Introduction to Computational Complexity

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU Math 400

Subsection 1

Introduction

Space Complexity

- We consider the complexity of computational problems in terms of the amount of space, or memory, that they require.
- Space complexity shares many of the features of time complexity and serves as a further way of classifying problems.
- We again select the Turing machine model for measuring space.

Definition (Space Complexity)

Let M be a deterministic Turing machine that halts on all inputs. The **space complexity** of M is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of tape cells that M scans on any input of length n. If the space complexity of M is $f(n)$, we also say that M **runs in space** $f(n)$. If M is a nondeterministic Turing machine wherein all branches halt on all inputs, we define its **space complexity** $f(n)$ to be the maximum number of tape cells that M scans on any branch of its computation for any input of length n.

Space Complexity Classes

- We estimate the space complexity of Turing machines by using asymptotic notation.

Definition (Space Complexity Classes)

Let $f : \mathbb{N} \to \mathbb{R}^+$ be a function. The **space complexity classes** SPACE($f(n)$) and NSPACE($f(n)$), are defined as follows:

- SPACE($f(n)$) $= \{L : L$ is a language decided by a O$(f(n))$ space deterministic Turing machine$\}$.
- NSPACE($f(n)$) $= \{L : L$ is a language decided by a O$(f(n))$ space nondeterministic Turing machine$\}$.

Satisfiability is in Linear Space

- We have already seen the NP-complete problem $\textsc{Sat}$.
- We now show that $\textsc{Sat}$ can be solved with a linear space algorithm.
- We believe that $\textsc{Sat}$ cannot be solved with a polynomial time algorithm, let alone in linear time, because $\textsc{Sat}$ is NP-complete.
- Space appears to be more powerful than time because space can be reused, whereas time cannot.

  M_1: On input $\langle \phi \rangle$, where ϕ is a Boolean formula:
  1. For each truth assignment to the variables $x_1, \ldots, x_m$ of ϕ:
     2. Evaluate ϕ on that truth assignment.
  3. If ϕ is ever evaluated to 1, accept; if not, reject.
- Machine M_1 clearly runs in linear space because each iteration of the loop can reuse the same portion of the tape:
  - The machine needs to store only the current truth assignment and that can be done with $O(m)$ space.
  - The number of variables m is at most n, the length of the input.

  So this machine runs in space $O(n)$.

Determining a Nondeterministic Space Complexity

- Determining the nondeterministic space complexity can be useful in determining its deterministic space complexity.
- Consider the problem of testing whether a nondeterministic finite automaton accepts all strings:

$$\text{ALL}_{\text{NFA}} = \{\langle A \rangle : A \text{ is an NFA and } L(A) = \Sigma^*\}.$$

- We give a nondeterministic linear space algorithm that decides the complement of this language, $\overline{\text{ALL}_{\text{NFA}}}$.
- The main idea involves:
  - Using nondeterminism to guess a string that is rejected by the NFA;
  - Using linear space to keep track of which states the NFA could be in at a particular time.
- This language is not known to be in NP or in coNP.

The Nondeterministic Algorithm for $\overline{\text{ALL}_{\text{NFA}}}$

N: On input $\langle M \rangle$, where M is an NFA:

1. Place a marker on the start state of the NFA.
2. Repeat 2^q times, where q is the number of states of M:
   3. Nondeterministically select an input symbol and change the positions of the markers on M's states to simulate reading that symbol.
4. If a marker was ever placed on an accept state, reject; otherwise accept.

- If M accepts any strings, it must accept one of length at most 2^q:

  In any longer string that is accepted the locations of the markers described in the preceding algorithm would repeat. The section of the string between the repetitions can be removed to obtain a shorter accepted string.

  Hence N decides $\overline{\text{ALL}_{\text{NFA}}}$.

- The only space needed by this algorithm is for storing the location of the markers. This can be done using linear space.

Subsection 2

Savitch's Theorem

Savitch's Theorem: The Naïve Approach

- The simulation of nondeterministic machines by deterministic machines seems to require an exponential increase in time.

Theorem (Savitch's Theorem)

For any function $f : \mathbb{N} \to \mathbb{R}^+$, where $f(n) \geq n$,

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

- We need to simulate an $f(n)$ space NTM deterministically.
  A naive approach is to proceed by trying all the branches of the NTM's computation, one by one. The simulation needs to keep track of which branch it is currently trying so that it be able to go on to the next one. But a branch that uses $f(n)$ space may run for $2^{O(f(n))}$ steps, and each step may be a nondeterministic choice. Exploring the branches sequentially would require recording all the choices used on a particular branch in order to be able to find the next branch. Therefore this approach may use $2^{O(f(n))}$ space, exceeding the goal of $O\left(f^2(n)\right)$ space.

Savitch's Theorem: The Yieldability Problem Approach

- We consider the more general **yieldability problem**:
  Given two configurations c_1, c_2 of the NTM, together with a number t, test whether the NTM can get from c_1 to c_2 within t steps.
- By solving the yieldability, with c_1 the start configuration, c_2 the accept configuration, and t the max number of steps that the NTM can use, we can determine whether the machine accepts its input.
- We give a deterministic, recursive algorithm that solves yieldability: It operates by searching for an intermediate configuration c_m, and recursively testing whether:
  (1) c_1 can get to c_m within $\frac{t}{2}$ steps;
  (2) c_m can get to c_2 within $\frac{t}{2}$ steps.
- Reusing space for each of the two recursive tests saves enough space:
  The algorithm needs space for storing the recursion stack. Each level of the recursion uses $O(f(n))$ space to store a configuration. The depth is $\log t$, where t is the max time that the nondeterministic machine may use on any branch. We have $t = 2^{O(f(n))}$, so $\log t = O(f(n))$, whence, the deterministic simulation uses $O\left(f^2(n)\right)$ space.

