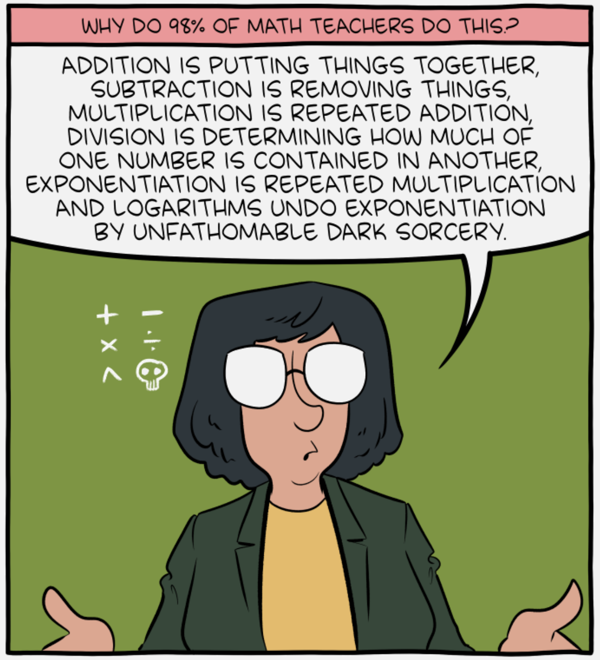
UMB CS622

Log Space (L and NL)

Wednesday, December 1, 2021



Announcements

- ~~HW 9 in~~
  - ~~Due Tues 11/30 11:59pm EST~~

- HW 10 out
  - Due Tues 12/7 11:59pm EST

- HW 11 will be the last assignment
  - Due Tues 12/14 11:59pm EST

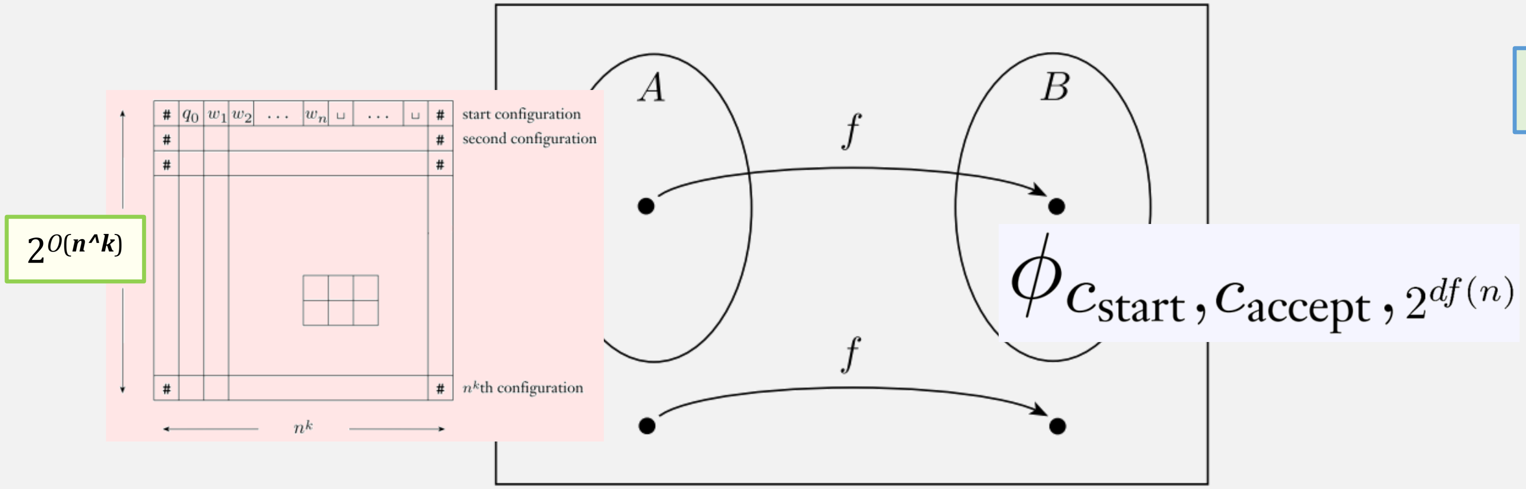
Last Time: **PSPACE**-Completeness

DEFINITION

A language B is ***PSPACE-complete*** if it satisfies two conditions:

1. B is in PSPACE, and
2. every A in PSPACE is polynomial time reducible to B.

If B merely satisfies condition 2, we say that it is ***PSPACE-hard***.



The first **PSPACE**-complete problem:

THEOREM

TQBF is PSPACE-complete.

PSPACE and Board Games

Note: These are for "generalized" games, e.g., $n \times n$ boards

Any problem for fixed size n is "solvable"

NIST Recommended Key Sizes

Date	Minimum security level (in bits)	Symmetric algorithm	RSA key size (in bits)
2010 (Legacy)	80	3DES with 2 keys	1,024
2011–2030	112	3DES with 3 keys	2,048
> 2030	128	AES-128	3,072
>> 2030	192	AES-192	7,680
>>> 2030	256	AES-256	15,360

The date is a projection of how far into the future the security level will be adequate. For example, to encrypt data now that should still be secret in 2031, use at least a security level of 128 bits.

