Lecture 14

Ben Rosenberg

June 29, 2021

Recall:

Definition 1: Language L is recursively enumerable if there exists a Turing machine that accepts L.

Definition 2: Language L is decidable if there exists a Turing machine that accepts L and halts on every input.

Definition 3: (*Halting problem*) $L_H = \{(M, w) | M \text{ is a Turing machine that halts on input } w\}$

Theorem 1: If L_H is recursively enumerable, then the Universal Turing Machine M_U accepts it.

$$M_U(M, w) \to \begin{cases} \text{halt if} & M(w) \searrow \\ \text{diverge if} & M(w) \nearrow \end{cases}$$

Let M_H be the Turing Machine that *decides* L_H:

$$M_H(M, w) \to \begin{cases} \text{halt and accept if} & M(w) \searrow \\ \text{halt and reject if} & M(w) \nearrow \end{cases}$$

M_H does not exist.

Rice's Theorem

Theorem 2: Let β be a nontrivial property of recursively enumerable languages such that $\beta(\emptyset) = 0$. Define a Turing Machine M_β as follows:

$$M_\beta(M) = \begin{cases} \text{halt and accept if} & \beta(L(M)) = 1 \\ \text{halt and reject if} & \beta(L(M)) = 0 \end{cases}$$

In brief: $M_\beta(M) = \beta(L(M))$

$M\beta$ **does not exist.**

Theorem 3: If both L and $\overline{L}$ are recursively enumerable, then both L and $\overline{L}$ are decidable.

Proof: Let M_1 accept L by halting, and let M_2 accept $\overline{L}$ by halting. (Given.)

We know that if L is decidable, then $\overline{L}$ must be too, because we can just flip the acceptance/rejecting states of L to obtain $\overline{L}$.

Construct M that $\underbrace{\textit{decides}}_{\text{will always halt}}$ L and $\overline{L}$.

$w \in L \implies M_1(w) \searrow$

$w \notin L \implies w \in \overline{L} \implies M_2(w) \searrow$

M operates as follows:

M will run M_1 and M_2 in parallel on input w.

$M_1(w)$

$M_2(w)$

Exactly one of M_1 and M_2 will halt, because either $w \in L \implies M_1(w) \searrow$ or $w \in \overline{L} \implies M_2(w) \searrow$.

Whichever halts → decision.

$\overline{L_H} = \{(M, w) | M(w) \nearrow\}$

Corollary 4: $\overline{L_H}$ is not recursively enumerable.

We know that L_H is recursively enumerable because M_U accepts it. But L_H is not decidable because M_H does not exist.

If $\overline{L_H}$ were recursively enumerable then by the previous theorem, both L_H and $\overline{L_H}$ would be decidable. But L_H is not.

Definition 4: A Turing Machine E enumerates a language L if E, starting on empty string as input, writes out exactly the elements of L on its designated tape (on which it moves only to the right).

$\forall x \in L,\ x$ will appear printed

We would call the enumerator tape a "stream" in Java or C++ – it is something from which we read.

`E >> S >> ...`

Definition 5: Lexicographic (dictionary) order of strings assumes an order on alphabet letters, and compares strings according to the leftmost mismatch.

Ex. $\Sigma = \{a, b, c\} \implies a < b < c$

- $a < b$
- $aa < b$

How many strings precede b? $\aleph_0$ of them.

Definition: In the *shortlex* order, the shorter string always precedes the longer; strings of equal length are compared lexographically.

If $|x| < |y| \implies x < y$

If $|x| = |y| \implies$ then compare them lexicographically

Under this system, $aa > b$.

So, every string has only finitely many predecessors under this system.

Algorithm ⑤:

Input: Turing Machine M that decides a language L.

Output: Turing Machine E that enumerates the language L in shortlex order.

Construction:

```
For each $w$ in $\Sigma^*$ in shortlex order {
    If $M(w)$ then print $w$
    else continue
}
```

Recursively, from strings of length k, we go to length $k + 1$ by prefixing each string with a, and then b, etcetera.

Algorithm ⑥:

Input: Turing Machine E that enumerates a language L in shortlex order.

Output: Turing Machine M that decides the language L.

Construction:

Given E such that `E >> S` gives L in shortlex

Need to construct $M(\text{string } w)$

$M(w)$ operates as follows:

```
do
    E >> S
    if (S == w) then
        return true;
    if (|S| > |w|) then
        return false;
while true
```

Algorithm ⑦:

Input: Turing Machine E that enumerates a language L (not necessarily in shortlex)

Output: Turing Machine M that accepts the language L by halting.

Construction:

```
do
    E >> S
    if (S == w) then return true;
while true
```

If $w \in L$, it will come out of E and $M(w) \searrow$. However, if $w \notin L$, then w will not come out of E, and $M(w) \nearrow$.

(And so, this is where the name **recursively enumerable** came from.)

Algorithm ⑧:

Input: Turing Machine M that accepts a language L by halting.

Output: Turing Machine E that enumerates the language L.

We cannot do as in Theorem 1, because for some $s \in \Sigma^*$, $M(s)$ may diverge, $\implies$ we never get any further.

```
For $k = 0, 1, 2, 3, \ldots$ $(k \in \mathbb{N})$ {
    run first $k$ steps of $M(w)$ on every string $|w|$ such that $|w| \leq k$;
    if halting found, print $w$
}
```

Every $w \in L$ ias accepted by M after some n steps, meaning that we will od that computation when our loops guard k reaches max(w, $|w|$).

(This is multithreading with an unbounded number of threads.)

Algorithm ⑨:

Input: DFA M that accepts a language L

Question: Is $L(M) = \emptyset$?

Construction:

Let k be the numberof states of M. Then, $L(M) \neq \emptyset$ if and only if M accepts at least one string with length less than k.

Proof:

Need to prove: If there exists $w \in L$, then there exists one of length $\leq k$.

Assume a shortest $w_0 \in L$ is such that $|w_0| \geq k$. Then, w_0 must pump. Then, we pump w_0 down once.

Then, we obtain $w_1 \in L$ but shorter.

Either $|w_1| \geq k$ or $|w_1| < k$. If $|w_1| \geq k$ then we continue to pump.

Algorithm ⑩:

Input: DFAs M_1 and M_2.

Question: Is $L(M_1) = L(M_2)$?

Reduction:

$L(M_1) = L_1; L(M_2) = L_2$

We know that $L_1 = L_2 \iff L_1 \subseteq L_2 \wedge L_2 \subseteq L_1 \iff L_2 \backslash L_2 = \emptyset \wedge L_2 \backslash L_1 = \emptyset \iff (L_1 \cup \overline{L_2}) \cup (L_2 \cap \overline{L_1}) = \emptyset$

This last language is regular, because regular languages are closed under union, intersection, and complement. As such, we can make an automaton out of this language and test it.

Likewise, $L(M_1) \subseteq L(M_2)$? We only need to use one side of the union in the above regular language.

Algorithm ⑪:

Input: DFA M that accepts a language L.

Question: Is $L(M)$ infinite?

Construction:

Let k be the number of states of M. $L(M)$ is infinite if and only iff M accepts at least on string with length no less than k but less than $2k$.

This is because this string will pump up $\implies$ there are infinitely many good strings $\implies$ $L(M)$ is infinite.

Now we need to prove the other direction – namely, that if $L(M)$ is infinite, then it has a good string of length $< 2k$ and $\geq k$.

Assume the opposite – that a shortest string which is larger than k is also larger than $2k$. Call this string w_0. w_0 must pump. Pump it down.

It is impossible to jump, in one pump, from more than $2k$ to less than k, because the pumping window is less than k by the pumping lemma.

Is $L(M) = \emptyset$? Is $L(M)$ infinite? Is $L(M) = L$? All these questions about $L(M)$ are unsolvable if M is a Turing Machine.

Algorithm ⑫:

Input: Context-free grammar G

Question: Is $L(G) = \emptyset$?

Construction:

We could use P.L. again.

Or, we could use the closure argument.

Input: $G = (V, \Sigma, P, S)$

Question: Is $L(G) = \emptyset$?

Construction:

Operation of *marking a symbol*

Markable symbols are elements of $V \cup \Sigma \cup \{\lambda\}$

How do we mark?

Base case:

mark λ, all elements of Σ

Recursively, until no further marking possible:

for each rule $[L \to D] \in P$

if the entire D (every symbol of D) is marked, then mark L.

once this stops:

return $L(G) \neq \emptyset$ if and only if the start symbol is marked.

Marking something means we can get to a terminal from it.

A variable is marked if and only if it can derive a terminal string.

Problem 1:

$G = (V, \Sigma, P, S)$, where $\Sigma = \{a, b, c\}$ and $V = \{S, T, A, B, D, H\}$ and the production set P is

- $S \to DA|DB|DT$
- $A \to AA|DA|D$
- $B \to AB|BA|H$
- $D \to aABDc|bBADc|HHc$
- $H \to bBH|cAH|T$
- $T \to TT|a|A$

The first rule that gives us something to mark is the last one, $T \to a$, as a is marked. Next, we look at rule $H \to T$ and we mark H because T has been marked. Then, we look at the rule $B \to H$, and then rule $D \to HHc$, and then $S \to DB$; so, the grammar can produce a terminal string.

And so, $L(G) \neq \emptyset$.

How many strings are there of length $\leq k$? There are about $|\Sigma|^k$ such strings.

The above algorithm runs in about $|V| \cdot |P|$ steps. Thus, it is more efficient than our previous ones.

But in order to use this, we need to convert the DFA into an NFA which is an exponential construction and thus still inefficient.