Proof of Savitch's Theorem I

- Let N be an NTM deciding a language A in space $f(n)$. We construct a deterministic TM M deciding A. M uses the procedure CanYield, which tests whether one of N's configurations can yield another within a specified number of steps.

  Let w be a string considered as input to N. For configurations c_1 and c_2 of N on w, and integer t, CanYield(c_1, c_2, t) outputs accept if N can go from c_1 to c_2 in t or fewer steps. If not, CanYield outputs reject. We assume that t is a power of 2.

  CanYield: On input c_1, c_2 and t:

  1. If $t = 1$, then test directly whether $c_1 = c_2$ or whether c_1 yields c_2 in one step according to the rules of N. Accept if either test succeeds; reject if both fail.
  2. If $t > 1$, then for each configuration c_m of N on w using space $f(n)$:
     3. Run CanYield$(c_1, c_m, \frac{t}{2})$.
     4. Run CanYield$(c_m, c_2, \frac{t}{2})$.
     5. If steps 3 and 4 both accept, then accept.
  6. If have not yet accepted, reject.

Proof of Savitch's Theorem II

- Now we define M to simulate N:
  - We first modify N so that when it accepts it clears its tape and moves the head to the leftmost cell, thereby entering a configuration c_{accept}.
  - We let c_{start} be the start configuration of N on w.
  - We select a constant d so that N has no more than $2^{df(n)}$ configurations using $f(n)$ tape, where n is the length of w. Then, $2^{df(n)}$ is an upper bound on the running time of any branch of N on w.

  M: On input w:
  1. Output the result of $\textsc{CanYield}(c_{\text{start}}, c_{\text{accept}}, 2^{df(n)})$.
- Algorithm $\textsc{CanYield}$ obviously solves the yieldability problem. So M correctly simulates N.
- We need to verify that M works within $O\left(f^2(n)\right)$ space.

  Whenever $\textsc{CanYield}$ invokes itself recursively, it stores the current stage number and the values of c_1, c_2, and t on a stack so that they may be restored upon return. So, each level uses $O(f(n))$ additional space. Since each level divides the size of t in half and t starts at $2^{df(n)}$, the depth of the recursion is $O\left(\log 2^{df(n)}\right)$ or $O(f(n))$. Therefore the total space used is $O\left(f^2(n)\right)$.

A Technical Difficulty Involving Knowledge of $f(n)$

- One technical difficulty arises in this argument because algorithm M needs to know the value of $f(n)$ when it calls CanYield.

  We can handle this difficulty by modifying M so that it tries $f(n) = 1, 2, 3, \ldots$.
  - For each value $f(n) = i$, the modified algorithm uses CanYield to determine whether the accept configuration is reachable.
  - In addition, it uses CanYield to determine whether N uses at least space $i + 1$ by testing whether N can reach any of the configurations of length $i + 1$ from the start configuration.
    - If the accept configuration is reachable, M accepts;
    - If no configuration of length $i + 1$ is reachable, M rejects;
    - Otherwise M continues with $f(n) = i + 1$.
- We could have handled this difficulty in another way by assuming that M can compute $f(n)$ within $O(f(n))$ space, but then we would need to add that assumption to the statement of the theorem.

Subsection 3

The Class PSPACE

Polynomial Space Complexity

- To the polynomial time complexity class P corresponds the space complexity class PSPACE:

Definition (The Class PSPACE)

PSPACE is the class of languages that are decidable in polynomial space on a deterministic Turing machine. I.e., $\text{PSPACE} = \bigcup_k \text{SPACE}(n^k)$.

- We define NPSPACE, the nondeterministic counterpart to PSPACE, in terms of the NSPACE classes.
- However, since the square of any polynomial is still a polynomial, by Savitch's Theorem, PSPACE = NPSPACE.
- Example: We showed that $\textsc{Sat}$ is in SPACE(n) and that $\textsc{All}_{\text{NFA}}$ is in coNSPACE(n). Hence, since deterministic space complexity classes are closed under complement, by Savitch's theorem, $\textsc{All}_{\text{NFA}}$ is also in SPACE(n^2). Therefore, both languages are in PSPACE.

Relations Between PSPACE, P and NP

- We look at the relationship of PSPACE with P and NP.
  - We have $\mathrm{P} \subseteq \mathrm{PSPACE}$:

    A machine can explore at most one new cell at each step of its computation. Thus, for $t(n) \geq n$, any machine that operates in time $t(n)$ can use at most $t(n)$ space.
  - Similarly, $\mathrm{NP} \subseteq \mathrm{NPSPACE}$, whence $\mathrm{NP} \subseteq \mathrm{PSPACE}$.
- Conversely, we can bound the time complexity of a Turing machine in terms of its space complexity.
  - We show that $\mathrm{PSPACE} \subseteq \mathrm{EXPTIME} = \bigcup_k \mathrm{TIME}(2^{n^k})$.

