CSE396 Spr26 Lecture Thu. Apr. 9: Decision Problems and Unecidability

[Lecture reviewed the following about decision problems involving NFAs before doing grammars.]

EQ_{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_{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:

$$\langle N_1, N_2 \rangle \in EQ_{NFA} \iff \langle N_1 \rangle \in ALL_{NFA}.$$

But we have already asserted above that ALL_{NFA} is NP**-hard.** So this blocks the attempt to solve EQ_{NFA}, and in fact, this shows that the EQ_{NFA} problem is 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_{Regexp}, A_{GNFA}, ALL_{Regexp}, NE_{GNFA}, and so on. They are all **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 "$_{Regexp}$" have much the same status as those subscripted "$_{NFA}$". 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

Let's first consider a problem that is fashioned around a task we've already seen in the coverage of Chomsky Normal Form (in the Tuesday Week 9 lecture). The problem name is not in the text, but it follows the text's general naming scheme.

Eps_{CFG}:
INST: A CFG $G = (V, \Sigma, \mathcal{R}, S)$.
QUES: Is $\epsilon \in L(G)$?

Before, we used the following loop to determine the set `NULLABLE` of variables that can derive ϵ. Then we simply append a line that accepts the given grammar G if and only if the final set `NULLABLE` includes the start symbol S:

```
bool changed = true;
set<V> NULLABLE = ∅; //constructed to be the empty set
while (changed) {
     changed = false;
     for (each rule A → W⃗ in ℛ such that A ∉ NULLABLE) {
          if (W⃗ is in NULLABLE*) {
               NULLABLE = NULLABLE ∪ {A};
               changed = true;
          }
     }
}
if (S ∈ NULLABLE) accept; else reject;
```

As previously discussed, this algorithm is correct (i.e., sound and comprehensive for building the true set of nullable variables) and terminates within $O(|V| \cdot |\mathcal{R}|)$ iterations, hence runs in time polynomial in the total bit-size of the encoding of the grammar G.

A little tweak---also briefly shown before---gives us a decision procedure for the nonemptiness problem of grammars over all strings, not just ϵ.

NE_{CFG}:
INST: A CFG $G = (V, \Sigma, \mathcal{R}, S)$.
QUES: Is $L(G) \neq \emptyset$? (Nerdy version: Is $L(G) \cap \Sigma^* \neq \emptyset$?)

The tweak is just in the initialization line (in blue):

```
bool changed = true;
set<V ∪ Σ> LIVE = Σ; //constructed to have the terminals
while (changed) {
     changed = false;
     for (each rule A → W⃗ in ℛ such that A ∉ LIVE) {
          if (W⃗ is in LIVE*) {
               LIVE = LIVE ∪ {A};
               changed = true;
          }
     } //LOOP INV: Every variable in LIVE can derive a terminal string
}
if (S ∈ LIVE) accept; else reject;
```

Call this code P, and let $\langle G \rangle$ be a standard way of encoding the given grammar G as a string (in binary or ASCII, say). Again, $P(\langle G \rangle)$ always terminates---indeed, it basically runs in $O(|V| \cdot |\mathcal{R}|)$ time---so it is a decider. 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 **sound**---that is, it never gives a false positive---and this tells us that $L(P) \subseteq NE_{CFG}$. For it to be **comprehensive**---and hence **correct**, meaning $L(P) = NE_{CFG}$---we need to argue that every grammar G with $L(G) \neq \emptyset$ gets accepted. The reasoning goes like this:

- Whenever $L(G) \neq \emptyset$, there is some terminal string $x \in \Sigma^*$ such that $S \Longrightarrow^* x$.
- The derivation $S \Longrightarrow^* x$ works in some finite number k of steps.
- The first step of that derivation is $S \rightarrow W$ for some $W \in (V \cup \Sigma)^*$.
- Every variable A in W derives some terminal string (which becomes a substring of x), and does so in a number j of steps that is at most $k - 1$.
- Hence we frame an induction hypothesis $H(k)$ saying, "If a variable B derives a terminal string within k steps, then the above algorithm adds it to the set `LIVE`---within k iterations of the outer `while` loop, in fact."
- The base case $H(0)$ puts terminals in `LIVE` --- or if it makes more sense, you can use $H(1)$ as the basis, since the first iteration will see all terminal rules $A \rightarrow u$ and put A into the set `LIVE` right away.
- Applying the principle of strong induction, we get $H(j)$ for all $j \leq k - 1$.
- In our particular case $S \rightarrow W$, this says that all variables A in W get placed into `LIVE` within the first $k - 1$ iterations (at worst).
- That means $W \in \texttt{LIVE}^*$, so the code itself puts S into `LIVE`.
- This---and parallel cases for other variables B besides S that derive terminals in k steps---proves $H(k)$ and make the induction $(\forall k) H(k)$ go through.
- That formally proves the correctness of the code. So $L(P) = NE_{CFG}$, so the NE_{CFG} problem is decidable.

The E_{CFG} problem is defined by flipping the question:

E_{CFG}:
INST: A CFG $G = (V, \Sigma, \mathcal{R}, S)$.
QUES: Is $L(G) = \emptyset$?

This is likewise decidable---and in polynomial time---since we just flip the final yes/no answers. Now we can tour other decision problems that correspond to other questions to pose about grammars:

A_{CFG}: (The "**A**cceptance Problem *for* **CFG**s")
INST: A CFG $G = (V, \Sigma, \mathcal{R}, S)$ 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 $\langle G, x \rangle$ iff it accepts $\langle G \rangle$.
2. Else, convert G into a Chomsky normal form grammar $G' = (V', \Sigma, \mathcal{R}', S')$ 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_j 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 $r - 2$ dedicated single-use variables:

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

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 ins ection 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 Theorem 7.16 on pages 290--291 later in the text, but it makes only the weaker statement that every individual context-free language belongs to **P**.)

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

ALL_{CFG}:
INST: A CFG $G = (V, \Sigma, \mathcal{R}, S)$.
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 $L(M) = \left\{\langle G \rangle : L(G) = \Sigma^*\right\}$, 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.

On the decidability side, we can itemize five basic algorithmic toolkit ideas:

1. Breadth-first search
2. The Cartesian Product Construction.
3. Loops that "grow" sets like `NULLABLE` and `LIVE`. Technically these are examples of "**least fixed-point**" algorithms, insofar as the outer while-loop iterates until the set stays fixed.
4. Exhaustive search---OK, unlike the above, it often takes exponential time.
5. **Dynamic Programming**---well, in this course this is FYI.

When we hit the level of problems about Turing machines, however, decidability becomes rare.

[Differences in my teaching philosophy here relative to the text:

- Less attention to the analogy with showing the real numbers are uncountable by diagonalization---skim/skip that.
- Instead of using the Halting Problem as the first undecidable problem, or its close cognate the "Acceptance Problem" $A_{TM} = \{\langle M, w\rangle : w \in L(M)\}$ for deterministic Turing machines M, it will start with the "diagonal language" $D_{TM} = \{\langle M\rangle : \langle M\rangle \notin L(M)\}$.
- The text implicitly uses D_{TM} in its undecidability proof at the end of chapter 4, but refers to it in terms of an impossible machine rather than a language. I will try to make clear what stuff is real and what is nonexistent/counterfactual/"quixotic". The language D_{TM} is real.

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 $D_{TM} = \{\langle M\rangle : M \textit{ does not accept } \langle M\rangle\}$. 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:

$$\begin{aligned} Q \textit{ accepts } q &\iff q \textit{ is in } D_{TM} && \text{by } L(Q) = D_{TM} \\ &\iff Q \textit{ does not accept } q && \text{by definition of } q \in D_{TM}. \end{aligned}$$

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. ☒

It is worth reworking this proof in several ways. One is to follow the chain of implications in both directions like a cat chasing its tail:

> "If Q accepts its own code q, then $q \in L(Q)$. But $L(Q) = D_{TM}$, which by definition is the language of codes of machines that do *not* accept their own code. So Q must not accept its own code q. But then q meets the definiton for being in D_{TM}. Since we have $D_{TM} = L(Q)$ to begin with, this means Q accepts its own code q. But if Q accepts its own code q, then..."

