Gödel Incompleteness

Logical Theory of Natural Numbers:
$\mathbb{N} = \{0, 1, 2, \ldots\}$
with addition & multiplication

Peano Axioms: symbols $S, +, \cdot, 0$

S: Successor means "+1"
$S0$ instead of 1
$SS0$ instead of 2
etc

Axioms:

$$\forall x\,(Sx \neq x)$$
$$\forall x \neq 0\ \exists y\ (Sy = x)$$
Successor

$$\forall x\,(x + 0 = x)$$
$$\forall x\,(x + Sy = S(x+y))$$
$+$

$$\forall x\,(x \cdot 0 = 0)$$
$$\forall x\,(x \cdot Sy = x \cdot y + x)$$
$\cdot$

Infinite number of axioms but easy to enumerate

Induction: For each formula P:
$$[\forall x((P(0) \wedge P(x) \rightarrow P(Sx)) \rightarrow \forall x\, P(x)$$

Methods of Proof:

Predicate Logic inference rules
(as covered in 311)

A proof of ψ is a sequence of formulas:
$\psi_1, \psi_2, \psi_3, \ldots, \psi_t = \psi$
s.t. every ψ_i is either
- An axiom
- or follows from previous formulas via an inference rule

ψ is provable iff there is such a proof

Defn $Th(\mathbb{N},+,\cdot)$ all statements that are <u>true</u> about $\mathbb{N},+,\cdot$

Thm The set of <u>provable</u> statements about $(\mathbb{N},+,\cdot)$ is Turing-recognizable

Proof <u>Recall</u>: T-rec $\Leftrightarrow$ recursively enumerable r.e.

We build an enumerator for the set of provable statements:

<u>Idea</u>: Create a TM that produces all possible proofs in lexicographic order. At each step can choose either an axiom to include or prior statements to apply an inference rule to. Output each <u>formula</u> produced

every formula in a <u>proof</u> is a provable <u>statement</u>

<u>Goal</u>

This needs refinement since we have an infinite # of axioms from the induction schema

<u>Dovetail</u>: Instead we could list possible proofs in order of the # of symbols by listing strings in order and, if the string is a properly formatted string, printing the final formula produced

Note: Though this tries all proofs in lexicographic order, the formulas being proved will not be in order. (Short statements may require very long proofs.)

Another way to do this as described in class would be to loop over all t, listing all axioms of size $\leq t$, and then trying all

BFS (derivations using up to t steps and printing the results

Every possible proof will be explored so all possible provable statements will eventually be produced

$\blacksquare$

Thm $Th(\mathbb{N}, +, \cdot)$ is undecidable

true statements about $\mathbb{N}, +, \cdot$

Proof: Reduction from A_{TM} using accepting computation histories

Idea: Can encode strings as numbers

Accepting computation histories of M on input w $\Longleftrightarrow$ natural numbers of a special form

x : candidate natural number of this special form

$$\underset{A_{TM}}{\langle M,w\rangle} \xmapsto{f} \underset{Th(\mathbb{N},+,\cdot)}{\exists x.\, \phi_{M,w}(x)}$$

encodes accepting computation of M on input w

s.t. $\phi_{M,w}(x) = \begin{cases} \text{true} & \text{if } x \text{ is of the special form} \\ \text{false} & \text{otherwise} \end{cases}$

Idea: using quantifiers, $+$, $\cdot$ can do things like mods to extract numbers representing individual configurations in the history and check that consecutive configurations satisfy $C_i \vdash_M C_{i+1}$

e.g. "$x<y$" $\exists z\,(y = x+z) \wedge (z \neq 0)$

"$r = x \bmod y$" $\exists q\, \exists r\,((x = q\cdot y + r) \wedge (r<y))$

i.e. $\exists q\, \exists r\,((x = q\cdot y + r) \wedge (\exists z.\, y = z+r))$

Can decode strings.

don't need all the axioms, a finite set Q is enough

Lots of tedious details that we will skip but the function f is computable

$A_{TM} \leq_m Th(\mathbb{N},+,\cdot)$. ∎

Thm $\exists$ a (true) statement in $Th(\mathbb{N},+,\cdot)$ that is not provable.