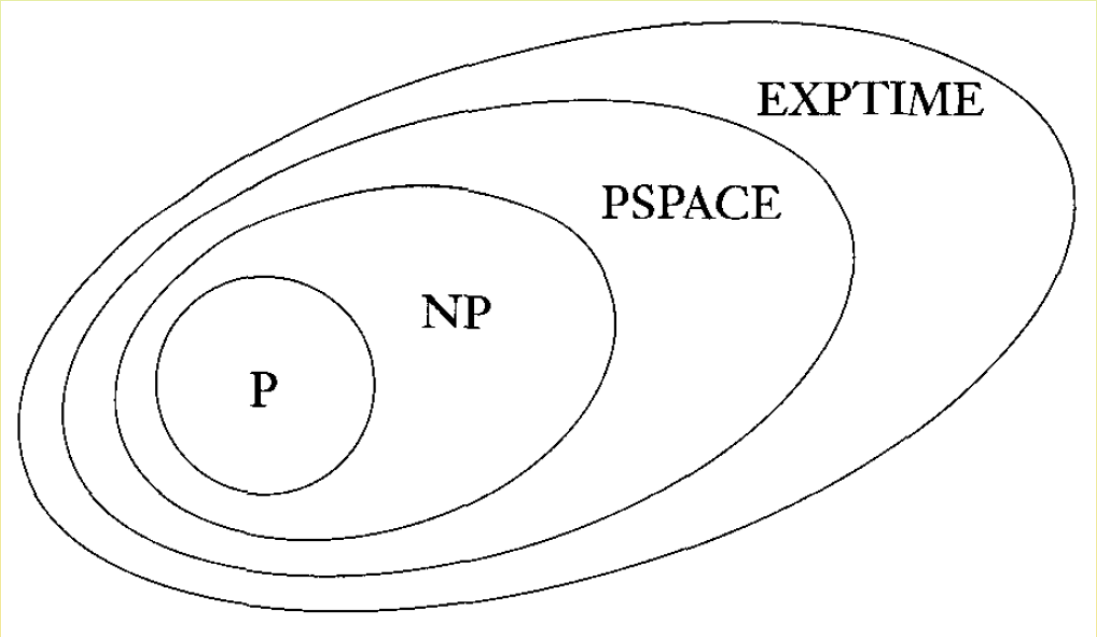
    For $f(n) \geq n$, a TM that uses $f(n)$ space can have at most $f(n)2^{\mathrm{O}(f(n))}$ different configurations. A TM computation that halts may not repeat a configuration. Therefore, a TM that uses space $f(n)$ must run in time $f(n)2^{\mathrm{O}(f(n))}$.

Depiction of the Relationships Between Classes

- We showed that:

$$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} = \mathrm{NPSPACE} \subseteq \mathrm{EXPTIME}.$$

- It is not known whether any of these "$\subseteq$" is actually an equality.
- However, we will prove that $\mathrm{P} \neq \mathrm{EXPTIME}$.
- So, at least one of "$\subseteq$" is proper, but we are unable to say which!
- Most researchers believe that all containments are proper, i.e., that:



Subsection 4

PSPACE-Completeness

PSPACE-Complete Languages

- NP-complete languages are the most difficult languages in NP: Demonstrating that a language is NP-complete provides strong evidence that the language is not in P.
- The analogous notion for the class PSPACE is PSPACE-completeness.

Definition (PSPACE-Complete Language)

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

1. B is in PSPACE;
2. Every A in PSPACE is polynomial time reducible to B.

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

- In the definition, we use polynomial time reducibility.
- We look now at why polynomial time reducibility is still used, instead of some "analogous" notion of "polynomial space reducibility".

Why Use Polynomial Time Reducibility

- Complete problems for a complexity class are important because they are examples of the most difficult problems in the class.
- Reason for difficulty: Since any other problem in the class is easily reduced to it, if we find an easy way to solve the complete problem, we can easily solve all other problems in the class.
- For this idea to work, the reduction must be easy, relative to the complexity of typical problems in the class.
- If the reduction itself were difficult to compute, an easy solution to the complete problem would not necessarily yield an easy solution to the problems reducing to it.
- This reasoning provides a rule for reductions:

> Whenever we define complete problems for a complexity class, the reduction model must be more limited than the model used for defining the class itself.

Universal and Existential Quantifiers

- A **Boolean formula** is an expression that contains Boolean variables, the constants 0 and 1, and the Boolean operations $\wedge, \vee$ and $\neg$.
- The **quantifiers** $\forall$ (for all) and $\exists$ (there exists) are often used in mathematical statements:
  - Writing the statement $\forall x\ \phi$ means that, for every value of the variable x, the statement ϕ is true.
  - Writing the statement $\exists x\ \phi$ means that, for some value of the variable x, the statement ϕ is true.
- Sometimes, $\forall$ is referred to as the **universal quantifier** and $\exists$ as the **existential quantifier**.
- We say that the variable x immediately following the quantifier is **bound** to the quantifier.

Universe of Interpretation

- Example: Considering the natural numbers, the statement

$$\forall x\ [x + 1 > x]$$

means that "the successor $x + 1$ of every natural number x is greater than the number itself". Obviously, this statement is true.
The statement

$$\exists y\ [y + y = 3]$$

is obviously false.

- When interpreting statements (i.e., assigning meaning to statements) involving quantifiers, the **universe** from which the values are drawn is extremely important.

  Example: If, instead of the natural numbers, we considered the set of real numbers as the universe, the preceding existentially quantified statement would become true.

