Theory of Computation (XIII)

Yijia Chen
Fudan University

Review

Definition

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

$$\text{PSPACE} = \bigcup_k \text{SPACE}\left(n^k\right).$$

We could also define

$$\text{NPSPACE} = \bigcup_k \text{NSPACE}\left(n^k\right).$$

Then by Savitch's Theorem,

$$\text{NSPACE} = \text{PSPACE}.$$

The relationship of PSPACE with P and NP

A machine which runs in time t can use at most space t.

Hence

$$\mathsf{P} \subseteq \mathsf{PSPACE} \quad \text{and} \quad \mathsf{NP} \subseteq \mathsf{NPSPACE} = \mathsf{PSPACE}.$$

PSPACE and EXPTIME

Let $f : \mathbb{N} \to \mathbb{R}^+$ satisfy $f(n) \geq n$. Then a TM uses $f(n)$ space can have at most

$$f(n) \cdot 2^{O(f(n))}$$

configurations. Therefore it must run in time $f(n) \cdot 2^{O(f(n))}$.

Hence

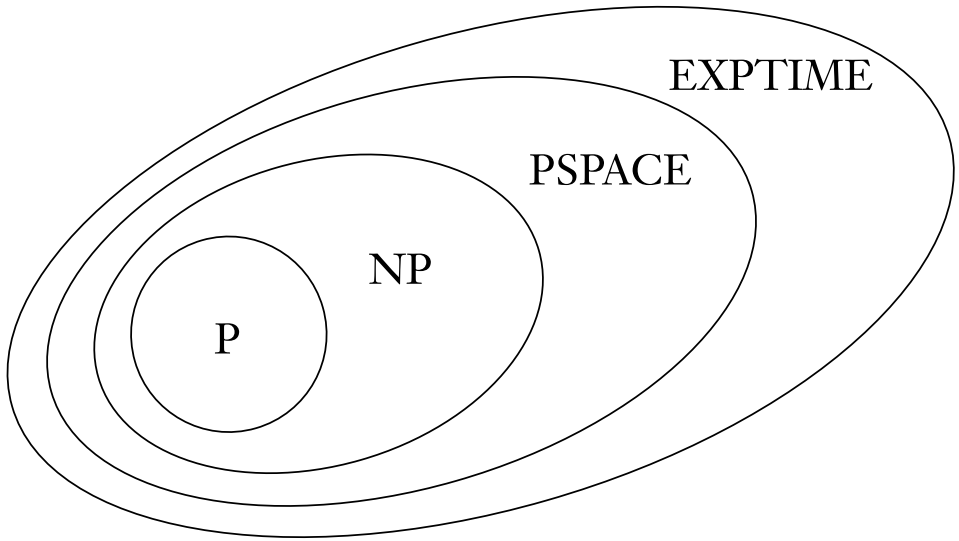
$$\text{PSPACE} \subseteq \text{EXPTIME} = \bigcup_k \text{TIME}\left(2^{n^k}\right).$$

We know

$$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} = \mathsf{NPSPACE} \subseteq \mathsf{EXPTIME},$$

and it is easy to show $\mathsf{P} \neq \mathsf{EXPTIME}$.

The general consensus is



PSPACE Completeness

Definition

A language B is PSPACE-complete if

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

If B merely satisfies condition 2, then it is PSPACE-hard.

The TQBF problem

- A Boolean formula contains Boolean variables, the constants 0 and 1, and the Boolean operations $\wedge$, *vee*, and $\neg$.
- The universal quantifiers $\forall$ in $\forall x \varphi$ means that φ is true for every value of x in the universe.
- The existential quantifiers $\exists$ in $\exists x \varphi$ means that φ is true for some value of x in the universe.
- Boolean formulas with quantifiers are quantified Boolean formulas. For such formulas, the universe is $\{0, 1\}$. E.g.

$$\forall x \exists y \left[(x \vee y) \wedge (\bar{x} \vee \bar{y}) \right].$$

- When each variable of a formula appears within the scope of some quantifier, the formula is fully quantified. A fully quantified Boolean formula is always either true or false.