Proof **Idea:** For each fully quantified statement φ about $(\mathbb{N},+,\cdot)$ exactly one of φ or $\neg\varphi$ is true

Suppose that every (true) statement in $Th(\mathbb{N},+,\cdot)$ were provable. ⊛

Claim: This would yield a decider for $Th(\mathbb{N},+,\cdot)$ as follows:

On input φ:
- Run enumerator for the set of provable statements in $Th(\mathbb{N},+,\cdot)$
- Either φ or $\neg\varphi$ is true so by assumption ⊛, one of φ or $\neg\varphi$ will be produced by the enumerator
- If φ is produced accept
- If $\neg\varphi$ is produced reject.

This would decide $Th(\mathbb{N},+,\cdot)$ which is impossible so the assumption is false ∎

Gödel found a single explicit statement that is true but not provable.
It expresses:

"This statement is not provable."

To produce this, one needs a statement that can talk about itself.
Can do this based on:

Recursion Theorem (see 6.1 in Sipser)

Can produce a program that prints out its own code. } Fun exercise

Related:

Gödel's Completeness Theorem:
If a set of formulas doesn't yield a provable contradiction then there is a world ("model") where it is true

Consider an enumerable set of axioms A for $Th(\mathbb{N},+,\cdot)$ and φ s.t. neither φ nor $\neg\varphi$ is provable

$\therefore$ $A \cup \{\varphi\}$ doesn't yield a contradiction

$\therefore$ $A \cup \{\neg\varphi\}$ doesn't yield a contradiction

true in different worlds

0 1 2 . . . standard model $\mathbb{N}$

$\therefore$ $\Rightarrow$ There exist "non-standard" models for A

$\mathbb{N}$: 0 1 2 . . . $\mathbb{Z}$: . . . -2 -1 0 1 2 . . . $\mathbb{Z}''$: . . . . .

This yields Gödel's incompleteness result that there is no theory that includes the $Th(\mathbb{N},+,\cdot)$ that has a unique model.

After Turing
Restricted computing models

McCulloch & Pitts (1943)
Neural Nets (precursor to deep nets)
as models of brain

Kleene (1951)
Neural Nets
complicated ≡ Finite State Machine (DFA) defined these and the terms
≡ Regular Expressions

Chomsky (1956)
Backus-Naur
Chomsky Context-Free languages as model of human languages
also did context-sensitive languages
Backus-Naur (aka Backus-Naur Form Grammars)
BNF for computer languages

Rabin & Scott (1959)
Turing Award winning work
Nondeterministic Finite Automata (NFA)
introduced nondeterministic machines
Idea really simplified Kleene's Thm

Following this Claim: CFGs ≡ Pushdown Automata (PDA)
Nondeterministic automata with a stack

For example, consider the grammar

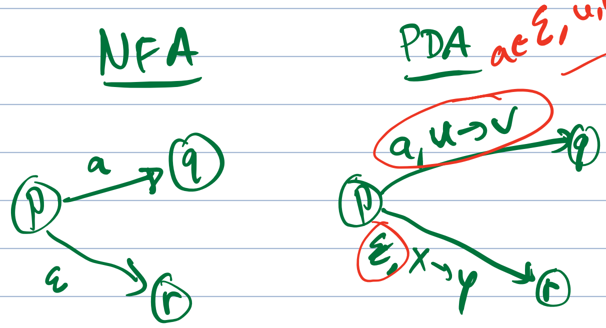
$S \to (S) \mid SS \mid \varepsilon$ for strings of balanced parentheses

Natural PDA algorithm:
push every (onto the stack
pop (when reading a)
reject if no (for a) or
if stack is not empty at end.