Multiple Quantifiers

- Statements may contain several quantifiers, as in

$$\forall x \, \exists y \, [y > x].$$

  Interpreted in the universe of the natural numbers, this statement says that "for every natural number there exists another natural number larger than it".
- The order of the quantifiers is important!
- Reversing the order, as in the statement

$$\exists y \, \forall x \, [y > x],$$

  gives an entirely different meaning. It says that "some natural number is greater than all others".
- Note that the first statement is true and the second statement is false.

Scope and Prenex Normal Form

- A quantifier may appear anywhere in a mathematical statement:

  It applies to the fragment of the statement appearing within the matched pair of parentheses or brackets following the quantified variable. This fragment is called the **scope** of the quantifier.
- If all quantifiers appear at the beginning of the statement and each quantifier's scope is everything following it, the statement is said to be in **prenex normal form**.
- Since any statement can be converted easily into prenex normal form, we only consider statements in this form.

Quantified Boolean Formulas

- Boolean formulas with quantifiers are called **quantified Boolean formulas**. For such formulas, the universe is $\{0, 1\}$.
  Example: Consider
  $$\phi = \forall x \; \exists y \; [(x \vee y) \wedge (\overline{x} \vee \overline{y})].$$
  ϕ is a quantified Boolean formula. ϕ is true, but it would be false if the quantifiers $\forall x$ and $\exists y$ were reversed.
- When each variable of a formula appears within the scope of some quantifier, the formula is said to be **fully quantified** or a **sentence** and is always either true or false.
  Example: The preceding formula ϕ is fully quantified. However, if the initial part, $\forall x$, of ϕ were removed, the formula would no longer be fully quantified and would be neither true nor false.
- The TQBF problem is to determine whether a fully quantified Boolean formula is true or false:
  $$\mathrm{TQBF} = \{\langle \phi \rangle : \phi \text{ is a true fully quantified Boolean formula}\}.$$

A PSPACE-Complete Language

Theorem

TQBF is PSPACE-complete.

- TQBF is in PSPACE: An algorithm assigns values to the variables and recursively evaluates the truth of the formula.
- Every language A in PSPACE reduces to TQBF in polynomial time: Let M be a polynomial space-bounded Turing machine for A. Polynomial time reduce an input w to a QBF ϕ that encodes a simulation of M on w, such that ϕ is true iff M accepts w.
  - In the Cook-Levin Theorem, we construct ϕ that simulates M on w by expressing the requirements for an accepting tableau. A tableau for M on w has width $O\left(n^k\right)$, the space used by M, but has exponential height in n^k because M can run for exponential time. This would result in a formula of exponential size, i.e., a non-polynomial time reduction.
  - A technique related to the proof of Savitch's theorem is used: The formula divides the tableau into halves and employs the universal quantifier to represent each half with the same part of the formula.

TQBF is in PSPACE

- The following polynomial space algorithm decides TQBF:
  T: On input $\langle \phi \rangle$, ϕ a fully quantified Boolean formula:
  1. If ϕ contains no quantifiers, then it is an expression with only constants, so evaluate ϕ and accept if it is true; otherwise, reject.
  2. If ϕ equals $\exists x\ \psi$, recursively call T on ψ, first with 0 substituted for x and then with 1 substituted for x. If either result is accept, then accept; otherwise, reject.
  3. If ϕ equals $\forall x\ \psi$, recursively call T on ψ, first with 0 substituted for x and then with 1 substituted for x. If both results are accept, then accept; otherwise, reject.
- Algorithm T obviously decides TQBF.
- The depth of the recursion is at most the number of variables. At each level we only store the value of one variable, so the total space used is $O(m)$, where m is the number of variables. So T runs in linear space.
- TQBF is PSPACE-hard: Let A be a language decided by a TM M in space n^k. We give a polynomial time reduction from A to TQBF.

TQBF is PSPACE-hard: $\phi_{c_1,c_2,t}$

- The reduction maps a string w to a quantified Boolean formula ϕ that is true iff M accepts w.

  To show how to construct ϕ we solve a more general problem:

  Using two collections of variables, denoted c_1 and c_2, representing two configurations, and a number $t > 0$, we construct a formula $\phi_{c_1,c_2,t}$. If we assign c_1 and c_2 to actual configurations, the formula is true iff M can go from c_1 to c_2 in at most t steps. Then, we can let ϕ be the formula $\phi_{c_{\text{start}},c_{\text{accept}},h}$, where $h = 2^{df(n)}$, for a constant d, chosen so that M has no more than $2^{df(n)}$ possible configurations on an input of length n. We let $f(n) = n^k$ and, for convenience, assume that t is a power of 2.

  The formula encodes the contents of tape cells as in the proof of the Cook-Levin Theorem. Each cell has several variables associated with it, one for each tape symbol and state. Each configuration has n^k cells and so is encoded by $O\left(n^k\right)$ variables.

$\phi_{c_1,c_2,t}$: The Case $t = 1$

- If t $= 1$, we can easily construct $\phi_{c_1,c_2,t}$:

  We design the formula to say that:
  - either c_1 equals c_2, or
  - c_2 follows from c_1 in a single step of M.

  Considering each case:
  - We express the equality by writing a Boolean expression saying that each of the variables representing c_1 contains the same Boolean value as the corresponding variable representing c_2.
  - We express the second possibility by using the technique presented in the proof of the Cook-Levin theorem, i.e., we can express that c_1 yields c_2 in a single step of M by writing Boolean expressions stating that the contents of each triple of c_1's cells correctly yield the contents of the corresponding triple of c_2's cells.

