Decidable Properties of Context-free Languages

1. Last time, we considered several problems related to regular languages described by DFAs or regular expressions and concluded that they were all decidable by identifying algorithms that could solve the problems.

Today, we will start by looking at similar problems involving context-free languages, but first let’s review a few ideas from our discussion of problems involving regular languages:

(a) The last two examples we were discussing were:
   - \( E_{DFA} = \{ \langle A \rangle | \text{A is a DFA and } L(A) = \emptyset \} \), and
   - \( EQ_{DFA} = \{ \langle A, B \rangle | \text{A and B are DFAs and } L(A) = L(B) \} \)

(b) While we saw that we could view \( E_{DFA} \) as a graph problem in which we sought to determine if any final state in the graph of a DFAs state machine was reachable, I also suggested a more brute-force approach — check every string.

(c) The brute force approach worked for three reasons:
   - It is possible for a Turing machine to enumerate all of the strings over an alphabet
   - It is possible for a Turing machine to simulate a DFA (or a PDA or for that matter another Turing machine) on a given input,
   - In the case of a regular language \( L(D) \), the Pumping Lemma allows us to argue that if \( D \) has \( n \) states than if any string belongs to \( L(D) \), then some string of length less than or equal to \( n \) must be in the language. So, we can stop enumerating strings once they exceed this length.

2. To further appreciate this brute force approach, before we consider \( EQ_{DFA} \), let’s think about its “almost” complement \( NEQ_{DFA} = \{ \langle A, B \rangle | L(A) \neq L(B) \} \). \[1\]

3. Given a descriptions of \( A \) and \( B \), observe that if we apply the brute force approach of checking every string \( w \) until we find a string that is in one of \( L(A) \) or \( L(B) \) but not the other and then accepting, we end up with a machine that recognizes \( NEQ_{DFA} \) but does not decide \( NEQ_{DFA} \) because it goes on trying forever when processing a \( \langle A, B \rangle \) that is not a member of \( NEQ_{DFA} \).

4. We can build a TM that decides \( EQ_{DFA} \) (or \( NEQ_{DFA} \)) by observing that \( (L(A) \cap \overline{L(B)}) \cup (L(B) \cap \overline{L(A)}) \) describes the set of all counter examples to the equality of the languages of the two machines. This language contains any string \( w \) that either is an element of \( L(A) \) but not \( L(B) \) or is an element of \( L(B) \) but not of \( L(A) \)

5. Since regular languages are closed under complement, union and intersection, we know that the language \( (L(A) \cap \overline{L(B)}) \cup (L(B) \cap \overline{L(A)}) \) must be regular. In fact, by following the details of the proofs of these closure properties, we can construct a DFA to recognize \( (L(A) \cap \overline{L(B)}) \cup (L(B) \cap \overline{L(A)}) \). Therefore, we can decide whether \( L(A) = L(B) \) by deciding whether the DFA we construct to recognize \( (L(A) \cap \overline{L(B)}) \cup (L(B) \cap \overline{L(A)}) \) accepts the empty language.

   - Note that we are employing a technique here that we will be using quite a bit — passing the buck.
   - While the algorithm we just described to decide \( EQ_{DFA} \) must build a description of a DFA whose language is \( (L(A) \cap \overline{L(B)}) \cup (L(B) \cap \overline{L(A)}) \), it doesn’t explicitly simulate or analyze the structure of this DFA.
   - Instead, having already concluded earlier that \( E_{DFA} \) is decidable, we just feed the description of the machine for \( (L(A) \cap \overline{L(B)}) \cup (L(B) \cap \overline{L(A)}) \) to some Turing machine (i.e., algorithm) that can

\[1\]I said “almost” because the complement of \( EQ_{DFA} \) is actually the union of \( NEQ_{DFA} \) and all strings that are not valid encodings of a pair of DFAs given the encoding scheme chosen.
decide $E_{DFA}$. We don’t need to know how this algorithm works. We just need to know it exists.

6. For $A_{CFG}$, we can easily imagine a nondeterministic TM that would recognize the language. It would just guess a derivation using the grammar’s productions and verify that it yields $w$. Unfortunately, if a grammar has rules like $P \rightarrow Q$ and $Q \rightarrow P$, there can be derivations of unbounded length, so the nondeterministic machine would be a recognizer for this language, but not a decider. Therefore, we need some way to put a bound on the length of the derivations we need to consider.

7. In class, we saw an algorithm to convert a CFG into Chomsky normal form. If we build a TM that converts its input CFG into Chomsky normal form, it can then limit its search to derivations of length less than twice the length of $w$ plus 1. This makes our machine a decider.

8. $E_{CFG}$ is a bit trickier. The decidability of this language depends on an algorithm that determines for each variable or terminal in a CGF whether that symbol derives any string composed entirely of terminals. Given a CFG $G = (V, \Sigma, R, S)$, we iteratively compute a set $USEFUL$ of symbols known to derive some string of terminals as follows:

- Initially, set $USEFUL = \Sigma$
- Repeatedly (until the following process does not increase the size of $USEFUL$)
  - For each $V \rightarrow \beta \in R$ where $V \notin USEFUL$, *
    - if every $s \in (\Sigma \cup V)$ in $\beta$ is already in $USEFUL$, then add $V$ to $USEFUL$

If at the end of this algorithm, $S \in USEFUL$, then we know that $L(G) \neq \emptyset$, otherwise, $L(G) = \emptyset$.

Recursive, Recursively Enumerable, not even R.E.

1. Today, we will continue to explore how to categorize the computability of various languages into three categories:

I want to start by exploring the relationships between these categories and giving some plausible examples of language that fall in each category (without proving the truth of these claims). The goal is to give you an overview of the landscape we will be exploring.

Turing-decidable/recursive We say a language $L$ is Turing-decidable if there exists a TM $M$ that halts on all inputs for which $L = L(M)$. We have just seen many examples of such languages.

Turing-recognizable/recursively enumerable We say a language $L$ is Turing-recognizable if there exists a TM $M$ for which $L = L(M)$. One important example of a recognizable language that is not decidable is $A_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM and } w \in L(M)\}$. We can recognize this language because it is possible to build a TM, $U$, that can simulate the behavior of another TM on input $w$. It isn’t clear how the simulator could ever conclude with certainty that $M$ is stuck in a loop, so the language may not be (it isn’t!) decidable.

Not Turing-recognizable The language $E_{TM} = \{\langle M \rangle \mid L \text{ is a TM and } L(M) = \emptyset\}$ is not recognizable, but its complement is.
Eventually, we will further distinguish languages that are not recognizable but whose complements are recognizable from those where both the language itself and its complement are not recognizable.

**Why Recursively Enumerable is Equivalent to Recognizable**

1. We say that a language is recursively enumerable if we can build a TM, $E$, that will write a sequence of strings that belong to $L$ on one of its tapes in such a way that every $w \in L$ will eventually appear in this sequence. As a result, we can think of the machine as numbering the elements of $L$ (it would be easy though time consuming to eliminate any duplicates). Each $w \in L$ is associated with the number of the position at which it appears within the sequence output by $E$.

2. It should be clear that every recursively enumerable language $L$ is recognizable.
   - Given an enumerator $E$ for a language, we can build a recognizer $R$ for the same language by having $R$ run $E$ as a sub-machine and every time $E$ writes a new member of $L$ on its tape compare that member to $R$’s input. If they match, $R$ accepts.

3. Slightly more surprising (and subtle to prove) is the fact that every Turing-recognizable language is also recursively enumerable.

4. The basic idea is that given a machine $R$ that recognizes some language $L$, we can build a machine $E$ that uses $R$ to check every string over its alphabet to see if $R$ accepts and writes all the accepted strings on its tape.

5. We have to be very careful because $R$ may loop on any $w \notin L$. If we just simulate $R$ on every element of $w_0, w_1, w_2, \ldots$ in order our simulator may get stuck in a loop on some early member of the sequence.

6. We solve this using a technique called dovetailing. We will design a simulator that simulates $R$ processing many strings at a time. At each round, our simulator will simulate one step of $R$ on each string it is currently simulating and then add one more string to the mix.