG G with the rules

$$\begin{aligned} S &\to T \mid B \\ T &\to t_1 T \mathrm{a}_1 \mid \cdots \mid t_k T \mathrm{a}_k \mid t_1 \mathrm{a}_1 \mid \cdots \mid t_k \mathrm{a}_k \\ B &\to b_1 B \mathrm{a}_1 \mid \cdots \mid b_k B \mathrm{a}_k \mid b_1 \mathrm{a}_1 \mid \cdots \mid b_k \mathrm{a}_k\ , \end{aligned}$$

where $\mathrm{a}_1, \ldots, \mathrm{a}_k$ are new terminal symbols. Prove that this reduction works.)

5.22 Show that A is Turing-recognizable iff $A \leq_{\mathrm{m}} A_{\mathrm{TM}}$.

5.23 Show that A is decidable iff $A \leq_{\mathrm{m}} 0^*1^*$.

5.24 Let $J = \{w \mid \text{either } w = 0x \text{ for some } x \in A_{\mathrm{TM}}, \text{ or } w = 1y \text{ for some } y \in \overline{A_{\mathrm{TM}}}\ \}$. Show that neither J nor $\overline{J}$ is Turing-recognizable.

5.25 Give an example of an undecidable language B, where $B \leq_{\mathrm{m}} \overline{B}$.

5.26 Define a ***two-headed finite automaton*** (2DFA) to be a deterministic finite automaton that has two read-only, bidirectional heads that start at the left-hand end of the input tape and can be independently controlled to move in either direction. The tape of a 2DFA is finite and is just large enough to contain the input plus two additional blank tape cells, one on the left-hand end and one on the right-hand end, that serve as delimiters. A 2DFA accepts its input by entering a special accept state. For example, a 2DFA can recognize the language $\{\mathrm{a}^n\mathrm{b}^n\mathrm{c}^n \mid n \geq 0\}$.

a. Let $A_{\mathrm{2DFA}} = \{\langle M, x\rangle \mid M \text{ is a 2DFA and } M \text{ accepts } x\}$. Show that A_{2DFA} is decidable.

b. Let $E_{\mathrm{2DFA}} = \{\langle M\rangle \mid M \text{ is a 2DFA and } L(M) = \emptyset\}$. Show that E_{2DFA} is not decidable.

5.27 A ***two-dimensional finite automaton*** (2DIM-DFA) is defined as follows. The input is an $m \times n$ rectangle, for any $m, n \geq 2$. The squares along the boundary of the rectangle contain the symbol # and the internal squares contain symbols over the input alphabet Σ. The transition function is a mapping $Q \times \Sigma \to Q \times \{\mathrm{L, R, U, D}\}$ to indicate the next state and the new head position (Left, Right, Up, Down). The machine accepts when it enters one of the designated accept states. It rejects if it tries to move off the input rectangle or if it never halts. Two such machines are equivalent if they accept the same rectangles. Consider the problem of determining whether two of these machines are equivalent. Formulate this problem as a language, and show that it is undecidable.

A***5.28** **Rice's theorem.** Let P be any nontrivial property of the language of a Turing machine. Prove that the problem of determining whether a given Turing machine's language has property P is undecidable.

In more formal terms, let P be a language consisting of Turing machine descriptions where P fulfills two conditions. First, P is nontrivial—it contains some, but not all, TM descriptions. Second, P is a property of the TM's language—whenever $L(M_1) = L(M_2)$, we have $\langle M_1\rangle \in P$ iff $\langle M_2\rangle \in P$. Here, M_1 and M_2 are any TMs. Prove that P is an undecidable language.

5.29 Show that both conditions in Problem 5.28 are necessary for proving that P is undecidable.

5.30 Use Rice's theorem, which appears in Problem 5.28, to prove the undecidability of each of the following languages.

$^{\mathrm{A}}$**a.** $INFINITE_{\mathrm{TM}} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) \text{ is an infinite language}\}$.

b. $\{\langle M\rangle \mid M \text{ is a TM and } 1011 \in L(M)\}$.

c. $ALL_{\mathrm{TM}} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) = \Sigma^*\}$.

5.31 Let

$$f(x) = \begin{cases} 3x + 1 & \text{for odd } x \\ x/2 & \text{for even } x \end{cases}$$

for any natural number x. If you start with an integer x and iterate f, you obtain a sequence, $x, f(x), f(f(x)), \ldots$ Stop if you ever hit 1. For example, if $x = 17$, you get the sequence 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. Extensive computer tests have shown that every starting point between 1 and a large positive integer gives a sequence that ends in 1. But, the question of whether all positive starting points end up at 1 is unsolved; it is called the $3x + 1$ problem.

Suppose that A_{TM} were decidable by a TM H. Use H to describe a TM that is guaranteed to state the answer to the $3x + 1$ problem.

5.32 Prove that the following two languages are undecidable.

a. $OVERLAP_{\mathrm{CFG}} = \{\langle G, H\rangle \mid G \text{ and } H \text{ are CFGs where } L(G) \cap L(H) \neq \emptyset\}$. (Hint: Adapt the hint in Problem 5.21.)

b. $PREFIX\text{-}FREE_{\mathrm{CFG}} = \{G \mid G \text{ is a CFG where } L(G) \text{ is prefix-free}\}$.

5.33 Let $S = \{\langle M\rangle \mid M \text{ is a TM and } L(M) = \{\langle M\rangle\}\ \}$. Show that neither S nor $\overline{S}$ is Turing-recognizable.

5.34 Consider the problem of determining whether a PDA accepts some string of the form $\{ww \mid w \in \{0,1\}^*\}$. Use the computation history method to show that this problem is undecidable.

