CS 4410

Automata, Computability, and Formal Language

Chapter 12: Limits of Algorithmic Computation

Dr. Xuejun Liang

Spring 2021

Chapter 12: Limits of Algorithmic Computation

1. Some Problems That Cannot Be Solved By Turing Machines
    - Computability and Decidability
    - The Turing Machine Halting Problems
    - Reducing One Undecidable Problem to Another
2. Undecidable Problems for Recursively Enumerable Languages
3. The Post Correspondence Problem
4. Undecidable Problems for Context-Free Languages
5. A Question of Efficiency

Learning Objectives

At the conclusion of the chapter, the student will be able to:

1. Explain and differentiate the concepts of computability and decidability
2. Define the Turing machine halting problem
3. Discuss the relationship between the halting problem and recursively enumerable languages
4. Give examples of undecidable problems regarding Turing machines to which the halting problem can be reduced
5. Give examples of undecidable problems regarding recursively enumerable languages
6. Determine if there is a solution to an instance of the Post correspondence problem
7. Give examples of undecidable problems regarding context-free languages

Computability and Decidability

- Are there questions which are clearly and precisely stated, yet have no algorithmic solution?
- As stated in chapter 9, a function *f* is *computable* if there exists a Turing machine that computes the value of *f* for all arguments in its domain
- Since there may be a Turing machine that can compute *f* for part of the domain, it is crucial to define the domain of *f* precisely
- The concept of decidability applies to computations that result in a "yes" or "no" answer: a problem is *decidable* if there exists a Turing machine that gives the correct answer for every instance in the domain

The Turing Machine Halting Problem (1)

- The Turing machine *halting problem* can be stated as: Given the description of a Turing machine M and an input string w, does M, when started in the initial configuration q_0w, perform a computation that eventually halts?
- The domain of the problem is the set of all Turing machines and all input strings.
- Any attempts to simulate the computation on a universal Turing machine face the problem of not knowing if/when M has entered an infinite loop
- By Theorem 12.1, there does not exist any Turing machine that finds the correct answer in all instances; the halting problem is therefore undecidable

The Turing Machine Halting Problem (2)

- Definition 12.1 (The Halting Problem)

Let w_M be a string that describes a Turing machine M = (Q, Σ, Γ, δ, q_0, □, 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

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

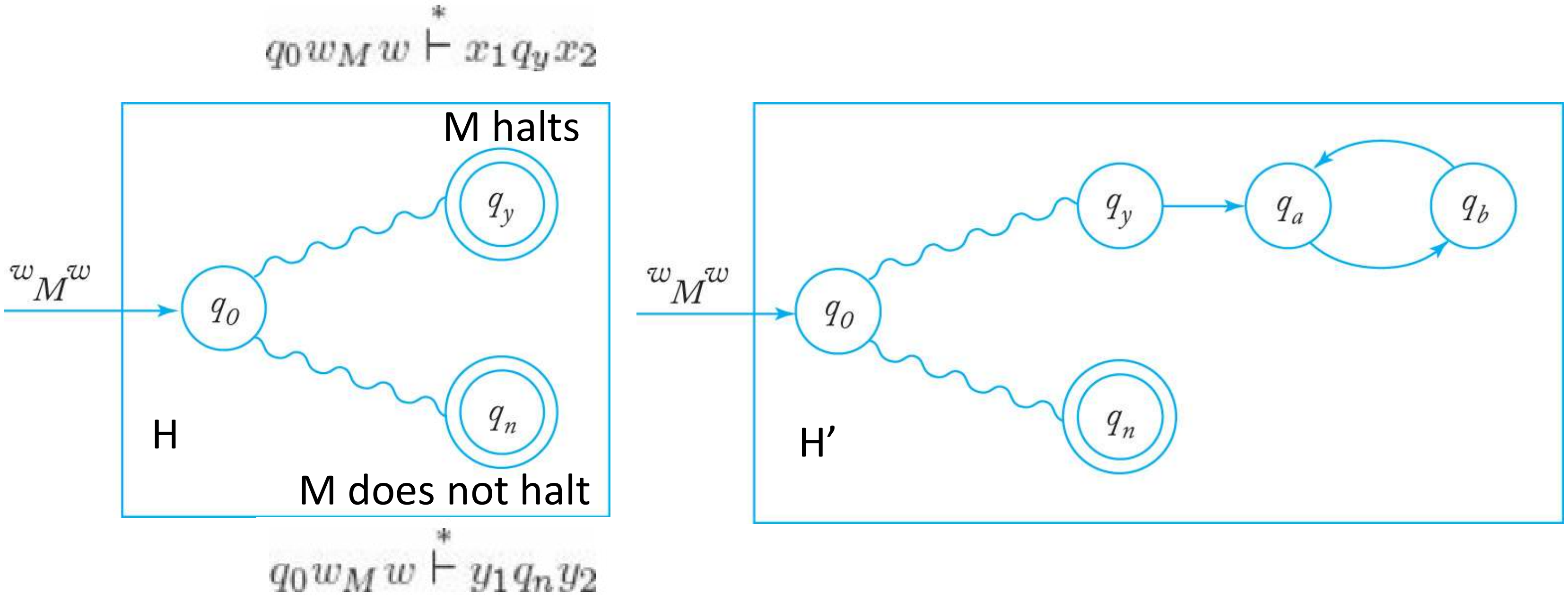
if M applied to w does not halt. Here q_y and q_n are both final states of H.

