Chapter 4: Decidability

Decidability

A Language L is **Turing-decidable** if there is a TM M that decides it: M accepts every string in L and rejects every string in $\overline{L}$.

A ***recursive*** language is a decidable language.

Recognizability

A Language L is **Turing-recognizable** if there is a TM M that recognizes it: M accepts every string in L and either rejects or fails to halt on every string in $\overline{L}$.

A ***recursively enumerable*** language is a recognizable language.

Co-Recognizability

A Language L is **co-recognizable** if there is a TM M that recognizes $\overline{L}$: M accepts every string in $\overline{L}$ and either rejects or fails to halt on every string in L.

A **co-r.e.** language is a co-recognizable language.

Decidability of Languages related to Finite Automata

The following are all Turing decidable:

1. A_{DFA} = {<M, w> | M is a DFA that accepts string w}
2. A_{NFA} = {<M, w> | M is a NFA that accepts string w}
3. A_{REX} = {<M, w> | M is a regular expression that generates string w}
4. E_{DFA} = {<M> | M is a DFA that satisfies L(M) = Φ}
5. ALL_{DFA} = {<M> | M is a DFA that satisfies L(M) = $\sum^*$ }
6. EQ_{DFA} = {<M, M'> | M and M' are two DFAs that satisfy L(M) = L(M') }

A_{DFA} = {<M, w> | M is a DFA that accepts w}

We simply need to present a TM T that decides A_{DFA} .

T = “On input <M, w>, where M is a DFA and w is a string:

1. Simulate M on input w.
2. If the simulation ends in an accept state, *accept*. If it ends in a non-accepting state, *reject*.”

- When T receives its input, T first determines whether it properly represents a DFA M and a string w. If not, T rejects.

A_{DFA} = {<M, w> | DFA M accepts w} (contd.)

- Then T carries out the simulation directly. It keeps track of M's current state and current position in the input w by writing this information down on its tape. The state and position are updated according to the transition function.
- When T finishes processing the last symbol of w, T accepts the input if it is in an accepting state; T rejects the input if it is in a non-accepting state.

A_{NFA} = {<M, w> | NFA M accepts w}

We can use the previous machine T as a subroutine.

TM N = "On input <M, w>, where M is a NFA, and w is a string:

1. Convert NFA M to an equivalent DFA C using the procedure for this conversion given in Theorem 1.39.
2. Run TM T from previous slide on input <C, w>.
3. If T accepts, *accept*; otherwise *reject*."

– Running TM T in stage 2 means incorporating T into the design of N as a sub-procedure.

A_{REX} = {<M, w> | M is a regular expression that generates string w}

TM *P* = "On input <M, w>, where M is a regular expression, and w is a string:

1. Convert regular expression M to an equivalent NFA A using the procedure for this conversion given in Theorem 1.54.
2. Run TM N from previous slide on input <A, w>.
3. If N accepts, *accept*; otherwise *reject*."

E_{DFA} = {<M> | DFA M satisfies $L(M) = \Phi$}

- A DFA accepts some string iff reaching an accept state from the start state by traveling along the arrows of the DFA is possible.
- TM T = “On input <M>, where M is a DFA
  1. Mark the start state of M.
  2. Repeat 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*.”

ALL_{DFA} = {<M> | DFA M satisfies $L(M) = \Sigma^*$ }

- Given DFA M, construct its complementary DFA M' such that $L(M') = \Sigma^* - L(M)$.
- Then ask whether $L(M') = \Phi$ (i.e. use the TM for E_{DFA}). If $L(M') = \Phi$, then $L(M) = \Sigma^*$; otherwise not.

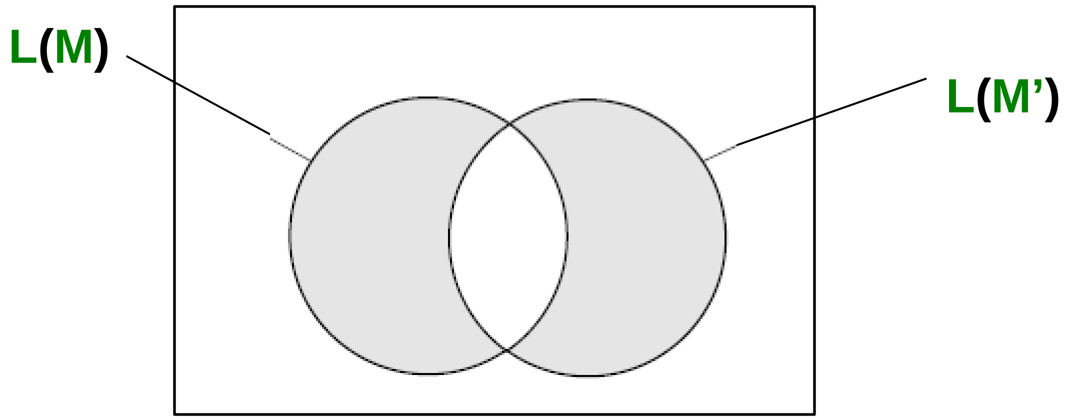
$EQ_{DFA} = \{\langle M, M' \rangle \mid$ DFAs M and M' satisfy $L(M) = L(M')\}$

- We use the proof for E_{DFA} to prove this theorem.
- We construct a new DFA C from M and M', where C accepts only those string that are accepted by either M or M' but not both. Thus, if M and M' accept the same language, C will accept nothing. The language of C is

$$L(C) = (L(M) \cap \overline{L(M')}) \cup (\overline{L(M)} \cap L(M'))$$

This expression is sometimes called symmetric difference of $L(M)$ and $L(M')$.

