CSE491/596 Lecture Mon. Oct. 9, 2023: Undecidable Languages

First, a portion of notes I skipped over when doing the Friday 10/06/23 lecture afresh from overhead projector drawings: The most basic kind of problems about automata or programs is whether they accept a given input. Here is the Sipser naming scheme exemplified for DFAs:

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

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

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---you could summarize it as "Just Do It":

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.

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 $|\langle N, x\rangle| \approx m + n$.

Now how about the same kind of problem for Turing machines? We presume deterministic ones:

A_{TM}: (The "**A**cceptance Problem *for* **T**uring **M**achines"---default is deterministic)
INST: A DTM $M = (Q, \Sigma, \Gamma, \delta, \_, s, q_{acc}, q_{rej})$ and a string $x \in \Sigma^*$.
QUES: Does M accept x?

Is this decidable? What if we just run M on x? It is deterministic after all. But it might not halt...
Another question is whether the language$A_{TM} = \langle M, x\rangle : x \in L(M)\}$ is computably enumerable.

The Diagonal Language---and the Problem It Causes

Define $D_{TM} = \{\langle M\rangle : M \textit{ does not accept } \langle M\rangle\}$. That is, $\{\langle M\rangle : \langle M, M\rangle \notin A_{TM}\}$.

There is a notation for the language of machines that do accept their own code:

$K_{TM} = \{\langle M\rangle : M \textit{ does accept } \langle M\rangle\}$. That is, $\{\langle M\rangle : \langle M, M\rangle \in A_{TM}\}$.

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. ☒

$D_{TM} = \{q : Q \textit{ does not accept } q \textit{ where } q = \langle Q \rangle\}$

The view this "in reverse": Suppose Q *does not accept* q. Since $q = \langle Q \rangle$, this means Q does not accept its own code $\langle Q \rangle$. This means in turn that $\langle Q \rangle$, i.e., q, belongs to the D_{TM} language. But if we maintain that $L(Q) = D_{TM}$, then Q must accept q after all. This is a contradiction.

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.

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 $\underline{\delta}(p, 0)$ 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 every subset of A is a value of f on some argument. But:

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:

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

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:

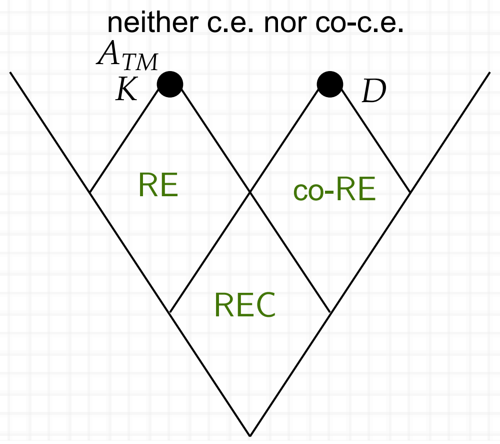
$$D_{Java} = \{p : p \textit{ compiles in Java to a program } P \textit{ such that } P(p) \textit{ does not execute } \texttt{System.exit}(0)\}.$$

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.