Aside added to notes but not part of lecture

Here's what a PDA looks like

You don't need to know this

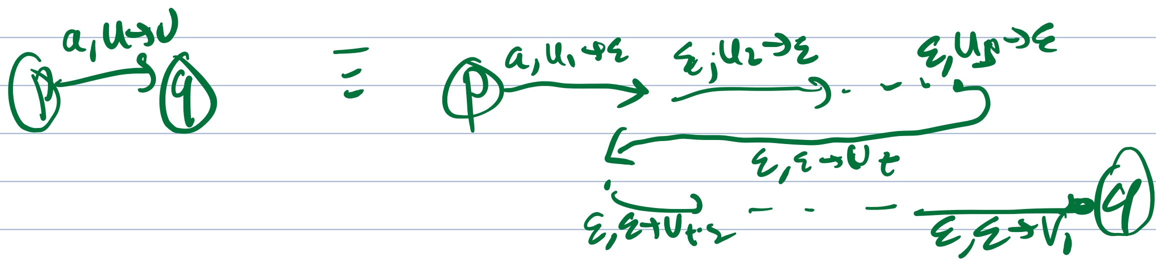


$a \in \Sigma_\varepsilon$, $u, v \in \Gamma^*$

meaning if next input char is an a and top of stack is u can replace u by v and move past a.

Note: Normally we think of only being able to see top symbol on the stack. but we can convert such a machine into one with that restriction:

Suppose $u = u_1 \cdots u_s$ $v = v_1 \cdots v_t$



In general CFGs / PDAs
need look-ahead
to know which rule to
.

Programming languages are designed so that one doesn't need to look ahead to know what rule to use

So far we claimed that A_{CFG} was decidable but didn't give a proof..

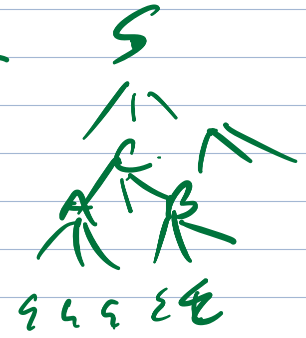
Thm A_{CFG} is decidable

Proof: Algorithm that doesn't work (yet)

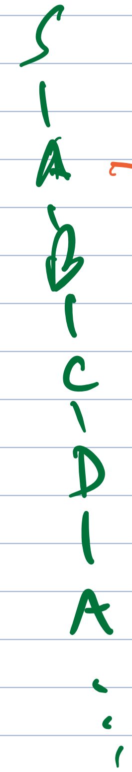
"On input $\langle G, w \rangle$ try all possible parse trees using breadth first search to see if w is generated."

Problem: No bound on size of parse trees that can produce w.
When to reject?

eg -



don't produce any char



looping behavior

<u>solution</u>: Special form of grammars so that the above algorithm can work because parse trees are of limited size in terms of $|w|$.

"Chomsky Normal Form"

- No use $A \to \varepsilon$ unless $w = \varepsilon$
- No $A \to B$ (unit) rules

New alg: On input <G, w> convert G to G' and run alg on <G', w>.

Chomsky Normal Form Conversion

rules of form

$A \to BC$ $B, C \in V$, $B, C \neq S$

$A \to a$

$S \to \varepsilon$

Problems for general rules

4 • $A \to \varepsilon$ for $A \neq S$

2 • Right-hand side of length >1 containing $a \in \Sigma$

3 • Right-hand sides of length >2

• Rules of form $A \to B$ (unit rules)

1 • S on RHS of a rule

We get rid of these step by step:

$$S \to (S) \mid SS \mid \varepsilon$$

① Add new S_0 and rule $S_0 \to S$

new start symbol

$$S_0 \to S, \quad S \to (S) \mid SS \mid \varepsilon$$

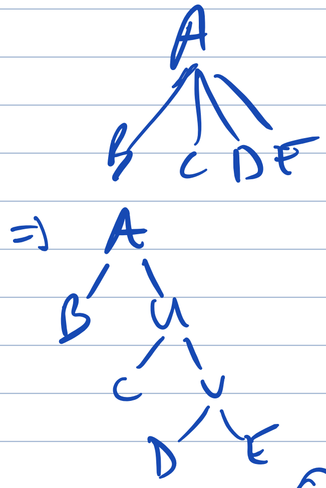
② Add new var U_a for each $a \in \Sigma$

replace a in long right hand sides with U_a and add rule $U_a \to a$

$$S_0 \to S, \quad S \to UST \mid SS \mid \varepsilon, \quad U \to (, \quad T \to)$$

③ Rules of length > 2

Add a chain of new intermediate variables to break up into size 2



$S_0 \to S$, $S \to UV \mid SS \mid \varepsilon$, $U \to ($, $T \to)$
$V \to ST$

④ Compute $\mathcal{E}$ the set of variables that can produce ε:

If $A \to \varepsilon$ is a rule add A to $\mathcal{E}$

Repeat: if $A \to BC$ is a rule with $B, C \in \mathcal{E}$ add A to $\mathcal{E}$

$\mathcal{E} = \{S_0, S\}$

Add

$S \to S$ not needed

$V \to T$

$S_0 \to \varepsilon$

(a) if $A \to BC$ where $B \in \mathcal{E}$ add rule $A \to C$

(b) if $A \to BC$ where $C \in \mathcal{E}$ add rule $A \to B$

(c) If $S_0 \in \mathcal{E}$ add rule $S_0 \to \varepsilon$

(d) remove all rules $A \to \varepsilon$ for $A \neq S_0$

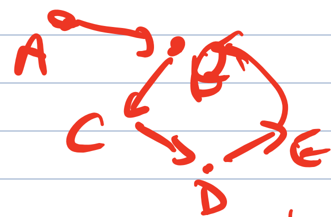
Any time such a rule is used it can be replaced by (a), (b), (c) rules above

$S_0 \to S \mid \varepsilon$, $S \to UV \mid SS \mid S$, $U \to ($, $T \to)$, $V \to ST \mid T$

Note: we added a number of unit rules that might not have been there before

5. Get rid of unit rules:

Create a directed graph on variables where there is an edge $A \to B$ iff $A \to B$ unit rule



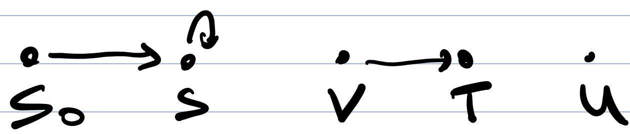
Note: can do rules that walk around this graph doing replacements but eventually need to do a non-unit rule at one of the vars reachable

$S_0 \to S \mid \varepsilon$, $S \to UV \mid SS \mid S$, $U \to ($, $T \to)$, $V \to ST \mid T$

non-unit rules marked.

- Add all non-unit RHS to any var that can reach them
- Remove unit rules

Graph



Final Grammar in Chomsky Normal Form

$$S_0 \to UV \mid SS \mid \varepsilon$$
$$S \to UV \mid SS$$
$$U \to ($$
$$T \to)$$
$$V \to ST \mid)$$

Proving that languages are not context-free.

Note: related lemma for regular languages is not as good as 311 methods

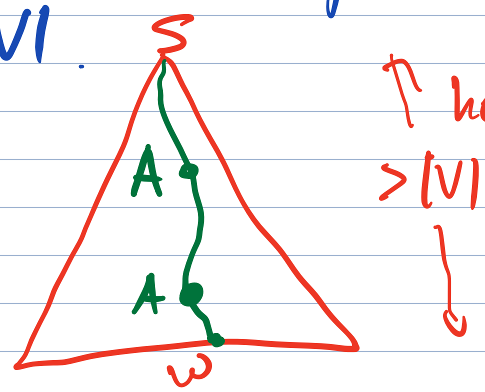
Pumping Lemma for Context-Free Languages

If L is a CFL then $\exists$ integer p (pumping length)
s.t.
$\forall w \in L$ with $|w| \geq p$
we can write $w = uvxyz$ s.t.
1. $|vy| \neq 0$ (can strengthen to $v \neq \varepsilon$)
2. $|vxy| \leq p$
3. $\forall i \geq 0$. $uv^ixy^iz \in L$

Proof Let G be CFG with $L = L(G)$
and assume that G is in
Chomsky Normal Form.

Let V = # of variables in G

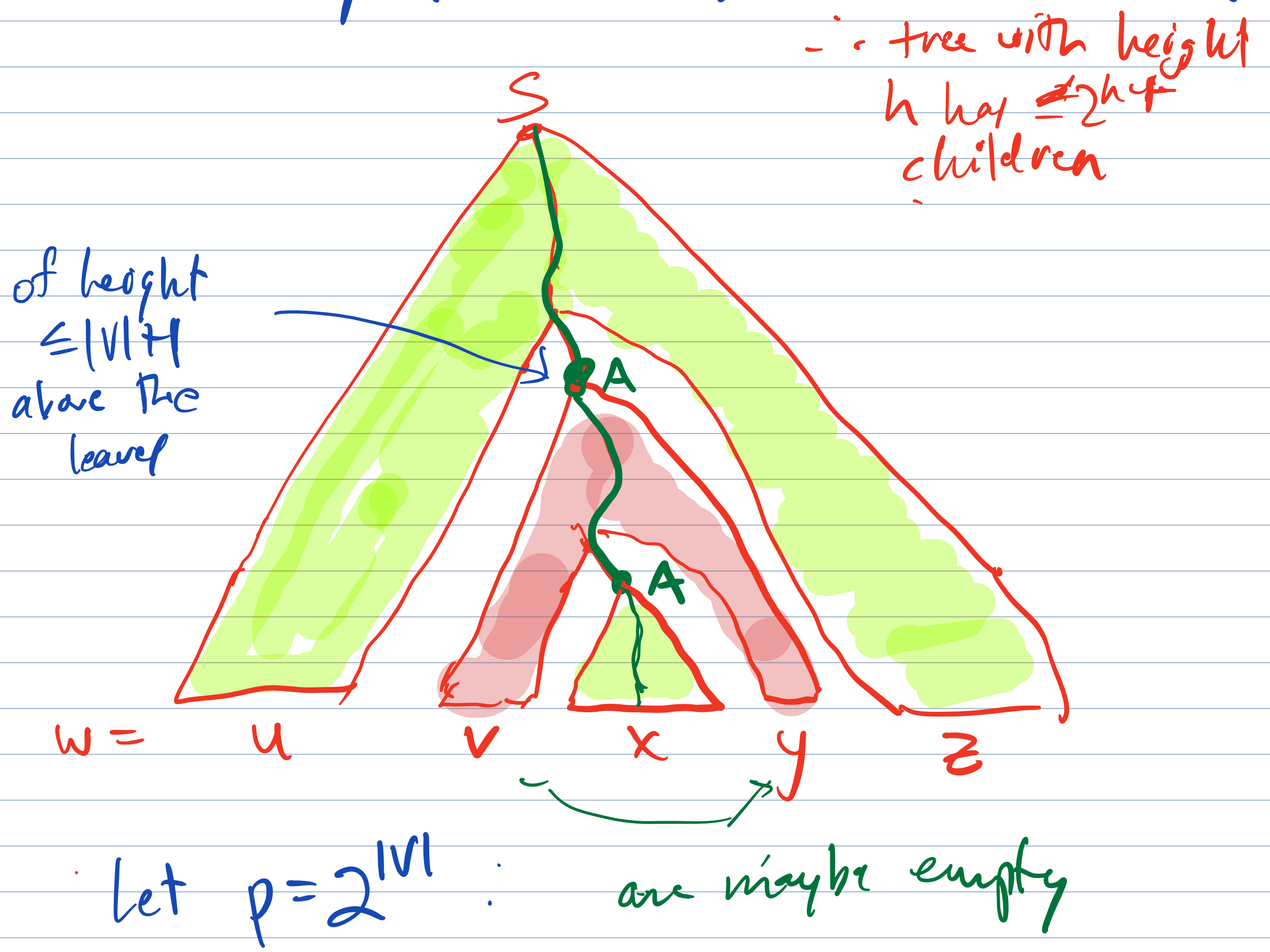
Suppose that $w \in L$ has a parse tree
of height $> |V|$.



$\Rightarrow$ root-leaf path
length $> |V|$
must contain repeated
variable

by the Pigeonhole
principle. Choose lowest repeat.

Note: Since Chomsky Normal Form
- every leaf has a symbol in Σ
- parent of each leaf has 1 child
- every other internal node has 2 children



Let $p = 2^{|V|}$.

- If $|w| \geq p$ then w requires parse tree height $\geq |V|+1$

$\Rightarrow$ parse tree has repeated variable on a path at height at most $|V|+1$ above leaves

Break up $w = uvxyz$ as in picture.

Since this is Chomsky normal form we must have either v or y (or both) non-empty since no unit rules

$\therefore$ ① is true

Since top A which generates vxy has height $\le |V|+1$

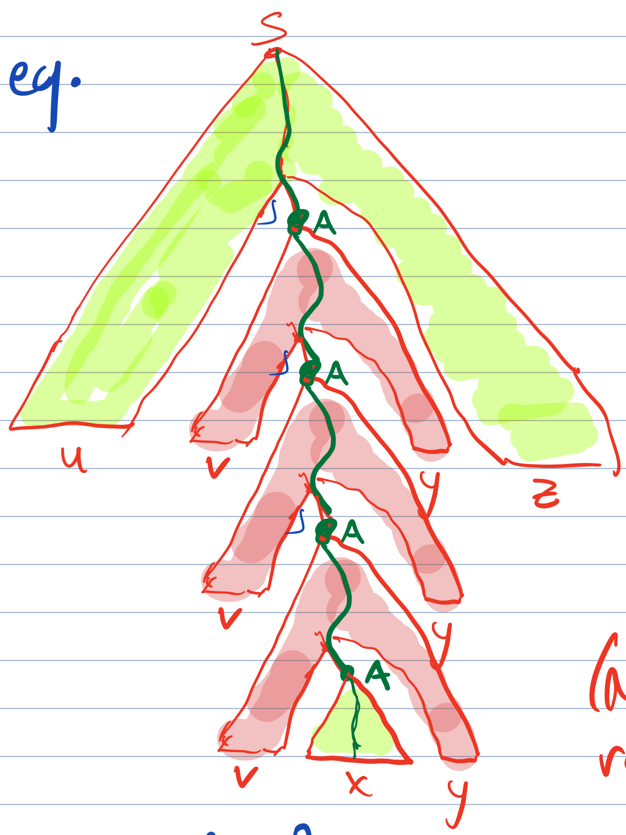
$\therefore |vxy| \le 2^{|V|} = p$

$\therefore$ ② is true

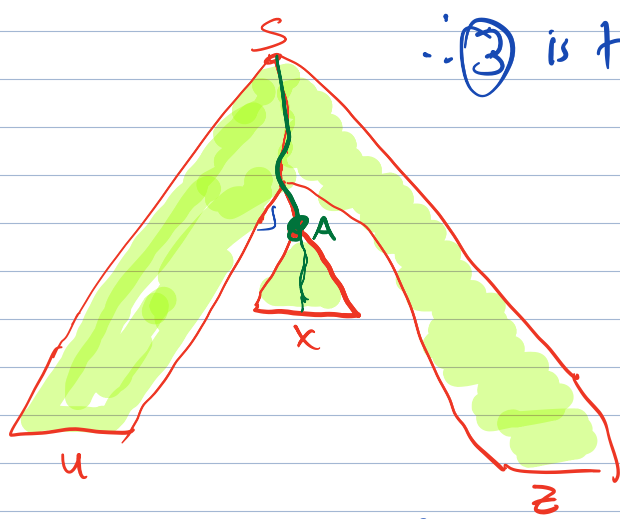
Finally, for ③, $w \in L$ is the case uv^1xy^1z, $i=1$.

We can repeat the red section that generates u and v any number of times

eg.



uv^3xy^3z



$\therefore$ ③ is true

$uxz = uv^0xy^0z$

(Note if $v=\varepsilon$ we can rewrite $w = u_1v_1x_1y_1z_1$ where $u_1 = uvx$, $v_1 = y$, $x_1 = y_1 = \varepsilon$, $z_1 = z$)

How to use this to show not context-free:

Show: $\forall p \exists w \in L$ $\forall$ ways of breaking up w into $uvxyz$,
$\exists i$ s.t. $uv^ixy^iz \notin L$

---

e.g. $L = \{x\#x : x \in \{0,1\}^*\}$ is not a CFL

Let p be the pumping length for L

Consider $w = \underbrace{0^p1^p}_{1^{st}\text{ block}} \# \underbrace{0^p1^p}_{2^{nd}\text{ block}} \in L$

What are options for vxy as part of w?

- if vxy all in 1^{st} block:
  For all $i \neq 1$, $uv^ixy^iz \notin L$
  part before # won't match part after
- if vxy all in 2^{nd} block:
  Same as above since parts won't match
- if v or y contains #:
  For $i \neq 1$, have too many/few #
- if v in 1^{st} block and y in 2^{nd} block
  then since $|vxy| \leq p$.
  v has only 1's
  y has only 0's

  and again # of 1's and 0's won't match when pumped $i \neq 1$.

only cases

detail not done in class

$\therefore$ L is not a CFL

Time Complexity

Def^n The running time of a NTM M is the function $T: \mathbb{N} \to \mathbb{N}$ given by:

$T(n) = \max \{$ # steps M takes on any input $w \in \Sigma^*$ with $|w| = n$ & and any computation path that M may take on input w. $\}$

This gives the definition for both deterministic and nondeterministic TMs

---

Def^n For $T: \mathbb{N} \to \mathbb{N}$ define

$NTIME(T(\cdot)) = \{ A$: there is a multitape NTM that decides A with running time that is $O(T(n)) \}$

language

add these for the nondeterministic case

Note: text uses single tape, but multitape is used by researchers. more like other models.

For example multitape TMs vs RAMs.

Random Access Machine (RAM)

(used in data structures and algs)

such that an operation costs time ≈ # of bits in words

Multitape TM

$T(n) \Longrightarrow O(T(n) \log T(n))$ time on multitape TM

typically words use $O(\log n)$ bits for inputs of "size" n

Why multitape?

<u>Examples</u> $A = \{x\#x : x \in \{0,1\}^*\}$

1-tape TM can't do this better

1-tape TM from 1st TM we produced running time $O(n^2)$ – need to shuttle back & forth

2-tape TM : copy part before # to tape 2
time $O(n)$ compare tapes 1 and 2

$\therefore A \in TIME(n)$ linear time

<u>Recall</u>: Simulation of k-tape TM by 1-tape TM

k-tape TM $T(n)$ $\Rightarrow$ 1-tape TM $O(T^2(n))$

Best possible simulation even to go from 2-tape to 1-tape because of above example

Can actually prove:

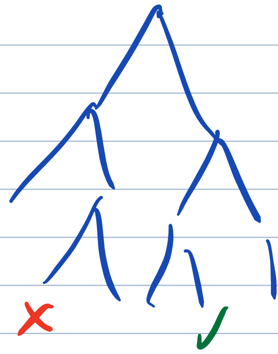
Hartmanis Lewis Stearns 1965

k-tape TM $T(n)$ $\Rightarrow$ 2-tape TM $O(T(n)\log T(n))$

• Also "oblivious" head position only depends on $|w|$, not w

Recall: NTM time $T(n)$ $\Rightarrow$ 1-tape TM time $2^{O(T(n))}$

$T(n)$



need to explore $2^{O(T(n))}$ leaves

Fact: If $P=NP$ then one could get a vastly better simulation:

we'll define these in a moment

e.g. $(T(n))^k$ for some k for each language A (k might depend on A)

Thm $\forall T$.

constant in the $O(\ )$ above

$$\mathrm{TIME}(T(n)) \subseteq \mathrm{NTIME}(T(n)) \subseteq \bigcup_c \mathrm{NTIME}(2^{cT(n)})$$

Polynomial Time

Edmonds, Cobham 1965:

polynomial-time = good algorithm

more than polynomial-time = bad algorithm

Def^n P polynomial time

$$P = \bigcup_k \mathrm{TIME}(n^k)$$

all languages that can be decided in time $O(n^k)$ for some constant k.

Def^n NP nondeterministic polynomial time

$$NP = \bigcup_k \mathrm{NTIME}(n^k)$$

Question: Does $P \stackrel{?}{=} NP$

Implicit in Edmonds work

Cook 1971

Karp 1972

Levin 1973

Note:
- $P \subseteq NP$ (every TM is an NTM)
- P is a set of languages (decision problems)

FP is the set of polynomial-time computable functions

We start with some examples:

PATH = { <G, s, t> : G is a directed graph with a path from s to t }

Thm PATH $\in$ P

Proof BFS, DFS, (or any general Graph Search) are all polynomial time

Note: Doesn't matter if graph is given by

input size n

poly time to convert betw

adjacency matrix $\Theta(N^2)$

adjacency lists $\Theta((N+M)\log N)$

edge lists $\Theta(M \underbrace{\log N}_{})$

N vertex, M edge graph

need $\Theta(\log N)$ bits to represent each vertex names ∎

Encoding < > : For computability it didn't matter much.
For complexity, we assume efficient use of an alphabet of size ≥ 2 (w.l.o.g. binary)

RELPRIME = {<a,b> : a,b are integers with gcd(a,b)=1}

binary encoding.

Thm RELPRIME $\in$ P

Proof: Claim: Euclid's Algorithm is polynomial time

Let's recall how Euclid's algorithm works

$a_0 = a \qquad a_1 = b$

Repeatedly Compute $\underbrace{a_i = q_i a_{i-1} + a_{i+1}}_{\text{long division with remainder}}$ for $0 \le a_{i+1} < a_{i-1}$
q_i, a_{i+1} integers

Computable in time $O(n^2)$
using grade school algorithm

$a_0 = q_1 a_1 + a_2$

$a_1 = q_2 a_2 + a_3$

$\vdots$

$a_{t-1} = q_{t-1} a_t + a_{t+1}$ $= \gcd(a,b)$ must be 1 for RELPRIME

$a_t = q_t a_{t+1} + \underbrace{0}_{a_{t+2}}$

How many steps t?

Recall Fibonacci numbers

$F_0 = 0 = a_{t+2}$

$F_1 = 1 = a_{t+1} \qquad F_i = F_{i-1} + F_{i-2}$

$a_{t-1} \geq a_{t+1} + a_{t+2} = F_1 + F_0 = F_2$
since $q_{t-1} \geq 1$

generally,

$a_i \geq a_{i+1} + a_{i+2}$

So $a_1 \geq F_t$ t-th Fibonacci #

Recall from CSE 311: $2^t \geq F_t \geq 2^{t/2} - 1$ for $t \geq 2$

Actually $F_t = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^t - \left(\frac{1-\sqrt{5}}{2}\right)^t\right)$

$\frac{1+\sqrt{5}}{2}$ = golden ratio $\approx 1.618\ldots$

exponential in t

absolute value $0 < \frac{\sqrt{5}-1}{2} < 1$

basically rounded to near integer

$\therefore$ F_t takes $\Omega(t)$ bits

$\therefore$ input size in bits n is $\Omega(t)$

$\therefore$ # steps t is $O(n)$

$\therefore$ # of steps in linear in n & total runtime is $O(n^3)$ ∎

By contrast consider

$\text{FACTOR} = \{\langle N, k, \ell\rangle$: integer N has an integer factor m with $k \le m \le \ell\}$

n-bits each

For example N is prime iff $\langle N, 2, N-1\rangle \in \text{FACTOR}$.

If we could decide factor efficiently we could binary search to find the factors.

Thm FACTOR $\in$ NP

Proof On input $\langle N, k, \ell \rangle$ of n bits

Use nondeterminism to write a string of length $\leq n$ representing an integer m

m is "guess", any n-bit string might be written.

Now check that

All are efficient checks for binary representations.

- $k \leq m$
- $m \leq \ell$
- m divides into N exactly

$\blacksquare$

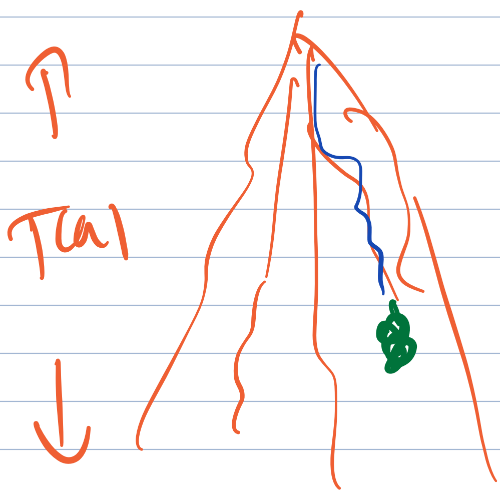
Open: Is FACTOR $\in P$?

If yes then this would break SSL and all of our current e-commerce security.

The NP algorithm for FACTOR split the nondeterministic algorithm into 2 parts

- A nondeterministic "guess" of a string that just depends on the input length
- A deterministic verification that the string is good

This can be done in general for NTIME(T(n)) without loss of generality:



- Guess a string of length $\leq T(n)$ representing the target address defining a potential accepting path
- Deterministically simulate for up to T(n) using this path to make choices and accept if node reached is accepting