CHAPTER 4
Decidability

Contents

• Decidable Languages
  • decidable problems concerning regular languages
  • decidable problems concerning context-free languages

• The Halting Problem
  • The diagonalization method
  • The halting problem is undecidable
  • A Turing unrecognizable languages

Decidability (intro.)

• We have introduced Turing machines as a model of a general purpose computer
• We defined the notion of algorithm in terms of Turing machines by means of the Church-Turing thesis
• In this chapter we
  • begin to investigate the power of algorithms to solve problems
  • demonstrate certain problems that can be solved algorithmically and others that cannot
• Our objective is to explore the limits of algorithmic solvability
• Why should we study unsolvability? Showing that a problem is unsolvable doesn’t appear to be of any use if we have to solve it. But …
• We need to study this phenomenon for two reasons:
  • First, knowing that a problem is algorithmically unsolvable is useful because then you realize that the problem must be simplified or altered before you can find an algorithmic solution.
  • The second reason is cultural. Even if you deal with problems that clearly are solvable, a glimpse of the unsolvable can stimulate your imagination and help you gain an important perspective on computation.
Decidable Languages

• In this section we give some examples of languages that are decidable by algorithms.
• For example, we present an algorithm which tests whether a string is a member of a context-free language.
• This problem is related to the problem of recognizing and compiling programs in a programming language.

Decidable Problems Concerning Regular Languages

• We begin with certain computation problems concerning finite automata.
• We give algorithms for testing whether a finite automaton accepts a string, whether the language of a finite automaton is empty, and whether two finite automata are equivalent.
• For convenience we use languages to represent various computational problems.
• For example, the acceptance problem for DFAs of testing whether a particular finite automaton accepts a given string can be expressed as a language, $A_{\text{DFA}}$.

$$A_{\text{DFA}} = \{<B,w>: B \text{ is a DFA that accepts input string } w\}.$$ 

• The problem of testing whether a DFA $B$ accepts an input $w$ is the same as the problem of testing whether $<B,w>$ is a member of the language $A_{\text{DFA}}$.
• Similarly, we can formulate other computational problems in terms of testing membership in a language. Showing that a language is decidable is the same as showing that the computation problem is decidable (= algorithmically solvable).

The Acceptance Problem for DFAs is Decidable

**Theorem 1** $A_{\text{DFA}}$ is a decidable language.

• We present a TM $M$ that decides $A_{\text{DFA}}$.

$M =$ “on input $<B,w>$, where $B$ is a DFA and $w$ is a string:

1. Simulate $B$ on input $w$.
2. If the simulation ends in an accept state, accept. If it ends in a non-accepting state, reject.“

A few implementation details:

• The input is $<B,w>$. It is a representation of a DFA $B$ together with a string $w$. One reasonable representation of $B$ is a list of its five components, $Q, \Sigma, \delta, q_0, F$.
• When $M$ receives its input, $M$ first checks on whether it properly represents a DFA $B$ and a string $w$. If not, it rejects.
• Then $M$ carries out the simulation in a direct way. It keeps track of $B$’s current state and $B$’s current position in the input $w$.
• Initially, $B$’s current state is $q_0$ and $B$’s current position is the leftmost symbol of $w$.
• The states and position are updated according to the specified transition function $\delta$.
• When $M$ finishes processing the last symbol of $w$, $M$ accepts if $B$ is in an accepting state; $M$ rejects if $B$ is in a non-accepting state.
The Acceptance Problem for NFAs and REXs.

We can prove similar result for NFAs and Regular Expressions.

\[ A_{NFA} = \{ <B,w> : B \text{ is a NFA that accepts input string } w \} \]

**Theorem 2:** \( A_{NFA} \) is a decidable language.

\( N = \) “on input \(<B,w>\), where \( B \) is a NFA and \( w \) is a string:

1. Convert NFA \( B \) to an equivalent DFA \( C \) using the procedure for this conversion given in Theorem “subset construction”.
2. Run TM \( M \) from Theorem 1 on input \(<C,w>\).
3. If \( M \) accepts, accept, otherwise reject.”

Running TM \( M \) in stage 2 means incorporating \( M \) into the design of \( N \) as a subprocedure.

\[ A_{REX} = \{ <R,w> : R \text{ is a regular Expression that generates string } w \} \]

**Theorem 3:** \( A_{REX} \) is a decidable language.

\( P = \) “on input \(<R,w>\), where \( R \) is a reg.expr. and \( w \) is a string:

1. Convert \( R \) to an equivalent DFA \( C \) using the procedure for this conversion given in Theorem earlier.
2. Run TM \( M \) from Theorem 1 on input \(<C,w>\).
3. If \( M \) accepts, accept, otherwise reject.”

