CSE396 Thu. April 15: Decidability and Undecidability

This is the first of numerous problems in which the instance type is "Just a Machine."

**NE\textsubscript{DFA}:**

INST: (The string code \(\langle M \rangle\) of) A DFA \(M = (Q, \Sigma, \delta, s, F)\).

QUES: Is \(L(M) \neq \emptyset\)?

The QUESTION is worded oppositely from the text's wording of \(E\textsubscript{DFA}\), which we'll come to. Here is an efficient decision procedure:

1. On input \(\langle M \rangle\), treat \(M\) as a directed graph without caring about the character labels on arcs.
2. Execute a breadth-first search in that graph from the start node \(s\) of (the graph of) \(M\).
3. If the search terminates having visited at least one state in \(F\), accept \(\langle M \rangle\), else reject.

The BFS in step 2 terminates—indeed, in time \(O(m^2)\) at worst since the graph has \(m\) nodes. [Well, it has \(O(m)\) edges, so you can get better time with random access to good data structures.] The procedure is correct because if BFS finds a path from \(s\) to a state \(q\) in \(F\), then the chars along that path form a string in \(L(M)\), so \(L(M) \neq \emptyset\).

The complementary problem ("E" for emptiness) is:

**E\textsubscript{DFA}:**

INST: A DFA \(M = (Q, \Sigma, \delta, s, F)\).

QUES: Is \(L(M) = \emptyset\)?

The solution is to use the same decision procedure, but switch the "accept" and "reject" cases:

1. On input \(\langle M \rangle\), treat \(M\) as a directed graph without caring about the character labels on arcs.
2. Execute a breadth-first search in that graph from the start node \(s\) of (the graph of) \(M\).
3. If the search terminates having visited at least one state in \(F\), reject \(\langle M \rangle\), else accept.

The corresponding problems for NFAs are just as easy: they have the same algorithms:

**NE\textsubscript{NFA}:**

INST: An NFA \(N = (Q, \Sigma, \delta, s, F)\).

QUES: Is \(L(N) \neq \emptyset\)?

**Solution:**

1. On input \(\langle N \rangle\), treat \(N\) as a directed graph without caring about the character labels on arcs.
2. Execute a breadth-first search in that graph from the start node \(s\) of (the graph of) \(N\).
3. If the search terminates having visited at least one state in \(F\), accept \(\langle N \rangle\), else reject.
This is BFS explicitly in the graph of $N$ with node set $Q$. It is not the same as the BFS used to convert an NFA into a DFA, which ran implicitly on the power set $2^Q$ of $Q$. Also "the same" is:

$$E_{NFA}:$$
INST: An NFA $N = (Q, \Sigma, \delta, s, F)$.
QUES: Is $L(N) = \emptyset$?

Solution: run the decision procedure for $NE_{NFA}$ but interchange the yes/no answers.

Now we consider a different kind of complementation:

$$ALL_{DFA}:$$
INST: A DFA $M = (Q, \Sigma, \delta, s, F)$.
QUES: Is $L(M) = \Sigma^*$?

Solution:
1. On input $\langle M \rangle$, form the complementary DFA $M' = (Q, \Sigma, \delta, s, F')$ with $F' = Q \setminus F$.
2. Feed $\langle M' \rangle$ to the decision procedure for $E_{DFA}$. Works because $L(M') = \sim L(M)$, so $L(M) = \Sigma^* \iff L(M') = \emptyset$.
3. If that procedure accepts $\langle M' \rangle$, then accept $\langle M \rangle$, else reject $\langle M \rangle$.

This embodies what in Chapter 5 we will call a mapping reduction from $ALL_{DFA}$ to $E_{DFA}$. The reduction and the whole procedure are correct because $L(M) = \Sigma^* \iff L(M') = \emptyset$.

This is not the same as the way we complemented $NE_{DFA}$ to $E_{DFA}$, and the best way to see why it's not so simple is to consider the analogous problem for NFAs.

$$ALL_{NFA}:$$
INST: An NFA $N = (Q, \Sigma, \delta, s, F)$.
QUES: Is $L(N) = \Sigma^*$?