Source: http://www.keylength.com/en/4/

GO is Polynominal-Space Hard

DAVID LICHTENSTEIN AND MICHAEL SIPSER

University of California, Berkeley, California

ABSTRACT. It is shown that, given an arbitrary GO position on an $n \times n$ board, the problem of determining the winner is Pspace hard New techniques are exploited to overcome the difficulties arising from the planar nature of board games In particular, it is proved that GO is Pspace hard by reducing a Pspace-complete set, TQBF, to a game called generalized geography, then to a planar version of that game, and finally to GO.

The Complexity of Checkers on an N × N Board - Preliminary Re

A. S. Fraenkel,[1] M. R. Garey,[2] D. S. Johnson,[2]

T. Schaefer,[3] and Y. Yesha[1]

Abstract

We consider the game of Checkers generalized to an $N \times N$ board. Although certain properties of positions are efficiently computable (e.g., can Black jump all of White's pieces in a single move?), the general question, given a position, of whether a specified player can force a win against best play by his opponent, is shown to be PSPACE-hard. Under certain reasonable assumptions about the "drawing rule" in force, the problem is itself in PSPACE and hence is PSPACE-complete.

nonconstructive and no polynom strategy is known. It might well initial position like that pictured in

For this reason, we shall regarded as "end-game" problems given position whether or not a p A *position* is specified by giving squares on the $N \times N$ checkerboa (i.e., Black piece, Black king, Wh and (2) the name (Black or White)

On the Complexity of Chess

JAMES A. STORER*

Bell Laboratories, Murray Hill, New Jersey 07974

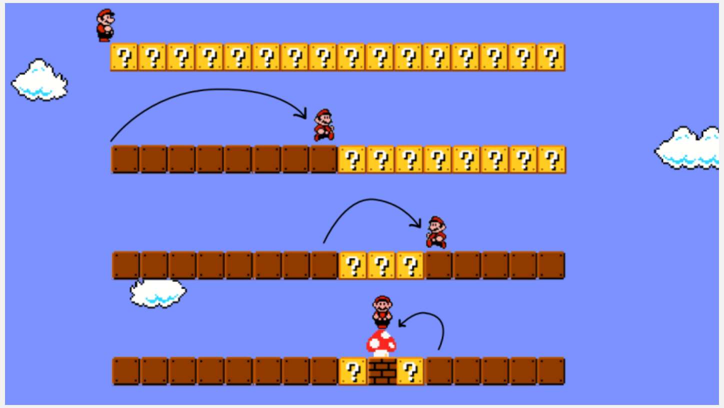
Received June 29, 1979; revised December 12, 1980

It is shown that for any reasonable generalization of chess to an *NxN* board, deciding for a given position which player has a winning strategy it is PSPACE-complete.

1. Introduction

Most past work analyzing games from the point of view of computational complexity has dealt with combinatorial games on graphs (e.g., Even and Tarjan [3], Schaefer [10], Chandra and Stockmeyer [2]). However, recently Fraenkel *et al.* [5], and Lichtenstein and Sipser [8] have considered the game of checkers and GO, respectively. These authors show that for generalizations of checkers and GO to an *NxN* board, it is PSPACE-hard[1] to determine if a specified player has a winning strategy. This paper shows that for a wide class of generalizations of chess to an *NxN* board, it is PSPACE-complete to determine if a specified player has a winning

What About Sublinear Algos?



TIME:

- Need at least ***n*** steps to read input of length ***n***
- We won't look at this for CS622

THEOREM

Savitch's theorem For any function $f : \mathcal{N} \longrightarrow \mathcal{R}^+$, where $f(n) \geq n$,

$$\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n)).$$

SPACE:

- Need at least ***n*** tape cells to store input of length ***n***
- To model sublinear space algorithms (e.g., log space):
  - Modify TM to only count <u>extra</u> non-input space usage

A Read-only Input, 2-Tape TM

2 tapes

- Tape 1: Read-only input tape
- Tape 2: Read/write work tape

Space complexity: only counts the work tape

We use this TM to model sublinear space algorithms

L and **NL**

DEFINITION

L is the class of languages that are decidable in logarithmic space on a deterministic Turing machine. In other words,

$$\text{L} = \text{SPACE}(\log n).$$

the class of languages that are decidable in logarithmic space ondeterministic Turing machine. In other words,

$$\text{NL} = \text{NSPACE}(\log n).$$

In this lecture:
"Turing machine"
=
Read-only Input, 2-Tape TM

Flashback: $A = \{0^k 1^k \mid k \geq 0\}$

M_1 = "On input string w:
1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*."

- "Crossing off" uses $O(n)$ space …
  - … because the input is modified,
  - … so it must first get copied to "work tape"

Any algorithm that modifies input is $O(n)$ space at minimum

$A = \{0^k 1^k \mid k \geq 0\}$ is a member of L

- Instead of crossing off input directly, keep <u>two counters</u>
  - Counter for `0`s
  - Counter for `1`s
- Each counter requires …
- … log space!

In general, the space required for storing a number $x = \log(x)$ (i.e., its binary representation)

Flashback: Theorem: $PATH \in \mathrm{P}$

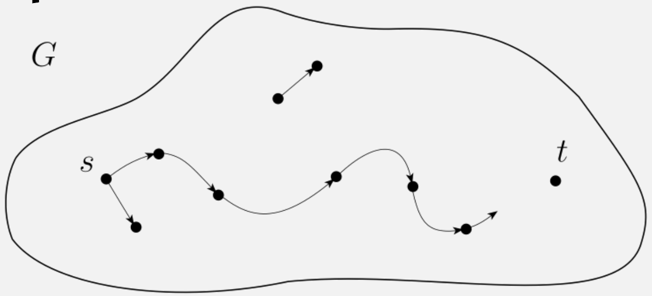
$PATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$

PROOF A polynomial time algorithm M for $PATH$ operates as follows.