$\phi_{c_1,c_2,t}$: A First Attempt at $t > 1$

- If $t > 1$, we construct $\phi_{c_1,c_2,t}$ recursively.
- We try one idea that does not work and then fix it: Let $\phi_{c_1,c_2,t} = \exists m_1 \, [\phi_{c_1,m_1,\frac{t}{2}} \wedge \phi_{m_1,c_2,\frac{t}{2}}]$. The symbol m_1 represents a configuration of M. $\exists m_1$ is shorthand for $\exists x_1, \ldots, x_\ell$, where $\ell = \mathrm{O}\left(n^k\right)$ and $x_1, \ldots, x_\ell$ are the variables that encode m_1. Then, we construct the two formulas $\phi_{c_1,m_1,\frac{t}{2}}$ and $\phi_{m_1,c_2,\frac{t}{2}}$ recursively.
- The formula $\phi_{c_1,c_2,t}$ has the correct value: it is TRUE whenever M can go from c_1 to c_2 within t steps.
- The problem is that it is too big. Every level of the recursion cuts t in half but roughly doubles the size of the formula. Hence we end up with a formula of size roughly t. Initially, $t = 2^{df(n)}$, so this method gives an exponentially large formula.
- To reduce the size of the formula we use the $\forall$ quantifier in addition to the $\exists$ quantifier.

$\phi_{c_1,c_2,t}$: The Case $t > 1$

- Let $\phi_{c_1,c_2,t} = \exists m_1 \ \forall (c_3, c_4) \in \{(c_1, m_1), (m_1, c_2)\} \ [\phi_{c_3,c_4,\frac{t}{2}}]$.
- The introduction of the new variables representing the configurations c_3 and c_4 allows us to "fold" the two recursive subformulas into a single subformula, while preserving the original meaning.
- We may replace the construct $\forall x \in \{y, z\}[\ldots]$ by the equivalent construct $\forall x \ [(x = y \vee x = z) \rightarrow \ldots]$ to obtain a syntactically correct quantified Boolean formula. Moreover, Boolean implication $\rightarrow$ and Boolean equality $=$ can be expressed in terms of AND and NOT.
- To calculate the size of the formula $\phi_{c_{\text{start}},c_{\text{accept}},h}$, where $h = 2^{df(n)}$, note that:
  - Each level of the recursion adds a portion of the formula that is linear in the size of the configurations and is thus of size $\mathrm{O}(f(n))$.
  - The number of levels of the recursion is $\log(2^{df(n)})$ or $\mathrm{O}(f(n))$.

  Hence, the size of the resulting formula is $\mathrm{O}\left(f^2(n)\right)$.

Quantifiers and Games

- A **game** is loosely defined to be a competition in which opposing parties attempt to achieve some goal according to prespecified rules.
- Games are closely related to quantifiers.
  - A quantified statement has a corresponding game.
  - Conversely, a game often has a corresponding quantified statement.
- These correspondences are helpful in several ways.
  - Expressing a mathematical statement that uses many quantifiers in terms of the corresponding game may facilitate in understanding its meaning.
  - Expressing a game in terms of a quantified statement aids in understanding the complexity of the game.
- To illustrate the correspondence between games and quantifiers, we consider a simple artificial game, called the **formula game**.

The Formula Game

- Let

$$\phi = \exists x_1 \ \forall x_2 \ \exists x_3 \cdots \mathrm{Q} x_k \ [\psi]$$

be a quantified Boolean formula in prenex normal form, where Q represents either a $\forall$ or $\exists$ quantifier.

We associate a **formula game** with ϕ as follows:

- Two players, called Player A and Player E, take turns selecting the values of the variables $x_1, \ldots, x_k$.
  - Player A selects values for the variables that are bound to $\forall$ quantifiers;
  - Player E selects values for the variables that are bound to $\exists$ quantifiers.
- The order of play is the same as that of the quantifiers at the beginning of the formula.
- At the end of the play, we use the values that the players have selected for the variables and declare that:
  - Player E has won the game if ψ, the part of the formula with the quantifiers stripped off, is now TRUE.
  - Player A has won if ψ is now FALSE.

Winning Strategy

- Suppose $\phi_1 = \exists x_1\ \forall x_2\ \exists x_3\ [(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3})]$.
- In the formula game for ϕ_1,
  - Player E picks the value of x_1;
  - Player A picks the value of x_2;
  - Player E picks the value of x_3.
- Represent the Boolean value TRUE with 1 and FALSE with 0.
- Suppose Player E picks $x_1 = 1$, then Player A picks $x_2 = 0$, and, finally, Player E picks $x_3 = 1$.

  With these values for x_1, x_2 and x_3, $(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3})$ evaluates to 1, so Player E has won the game.
- In fact, Player E may always win this game by selecting $x_1 = 1$ and then selecting x_3 to be the negation of whatever Player A selects for x_2. We say that Player E has a *winning strategy* for this game.
- In general, a player has a **winning strategy** for a game if that player wins when both sides play optimally.