We can solve this by converting $N$ into an equivalent DFA $M$ and running the decider for $ALL_{DFA}$ on $\langle M \rangle$. But that can take exponential time. Can we use the same idea as for $ALL_{DFA}$ of reducing to the corresponding emptiness problem, $E_{NFA}$, which we solved just as efficiently as for $E_{DFA}$? The problem is that we can't directly complement an NFA. Surely some other idea can help? The fact is, this problem is NP-hard. Nobody (on Earth) knows a polynomial-time algorithm, and most (on Earth) believe that no such algorithm exists.

Two-Machine Problems
Here the input \( w \) has type "Two Machines", meaning a pair \( \langle M_1, M_2 \rangle \). If the input \( w \) does not have this pair form, it is rejected to begin with.

**EQ\text{\_DFA}:**

**INST:** Two DFAs \( M_1 = (Q_1, \Sigma, \delta_1, s_1, F_1) \) and \( M_2 = (Q_2, \Sigma, \delta_2, s_2, F_2) \).

**QUES:** Is \( L(M_1) = L(M_2) \)?

The fact that gives an efficient decision procedure is that two sets \( A \) and \( B \) are equal if and only if their symmetric difference \( A \Delta B = (A \setminus B) \cup (B \setminus A) = (A \cup B) \setminus (A \cap B) \) is empty. The symmetric difference is often written \( A \oplus B \), with \( \oplus \) also used to mean XOR. Thus if we apply the Cartesian product construction to \( M_1 \) and \( M_2 \) with XOR as the operation, to produce a DFA \( M_3 \), then the answer is yes if and only if \( L(M_3) = \emptyset \).

**Solution:**

1. Decode a given input string \( w = \langle M_1, M_2 \rangle \) into DFAs \( M_1 \) and \( M_2 \). (If \( w \) does not have that form, reject.)
2. Create the Cartesian product DFA \( M_3 = (Q_3, \Sigma, \delta_3, s_3, F_3) \) with \( Q_3 = Q_1 \times Q_2 \) and \( F_3 = \{ (q_1, q_2) : q_1 \in F_1 \text{ XOR } q_2 \in F_2 \} \).
3. Feed \( \langle M_3 \rangle \) to the decision procedure for \( E\text{\_DFA} \), and accept \( \langle M_1, M_2 \rangle \) if and only if that accepts \( \langle M_3 \rangle \).

If \( m \) is the maximum of the number of states in \( Q_1 \) and in \( Q_2 \), then step 2 runs in \( O(m^2) \) time (ignoring the \( \log m \) length of state labels). Step 3 is run on a quadratically bigger machine, so its own quadratic time becomes \( O(m^4) \) overall, but that's AOK---still polynomial in \( m \). But how about:

**EQ\text{\_NFA}:**

**INST:** Two NFAs \( N_1 = (Q_1, \Sigma, \delta_1, s_1, F_1) \) and \( N_2 = (Q_2, \Sigma, \delta_2, s_2, F_2) \).

**QUES:** Is \( L(N_1) = L(N_2) \)?

We can get a decision procedure by converting the NFAs into DFAs \( M_1 \) and \( M_2 \) and testing whether \( L(M_1) = L(M_2) \). For decidability purposes, that is all we need to say, but it is inefficient. Can't we apply the Cartesian product idea directly to \( N_1 \) and \( N_2 \)? If the operation is intersection or union, this makes a good self-study question, but for difference or symmetric difference/XOR, there is a clear reason for doubt: If we could solve \( EQ\text{\_NFA} \) efficiently in general, then we could solve it efficiently in cases where \( N_2 \) is a fixed NFA that accepts all strings. Then we would have:

\[
\langle N_1, N_2 \rangle \in EQ\text{\_NFA} \iff \langle N_1 \rangle \in ALL\text{\_NFA}.
\]

But we have already asserted above that \( ALL\text{\_NFA} \) is \textbf{NP}-hard. So this blocks the attempt to solve \( EQ\text{\_NFA} \), and in fact, this shows that the \( EQ\text{\_NFA} \) problem is \textbf{NP}-hard as well.
One can define all these problems when the givens are regular expressions or GNFAs rather than DFAs or NFAs. The Sipser naming scheme will write the problems as \( EQ_{\text{Regexp}}, A_{\text{GNFA}}, ALL_{\text{Regexp}}, NE_{\text{GNFA}} \), and so on. They are all \textbf{decidable} because regular expressions and GNFAs are convertible to NFAs and DFAs, but not always efficiently to the latter. Regular expressions and NFAs convert to and from each other especially efficiently, and so the problems subscripted "\text{Regexp}" have much the same status as those subscripted "\text{NFA}". When we extend the problems to context-free grammars, pushdown automata, and general (deterministic) Turing machines, however, we will "lose" a lot more.