M = "On input $\langle G, s, t\rangle$, where G is a directed graph with nodes s and t:

1. Place a mark on node s.
2. Repeat the following until no additional nodes are marked:
3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.
4. If t is marked, *accept*. Otherwise, *reject*."

- Modifying input requires moving it onto work tape
- So, again, this also uses $O(\boldsymbol{n})$ space

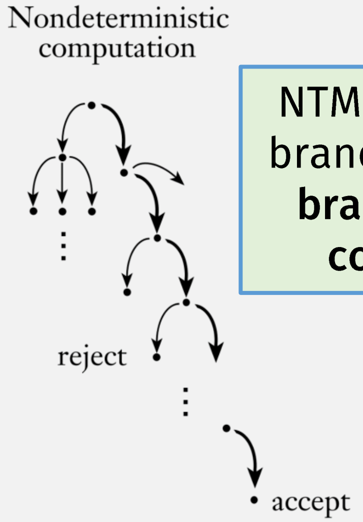


Theorem: *PATH* is in NL

(It's not known whether *PATH* is in *L*)

$PATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$

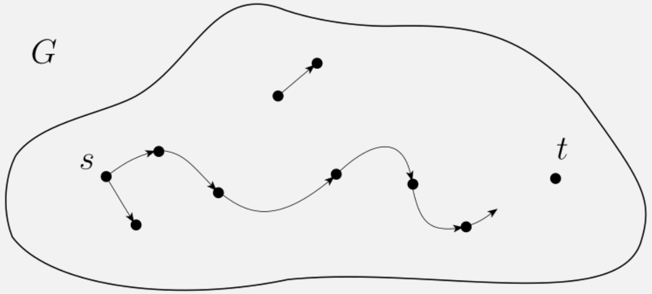
- Don't directly modify input
- Instead, just remember "current" node
  - Don't need to remember all nodes …
  - … so long as we start at s, and each step is a valid edge



NTMs accept if <u>any</u> branch accepts; but **branches cannot communicate**

On input <G, s, t>:

- Starting at "current" node = s:
  - nondeterministically follow edges
- Each branch remembers:
  - Current node ← $O(1)$ space
  - # of steps ← $O(\log m)$ space
- <u>Accept</u> if: any "current" node is t
- <u>Reject</u> if: # of steps = m (# nodes)



Flashback: Facts About Time vs Space

TIME → SPACE

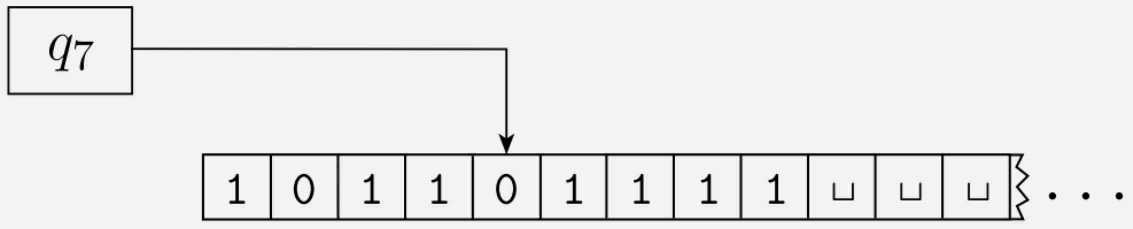
- If a decider runs in <u>time</u> $t(\boldsymbol{n})$, then its maximum <u>space</u> usage is …
- … $t(\boldsymbol{n})$
- … because it can add at most 1 tape cell per step

SPACE → TIME

<u>Note</u>: This assumes $f(n) \geq O(n)$

- If a decider runs in <u>space</u> $f(\boldsymbol{n})$, then its maximum <u>time</u> usage is …
- … $(|\Gamma| + |Q|)^{f(\boldsymbol{n})} = 2^{df(\boldsymbol{n})}$
- … because that's the number of possible configurations
- (and a decider cannot repeat a configuration)

Flashback: TM Config = State + Head + Tape



$$1011q_7 01111$$

Textual representation of "configuration"

1st char after state is current head position

Read-only Input 2-Tape TM Configurations

- State
  - Let $\boldsymbol{q}$ = # states
- 2 head positions
  - Let $\boldsymbol{n}$ = input length
  - Let $\boldsymbol{f(n)}$ = work tape length
- Work tape contents only (not input tape)
  - Let g = # tape alphabet chars
  - Maximum number of different work tape contents = $\boldsymbol{g^{f(n)}}$

Maximum configurations = $q \cdot n \cdot f(n) \cdot g^{f(n)}$

= $\boldsymbol{O(n)}$ (if $f(n) = O(1)$)

= $\boldsymbol{O(n^2)}$ (if $f(n) = O(\log n)$)

= $\boldsymbol{2^{O(f(n))}}$ (if $f(n) \geq O(n)$)

Flashback: Deterministic vs Non-Det. Space

THEOREM

Savitch's theorem For any function $f: \mathcal{N} \longrightarrow \mathcal{R}^+$, where $f(n) \geq$ ~~n~~, $\log n$

$$\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n)).$$

- If a non-deterministic TM runs in: $f(n)$ space
- Then an equivalent deterministic TM runs in: $f^2(n)$ space
  - ~~**Exponentially**~~ Only **Quadratically** slower!

Flashback: Deterministic = Non-deterministic?

- **P = NP**? Unknown?

- **PSPACE** = **NPSPACE**? Yes!

THEOREM

Savitch's theorem For any function $f : \mathcal{N} \longrightarrow \mathcal{R}^+$, where $f(n) \geq n$,

$$\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n)).$$

- **L** = **NL**? Unknown?

NL-Completeness

DEFINITION

A language B is ***NL-complete*** if

1. $B \in$ NL, and
2. every A in NL is log space reducible to B.

Because poly time is too much!

(We'll show that <u>every</u> NL problem is solvable in poly time!)

Flashback: Mapping Reducibility

Language A is ***mapping reducible*** to language B, written $A \leq_m B$, if there is a computable function $f\colon \Sigma^* \longrightarrow \Sigma^*$, where for every w,

$$w \in A \iff f(w) \in B.$$

The function f is called the ***reduction*** from A to B.

Flashback: Computable Functions

- A TM that, instead of accept/reject, "outputs" final tape contents

A function $f: \Sigma^* \longrightarrow \Sigma^*$ is a ***computable function*** if some Turing machine M, on every input w, halts with just $f(w)$ on its tape.

Log Space Computable Functions

Needs **3 tapes**

1. Read-only input tape
2. Write-only output tape
3. Read/write work tape

Space complexity: only counts the work tape

DEFINITION

A ***log space transducer*** is a Turing machine with a read-only input tape, a write-only output tape, and a read/write work tape. The head on the output tape cannot move leftward, so it cannot read what it has written. The work tape may contain $O(\log n)$ symbols. A log space transducer M computes a function $f\colon \Sigma^* \longrightarrow \Sigma^*$, where $f(w)$ is the string remaining on the output tape after M halts when it is started with w on its input tape. We call f a ***log space computable function***.

Log Space Reducibility

DEFINITION

Language A is ***log space reducible*** to language B, written $A \leq_L B$, if A is mapping reducible to B by means of a log space computable function f.

Log space reducibility
=
mapping reducibility with a log space computable function

NL-Completeness and L=NL?

DEFINITION

A language B is ***NL-complete*** if

1. $B \in \mathrm{NL}$, and
2. every A in NL is log space reducible to B.

THEOREM

If $A \leq_{\mathrm{L}} B$ and $B \in \mathrm{L}$, then $A \in \mathrm{L}$.

COROLLARY

If any NL-complete language is in L, then $\mathrm{L} = \mathrm{NL}$.

unknown

known

Flashback:

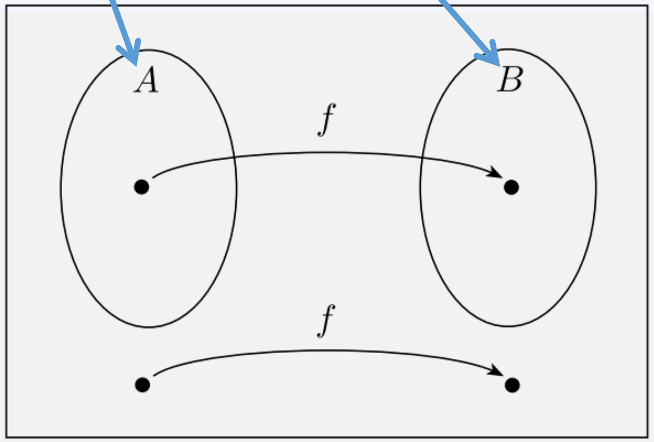
If $A \leq_P B$ and $B \in P$, then $A \in P$.

PROOF Let M be the polynomial time algorithm deciding B and f be the polynomial time reduction from A to B. We describe a polynomial time algorithm N deciding A as follows.

$N =$ "On input w:

1. Compute $f(w)$.
2. Run M on input $f(w)$ and output whatever M outputs."

This won't work for the log version because $f(w)$ may produce an output (which is now part of N's work tape) that uses more than $\log(n)$ space!



THEOREM

If $A \leq_L B$ and $B \in \mathrm{L}$, then $A \in \mathrm{L}$.

PROOF Let M be the ~~polynomial time~~ log space algorithm deciding B and f be the ~~polynomial time~~ log space reduction from A to B. We describe a ~~polynomial time~~ log space algorithm N deciding A as follows.

N = "On input w:

1. ~~Compute $f(w)$.~~
2. Run M on input $f(w)$ and output whatever M outputs."

Instead, N computes $f(w)$ output chars as needed by M, discarding everything else

(this may require recomputing $f(w)$ every time M needs part of it)

NL-Completeness

DEFINITION

A language B is ***NL-complete*** if

1. $B \in \mathrm{NL}$, and
2. every A in NL is log space reducible to B.

The <u>first</u> **NL**-complete problem?

Theorem: *PATH* is NL-complete

$PATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$

DEFINITION

A language B is ***NL-complete*** if

☑ **1.** $B \in$ NL, and

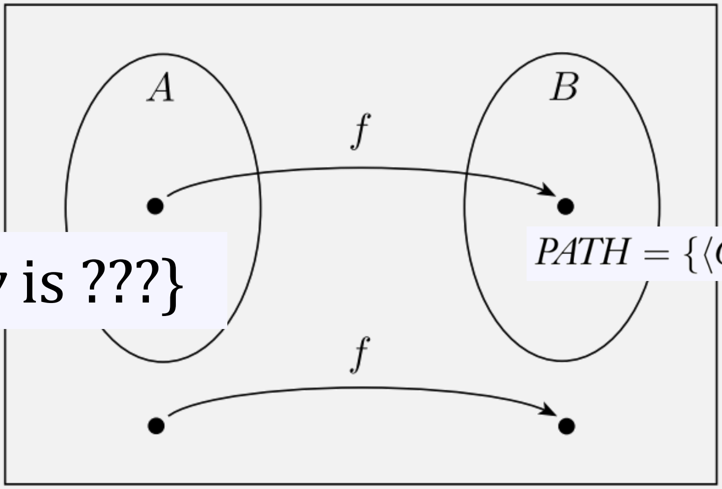
➡ **2.** every A in NL is log space reducible to B.

