CS 361 Meeting 23 — 4/22/20

Encoding Questions about Automata as Languages

(Click for video)

1. We are getting close to the goal of showing an example of a problem (i.e., language) that is not computable (i.e., decidable). The first example of such a language (and many of the examples of such languages) is a language that involves statements about automata.

2. To get ready for such languages, it makes sense to spend a little time talking about languages that make simpler statements about automata. That is, to talk about decidable languages about automata.

3. For example, given a deterministic finite automaton we might ask whether a particular string belongs to its language. That is, whether it accepts a particular string.

   - To answer this question, we would need a description of the automaton.
   - While our first inclination might be to draw a diagram of the automaton, we have also seen that we can describe an automaton in textual form as a tuple $M = (Q, \Sigma, \delta, s_0, F)$.
   - Suppose that M is a string (i.e., a bunch of text) describing a DFA as a tuple and w is any string over the DFA's alphabet.
   - The set of strings M, w formed by joining together a description of M followed by any w such that $w \in L(M)$ forms a language.
   - Figuring out whether some M, w belongs to this language is equivalent to answering the question does M accept w.

4. A few examples of the sorts of languages I have in mind include:

   - $A_{DFA} = \{\langle M, w\rangle \mid M$ is a DFA and $w \in \mathcal{L}(M).\}$
   - $A_{REX} = \{\langle R, w\rangle \mid R$ is a regular expression and $w \in \mathcal{L}(R).\}$
   - $E_{DFA} = \{\langle M\rangle \mid M$ is a DFA and $\mathcal{L}(M) = \emptyset\}$
   - $EQ_{DFA} = \{\langle A, B\rangle \mid A$ and B are DFAs and $\mathcal{L}(A) = \mathcal{L}(B)\}$
   - $A_{CFG} = \{\langle G, w\rangle \mid G$ is a CFG and $w \in \mathcal{L}(G).\}$
   - $E_{CFG} = \{\langle G\rangle \mid G$ is a CFG and $\mathcal{L}(G) = \emptyset\}$
   - $EQ_{CFG} = \{\langle G, H\rangle \mid G$ and H are CFGs and $\mathcal{L}(A) = \mathcal{L}(B)\}$

5. In the descriptions of all of these sets, the angle brackets, $\langle ...\rangle$ are included to suggest that descriptions of the grammars, automata, etc. that are included in these languages must somehow be encoded in a precise way.

   - To determine that any of these languages is Turing-recognizable or Turing-decidable, we would need to describe some Turing machine that recognized the language in questions. This machine would have some fixed alphabet Σ, but we wish to include machines and grammars over all alphabets in these languages. Accordingly, we will have to somehow encode arbitrary finite alphabets in some single alphabet.

6. The exact encoding scheme used is usually unimportant and rarely explicitly discussed in the proof of a language's decidability, but before we start ignoring these details, I thought it would be helpful to think concretely about how we might represent one of these languages. So, let't think about how we might represent the strings in A_{CFG}.

   - This mainly boils down to how do we represent an arbitrary CFG in some fixed alphabet.
   - I would like you all to take a few minutes to design a scheme for representing arbitrary CFGs given a fixed alphabet like:

   $$\Sigma = \{\rightarrow, \#, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, V, T\}$$

   - As an example, consider the grammar $G =$

     $L \rightarrow 1L1 \mid +R$
     $R \rightarrow 1R1 \mid =$

     That is, be prepared to show how to construct the representation of $< G, 1 + 1 = 1 >$

Click here to view the slides for this class

- There are many ways to do this. For example, we could number the elements of the alphabets of terminals and variables (non-terminals) and then use "Tn" to encode the nth terminal and "Vn" for the nth variable.
- We would then give up on the | symbol and just list all of the productions separated by # as in:

$$V1 \to T1V1T1\#V1 \to T2V2\#V2 \to T1V2T1\#V2 \to T3$$

- We could complete the representation by adding w separated by the remaining symbols with a pair of #'s.

$$V1 \to T1V1T1\#V1 \to T2V2\#V2 \to T1V2T1\#V2 \to T3\#\#T1T2T3T1$$

- Of course, we could have used binary numbers instead of decimal, or unary, or ...

7. All of the language we are interested in should only contain strings that are valid given the representation scheme we have chosen. For example, given our scheme for CFGs, a string that contained two consecutive T's should immediately be rejected.

8. Fortunately, for reasonable representation schemes, the language of syntactically correct encodings is regular, context-free, or at worst decidable, so we can assume that any TM we describe to recognize one of these languages begins by rejecting anything with invalid format.

Decidable Properties of DFAs and Regular Expressions

(Click for video)

(**Errata:** As I reviewed the video for this topic (and the next) I noticed that I said "DFA" when I should have said "NFA" on many occasions while describing the argument supporting the claims that regular languages are closed under union, product and closure.)

1. OK. Enough of that! Let's get back to talking about language based on questions about automata, grammars, etc.

2. Assuming now that the particular representation we use is not critical, let's consider whether some of the problems mentioned above are decidable.

3. Consider the first language:

$$A_{DFA} = \{\langle B, w\rangle \,|\, D \text{ is a DFA and } w \in \mathcal{L}(B).\}$$

4. We can show that this language is decidable by arguing that a TM can read in a description of a DFA from its input tape and then simulate that DFA. We will assume that the representation used for a DFA is essentially a list of triples of the form $(q, x, \delta(q, x))$ describing the machine's transition function together with descriptions of its initial state, its final states and the number of states and symbols in its alphabet.

   To make the explanation of the simulation as simple as possible, we will assume a 3-tape machine.

   - First, the machine will find w at the end of its first/input tape, copy it to its second tape, and erase it from the first tape, leaving mainly a list of transition function triples on the first tape.
   - Next, the machine will write the description of the initial state on its third tape.
   - Mainly, the machine will repeatedly (until it reaches the last input symbol):
     - Scan its first tape to find a $(q, x, \delta(q, x))$ with x matching the symbol encoded starting with the position of the second tape head and q matching the symbol on the third tape.
     - Move the second tape head to the beginning of the next encoded symbol
     - Copy the state $\delta(q, x)$ from the triple found on the first tape over the state that had been stored on the third tape.

5. Consider the language

$$A_{REX} = \{\langle R, w\rangle \,|\, R \text{ is a regular expression and } w \in \mathcal{L}(R).\}$$

- In some sense, this is the same as A_{DFA}, since the language recognized by DFAs are exactly the same as those described by regular expressions.
- In particular, we know that there is an algorithm that can convert a regular expression into a NDFA and then the subset construction can convert this NDFA into a DFA.
- Turing machines can be used to implement algorithms!
- Therefore, we can say that A_{REX} is decidable because we can build a TM that uses the algorithms we studied earlier to convert the regular expression in its input into a DFA and then uses the machine we described above for A_{DFA} to finish the job!

6. Next, let's think a bit about how we could argue that the following languages is decidable.
   - $E_{DFA} = \{\langle A\rangle \mid A \text{ is a DFA and } \mathcal{L}(A) = \emptyset\}$, and
7. We can treat E_{DFA} as a graph search problem. Viewing the machine's state diagram as a graph, if there is any path from the start symbol to a final state in the graph, then the machine must accept the string whose symbols label the edges of the path. A breadth-first search of the graph will find all reachable states.
8. We can also take a more "brute force" approach in which we use a decider for A_{DFA} to check to see any member of a large group of strings belongs to the DFAs language and accept $\langle A\rangle$ only if no members of $L(A)$ are found.

   The feasibility of this approach depends on several factors:

   - 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.

More Decidable Questions Involving DFAs and CFGs

(Click for video)

1. Next, let's consider $EQ_{DFA} = \{\langle A, B\rangle \mid L(A) = L(B)\}$.
2. Actually, to further appreciate the option of using brute force for some problems, let's first consider the "almost" complement of EQ_{DFA}, the language $NEQ_{DFA} = \{\langle A, B\rangle \mid L(A) \neq L(B)\}$. [1]
3. Given description 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 $< A, B >$ in the case that $L(A) = L(B)$.
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.

[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.

- Note that we are employing a technique here that we will be using quite a bit — passing the buck. Given one problem, we construct an instance of another problem that we already know how to solve.
- 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 decide E_{DFA}. We don't need to know how this algorithm works. We just need to know it exists.

6. I want to conclude our introduction to decidable problems about languages by briefly considering two questions related to context-free grammars.

7. For
$$A_{CFG} = \{\langle G, w\rangle \,|\, G \text{ is a CFG and } w \in \mathcal{L}(G).\}$$
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.

8. Unfortunately, if a grammar has rules like $P \rightarrow Q$ and $Q \rightarrow P$, there can be derivations of unbounded length. If we design a nondeterministic machine so that it can choose any option in searching for a derivation, that machine will have non-terminating computation sequences. So the nondeterministic machine would be a recognizer for this language, but not a decider. To build a decider, we need some way to put a bound on the length of the derivations we need to consider.

9. 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.

10. The language
$$E_{CFG} = \{\langle G\rangle \,|\, G \text{ is a CFG and } \mathcal{L}(G) = \emptyset\}$$
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 β 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, $L(G) = \emptyset$.

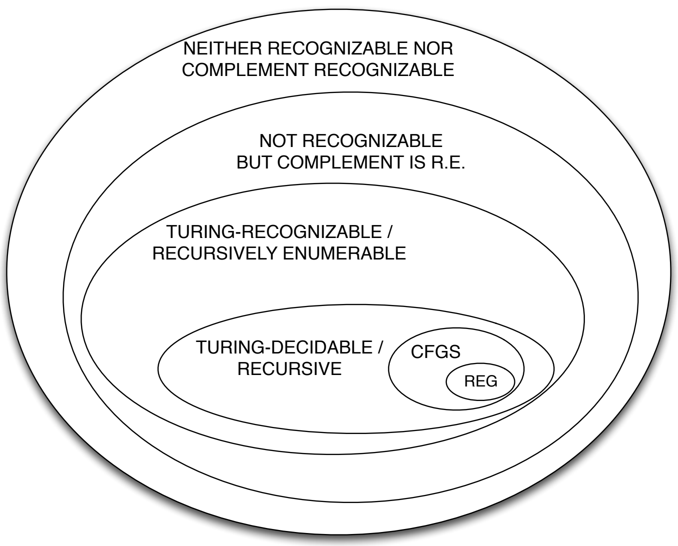
Recursive, Recognizable, not even Recognizable, and worse!

(Click for video)

1. We have not considered one of the languages in our original list:
    - $EQ_{CFG} = \{\langle G, H\rangle \,|\, G \text{ and } H \text{ are CFGs and } \mathcal{L}(A) = \mathcal{L}(B)\}$
2. In addition, we have not considered the decidability of similar languages about TMs:
    - $A_{TM} = \{\langle M, w\rangle \,|\, M \text{ is a TM and } w \in \mathcal{L}(M).\}$
    - $E_{TM} = \{\langle M\rangle \,|\, M \text{ is a TM and } \mathcal{L}(M) = \emptyset\}$
    - $EQ_{TM} = \{\langle M, N\rangle \,|\, M \text{ and } N \text{ are TMs and } \mathcal{L}(M) = \mathcal{L}(N)\}$
3. In case you cannot guess, these are all examples of languages that are not decidable. In fact, one of them is not even Turing-recognizable.

4. Before proving these facts, I want to take some time to make sure we are as comfortable as we can get with the terminology we will be using to distinguish different degrees of computability relative to the Turing machine model.

5. We will focus on three types of languages. The diagram below summarizes the inclusion relationships that exist between these three sets of languages and several others we have (or will) encounter.



Turing-decidable We say a language L is Turing-decidable if there exists a TM M that halts on all inputs for which $L = L(M)$. The terms "decidable" and "recursive" are synonymous with "Turing-decidable". All of the languages we discussed so far this semester including regular and context-free language belong to this group.

Turing-recognizable We say a language L is Turing-recognizable if there exists a TM M for which $L = L(M)$. Basically, we have simply dropped the requirement that M halt on inputs that are not in L. Therefore, all Turing-decidable languages are also Turing-recognizable. The terms "recognizable" and "recursively enumerable" are synonymous with "Turing-recognizable".