**Problems Involving Grammars**

Again, let's "accentuate the positive" and start with the nonemptiness problem rather than the emptiness problem.

\( NE_{\text{CFG}} \):

\[ \text{INST: A CFG } G = (V, \Sigma, R, S). \]

\[ \text{QUES: Is } L(G) \neq \emptyset? \]

The following pseudocode to solve the problem is strongly analogous to breadth-first search:

```plaintext
bool changed = true;
set<\(V \cup \Sigma\)> LIVE = \(\Sigma\); //constructed to be the empty set
while (changed) {
    changed = false;
    for (each rule \(A \rightarrow \overrightarrow{W}\) in \(R\) such that \(A \notin LIVE\)) {
        if (\(\overrightarrow{W}\) is in \(LIVE\)) {
            LIVE = LIVE \cup \{A\};
            changed = true;
        }
    }  //LOOP INV: Every variable in LIVE can derive a terminal string
} // 
if (S \in LIVE) accept; else reject;
```

By the loop invariant, if \(S\) is ever added to \(LIVE\) then \(S\) derives some terminal string, which means that \(L(G) \neq \emptyset\). Hence the algorithm is \textbf{sound}---that is, it never gives a false positive. Why does it always terminate, and why is it comprehensive---that is, why does it halt and catch all "yes" cases?

- Each iteration of the while loop either adds a new variable to \(LIVE\) or leaves \(changed\) false.
- If \(changed\) is left false, the loop terminates right there.
- The number of times it can add a new variable is limited by the size of \(V\).
• Hence the while loop must terminate within \(|V| + 1\) iterations.
• So the pseudocode defines a **total** Turing machine, that is, a **decider**.

Now why is it comprehensive? Suppose \(L(G) \neq \emptyset\). Then there is a derivation of a terminal string \(x\) from \(S\). The string \(x\) can be \(\varepsilon\); this won't matter to the logic. The derivation has some number \(k\) of steps and can be represented abstractly as

\[
S \Rightarrow X_{k-1} \Rightarrow X_{k-2} \Rightarrow \cdots \Rightarrow X_3 \Rightarrow X_2 \Rightarrow X_1 \Rightarrow x
\]

The vector signs are to remind that each **sentential form** \(X_i\) can include multiple variables and its own terminals as well. Note the indexing of \(i\) in reverse order. Each \(X_i\) has one variable that was expanded in the step---wlog. it is the leftmost variable in \(X_i\)---and we can call it \(A_i\). It can be the same variable in different steps but we'll still call them \(A_i\). The first point is that \(A_1\) must be the only variable in \(X_1\), because replacing it leaves a terminal string. Put another way, the right-hand side of the rule \(A_1 \rightarrow W_1\) that was applied in the last step must be all terminals (we could have \(W_1 = \varepsilon\), that's fine) and everything else in \(X_1\) must be terminals. Since all terminals are initially in the set \(\text{LIVE}\), we have the following facts:

- The rule \(A_1 \rightarrow W_1\) has \(W_1 \in \text{LIVE}^*\).
- Hence the algorithm on the first iteration includes \(A_1\) into \(\text{LIVE}\).
- On the next iteration, \(X_1\) belongs to \(\text{LIVE}^*\).
- Whatever rule \(A_2 \rightarrow W_2\) was applied at the next-to-last step, it has \(W_2 \in \text{LIVE}^*\) in that iteration.
- Hence the second iteration adds \(A_2\) to \(\text{LIVE}^*\) (if \(A_2\) wasn't there already by virtue of being the same variable as \(A_1\)).