Theorem: *PATH* is NL-complete

$PATH = \{\langle G, s, t\rangle |\ G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$

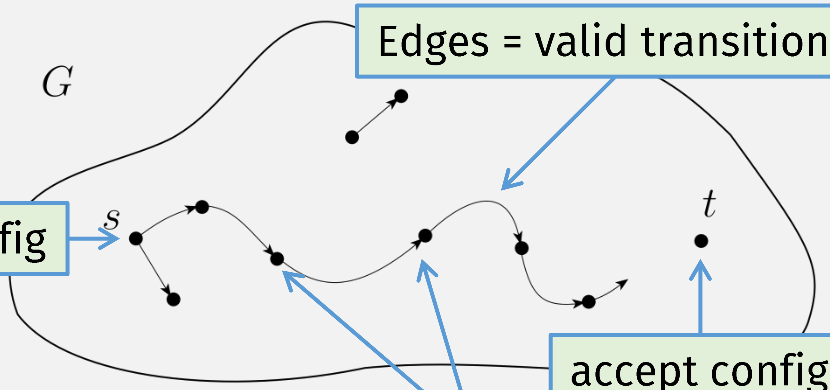
__We know__:
Some TM M decides A in $O(\log(n))$ space

Some **NL** lang $A = \{w \mid w \text{ is ???}\}$



A graph where a path from s to t encodes accepting config sequence of M on w

$PATH = \{\langle G, s, t\rangle |\ G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$



Can we compute this graph in log space?

- __Nodes__: M runs in $O(\log(n))$ space, so each config/node takes …
  … $O(\log(n))$ space to compute
- __Edges__: check every pair of configs for valid transitions …
  … $O(\log(n))$ space

Key: Nodes/configs can be output independently

Corollary: $\mathrm{NL} \subseteq \mathrm{P}$

(Every NL problem is solvable in poly time!)

- Every language in **NL** is reducible in log space to *PATH*
  - Justification?
- A log space reduction takes poly time
  - Justification?
- A language that is poly time reducible to a lang in **P** is in **P**
  - Justification?
- *PATH* is in **P**
  - Justification?
- Every language in **NL** is in **P**
  - Justification?

Future HW question?

NL-Completeness

DEFINITION

A language B is ***NL-complete*** if

1. $B \in \text{NL}$, and
2. every A in NL is log space reducible to B.

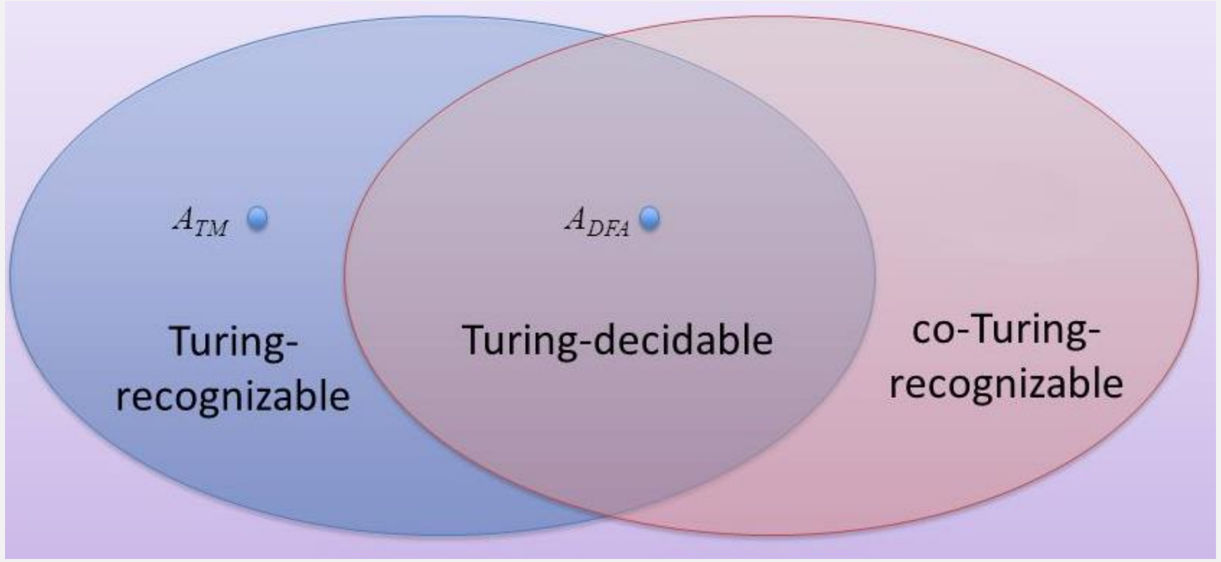
Poly time is too much!

Every NL problem is solvable in poly time …
so it's pointless to poly time reduce it to another problem!