EQ_{DFA} = {<M, M'> | DFAs M and M' satisfy L(M) = L(M’) } (contd.)



F = “On input <M, M'>, where M and M‘ are DFAs:

1. Construct DFA C as described.
2. Run TM T from E_{DFA} on input <C>.
3. If T accepts, accept. If T rejects, reject.”

Turing Decidability of languages related to CFLs

1. A_{CFG} = {< G, w > | CFG G generates w}.
2. A_{PDA} = {<M, w> | PDA M accepts w}.
3. E_{CFG} = {<G> | CFG G satisfies L(G) = Φ}.

The following are undecidable:

4. ALL_{CFG} = {<G> | CFG G satisfies L(G) = ∑* }.
5. EQ_{CFG} = {<G, G'> | CFGs G and G' satisfy L(G) = L(G')}.
6. A_{TM} = {<M, w> | TM M accepts w}.

$A_{CFG} = \{< G, w > \mid \text{CFG } G \text{ generates } w\}$

- One idea is to use G to go through all derivations to determine whether any is a derivation of *w*. This idea does not work, as infinitely many derivations may have to be tried.
- This idea gives a Turing Machine that is a recognizer, but not a decider, for A_{CFG}.
- To make a machine that is a decider we need to ensure that algorithm tries only finitely many derivations.

A_{CFG} = {< G, w > | CFG G generates w} (contd.)

- TM S = "On input <G, w>, where G is a CFG and w 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*."

A_{PDA} = {<M, w> | PDA M accepts w}

- Let M be a PDA and G be a CFG for CFL A. Now design TM R that decides A_{PDA}. G can be obtained via the conversion from a PDA to an equivalent CFG that is discussed in Chapter 2.
- R = "On input <M, w>, where M is a PDA and w is a string:

    0. Construct the CFG G that is equivalent to M.
    1. Run TM S (that decides A_{CFG}) on input <G, w>
    2. If S accepts, *accept*; if S rejects, *reject*. "

E_{CFG} = {<G> | CFG G satisfies $L(G) = \Phi$}

- We can not use the TM S for A_{CFG}, because the algorithm might try going through all possible w’s, one by one. But there are infinitely many w’s to try, so this method could end up running forever.
- We need to test whether the start variable can generate a string of terminals.
- It determines for each variable whether that variable is capable of generating a string of terminals.

E_{CFG} = {<G> | CFG G satisfies $L(G) = \Phi$} (contd.)

TM R = "On input <G>, where G is a CFG

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

Proof for $A_{TM} = \{<M, w> \mid M \text{ accepts } w\}$

- Suppose that H is a decider for A_{TM}.
- On input <M, w>, where M is a TM and w is a string, H halts and accepts if M accepts w, and H halts and rejects if M rejects w.

$$H(<M, w>) = \begin{cases} \text{accept} & \text{if } M \text{ accepts } w \\ \text{reject} & \text{if } M \text{ rejects } w. \end{cases}$$

- Now we construct a new TM D with H as a subroutine. D calls H to determine what M does when input to M is its own description <M>. Once D has determined this information, it does the opposite.

Proof for A_{TM} (contd.)

D = “On input <M>, where M is a TM:

1. Run H on input <M, <M>>.
2. Output the opposite of what H outputs; that is, if H accepts, *reject*; and if H rejects, *accept*.”

$$D(<M>) = \begin{cases} \text{accept} & \text{if } M \text{ does not accept } <M> \\ \text{reject} & \text{if } M \text{ accepts } <M>. \end{cases}$$

What happens when we run D with its own description <D> as input? In that case, we get

$$D(<D>) = \begin{cases} \text{accept} & \text{if } D \text{ does not accept } <D> \\ \text{reject} & \text{if } D \text{ accepts } <D>. \end{cases}$$

Proof for A_{TM} (contd.)

- Let's review the steps of this proof:
  - Assume that a TM H decides A_{TM}.
  - Then use H to build a TM D that takes an input <M>, where D accepts its input <M> exactly when M does not accept input <M>.
  - Finally, run D on itself. No matter what D does, it is forced to do the opposite. Thus, neither TM D nor TM H can exist.
- The machine take the following actions, with the last line being the contradiction.
  - H accepts <M, w> exactly when M accepts w.
  - D rejects < M > exactly when M accepts < M >.
  - D rejects < D > exactly when D accepts < D >.

Reference

- www.cs.uh.edu/~rmverma by Dr. Rakesh Verma.
- www.cs.unm.edu/~gemmell/500.html by Dr. Peter Gemmell.