What this adds up to is that by induction on \(i\) we can prove the statement \(Q(i) \equiv "\text{the variable } A_i \text{ is added to the set LIVE on or before the } i\text{-th iteration}." \) Then with \(i = k\) the variable "\(A_k\)" is none other than \(S\).

- Thus the algorithm adds \(S\) to \(\text{LIVE}\), and so it gives the true-positive answer "yes."
- This means the algorithm captures all true positives, so it is comprehensive.
- Since it has no false positives, it is correct.
- Thus the problem \(\text{NE}_{\text{CFG}}\) is decidable.
- Since whenever we have a total Turing machine, we can complement the language by interchanging \(q_{\text{acc}}\) and \(q_{\text{ rej}}\), the complementary problem \(E_{\text{CFG}} \equiv "\text{Given a CFG } G, \text{ is } L(G) = \emptyset?"\) is likewise decidable.
- The algorithm runs within time \(O(|V| \cdot |\mathcal{R}| \cdot r)\), where \(|\mathcal{R}|\) means the number of rules but we also
have to allow for the maximum length $r$ of the right-hand side of a rule. Since the size of the (string encoding $\langle G \rangle$ of the) grammar $G$ can be reckoned as order-of $(|V| + r|R|)$, this is at worse quadratic. Anyway, it is a polynomial-time decider for $NE_{CFG}$ and $E_{CFG}$.

Now we consider a different problem but with a closely related solution. The name is not standard but is compatible with Sipser's naming scheme.

$Eps_{CFG}$:

INST: A CFG $G = (V, \Sigma, R, S)$.

QUES: Is $\epsilon \in L(G)$?

The algorithm needs changing only one line:

```python
bool changed = true;
set<V, \Sigma> NULLABLE = \emptyset; //constructed to be the empty set
while (changed) {
    changed = false;
    for (each rule $A \rightarrow W$ in $R$ such that $A \notin NULLABLE$) {
        if ($W$ is in $NULLABLE'$) {
            NULLABLE = NULLABLE \cup \{A\};
            changed = true;
        }
    }
}
```

if ($S \in NULLABLE$) accept; else reject;

The trick that gets this off the ground is our old friend $\emptyset^* = \{\epsilon\}$. Thus, in the first iteration, all variables $A$ such that $A \rightarrow \epsilon$ is a rule get added to NULLABLE. (We could also have initialized the set NULLABLE this way.) Every variable $B$ that is later added to NULLABLE truly derives $\epsilon$, so the soundness of this algorithm is clear, and the reason it terminates within $|V| + 1$ iterations is the same. Its correctness is a self-study exercise. Now we are ready to address the problem $A_{CFG}$.

$A_{CFG}$: (The "Acceptance Problem for CFGs")

INST: A CFG $G = (V, \Sigma, R, S)$ and a string $x \in \Sigma^*$.

QUES: Is $x \in L(G)$?

A decision procedure:

1. If $x = \epsilon$, apply the decision procedure for $Eps_{CFG}$ and accept $\langle G, \epsilon \rangle$ iff it accepts $\langle G \rangle$.
2. Else, convert $G$ into a Chomsky normal form grammar $G' = (V', \Sigma, R', S')$ such that $L(G') = L(G) \setminus \{\epsilon\}$, so that $x \in L(G) \iff x \in L(G')$.
3. Noting that $S' \Longrightarrow^* x$ if and only if $S'$ derives $x$ in exactly $2n - 1$ steps, where $n = |x|$, we can
exhaustively try all derivations of $2n - 1$ steps, and accept if and only if at least one of them derived $x$.

Step 1 runs in polynomial time, but as-stated, steps 2 and 3 do not. The issue with step 2 as presented in many other sources is that if we have a "long rule" like $A \rightarrow B_1 B_2 \cdots B_r$ where each $B_i$ is nullable, the conversion says to add all rules obtained by deleting any sublist of $(B_1, \ldots, B_r)$. This makes $2^r$ sublists, each of which might produce a different rule, and so takes exponential time. But a nifty trick is that we can first shorten the rule using dedicated single-use variables:

$$A \rightarrow B_1 D_1, \quad D_1 \rightarrow B_2 D_2, \quad D_2 \rightarrow B_3 D_3, \ldots, \quad D_{r-3} \rightarrow B_{r-2} D_{r-2}, \quad D_{r-2} \rightarrow B_{r-1} B_r.$$ 

Then the overall number of rules is multiplied by at most $2r$, which keeps the expansion of the grammar within a polynomial factor of the original data-size of $G$. The text does something related to this in its own incremental way of handling nullable variables when it describes the conversion to Chomsky normal form in section 2.1. (The remaining details of that are still FYI, skim/skip.)