Formula Game is Complete for Polynomial Space

- We consider the problem of determining which player has a winning strategy in the formula game associated with a formula:

$$\text{FORMULAGAME} = \{\langle\phi\rangle : \text{Player E has a winning strategy in the formula game associated with } \phi\}.$$

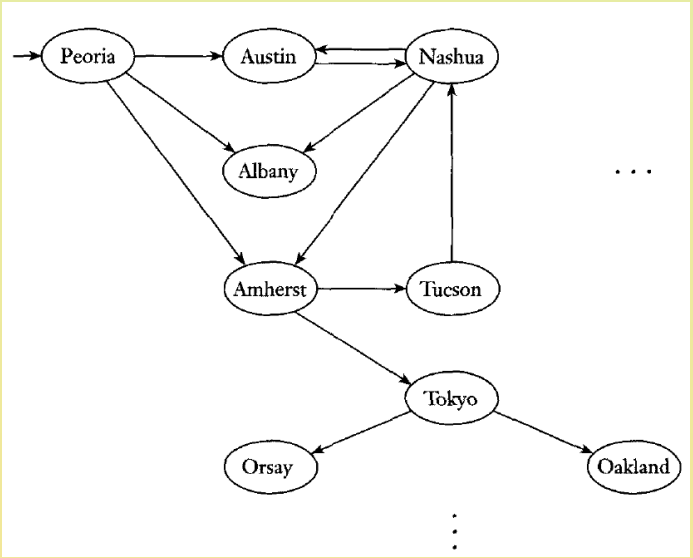
Theorem

FORMULAGAME is PSPACE-complete.

- FORMULAGAME is PSPACE-complete because it is TQBF.
- The formula $\phi = \exists x_1\ \forall x_2\ \exists x_3\ \cdots [\psi]$ is TRUE when some setting for x_1 exists such that, for any setting of x_2, a setting of x_3 exists such that, and so on, where ψ is TRUE under the settings of the variables. Similarly, Player E has a winning strategy in the game associated with ϕ when Player E can make some assignment to x_1, such that, for any setting of x_2, Player E can make an assignment to x_3, such that, and so on, ψ is TRUE under these settings of the variables. The same reasoning applies when the formula does not alternate quantifiers.

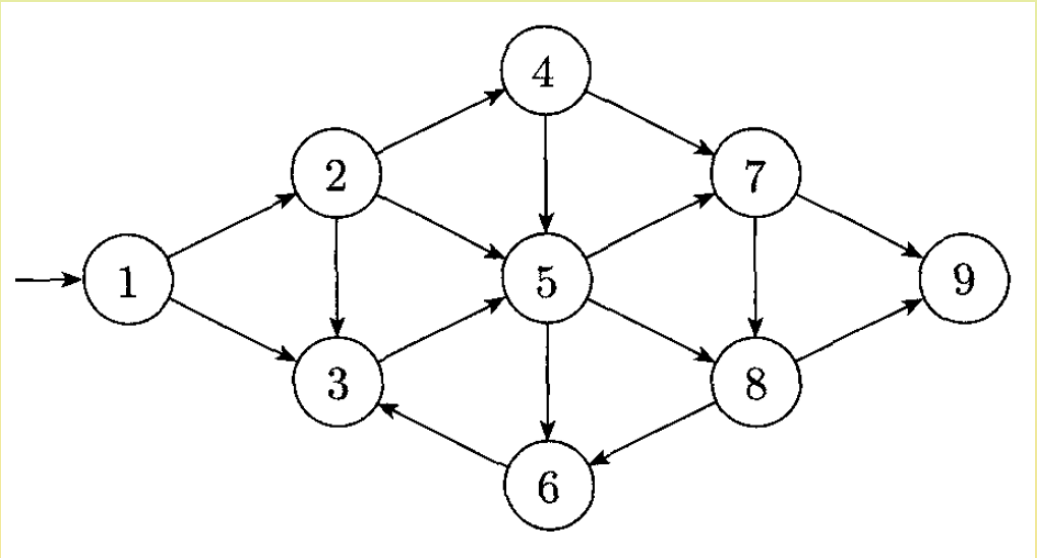
The Game of Geography

- **Geography** is a child's game in which players take turns naming cities from anywhere in the world. Each city chosen must begin with the same letter that ended the previous city's name. Repetition is not permitted.
  - The game starts with some designated starting city.
  - It ends when some player loses because he/she is unable to continue.
- Example: Peoria; Amherst; Tucson; Nashua; $\cdots$; Until one player gets stuck and thereby loses.
- We can model this game with a directed graph whose nodes are the cities of the world. We draw an arrow from one city to another if the first can lead to the second according to the game rules, i.e., the graph contains an edge from a city X to a city Y if city X ends with the same letter that begins city Y.



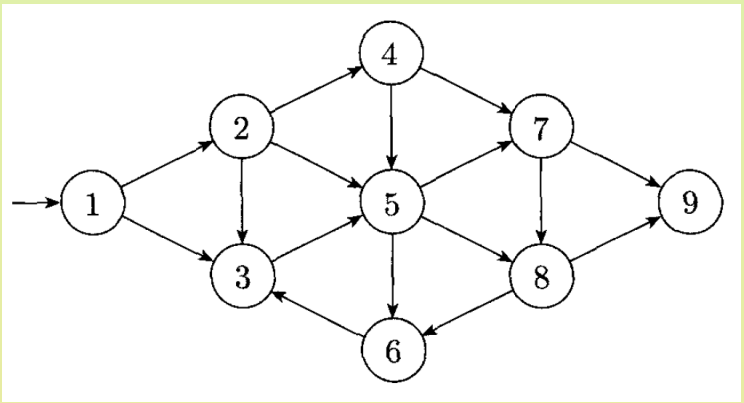
Generalized Geography

- In Geography, as interpreted in terms of the graphic representation:
  - A player starts by selecting the designated start node.
  - The players take turns alternately by picking nodes that form a simple path in the graph. The path being simple (i.e., not using any node more than once) reflects the requirement that a city not be repeated.
  - The first player unable to extend the path loses the game.
- In **generalized geography** we take an arbitrary directed graph with a designated start node instead of the graph associated with the actual cities:



Winning Strategy in Generalized Geography

- Say that Player I moves first and Player II second.