5.35 Let $X = \{\langle M, w\rangle |\ M$ is a single-tape TM that never modifies the portion of the tape that contains the input $w\}$. Is X decidable? Prove your answer.

SELECTED SOLUTIONS

5.5 Suppose for a contradiction that $A_{\mathrm{TM}} \leq_{\mathrm{m}} E_{\mathrm{TM}}$ via reduction f. It follows from the definition of mapping reducibility that $\overline{A_{\mathrm{TM}}} \leq_{\mathrm{m}} \overline{E_{\mathrm{TM}}}$ via the same reduction function f. However $\overline{E_{\mathrm{TM}}}$ is Turing-recognizable and $\overline{A_{\mathrm{TM}}}$ is not Turing-recognizable, contradicting Theorem 5.28.

5.6 Suppose $A \leq_{\mathrm{m}} B$ and $B \leq_{\mathrm{m}} C$. Then there are computable functions f and g such that $x \in A \iff f(x) \in B$ and $y \in B \iff g(y) \in C$. Consider the composition function $h(x) = g(f(x))$. We can build a TM that computes h as follows: First, simulate a TM for f (such a TM exists because we assumed that f is computable) on input x and call the output y. Then simulate a TM for g on y. The output is $h(x) = g(f(x))$. Therefore h is a computable function. Moreover, $x \in A \iff h(x) \in C$. Hence $A \leq_{\mathrm{m}} C$ via the reduction function h.

5.7 Suppose that $A \leq_{\mathrm{m}} \overline{A}$. Then $\overline{A} \leq_{\mathrm{m}} A$ via the same mapping reduction. Because A is Turing-recognizable, Theorem 5.28 implies that $\overline{A}$ is Turing-recognizable, and then Theorem 4.22 implies that A is decidable.

5.8 You need to handle the case where the head is at the leftmost tape cell and attempts to move left. To do so add dominos

$$\left[\frac{\#qa}{\#rb}\right]$$

for every $q, r \in Q$ and $a, b \in \Gamma$, where $\delta(q, a) = (r, b, \mathrm{L})$.

5.10 Let $B = \{\langle M, w\rangle |\ M$ is a two-tape TM which writes a nonblank symbol on its second tape when it is run on $w\}$. Show that A_{TM} reduces to B. Assume for the sake of contradiction that TM R decides B. Then construct TM S that uses R to decide A_{TM}.

```
$S$ = "On input $\langle M, w\rangle$:
    1. Use $M$ to construct the following two-tape TM $T$.
       $T$ = "On input $x$:
           1. Simulate $M$ on $x$ using the first tape.
           2. If the simulation shows that $M$ accepts, write a non-
              blank symbol on the second tape."
    2. Run $R$ on $\langle T, w\rangle$ to determine whether $T$ on input $w$ writes a
       nonblank symbol on its second tape.
    3. If $R$ accepts, $M$ accepts $w$, therefore accept. Otherwise reject."
```

5.11 Let $C = \{\langle M\rangle |\ M$ is a two-tape TM which writes a nonblank symbol on its second tape when it is run on some input$\}$. Show that A_{TM} reduces to C. Assume for the sake of contradiction that TM R decides C. Construct TM S that uses R to decide A_{TM}.

```
$S$ = "On input $\langle M, w\rangle$:
    1. Use $M$ and $w$ to construct the following two-tape TM $T_w$.
       $T_w$ = "On any input:
           1. Simulate $M$ on $w$ using the first tape.
           2. If the simulation shows that $M$ accepts, write a non-
              blank symbol on the second tape."
    2. Run $R$ on $\langle T_w\rangle$ to determine whether $T_w$ ever writes a nonblank
       symbol on its second tape.
    3. If $R$ accepts, $M$ accepts $w$, therefore accept. Otherwise reject."
```

5.28 Assume for the sake of contradiction that P is a decidable language satisfying the properties and let R_P be a TM that decides P. We show how to decide A_{TM} using R_P by constructing TM S. First let $T_\emptyset$ be a TM that always rejects, so $L(T_\emptyset) = \emptyset$. You may assume that $\langle T_\emptyset\rangle \notin P$ without loss of generality, because you could proceed with $\overline{P}$ instead of P if $\langle T_\emptyset\rangle \in P$. Because P is not trivial, there exists a TM T with $\langle T\rangle \in P$. Design S to decide A_{TM} using R_P's ability to distinguish between $T_\emptyset$ and T.

```
$S$ = "On input $\langle M, w\rangle$:
    1. Use $M$ and $w$ to construct the following TM $M_w$.
       $M_w$ = "On input $x$:
           1. Simulate $M$ on $w$. If it halts and rejects, reject.
              If it accepts, proceed to stage 2.
           2. Simulate $T$ on $x$. If it accepts, accept."
    2. Use TM $R_P$ to determine whether $\langle M_w\rangle \in P$. If YES, accept.
       If NO, reject."
```

TM M_w simulates T if M accepts w. Hence $L(M_w)$ equals $L(T)$ if M accepts w and $\emptyset$ otherwise. Therefore $\langle M, w\rangle \in P$ iff M accepts w.

5.30 (a) *INFINITE*$_{\mathrm{TM}}$ is a language of TM descriptions. It satisfies the two conditions of Rice's theorem. First, it is nontrivial because some TMs have infinite languages and others do not. Second, it depends only on the language. If two TMs recognize the same language, either both have descriptions in *INFINITE*$_{\mathrm{TM}}$ or neither do. Consequently, Rice's theorem implies that *INFINITE*$_{\mathrm{TM}}$ is undecidable.