The Turing Machine Halting Problem (3)

- 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.

- Idea of Proof



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

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

The Turing Machine Halting Problem (4)

From H' we construct another Turing machine $\widehat{H}$.

$\widehat{H}$

$q_0 w_M \overset{*}{\vdash}_{\widehat{H}} q_0 w_M w_M \overset{*}{\vdash}_{\widehat{H}} \infty$ M halts if applied to w_M

$q_0 w_M \overset{*}{\vdash}_{\widehat{H}} q_0 w_M w_M \overset{*}{\vdash}_{\widehat{H}} y_1 q_n y_2$ M does not halt if applied to w_M

Now $\widehat{H}$ is a Turing machine, so it has a description in $\{0,1\}^*$, say, $\widehat{w}$.

$\widehat{H}$ is applied to $\widehat{w}$, identifying M with $\widehat{H}$, we get

$\widehat{H}$

$q_0 \widehat{w} \overset{*}{\vdash}_{\widehat{H}} \infty$ $\widehat{H}$ halts if applied to $\widehat{w}$

$q_0 \widehat{w} \overset{*}{\vdash}_{\widehat{H}} y_1 q_n y_2$ $\widehat{H}$ does not halt if applied to $\widehat{w}$

The Halting Problem and Recursively Enumerable Languages

Theorem 12.2 states that, if the halting problem were decidable, then every recursively enumerable language would be recursive

- Assume that L is a recursively enumerable language and M is a Turing machine that accepts L
- Let H be a Turing machine that solves the halting problem, then we can apply H to the accepting machine M (i.e. $w_M w$)
  - If H concludes that M does not halt, then w is not in L
  - If H concludes that M halts, then M will determine If w is in L
- Consequently, we would have a membership algorithm for L. This makes 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

Reducing One Undecidable Problem to Another

- A problem A is *reduced* to a problem B if the decidability of A follows from the decidability of B
- An example is the *state-entry problem*: given any Turing machine M and string w, decide whether or not the state q is ever entered when M is applied to w
- If we had an algorithm that solves the state-entry problem, it could be used to solve the halting problem
- However, because the halting problem is undecidable, the state-entry problem must also be undecidable

Example 12.1: Reduce the halting problem to the state-entry problem

- The state-entry problem (M, q, w)

  If the state q is ever entered when M is applied to w?
- Suppose that we have an algorithm A that solves the state-entry problem
- Given any M and w, modify M to get $\widehat{M}$ in such a way that $\widehat{M}$ halts in q if and only if M halts by doing
  - If $\delta(q_i, a)$ is undefined in M, define in $\widehat{M}$: $\delta(q_i, a) = (q, a, R)$, where q is a final state.
- Apply the state-entry algorithm A to ($\widehat{M}$, q, w)
  - 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.