Player I has a winning strategy: He starts at node 1. His first move must be to node 2 or 3 and he chooses 3. Now Player II must move, and she is forced to select node 5. Then Player I selects 6. And Player II is stuck. Thus, Player I wins.

- The problem of determining which player has a winning strategy in a generalized geography game is PSPACE-complete:

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

Generalized Geography is PSPACE-Complete

Theorem

GG is PSPACE-complete.

- A recursive algorithm, similar to the one used for TQBF, determines which player has a winning strategy. It runs in polynomial space and, thus, $\mathrm{GG} \in$ PSPACE.
- To prove that GG is PSPACE-hard, we give a polynomial time reduction from $\textsc{FormulaGame}$ to GG.

  This reduction converts a formula game to a generalized geography graph so that play on the graph mimics play in the formula game. The players in the generalized geography game are really playing an encoded form of the formula game.

Generalized Geography is in PSPACE

- The following algorithm decides whether Player I has a winning strategy in instances of generalized geography, i.e., it decides GG:

  M: On input $\langle G, b \rangle$, with G a directed graph and b a node of G:

  1. If b has outdegree 0, reject, because Player I loses immediately.
  2. Remove node b and all connected arrows to get a new graph G_1.
  3. For each of the nodes $b_1, b_2, \ldots, b_k$ that b originally pointed at, recursively call M on $\langle G_1, b_i \rangle$.
  4. If all of these accept, Player II has a winning strategy in the original game, so reject. Otherwise, Player II does not have a winning strategy, so Player I must; therefore accept.

- The only space required is for storing the recursion stack.
  - Each level of the recursion adds a single node to the stack;
  - At most m levels occur, where m is the number of nodes in G.

  Hence the algorithm runs in linear space.

Generalized Geography is PSPACE-hard

- We show that FORMULAGAME is polynomial time reducible to GG.
- The reduction maps the formula

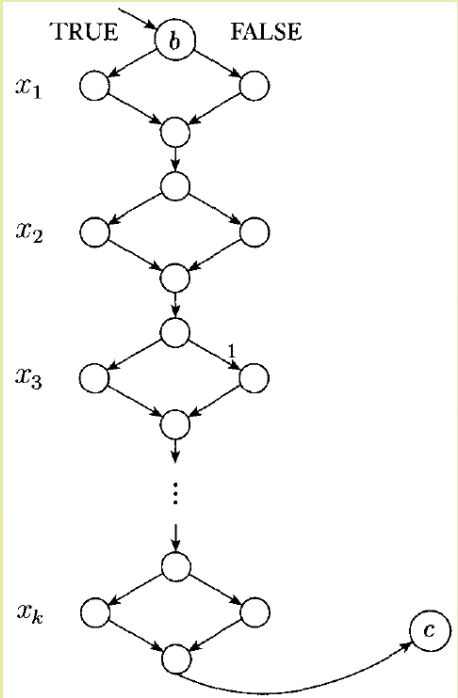
$$\phi = \exists x_1 \ \forall x_2 \ \exists x_3 \ \cdots \mathsf{Q} x_k [\psi]$$

to an instance $\langle G, b \rangle$ of generalized geography.
We assume for simplicity that:
  - ϕ's quantifiers begin and end with $\exists$;
  - They strictly alternate between $\exists$ and $\forall$.

A formula that does not conform to this assumption may be converted to a slightly larger one that does so, by adding extra quantifiers binding otherwise unused or "dummy" variables.
  - We assume also that ψ is in conjunctive normal form.
- The reduction constructs a geography game on a graph G where optimal play mimics optimal play of the formula game on ϕ.
  - Player I in the geography game takes the role of Player E in the formula game;
  - Player II takes the role of Player A.

Left Part of the Geography Graph G

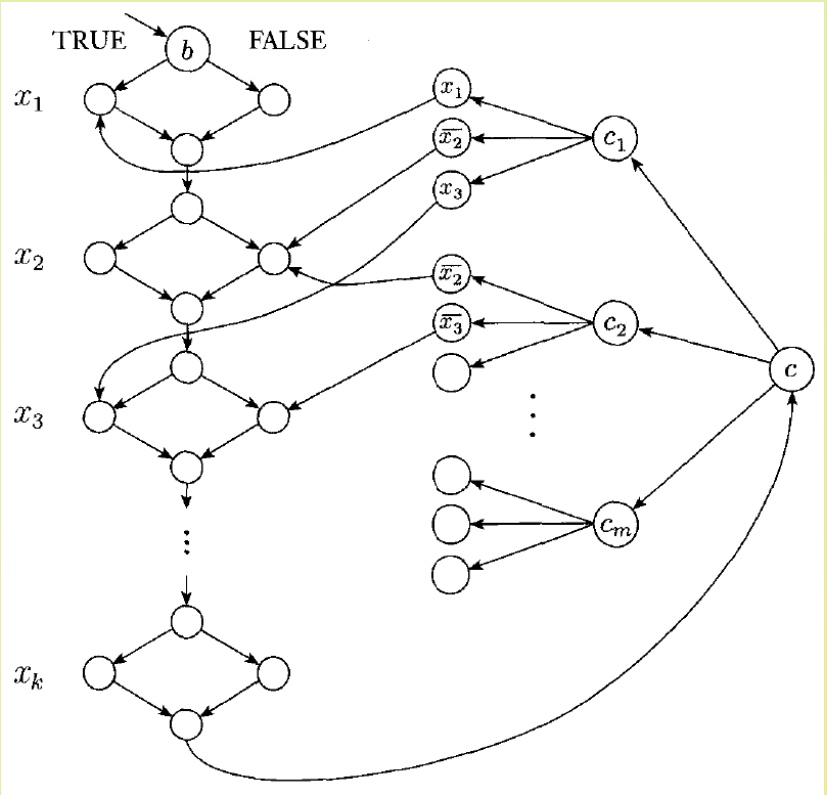
- The structure of graph G is partially shown below:



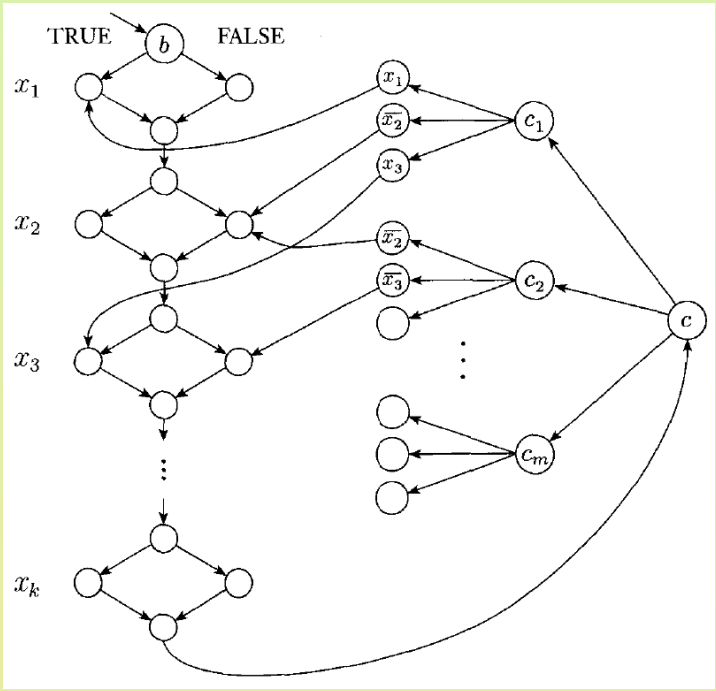
  - Play starts at b. Player I must select one of the two edges going from b. These correspond to Player E's initial possible choices:
    - The left-hand choice for Player I corresponds to TRUE for Player E in the formula game;
    - The right-hand choice to FALSE.
  - After Player I has selected one of these edges, say, the left-hand one, Player II moves. Only one outgoing edge is present, so this move is forced.
    - Player I's next move is also forced.
    - Now two edges again are present, but Player II gets the choice. This choice corresponds to Player A's first move in the formula game.
    - Players I and II choose a path through each of the diamonds.
- At the bottom node, it is Player I's turn because the last quantifier is $\exists$. Player I's next move is forced and we end up at c.

Right Part of the Geography Graph G

- At this point Player II has the next move. This point in the geography game corresponds to the end of play in the formula game. The chosen path corresponds to an assignment to ϕ's variables:
  - If ψ is TRUE, Player E wins;
  - If ψ is FALSE, Player A wins.
- The righthand side of the graph guarantees that
  - Player I can win if Player E has won;
  - Player II can win if Player A has won.



Gaming on the Right Side



- At c, Player II chooses a node corresponding to one of ψ's clauses.
- Then Player I chooses a node corresponding to a literal in clause.
  - The nodes corresponding to unnegated literals are connected to the left-hand (TRUE) sides of the diamond for associated variables;
  - For negated literals to right-hand (FALSE) sides.

- If ψ is FALSE, Player II may win by selecting the unsatisfied clause. Any literal that Player I may pick is FALSE. So it is connected to the side of the diamond that has not yet been played. Player II may play the node in the diamond. Player I is unable to move and loses.
- If ψ is TRUE, any clause that Player II picks contains a TRUE literal. Player I selects that literal. Being TRUE, it is connected to the side that has already been played. Player II is unable to move and loses.

Subsection 5

The Classes L and NL

Turing Machine Model for Sublinear Space

- We have considered only time and space complexity bounds that are at least linear. Now we turn to smaller, sublinear space bounds.
  - In time complexity, sublinear bounds are insufficient for reading the entire input.
  - In sublinear space complexity the machine is able to read the entire input but it does not have enough space to store the input. For this to be meaningful, we must modify the model of computation.
- We introduce a Turing machine with two tapes:
  - A read-only input tape;
  - A read/write work tape.
- On the read-only tape the input head can detect symbols but not change them. The machine can detect when the head is at the left-hand and right-hand ends of the input so that 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.

The Classes L and NL

- Sublinear space algorithms allow the computer to manipulate the data without storing all of it in main memory.
- For space bounds that are at least linear, the two-tape TM model is equivalent to the standard one-tape model.
- For sublinear space bounds, we use only the two-tape model:

Definition (The Classes L and NL)

- L is the class of languages that are decidable in 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 Language in Log Space

- The choice of $\log n$ space is based on reasons similar to those for selecting polynomial time and space:
  - