NL = coNL: Surprising?

coNL = set of problems / languages whose complement is in NL.
We’ve seen checking language \( A \) and \( \overline{A} \) is asymmetric:
If \( A \) has easy certificate/witness that \( w \in A \), then \( \overline{A} \) may require expensive exhaustive checking
Recall NP vs coNP: SAT, CLIQUE, etc.

We’ll prove \( PATH \in NL \). Insights:
▶ Repeatedly decompose problem
▶ Use PATH as subproblem (in NL)!
▶ Extensively use guessing / nondeterminism

First transformation: Reachability

Given: graph \( G \), nodes \( s, t \)
Goal: construct log space NTM that accepts if no \( s \leadsto t \) path

Rephrase as reachability:
find all nodes of \( G \) that are reachable from \( S \).

Can’t store set of reachable nodes in \( \log n \) space.
Could we count?
Assume we could know/guess count of reachable nodes.
Does this help?

Check reachable nodes knowing count

Given any node \( u \), we can check in \( \log n \) space if reachable
this is \( PATH \)!

Idea: we can do this sequentially!
Iterate through all nodes, at each flip bit, guess if reachable.
if guessed unreachable, go to next
if guessed reachable and node is \( t \), reject (don’t want that)
else verify: if not reachable, reject.
else increment count
Finally, check if count is expected value, accept/reject.

Summary: if nondeterministically:
we can select the right number of reachable nodes
and we did not select \( t \) and each is indeed reachable
then we know the other nodes (incl. \( t \)) are not reachable, accept.

Finding the count of reachable nodes

Idea: do BFS, count the nodes reached after each level
Let \( A_i \) = set of nodes at distance at most \( i \) from \( s \) in BFS
We will count \( c_i = |A_i| \).
How? Like before: (again!) guess \( c_i \) and verify
Loop through all nodes in \( G \), guess if in \( A_{i+1} \).
How to check? Need to know predecessor in \( A_i \).
Catch-22? Recurrence? No, can guess!

Finding node count at each level

We already know \( c_i \) (from previous iteration, \( c_0 = 1 \)).
Will re-count and re-find them for each candidate for \( A_{i+1} \)

Compute \( c_{i+1} \) (nodes of \( A_{i+1} \)):
loop through all nodes \( v \) (candidate for \( A_{i+1} \))
start re-counting \( A_i \) \((r_i = 0)\)
loop through all \( u \) (candidate for \( A_i \))
if edge \((u, v)\), increment \( c_{i+1} \), take next \( v \)
if recount \( r_i \neq c_i \), reject
We compute \( c_i \) until \( c_{i+1} = c_i \) (at most \( c_m \))
To implement, need these counters:

- \( i \): counts distance (\( A_i \)), up to \( \leq m \)
- \( c_i \): count of \( A_i \) (distance \( \leq i \))
- \( c_{i+1} \): count of \( A_{i+1} \)
- temp counter to re-count/check \( c_i \) and \( c_m \)
- \( u, v \): previous / current nodes

\( L \subseteq NL = coNL \subseteq P \subseteq NP \subseteq PSPACE \subseteq EXPTIME \)

We'll see \( NL \not\subseteq PSPACE \)
so \( P \) (separating them must be different from one of them).