CSE491/596 Lecture 10/6/23 Friday: Decidable and Undecidable Languages

The standard format for specifying decision problems came from a famous text by Michael Garey and David S. Johnson titled *Computers and Intractability*:

[Name of problem in small caps]
INSTANCE: [a description of the input(s) to the problem: strings, numbers, machines, graphs, etc.]
QUESTION: [a *yes*/*no* condition where *yes* means the input is accepted]

INSTANCE is also called INPUT; one can abbreviate it to INST and QUESTION to QUES. The **language** of the problem is the set of valid instances for which the answer is *yes*. Sometimes confusingly, the name of the problem usually doubles as the name of the language. The Sipser text also *established* a standard scheme for naming various decision problems that arise with the various machine, regexp, and grammar classes in this subject. It is best described by example.

A_{DFA}: (The "**A**cceptance Problem *for* **DFA**s")
INST: A DFA $M = (Q, \Sigma, \delta, s, F)$ and a string $x \in \Sigma^*$.
QUES: Does M accept x?

The input to a decision procedure for this problem is given in the form $\langle M, x \rangle$. The language is

$$A_{DFA} = \{\langle M, x \rangle : M \textit{ is a DFA and } M \textit{ accepts } x\}.$$

The length N of $\langle M, x \rangle$ can be reckoned as roughly of order $m + n$ where m is the number of states in Q (note that the number of instructions for a DFA is m times $|\Sigma|$ and we can treat $|\Sigma|$ as a fixed constant such as 2) and $n = |x|$ as usual. The alphabet of the A_{DFA} language can be reckoned as ASCII or even as $\{0, 1\}$. Here is a simple statement of an algorithm to solve the A_{DFA} problem:

1. Given $\langle M, x \rangle$, first decode M and x individually. (If not possible, reject.)
2. Run $M(x)$ (using a simulator like the *Turing Kit*) until the DFA reaches the end of x.
3. Accept $\langle M, x \rangle$ if M accepted x, else halt and reject $\langle M, x \rangle$.

This pseudocode always halts because a DFA M always halts. To simulate a step of $M(x)$ takes time *at most* order-m; really it can be $O(\log m)$ time per step using good data structures (mainly being able to assign a pointer to the destination state in any executed instruction). So the running time is $O(mn)$ which gives time $O\left(N^2\right)$ taking the length $N = |\langle M, x \rangle|$ into account. Thus we can say:

- The algorithm is a **decision procedure** to solve the A_{DFA} problem.
- Hence the A_{DFA} *problem* and the A_{DFA} *language* are called **decidable**.
- In fact, they are *decidable in polynomial time*.

Now suppose we have an NFA in place of the DFA.

A_{NFA}: (The "**A**cceptance Problem *for* **NFA**s")
INST: An NFA $N = (Q, \Sigma, \delta, s, F)$ and a string $x \in \Sigma^*$.
QUES: Does N accept x?

The following qualifies as a decision procedure, albeit highly inefficient:

1. Given $\langle N, x \rangle$, first decode N and x individually.
2. Convert N into an equivalent DFA M.
3. Then run the decision procedure for A_{DFA} on $\langle M, x \rangle$ and give the same yes/no answer.

Step 3 will later be called **reducing** the (instance of the) latter problem **to** the (equivalent "mapped" instance of the) former problem. But step 2 makes this an inefficient reduction---it can require order-of 2^m time where we are now calling m the number of states in N. Then again, step 2 does always halt, so if halting is all you care about, it goes as a decision procedure. But faster is:

1. Given $\langle N, x \rangle$, first decode N and x individually.
2. Initialize R_0 to be the ϵ-closure of the start state of N.
3. For each char x_i of x, build the set R_i of states reachable from a state in R_{i-1} by processing x_i.
4. Accept $\langle N, x \rangle$ if and only if $R_n \cap F \neq \emptyset$, which is if and only if N accepts x.

For each char i, step 3 runs in time at worst $O(m^2)$ (again, one can do better with smarter data structures), so the whole time is $O(m^2 n)$, which is polynomial in $|\langle N, x \rangle| \approx m + n$.

(Non-)Emptiness Problems

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

NE_{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_{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_{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_{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}.
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 reducting 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_{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 $A \triangle B = (A \setminus B) \cup (B \setminus A) = (A \cup B) \setminus (A \cap B)$ *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 $F_3 \ = \ \{(q_1, q_2) : \ q_1 \in F_1 \ XOR \ q_2 \in F_2\}$.
3. Feed $\langle M_3 \rangle$ to the decision procedure for E_{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_{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 "$_{NFA}$".

Undecidable Languages

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

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

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. Another is to use the recursive enumeration $M_0, M_1, M_2, \ldots$ of DTMs, that is, to treat their codes as "Gödel Numbers." Then the definition looks like:

$$D_{TM} = \{i : i \notin L(M_i)\}.$$

The proof then goes: if Q existed, it would equal M_q for some number q. But then Q accepts $q \iff$ ... as above.

We compare with an abstract proof about sets. Consider functions f whose arguments belong to a set A and whose outputs are subsets of A. The $\underline{\delta}(p, c)$ function from an NFA becomes such a function when you fix the char c. 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 had such an A and f. Then we would have the subset

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

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

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

The contradiction rolls back to the beginning, so there cannot be such an A and f. ☒

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.]

Yet another variation is to define D with regard to other progarmming formalisms besides Turing machines, for instance:

$$D_{Java} = \{p : p \text{ compiles in Java to a program } P \text{ such that } P(p) \text{ does not execute } \texttt{System.exit}(0)\}.$$

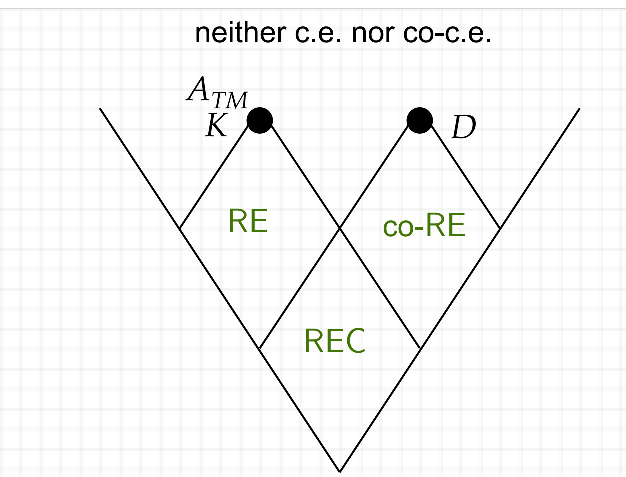
If D_{Java} were c.e. then by the equivalence of Java and TMs, there would be a Java program Q such that $L(Q) = D_{Java}$ (where acceptance means exiting normally). Then Q would have a valid code q that compiles to Q and ... the logic is the same as before.

One nice aspect of Gödel Numbers is that you don't have to worry about strings that are not valid codes. So if we define

$$D = D_{TM} = \{i : i \notin L(M_i)\}$$

$$K = K_{TM} = \{i : i \in L(M_i)\} = \{i : \langle M_i, i \rangle \in A_{TM}\}$$

then K_{TM} is literally the complement of D_{TM}. Now we can label our basic "class diagram" from before a little further, populating it with some languages:



This diagram conveys some extra information:
◎ REC is closed under complements,
◎ RE $\cap$ co $-$ RE $=$ REC, and
◎ All three classes are closed downward under *computable many-one/mapping reductions*.