Flashback: Co-Turing-Recognizability

- A language is **co-Turing-recognizable** if …
- … it is the complement of a Turing-recognizable language.

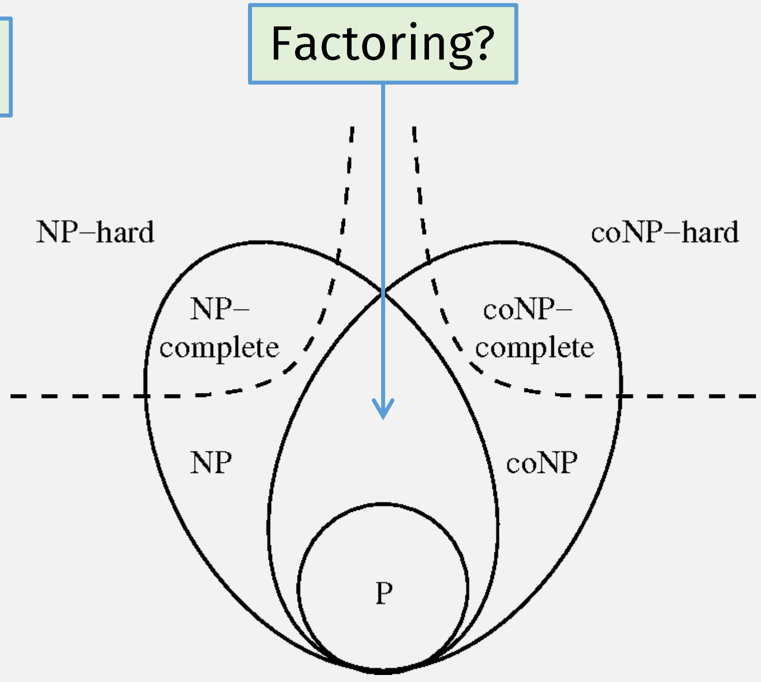
Flashback: Decidable ⇔ Recognizable & co-Recognizable



co**NP**

co**NP** has languages whose complement is in **NP**

It's believed that **NP** ≠ co**NP** (but not known)



Example:

$$SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$$

- SAT ∈ **NP**
  - Verifiable in poly time
    - Certificate = a satisfying assignment

- $\overline{SAT}$ ∈ co**NP** but ∉ **NP**
  - Not verifiable in poly time
    - There's no certificate (must always check all possible assignments)

$\mathbf{NL} = \mathrm{co}\mathbf{NL}$

Proof:

- $PATH$ is in $\mathbf{NL}$ and is $\mathbf{NL}$-complete
- $\overline{PATH}$ is in $\mathrm{co}\mathbf{NL}$ and is $\mathrm{co}\mathbf{NL}$-complete
- If we can show $\overline{PATH}$ is in $\mathbf{NL}$, then $\mathbf{NL} = \mathrm{co}\mathbf{NL}$

$\overline{PATH}$ (“No Path”) is in **NL**

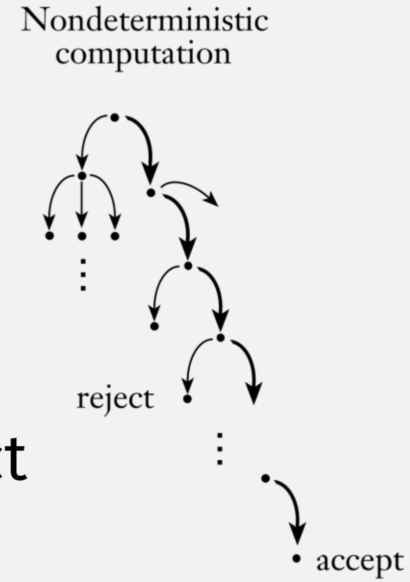
$PATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$

Naïve idea (doesn’t work):

×Nondeterministically check every path

- Similar to *PATH* is in **NL** proof
- Reject if any go from s to t

- But when to accept?
  - We need to know if all branches failed
  - But branches can’t communicate
- Remember, NTMs accept if any branch accepts
  - Each branch must independently determine accept/reject



$\overline{PATH}$ ("No Path") is in **NL**

Let m = # nodes of G

$PATH = \{\langle G, s, t\rangle \mid G$ is a directed graph that has a directed path from s to $t\}$

Better idea # 1 (still doesn't quite work):

❖ Count nodes reachable from s (at most m)

- Non-deterministically explore all paths from s to some node u
  - In any branch that reaches u, increment a counter
  - Then nondeterministically (with increased counter) check reachability of the "next" node

But each branch does not know what the "next" node is?

$\overline{PATH}$ ("No Path") is in **NL**

Let m = # nodes of G

$PATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$

Better idea # 2 (still doesn't quite work):

❖ Nondeterministically guess subset of reachable nodes

- In each branch:
  - Verify that each guessed reachable subset matches the count
  - And reject the bad guesses

But we didn't compute the count yet?

$\overline{PATH}$ ("No Path") is in **NL**

Let m = # nodes of G

$PATH = \{\langle G, s, t\rangle \mid G$ is a directed graph that has a directed path from s to $t\}$

Best idea (works) Part 1:

✓Compute reachability count and nodes incrementally

- for path lengths 0, 1, 2, …

- If c_i = # nodes reachable from s in i steps,
  - we know $c_0 = 1$
- Nondeterministically guess nodes reachable from s in i steps:
  - In each branch, verify that c_i nodes are reachable
  - Reject bad branches
- In correctly guessed branch:
  - compute c_{i+1} by checking edges from those nodes

$\overline{PATH}$ ("No Path") is in **NL**

Let m = # nodes of G

$PATH = \{\langle G, s, t\rangle |\ G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$

Best idea (works) Part 2:

✓Once we have computed c_m (# nodes reachable from s):

- nondeterministically guess (and re-count) reachable nodes
  - **this time excluding** t
- Accept in any branch where the re-count = c_m
  - (because this means t was not reachable)

$\overline{PATH}$ is in **NL**, Formally

PROOF Here is an algorithm for $\overline{PATH}$. Let m be the number of nodes of G.