[Lecture paused to take this all in, and I inserted a flyover of Turing's seminal 1936 paper that I had skipped on Tuesday. This included an example of how Turing thought of computing real numbers to any specified precision rather than deciding languages. If we think of (subsets of) the positive natural numbers rather than (languages of) binary strings, then we can key the first "binary decimal place" to the number 1, the second place to the nuymber 2, and so on. Given a set A, we make a real number r_A between 0 and 1 by putting a 1 in each "binary decimal place i" if and only if the number i belongs to A. For instance, if A is the set of even numbers, then

$$r_A = 0.01010101\ldots = \frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \cdots = \frac{1}{3}.$$

Then r_A is computable in Turing's sense if and only if A is decidable in the sense we've defined. If (and only if) A is c.e., then we can list out as an infinite process all the places i that eventually get a 1, but maybe in such a haphazard order that we cannot guarantee when we will resolve which of the first n places ultimately stay 0 or get a 1. Then we can't guarantee knowing r_A to n-place precision as a

function of n. I did also attest that although Turing's own notation used weird capital latters and was gargantuan by our standards, his way of thinking about state spaces was instrumental to modeling the unseen settings of the Enigma coding device during WW II. All this was FYI, but the example I finished with below is important.]

Another help is to compare with an abstract proof about sets. Consider functions f whose arguments are elements of a set A and whose outputs are subsets of A. The $\underline{\delta}(p, a)$ function from an NFA becomes such a function when you fix the char $c = a$. Thus we write $f : A \to \mathcal{P}(A)$ where $\mathcal{P}$ denotes the power set. Then f being *onto* would mean that every subset of A is a value of f on some argument(s). But we have:

Theorem: No function $f : A \to \mathcal{P}(A)$ can ever be onto $\mathcal{P}(A)$.

Proof: Suppose we have a function $f : A \to \mathcal{P}(A)$. Then we *do* have the subset

$$D = \{a \in A : a \text{ is not in the set } f(a)\}.$$

By f being onto, there would exist $d \in A$ such that $f(d) = D$. But then:

$$\begin{aligned} d \in D &\iff d \text{ is in the set } f(d) && \text{by } f(d) = D \\ &\iff d \text{ is not in the set } f(d) && \text{by definition of } d \in D. \end{aligned}$$

The contradiction rolls back to the beginning, so f cannot be onto the power set $\mathcal{P}(A)$. ☒

When A is a finite set, this is obvious just by counting. Suppose $A = \{1, 2, 3, 4, 5\}$. Then there are $2^5 = 32$ subsets but only 5 elements of A to go around. As the size of A increases this becomes "more and more obvious." The historical kicker is that the proof works even when A is infinite. Georg Cantor gave ironclad criteria by which it follows that $\mathcal{P}(A)$ always has higher **cardinality** than A. In the case where $A = \mathbb{N}$ or $A = \Sigma^*$ this tells us that the set of all languages has higher cardinality than A, i.e., is not countably infinite. Because we have only countably many (string codes or Gödel numbers of) Turing machines, this is an "existence proof" that many languages don't have machines. The function $f(\langle M \rangle) = L(M)$ cannot be onto $\mathcal{P}\left(\Sigma^*\right)$.

Many sources give the illustration where the real numbers $\mathbb{R}$ are used in place of $\mathcal{P}\left(\Sigma^*\right)$. There is a nagging technical issue that two different decimal or binary expansions like $0.01111...$ and $0.1000...$ can denote the same number (0.5 in this case) but in decimal one can avoid it. The real number that is "not counted" is pictured by going down the main diagonal of an infinite square grid, hence the name *diagonalization* for the whole idea. But I like to do without it.