The TQBF problem

The *TQBF* problem is to determine whether a fully quantified Boolean formula is true or false, i.e.,

$$TQBF = \{\langle\varphi\rangle \mid TQBF \text{ is a true fully quantifier Boolean formula}\}.$$

Theorem

TQBF is PSPACE*-complete.*

Winning strategies for games

The formula game

Let

$$\varphi = \exists x_1 \forall x_2 \exists x_3 \cdots Q x_k \left[\psi\right].$$

We associate a game with φ.

1. Two players, Player A and Player E, take turns selecting the values of the variables $x_1, \ldots, x_k$.
2. Player A selects values for the variables bound to $\forall$.
3. Player E selects values for the variables bounded to $\exists$.
4. At the end, if ψ is true, then Player E wins; otherwise Player A wins.

Let

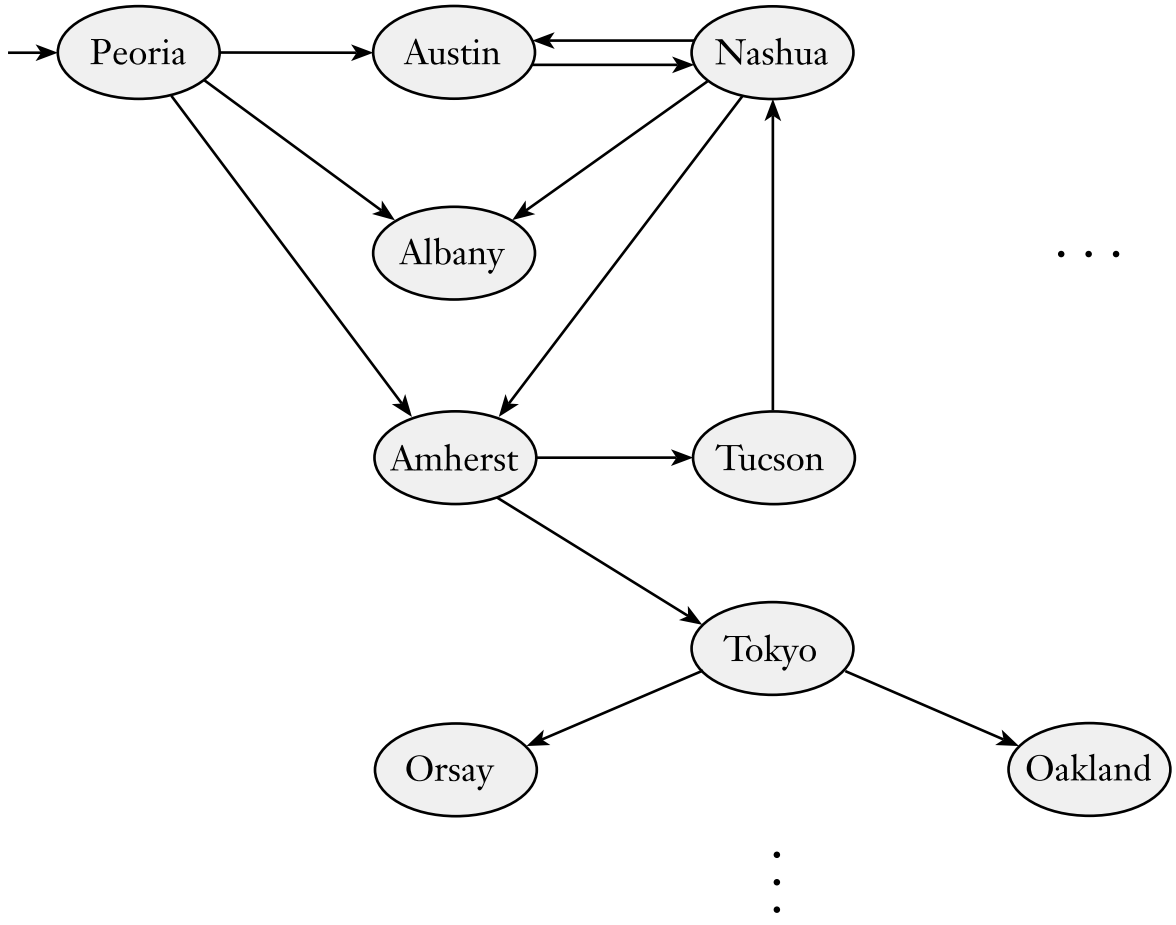
$$\mathit{FORMULA\text{-}GAME} := \{\langle\varphi\rangle \mid \text{Player E has a winning strategy in the formula game associated with } \varphi\}.$$