M = "On input $\langle G, s, t\rangle$:

1. Let $c_0 = 1$. [[$A_0 = \{s\}$ has 1 node]]
2. For $i = 0$ to $m - 1$: [[compute c_{i+1} from c_i]]
3. Let $c_{i+1} = 1$. [[c_{i+1} counts nodes in A_{i+1}]]
4. For each node $v \neq s$ in G: [[check if $v \in A_{i+1}$]]
5. Let $d = 0$. [[d re-counts A_i]]
6. For each node u in G: [[check if $u \in A_i$]]
7. Nondeterministically either perform or skip these steps:
8. Nondeterministically follow a path of length at most i from s and *reject* if it doesn't end at u.
9. Increment d. [[verified that $u \in A_i$]]
10. If (u, v) is an edge of G, increment c_{i+1} and go to stage 5 with the next v. [[verified that $v \in A_{i+1}$]]
11. If $d \neq c_i$, then *reject*. [[check whether found all A_i]]
12. Let $d = 0$. [[c_m now known; d re-counts A_m]]
13. For each node u in G: [[check if $u \in A_m$]]
14. Nondeterministically either perform or skip these steps:
15. Nondeterministically follow a path of length at most m from s and *reject* if it doesn't end at u.
16. If $u = t$, then *reject*. [[found path from s to t]]
17. Increment d. [[verified that $u \in A_m$]]
18. If $d \neq c_m$, then *reject*. [[check whether found all of A_m]]

Otherwise, *accept*."

Start with c_0

c_i = # nodes reachable from s in i steps

Need space for these variables; none larger than m = log(m) space

Guess reachable nodes in c_i

Verify and count reachable nodes

Compute c_{i+1} from c_i

Guess reachable nodes (without t) and re-count

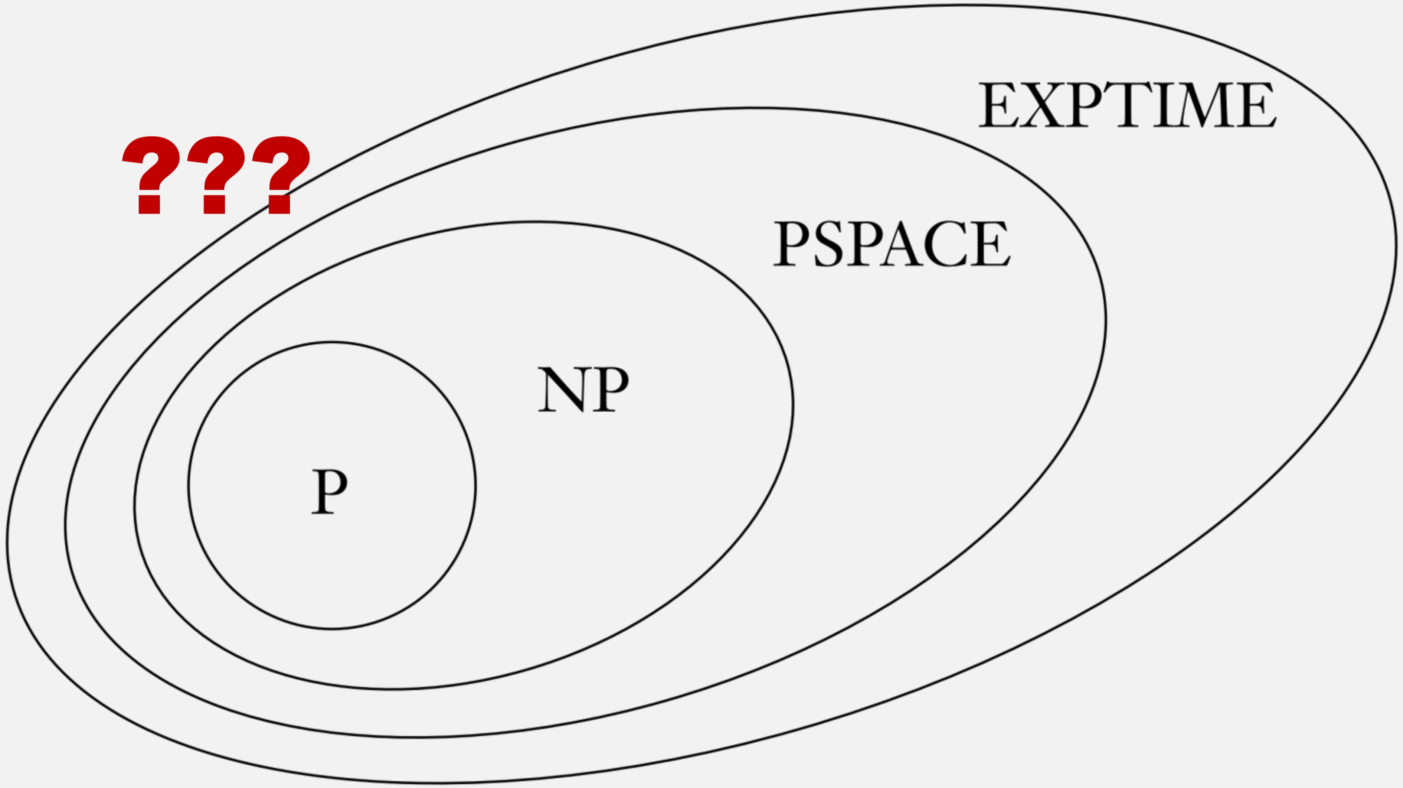
Accept if re-count matches c_m

$\mathbf{NL} = \mathrm{co}\mathbf{NL}$

Proof:

- $PATH$ is in $\mathbf{NL}$ and is $\mathbf{NL}$-complete
- $\overline{PATH}$ is in $\mathrm{co}\mathbf{NL}$ and is $\mathrm{co}\mathbf{NL}$-complete
- If we can show $\overline{PATH}$ is in $\mathbf{NL}$, then $\mathbf{NL} = \mathrm{co}\mathbf{NL}$ ■

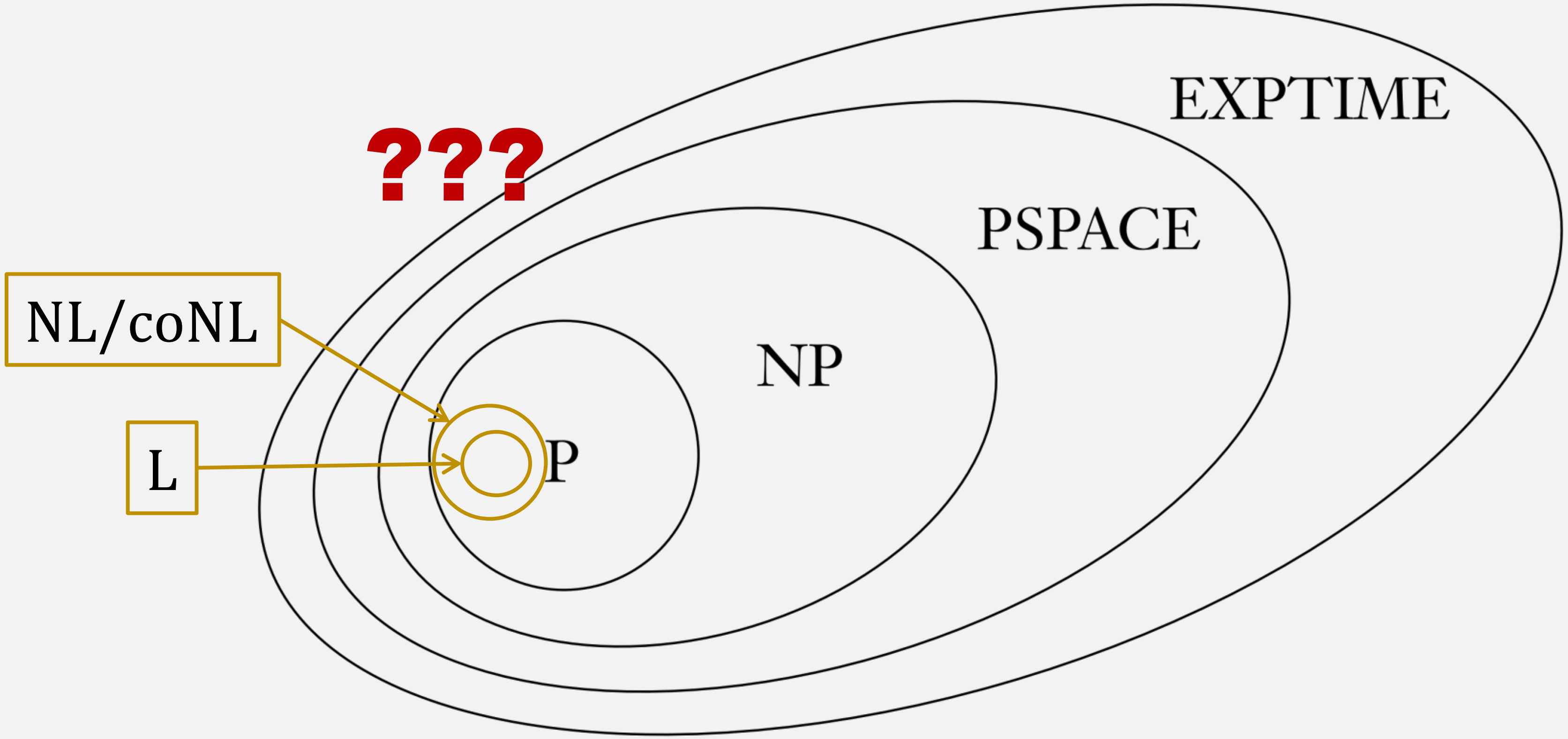
Space vs Time: Conjecture



We think: $P \subset NP \subset PSPACE = NPSPACE \subset EXPTIME$

We know: $P \subseteq NP \subseteq PSPACE = NPSPACE \subseteq EXPTIME$

Space vs Time: Conjecture



We think: **$L \subset NL = coNL \subset P \subset NP \subset PSPACE = NPSPACE \subset EXPTIME$**

We know: **$L \subseteq NL = coNL \subseteq P \subseteq NP \subseteq PSPACE = NPSPACE \subseteq EXPTIME$**

Check-in Quiz 12/1

On gradescope