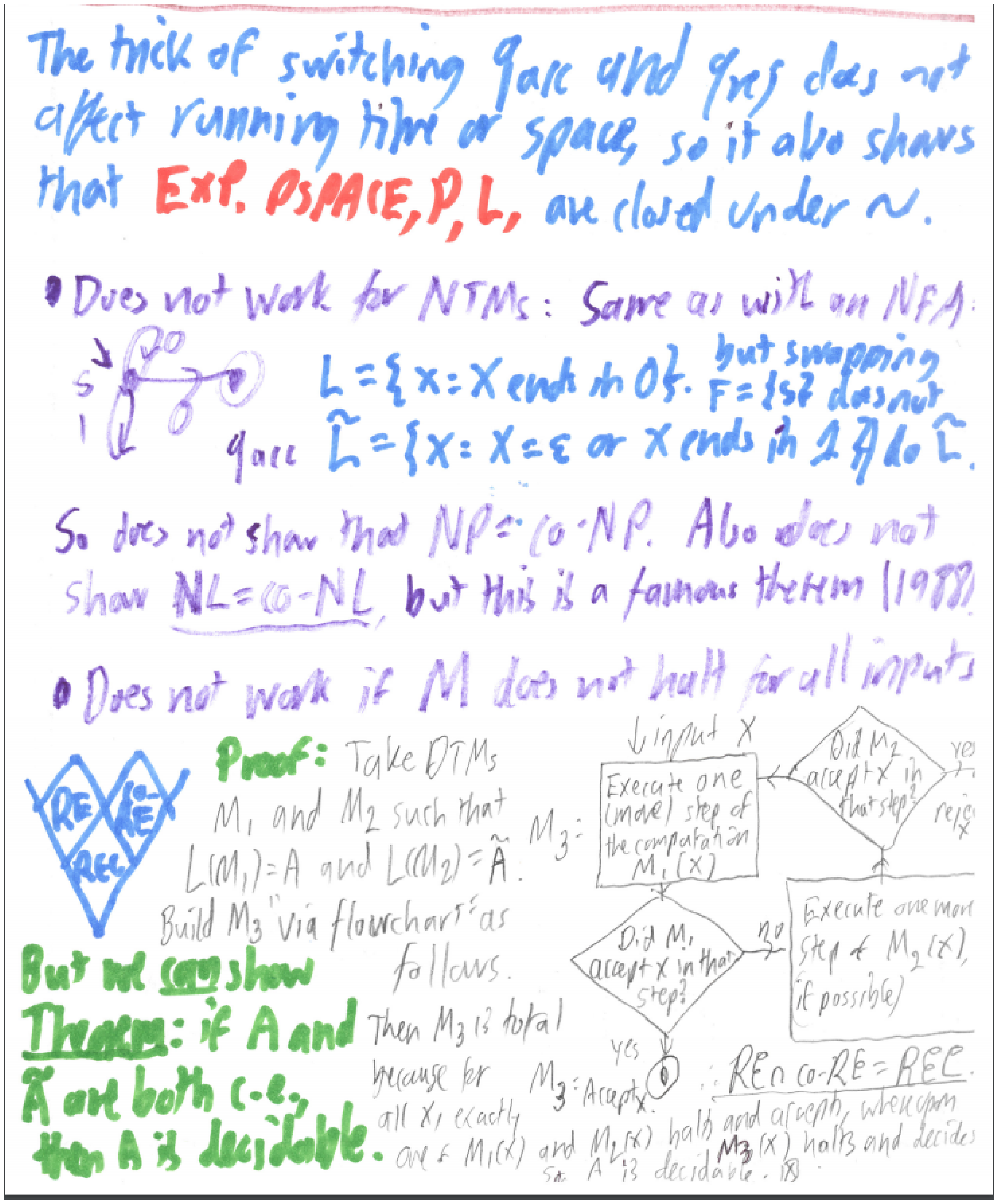
One nice aspect of Gödel Numbers is that you don't have to worry about strings that are not valid codes. So if we define

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

then K_{TM} is literally the complement of D_{TM}.



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



The trick of switching q_{acc} and q_{rej} does not affect running time or space, so it also shows that EXP, PSPACE, P, L, are closed under $\sim$.

- Does not work for NTMs: Same as with an NFA:

$L = \{x : x \text{ ends in } 0\}$. but swapping $F = \{s\}$ does not
$\tilde{L} = \{x : x = \epsilon \text{ or } x \text{ ends in } 1\}$ do $\tilde{L}$.

So does not show that $NP \overset{?}{=} co\text{-}NP$. Also does not show $\underline{NL = co\text{-}NL}$, but this is a famous theorem (1988)

- Does not work if M does not halt for all inputs

Proof: Take DTMs M_1 and M_2 such that $L(M_1) = A$ and $L(M_2) = \tilde{A}$. Build M_3 "via flowchart" as follows.

M_3:
input x
Execute one (more) step of the computation $M_1(x)$
Did M_1 accept x in that step?
yes
Accept x.
no
Execute one more step of $M_2(x)$, (if possible)
Did M_2 accept x in that step?
yes
reject x

But we can show

<u>Theorem</u>: if A and $\tilde{A}$ are both c.e., then A is decidable.

Then M_3 is total because for all x, exactly one of $M_1(x)$ and $M_2(x)$ halts and accepts, whereupon $M_3(x)$ halts and decides. So A is decidable. ☒

$\therefore \underline{RE \cap co\text{-}RE = REC}$.

Footnote: Bertrand Russell came up with the most vicious and viscous diagonal set of all:

$$D = \{x : x \notin x\}.$$

The original freewheeling approach to set theory allowed you to state the possibility of a set being a member of itself. If you expect that no set could ever be a member of itself, then D would become "the set of all sets"---but then D would be a member of itself. Indeed, we immediately get the logical equivalence $D \in D \iff D \notin D$. The resolution is not that "$x \notin x$" is always true, but rather that "$x \in x$" and "$x \notin x$" do not compile --- because the $\in$ relation and its negation are allowed to be used only between objects of type T and `set<`T`>` for some type T. This led to a hierarchy of types, which were called "ramified", and a book *Principia Mathematica* that can be analogized to reading the object code of a compiler, both painful...