Example 11.2: The Blank-Tape Halting Problem

Given a Turing machine M, determine whether or not M halts if started with a blank tape

- To show that the problem is undecidable,
- Given a machine M and input string w, construct from M and w a new machine M_w that starts with a blank tape, writes w on it, and acts like M
- Clearly, M_w will halt on a blank tape if and only if M halts on w
- If we start with M_w and apply the blank-tape halting problem algorithm to it, we would have an algorithm for the halting problem
- Since the halting problem is known to be undecidable, the same must be true for the blank-tape version

The Undecidability of the Blank-Tape Halting Problem

- Figure 12.3 illustrates the process used to establish the result that the blank-tape halting problem is undecidable
- After M_w is built, the presumed blank-tape halting problem algorithm would be applied to M_w, yielding an algorithm for the halting problem, which leads to a contradiction

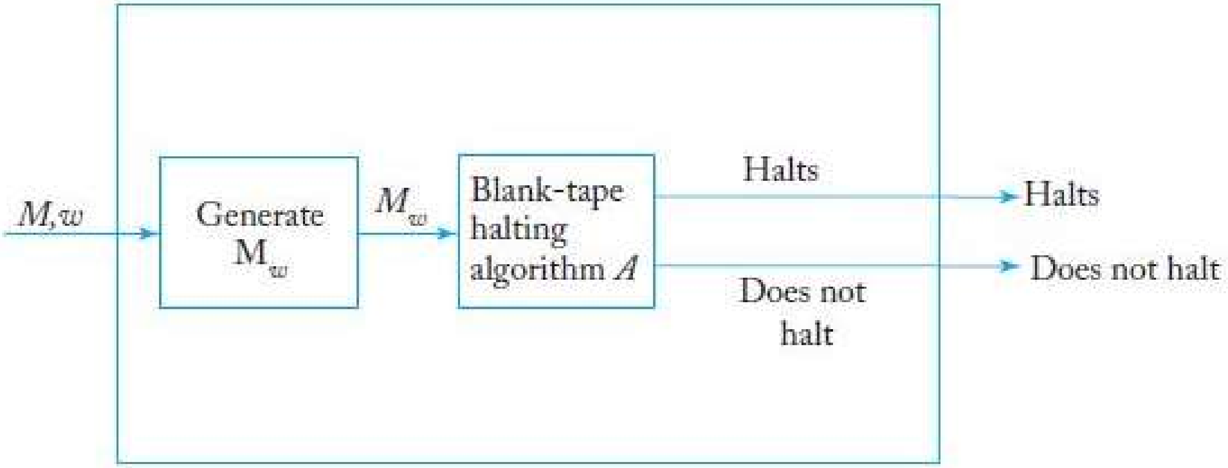


FIGURE 12.3 Algorithm for the halting problem.

Undecidable Problems for Recursively Enumerable Languages

- As illustrated before, there is no membership algorithm for recursively enumerable languages
- Recursively enumerable languages are so general that most related questions are undecidable
- Usually, there is a way to reduce the halting problem to questions regarding recursively enumerable languages, such as
  - Is the language generated by an unrestricted grammar empty?
  - Is the language accepted by a Turing machine finite?

Is the Language Generated by an Unrestricted Grammar Empty?

- Given an unrestricted grammar G, determine whether or not L(G) is empty
- To show that the problem is undecidable,
  - Given a Turing machine M and string w, modify M to create a new machine M_w, so that M_w saves its input on a special part of its tape, and then acts as M. Whenever M enters a final state, it accepts the input only if the input is equal to w. Clearly, $L(M_w) = L(M) \cap \{ w \}$,
  - Construct a grammar G_w that generates $L(M_w)$. So $L(G_w) = L(M_w)$ is nonempty *iff* $w \in L(M)$.
  - Assuming there is an algorithm A for deciding whether or not an arbitrary L(G) is empty, we could apply it to G_w, which would give us a membership algorithm for any recursively enumerable language
  - But this contradicts previous results that have established there is no such membership algorithm.

The Undecidability of the “L(G) = ∅” Problem

- Figure 12.5 illustrates the process used to establish the result that the “L(G) = ∅” problem is undecidable
- After G_w is built, the presumed emptiness algorithm A would be applied to G_w, giving a membership algorithm for recursively enumerable languages, which is impossible

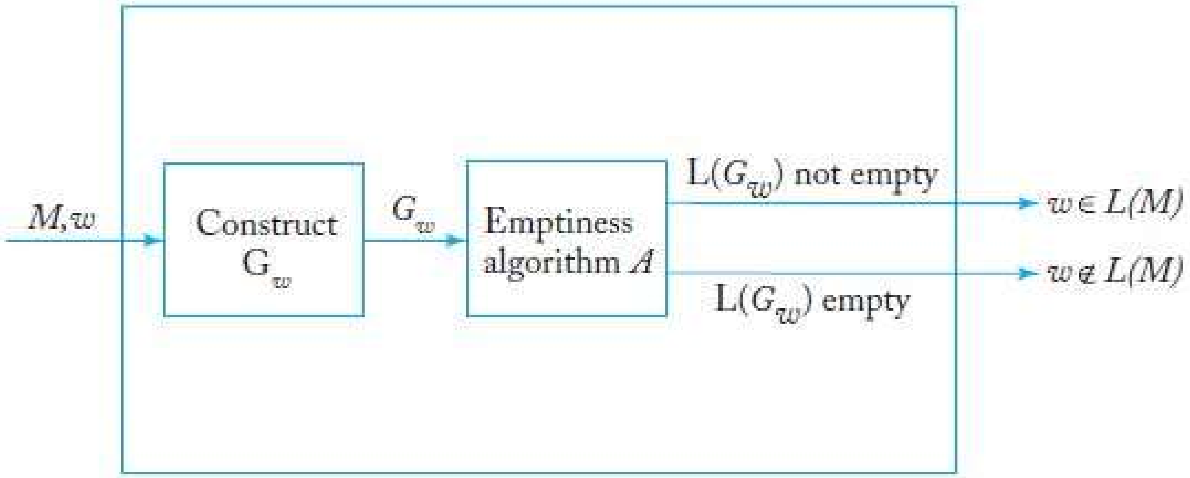


FIGURE 12.5 Membership algorithm.

Is the Language Accepted by a Turing Machine finite?

- Given a Turing machine M, determine whether or not L(M) is finite
- To show that the problem is undecidable,
  - Given a Turing machine M and string w, modify M to create a new machine $\widehat{M}$, as below.
    - $\widehat{M}$ generates w on an unused portion of its tape and perform the same computations as M starting with q_0w.
    - if M halts in any configuration, then $\widehat{M}$ halts in a final state and accepts all its inputs.
    - If M does not halt, then $\widehat{M}$ will not halt either.
  - As a result, $\widehat{M}$ either accepts $\varnothing$ or the infinite language Σ^+
  - Assuming there is an algorithm A for deciding whether or not L(M) is finite, we could apply it to $\widehat{M}$, which would give us a solution to the halting problem
  - But this contradicts previous results that have established that the halting problem is undecidable

The Undecidability of the “L(M) is Finite” Problem

- Figure 12.6 illustrates the process used to establish the result that the “L(M) is finite” question is undecidable
- After an algorithm generates $\widehat{M}$, the presumed finiteness algorithm A would be applied to $\widehat{M}$, resulting in a solution to the halting problem, which is impossible

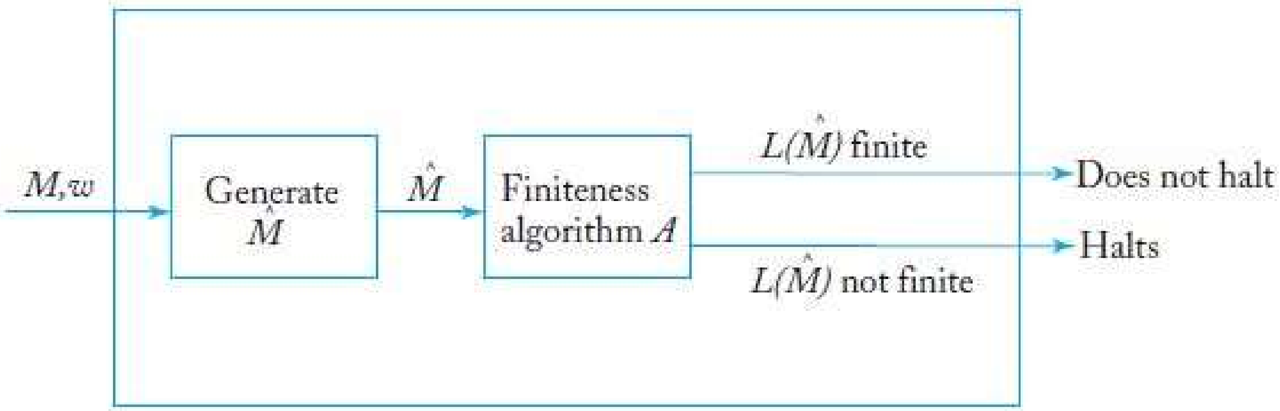


FIGURE 12.6

The Post Correspondence Problem

- Given two sequences of n strings on some alphabet Σ, for instance

  $A = w_1, w_2, \ldots, w_n$ and $B = v_1, v_2, \ldots, v_n$

  there is a Post correspondence solution (PC solution) for the pair (A, B) if there is a nonempty sequence of integers i, j, …, k, such that $w_i w_j \ldots w_k = v_i v_j \ldots v_k$
- As shown in Example 12.5, assume A and B consist of

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

  A PC solution for this instance of (A, B) exists, as shown below

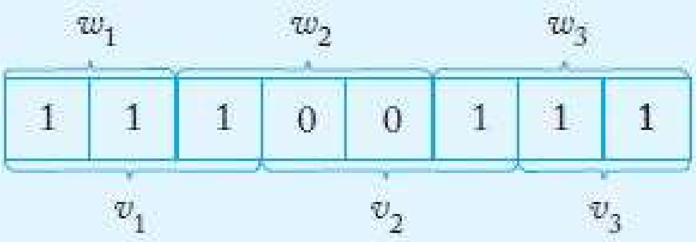


FIGURE 12.7

The Undecidability of the Post Correspondence Problem

- The Post correspondence problem is to devise an algorithm that determines, for any (A, B) pair, whether or not there exists a PC solution
- For example, there is no PC solution if A and B consist of $w_1 = 00$, w_2, $= 001$, $w_3 = 1000$ and $v_1 = 0$, v_2, $= 11$, $v_3 = 011$
- Theorem 12.7 states that there is no algorithm to decide if a solution sequence exists under all circumstances, so the Post correspondence problem is undecidable
- Although a proof of theorem 12.7 is quite lengthy, this very important result is crucial for showing the undecidability of various problems involving context-free languages

Undecidable Problems for Context-Free Languages

- The Post correspondence problem is a convenient tool to study some questions involving context-free languages
- The following questions, among others, can be shown to be undecidable
  - Given an arbitrary context-free grammar G, is G ambiguous?
  - Given arbitrary context-free grammars G_1 and G_2, is $L(G_1) \cap L(G_2) = \varnothing$?
  - Given arbitrary context-free grammars G_1 and G_2, is $L(G_1) = L(G_2)$?
  - Given arbitrary context-free grammars G_1 and G_2, is $L(G_1) \subseteq L(G_2)$?