The Emptiness Problem for the Language of a Finite Automaton.

\[ E_{DFA} = \{ <A> : A \text{ is a DFA and } L(A) = \emptyset \} \]

**Theorem 4:** \( E_{DFA} \) is a decidable language.

- A DFA accepts some string if and only if reaching an accept state from the start state by traveling along the arrows of the DFA is possible.
- To test this condition we can design a TM \( T \) that uses marking algorithm similar to that used in example “connectedness of a graph”.

\( T = \) “on input \(<A>\), where \( A \) is a DFA :

1. Mark the start state of \( A \).
2. Repeat the following stage until no new states get marked:
3. Mark any state that has a transition coming into it from any state that is already marked.
4. If no accept state is marked, accept; otherwise reject.”
**The Equivalence Problem for Finite Automata.**

\[ EQ_{DFA} = \{ < A, B > : A \text{ and } B \text{ are DFAs and } L(A) = L(B) \}. \]

**Theorem 5:** \( EQ_{DFA} \) is a decidable language.

- Consider a symmetric difference of \( L(A) \) and \( L(B) \), i.e, a language \( L(C) \)
  \[ L(C) = (L(A) \cap L(B)) \cup (\overline{L(A)} \cap L(B)). \]
- Hence, \( L(C) = \emptyset \) if and only if \( L(A) = L(B) \).
- We can construct \( C \) from \( A \) and \( B \) with the constructions for proving the class of regular languages closed under complementation, union, and intersection.
- These constructions are algorithms that can be carried out by Turing machines.

---

**Decidable Problems Concerning CFLs**

- Here we describe algorithms to test whether a CFG generates a particular string and to test whether the language of a CFG is empty.
- Let \( A_{CFG} = \{ < G, w > : G \text{ is a CFG that generates string } w \} \).

**Theorem 6:** \( A_{CFG} \) is a decidable language.

- For CFG \( G \) and string \( w \) we want to test whether \( G \) generates \( w \).
- One idea is to use \( G \) to go through all derivations to determine whether any is a derivation of \( w \). This idea doesn’t work, as infinitely many derivations may have to be tried. If \( G \) does not generate \( w \), this algorithm would never halt. Hence this idea gives a TM which is recognizer, not a decider.
- To make this TM into a decider we need to ensure that the algorithm tries only finite many derivations.
- If \( G \) is in Chomsky normal form, any derivation of \( w \) has \( 2n-1 \) steps, where \( n \) is the length of \( w \). Only finite many such derivations exist.
- We present a TM \( S \) that decides \( A_{CFG} \).

\( S = " \text{on input } < G, w >, \text{ where } G \text{ is a CFG and } w \text{ is a string:} ]

1. Convert \( G \) to an equivalent grammar in Chomsky normal form.
2. List all derivations with \( 2n-1 \) steps, where \( n \) is the length of \( w \), except if \( n=0 \), then instead list all derivations with 1 step.
3. If any of these derivations generate \( w \), accept; if not, reject."
Decidable Problems Concerning CFLs (cont.)

• Here we describe an algorithm to test whether the language of a CFG is empty.
• Let \( E_{\text{CFG}} = \{ <G> : G \text{ is a CFG and } L(G) = \emptyset \} \).

**Theorem 7:** \( E_{\text{CFG}} \) is a decidable language.

- For CFG \( G \) we need to test whether the start variable can generate a string of terminals.
- The algorithm does so by solving a more general problem. It determines for each variable whether that variable is capable of generating a string of terminals.
- When the algorithm has determined that a variable can generate some string of terminals, the algorithm keeps track of this information by placing a mark on that variable. First the algorithm marks all terminal symbols in the grammar.
- Then it scans all the rules of the grammar. If it ever finds a rule that permits some variable to be replaced by some string of symbols all of which are already marked, the algorithm knows that this variable can be marked, too.
- The algorithm continues in this way until it cannot mark any additional variables. The TM \( R \) implements this algorithm.

\[ R = \text{"on input } <G>, \text{ where } G \text{ is a CFG:} \]

1. Mark all terminals in \( G \). Repeat (2) until no new variables get marked:
2. Mark any variable \( A \) where \( G \) has a rule \( A \rightarrow U_1U_2\ldots U_k \) and each symbol \( U_1, U_2, \ldots, U_k \) has already been marked.
3. If the start symbol is not marked, accept; otherwise reject. “

• We can prove now the following.

**Theorem 8:** Every CFL is decidable.

• Let \( A \) be a CFL and \( G \) be a CFG for \( A \).

• Here is a TM \( M(G) \) that decides \( A \).
• We build a copy of \( G \) into \( M(G) \).
• \( S \) is a TM from Theorem 6.

\[ M(G) = \text{"on input } w:\]  

1. Run TM \( S \) on input \( <G, w> \)
2. If this machine accepts, accept; if it rejects, reject. “

Theory of Computation, Feodor F. Dragan, Kent State University