Decidability

- The power of algorithms to solve problems. The main objective of decidability is to explore the limits of algorithmic solvability.
- Why study decidability?
  1. To know when a problem is algorithmically unsolvable because that’s when you know the problem needs to be simplified or altered before you can find an algorithmic solution.
  2. The other reason is more cultural, a glimpse of the unsolvable can help gain an important perspective on computation and will better the understanding.

Decidability with Regular Languages

We chose to represent various computational problems by languages. Doing so is convenient because we have already set up terminology for dealing with languages. For example, the acceptance problem for DFAs of testing whether a deterministic finite automaton accepts a given string can be expressed as a language, \(A_{DFA}\). This language contains the encodings of all DFAs together with strings that the DFAs accept.

\[ A_{DFA} = \{ (B, w) \mid B \text{ is a DFA that accepts input string } w \}. \]

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 \( A_{DFA} \). Similarly, we can formulate other computational problems in terms of testing membership in a language. Showing that the language is decidable is the same as showing that the computational problem is decidable.

**Theorem 4.1**

\( A_{DFA} \) is a decidable language.

We simply need to present a TM \( M \) that decides \( A_{DFA} \).

\( M = \) “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 nonaccepting state, \( reject \).”

Simulate the \( B \) on input \( w \)

If the simulation ends in an accept state, \( accept \); otherwise, \( reject \);

We can prove a similar theorem for nondeterministic finite automata. Let

\[ A_{NFA} = \{ (B, w) \mid B \text{ is an NFA that accepts input string } w \}. \]

**Theorem 4.2**
$A_{NFA}$ is a decidable language.

$N = \text{"On input } (B, w), \text{ where } B \text{ is an NFA and } w \text{ is a string:} \newline
1. \text{ Convert NFA } B \text{ to an equivalent DFA } C, \text{ using the procedure for this conversion given in Theorem 1.39.} \newline
2. \text{ Run TM } M \text{ from Theorem 4.1 on input } (C, w). \newline
3. \text{ If } M \text{ accepts, accept; otherwise, reject."} \newline

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

**Theorem 4.3**

$A_{REX}$ is a decidable language.

**Proof** The following TM $P$ decides $A_{REX}$.

$P = \text{"On input } (R, w), \text{ where } R \text{ is a regular expression and } w \text{ is a string:} \newline
1. \text{ Convert regular expression } R \text{ to an equivalent NFA } A \text{ by using the procedure for this conversion given in Theorem 1.54.} \newline
2. \text{ Run TM } N \text{ on input } (A, w). \newline
3. \text{ If } N \text{ accepts, accept; if } N \text{ rejects, reject."} \newline

**Theorem 4.4**

$L_{DFA}$ is a decidable language.

**Proof** A DFA accepts some string if reaching an accept state from the start state by travelling along arrow of the DFA is possible. To test this condition, we can design a TM $T$:

$T = \text{"On input } (A), \text{ where } A \text{ is a DFA:} \newline
1. \text{ Mark the start state of } A. \newline
2. \text{ Repeat until no new states get marked:} \newline
3. \text{ Mark any state that has a transition coming into it from any state that is already marked.} \newline
4. \text{ If no accept state is marked, accept; otherwise, reject."} \newline

**Theorem 4.5**

$EQ_{DFA}$ is a decidable language.

**Proof** To prove this theorem, we use Theorem 4.4. We construct a new DFA $C$ from $A$ and $B$, where $C$ accepts only those strings that are accepted by either $A$ or $B$ but not by both. Thus, if $A$ and $B$ recognize the same language, $C$ will accept nothing. The language of $C$ is

$$L(C) = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B)).$$

This expression is sometimes called the symmetric difference of $L(A)$ and $L(B)$.
if \( L(C) \) is empty, \( L(A) \) and \( L(B) \) are equal

\[
F = \text{"On input } (A, B), \text{ where } A \text{ and } B \text{ are DFAs:} \\
1. \text{ Construct DFA } C \text{ as described.} \\
2. \text{ Run TM } T \text{ from Theorem 4.4 on input } (C). \\
3. \text{ If } T \text{ accepts, } \text{accept.} \text{ If } T \text{ rejects, reject."}
\]

**Theorem 4.7**

\( A_{CFG} \) is a decidable language.

**Proof** The TM \( S \) for \( A_{CFG} \) follows.

\( S = \text{"On input } (G, w), \text{ where } G \text{ is a CFG and } w \text{ is a string:} \\
1. \text{ Convert } G \text{ to an equivalent grammar in Chomsky normal form.} \\
2. \text{ List all derivations with } 2n - 1 \text{ steps, where } n \text{ is the length of } w; \\
\text{ except if } n = 0, \text{ then instead list all derivations with one step.} \\
3. \text{ If any of these derivations generate } w, \text{ accept; if not, reject."}

The problem of determining whether a CFG generates a string is related to a problem of compiling programming languages. The Algorithm in TM \( S \) is very inefficient and would never be used in practice.

**Theorem 4.8**

\[
E_{CFG} = \{ (G) \mid G \text{ is a CFG and } L(G) = \emptyset \}.
\]

- Can’t test infinite number of strings
- Track whether the start variable ever generate any terminals.

\( E_{CFG} \) is a decidable language.

**Proof**

\( R = \text{"On input } (G), \text{ where } G \text{ is a CFG:} \\
1. \text{ Mark all terminal symbols in } G. \\
2. \text{ Repeat until no new variables get marked:} \\
3. \text{ Mark any variable } A \text{ where } G \text{ has a rule } A \rightarrow U_1 U_2 \ldots U_k \text{ and} \\
\text{ each symbol } U_1, \ldots, U_k \text{ has already been marked.} \\
4. \text{ If the start variable is not marked, } \text{accept; otherwise, reject."}
\)