- It should be clear that

$$A_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM and } w \in \mathcal{L}(M).\}$$

  is Turing-recognizable. Just as a TM can simulate the computation of a DFA, we can design a TM to simulate any other TM as long as it is given a description of that TM and the input that machine should process. If $w \in L(M)$, then the simulation will eventually reach an accepting state and the simulator can accept $< M, w >$. However, it is not clear that this language is decidable since if M does not halt on w, there is no obvious way that the simulator can distinguish this from a very long computation that will eventually accept so it will not halt either.
  Note: This is not intended as a proof that A_{TM} is not decidable. The goal is just to help us clearly understand the structure of these classes of language before we do prove that various language fall into various categories.

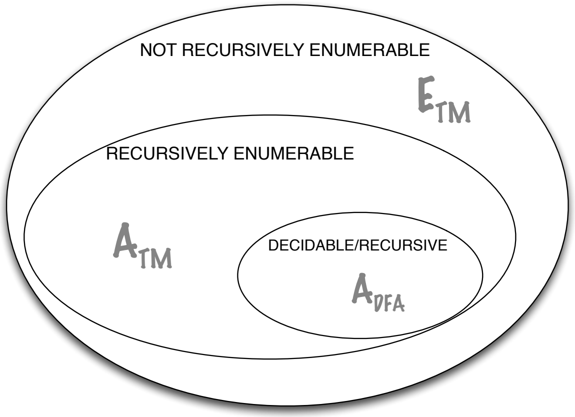
Not Turing-recognizable Each Turing-recognizable language is associated with one or more (actually always more) Turing machines. We know that there are only countably many TMs. Therefore, there can only be countably many Turing-recognizable languages. Since there are uncountably many subsets of the set of all strings over any alphabet, there must be many languages that are not Turing-recognizable (a.k.a. not recursively enumerable).

Intuitively, it should seem likely that

$$E_{TM} = \{\langle M\rangle \mid M \text{ is a TM and } \mathcal{L}(M) = \emptyset\}$$

is such a language. The only obvious way to determine that the language of a TM is empty is to check every string to make sure that there is no string that the machine accepts. Since there are infinitely many strings, this process will not terminate if the language is actually empty.

The diagram below summarizes the language we have suggested might belong to each of these three sets of languages.



Mr. Goldbach (Optional Section)

(Click for video)

1. While we know that there must be (plenty of) languages that are not recursively enumerable, the argument given above that E_{TM} is unlikely to be recursively enumerable is not that compelling. We have seen several results already this semester that run counter to intuition, so it might not be a surprise to discover there was some sneaky way to determine that a TM was not going to halt on an input (other than waiting for eternity to see what happens).

2. To give you a more compelling reason to believe no such secret way to predict an infinite loop exists, consider the following open problem from mathematics:

   > **Goldbach's Conjecture:** Every even integer greater than 2 can be written as the sum of two primes.

   This simple statement has been an open question since June 7, 1742.

3. To appreciate how this conjecture relates to computability, consider the TM:

   M_{GB} =

   - On input n:
     - if n is odd, reject
     - For all $1 < i < n$
       - if i is prime and $n - i$ is prime reject
     - accept

4. To determine if $\langle M_{GB}\rangle \in E_{TM}$ we would have to build a machine that could prove or disprove the Goldbach Conjecture (and any other open mathematical program for which we could write a similar algorithm to search for counter examples). Thus, deciding or recognizing E_{TM} is probably very hard!

5. A related algorithm suggests that A_{TM} is also difficult to decide.

   M'_{GB} =

   - On input n:
     - ignore n.
     - for every even $i > 2$
       - for all $2 < j < i$
         - if i is a counter-example to the Goldbach conjecture, accept
     - reject (by looping infinitely)

6. If you think about it, you will realize that $L(M'_{GB}) = \emptyset$ if the Goldbach conjecture is true and $L(M'_{GB}) = \Sigma^*$ if it is false. Therefore, deciding $\langle M'_{GB}, w\rangle \in A_{TM}$ for any w would require resolving the Goldbach conjecture.