Theorem

FORMULA-GAME is PSPACE*-complete.*

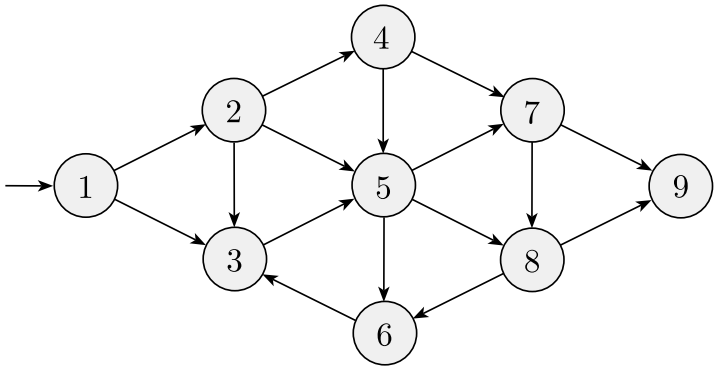
Generalized geography

1. Two players take turns to name cities from anywhere in the world.
2. Each city chosen must begin with the same letter that ended the previous city's name.
3. Repetition isn't permitted.
4. The game starts with some designated starting city and ends when some player can't continue and thus loses the game.


Peoria
Austin
Nashua
Albany
Amherst
Tucson
Tokyo
Orsay
Oakland

Generalized Geography

1. Take an arbitrary directed graph with a designated start node.



2. Player I starts by selecting the designated start node.
3. Then the players take turns picking nodes that form a simple path in the graph.
4. The first player unable to extend the path loses the game.

Let

$$GG := \{\langle G, b\rangle \mid \text{Player I has a winning strategy for the geography game played on graph } G \text{ starting at } b\}.$$

Theorem

GG is PSPACE*-complete.*

Proof (1)

M on $\langle G, b \rangle$:

1. If b has outdegree 0, then reject.
2. Remove node b and all connected arrows to get a new graph G'.
3. For each of the nodes $b_1, b_2, \ldots, b_k$ that b originally pointed at, recursively call M on $\langle G', b_i \rangle$.
4. If all of these accept, then Play II has a winning strategy in the original game, so reject. Otherwise, accept.

M runs in linear space.

Proof (2)

To show the hardness, we give a reduction from *FORMULA-GAME*.

Let

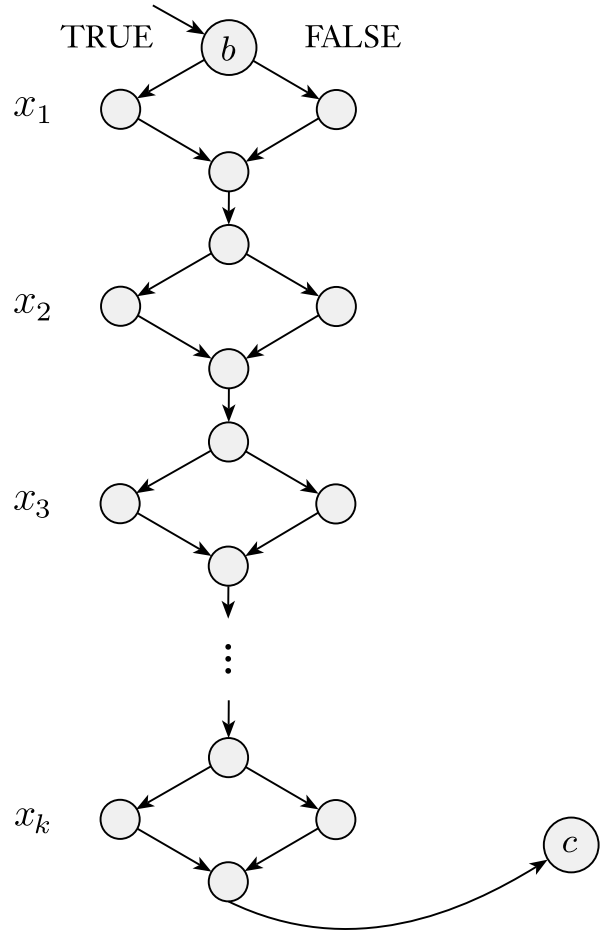
$$\varphi = \exists x_1 \forall x_2 \exists x_3 \cdots Q x_k \left[\psi\right],$$