Step 3 can exponentiate $2^{n-1}$ or worse if there are at least 2 choices for the $n - 1$ applications of a non-terminal rule in the derivation. However, there is a nifty dynamic programming algorithm that is sometimes mentioned in CSE331 or software-systems courses, called CYK or CKY for its authors Cocke, Kasami, and Younger. It does step 3 in polynomial time, thus completing a polynomial-time decider for $A_{CFG}$. (This is mentioned later in the text, but IMHO not so clearly.)

Since we did $A_{CFG}$, how about the corresponding "all"-type problem?

$ALL_{CFG}$:
INST: A CFG $G = (V, \Sigma, R, S)$.
QUES: Is $L(G) = \Sigma^*$?

Shock fact: This problem is not decidable at all. Indeed, there does not even exist a Turing machine $M$ such that $L(M) = \{ (G) : L(G) = \Sigma^* \}$, let alone one that is total. The proof of this will come later in Chapter 5, but we will reach it by starting with undecidable problems that involve Turing machines themselves as the data objects.

Problems About Turing-Machines Programs In General

$A_{TM}$:
INST: A deterministic Turing machine $M$ and an input $w$ to $M$.
QUES: Does $M$ accept $w$?

This is called the Acceptance Problem for Turing machines. Sometimes it is regarded as being the same as the Halting Problem, but we prefer to keep a separate designation for it:
**HP\_TM**: 
INST: A deterministic Turing machine \(M\) and an input \(w\) to \(M\).  
QUES: Does \(M(w) \downarrow\) ?

The \(A\_TM\) language is \(\{\langle M, w \rangle : w \in L(M)\}\). There is indeed a Turing machine that accepts (exactly) it: \(A\_TM = L(U)\) where \(U\) is any universal Turing machine, such as the one in Tuesday's lecture. But recall that \(U\) is not total: whenever \(M(w) \uparrow\), \(U(\langle M, w \rangle) \uparrow\) too. We will see that there is no decider for \(A\_TM\). The text states and proves this directly, but we will get there by a parallel road: the following defines the "diagonal language" in place of the text's "diagonal machine \(D\)."

**\(D\_TM\):** 
INST: A deterministic Turing machine \(M\).  
QUES: Does \(M\) not accept its own code \(\langle M \rangle\)?

The language is \(D\_TM = \{\langle M \rangle : M \text{ does not accept } \langle M \rangle\}\). Note that the case \(M(\langle M \rangle) \uparrow\), that is, \(M\) not halting on its own code, counts as \(\langle M \rangle\) being in the language \(D\_TM\) even though you can't immediately "register" that condition.

**Theorem:** The language \(D\_TM\) is not c.e.---that is, there does not exist a TM \(Q\) such that \(L(Q) = D\_TM\).

I am using the letter \(Q\) in a new way, to refer to a whole machine rather than its set of states, in order to reinforce the point that this machine does not actually exist although the proof involves talking about it as if it did. We can say \(Q\) is quixotic, after Don Quixote.

**Proof.** Suppose such a \(Q\) existed. Then it would have a string code \(q = \langle Q \rangle\). Then we could run \(Q\) on input \(q\). The logical analysis of that run, on hypothesis \(L(Q) = D\_TM\), is:

\[
Q \text{ accepts } q \iff q \text{ is in } D\_TM \quad \text{by } L(Q) = D\_TM \\
\iff Q \text{ does not accept } q \quad \text{by definition of } q \in D\_TM.
\]

The analysis makes a statement equivalent to its negation, which is a "logical rollback" condition. The rollback goes all the way to the first sentence of the proof. So such a \(Q\) cannot exist. \(\Box\)

It is worth reworking this proof in several ways...but for now we leave it as an end-of-lecture "cliffhanger." We will redo it with the several different ways next time.