For context-free grammars, only these two questions are *solvable* (pumping lemma):

- Is $L(G) = \emptyset$?
- Is $L(G)$ infinite?

Anything about complement or intersection is *unsolvable*.

Recall Rice's Theorem: How to make M_H

M_H (M, w)$:

Construct D where $D(x) : M(w); M_1(x)$

return $M_\beta(0)$

Anything that can simulate universal computation like D can is **undecidable**.

Post's Correspondence Problem

Definition:

Input: A finite set of dominoes, where domino j contains a pair of strings p_j (upper) and ℓ_j (lower).

A win (solution) is a finite sequence in which every domino appears (placed vertically) zero or more times, such that the concatenation of upper strings is equal to the concatenation of lower strings.

Question: Does this set of dominoes have a win (solution)?

Post's Correspondence Problem is undecidable.

There is no algorithm to tell whether a given instance of P.C.P. has a win.

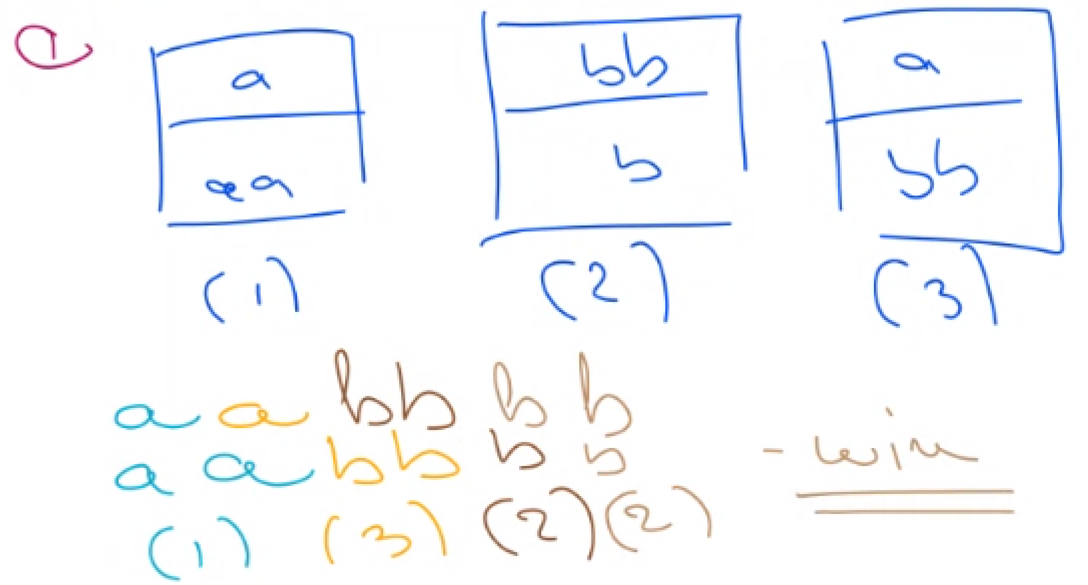
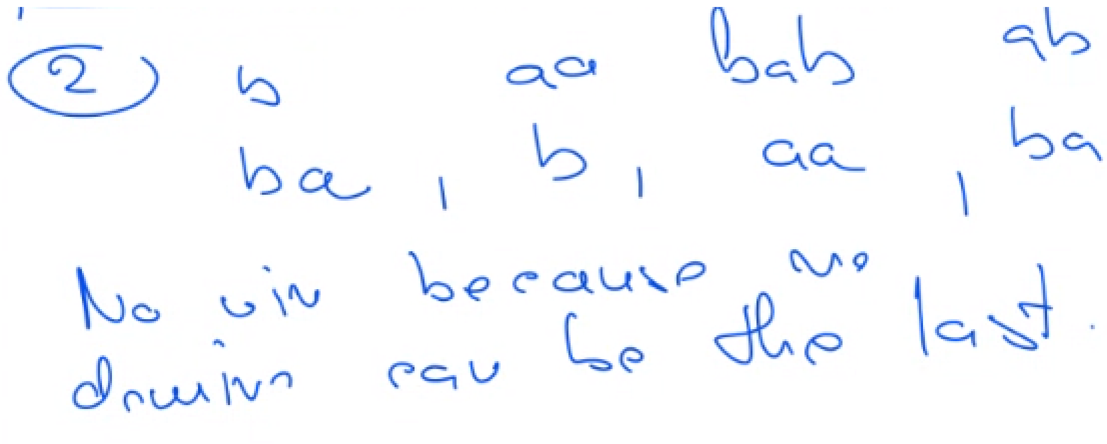


Figure 1: Example P.C.P. solution

If we are *told* there is a win, then we can generate all the finite sequences and eventually terminate.



② b aa bab ab
ba , b , aa , ba

No win because no domino can be the last.

Figure 2: Example impossible P.C.P.