where

- the quantifiers begin and end with $\exists$, and alternate between $\exists$ and $\forall$,
- ψ is in conjunctive normal form.

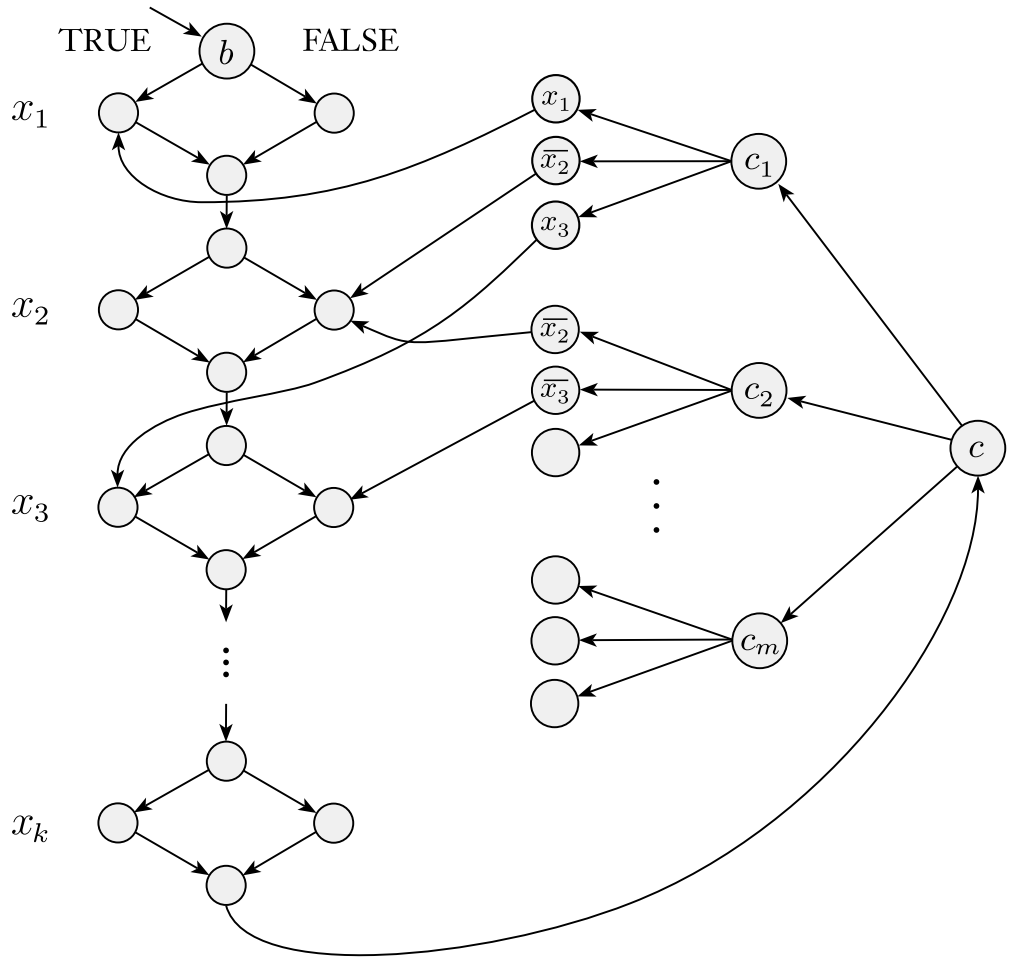
We constructs a geography game on a graph G where optimal play mimics optimal play of the formula game on φ, in which Player I in the geography game takes the role of Player E in the formula game, and Player II takes the role of Player A.

Proof (3)



At the last node of the left diamonds, it is Player I's move.

Proof (4)



$$\exists x_1 \forall x_2 \cdots \exists x_k \left[(x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3} \vee \cdots) \wedge \cdots \right].$$

The classes L and NL

Until now, we have considered only time and space complexity bounds that are at least linear — that is, bounds where $f(n)$ is at least n.

Now we examine smaller sublinear space bounds.

Not enough space to store the input. To consider this situation meaningfully, we need to modify the computational model.

Machine model for NL

A Turing machine with two tapes:

1. a read-only input tape;
2. a read/write work tape.

On the input tape, the input head can detect symbols, but not change them. The input head must remain on the portion of the tape containing the input.

The work tape may be read and written in the usual way.

Only the cells scanned on the work tape contribute to the space complexity of this type of Turing machine.

For sublinear space bounds, we use only the two-tape model.

Definition

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

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

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

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

$A = \{0^k 1^k \mid k \geq 0\} \in \mathsf{L}$

Obviously, $A \in \mathsf{SPACE}(n)$. To make it sublinear:

The machine counts the number of 0s and separately, the number of 1s in binary on the work tape.

The only space required is that used to record the two counters. Hence the algorithm runs in space $O(\log n)$.

$PATH \in \mathrm{NL}$

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

The NTM starting at node s, nondeterministically guessing the nodes of a path from s to t.

- The machine only records the position of the current node at each step on the work tape, not the entire path.
- The machine nondeterministically selects the next node from among those pointed by the current node,
- Repeat this action until it reaches node t and accepts.

  Or, until it has gone on for m steps and rejects, where m is the number of nodes in the graph.

It is not clear whether $PATH \in \mathrm{L}$.

Space vs. Time

The result that any $f(n)$ space bounded Turing machine also runs in time $2^{Of(n)}$ is no longer true for very small space bounds.

For example, a TM that use $O(1)$ space may run for n steps.

Space vs. Time

Definition

If M is a Turing machine that has a separate read-only input tape and w is an input, a configuration of M on w is a setting of the state, the work tape, and the positions of the two tape heads.

The input w is not a part of the configuration of M on w.

If M runs in $f(n)$ space and w is an input of length n, the number of configurations of M on w is $n2^{O(f(n))}$.

Now, when $f(n) \geq \log n$, we still have that the time complexity of a machine is at most exponential in its space complexity.

Savitch's theorem can be also extended to the sublinear space case provided that $f(n) \geq \log n$.

NL-completeness

We don't use polynomial time reducibility, because:

Proposition

All problems in NL *except* $\emptyset$ *and* Σ^* *are polynomial time reducible to one another.*

Instead, we use a new type of reducibility called log space reducibility.

log space reduction

Definition

A log space transducer is a TM 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 $f : \Sigma^* \to \Sigma^*$, where $f(w)$ is the string remaining on the output tape after M halts when it is started with w on its input tape.

f is a log space computable function.

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

NL-complete

Definition

A language B is NL-complete if

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

NL-complete

Theorem
If $A \leq_L B$ and $B \in L$, then $A \in L$.

Proof.
$f(w)$ may be too large to fit within the log space bound!

Suppose the log space reduction function is f, and B is decided by a TM $M_B \in$L. We build a TM M_A for A:

- M_A computes individual symbols of $f(w)$ as required by M_B,
- M_A keeps track of where M_B's input head would be on $f(w)$,
- Every time M_B moves, M_A restarts the computation of f on w from the beginning and ignores all the output except for the desired location of $f(w)$.

Only a single symbol of $f(w)$ needs to be stored at any point, in effect trading time for space. □

Corollary

If any NL*-complete language is in* L*, then* L = NL*.*

NL-complete problem

Theorem
PATH is NL-complete.

Proof (1)

Let $A \in \mathrm{NL}$, say NTM M decides A in $O(\log n)$ space.

Given an input w we construct $\langle G, s, t \rangle$ in log space, where G is a directed graph such that

$$G \text{ contains a path from } s \text{ to } t \iff M \text{ accepts } w.$$

1. Nodes of G are the configurations of M on w.
2. For configurations c_1 and c_2 of M on w, the pair (c_1, c_2) is an edge of G if c_2 is one of the possible next configurations of M starting from c_1.
3. Node s is the start configuration of M on w.
4. M is modified to have a unique accepting configuration, which is node t.

Proof (2)

There is a log space transducer T that outputs $\langle G, s, t \rangle$ on input w.

1. List the nodes of G.

   Each node is a configuration of M on w and can be represented in $c \log n$ space for some constant c.

   T sequentially goes through all possible strings of length $c \log n$ and tests whether each is a legal configuration of M on w, and output those that pass the test.

2. List the edges of G.

   T tries all pairs (c_1, c_2), tests whether each is a legal configuration of M on w. Those that do are added to the output tape.

Corollary

NL $\subseteq$ P.

Proof.

1. $A \in$ NL, then $A \leq_L PATH$.
2. As any TM uses space $f(n)$ runs in time $n \cdot 2^{O(f(n))}$, a log space reducer also runs in polynomial time.
3. 1. and 2. imply A is polynomial time reducible to $PATH$.
4. $PATH \in$ P.

$\square$

$$\mathsf{NL} = \mathsf{coNL}$$

Theorem (Immerman-Szelepcsényi, 1988,1987)

NL = coNL.

Proof

$$\overline{PATH} = \{ \langle G, s, t \rangle \mid \text{There is no path from } s \text{ to } t \text{ in } G \}.$$

Suppose G has m nodes in all (represented by $[m]$). Let c be the number of nodes in G that are reachable from s. Consider the input $\langle G, s, t, c \rangle$.

The machine M:

- Initialize $\theta = 0$, for every node $u \in [m]$:
  1. M nondeterministically guess if u is reachable from s.
     - 1.1 if $u = t$ and the guess is YES, reject.
     - 1.2 if $u \neq t$ and the guess is YES, verify the guess:
       Guessing a path of length at most m from s to u.
       - *i*. If the verifying passes: $\theta \leftarrow \theta + 1$;
       - *ii*. If the verifying fails: reject.
- If $\theta = c$, accept; otherwise, reject.

How to get c

For $0 \leq i \leq m$

$$A_i = \left\{ v \;\middle|\; \text{the distance from } s \text{ to } v \text{ is at most } i \right\}$$

Then

$$\{s\} = A_0 \subseteq A_1 \subseteq \cdots \subseteq A_m.$$

Define $c_i := |A_i|$. Thus

$$c_0 = 1 \quad \text{and} \quad c_m = c.$$

How to get c (cont'd)

- Initialize $c_{i+1} = 0$.

  For every node v, repeat: $\theta'_i = 0$,

  1. for every node u in G, guess whether $u \in A_i$.

     1.1 If YES, verify the guess:

     Guessing the path of length at most i from s to u.

     If the verifying passes, $\theta'_i \leftarrow \theta'_i + 1$;

     - Test if $(u, v) \in G$.

     If YES, $c_{i+1} \leftarrow c_{i+1} + 1$ and return (try another v),

     otherwise return; (try another u)

     otherwise return. (try another u)

  2. If $\theta'_i \neq \theta_i$, reject. (start another branch of 1.)

  (try another v)

- Output c_{i+1}.

The final algorithm

Here is an algorithm for 'no *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*."

Complexity classes so far

$$\mathsf{L} \subseteq \mathsf{NL} = \mathsf{coNL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE}.$$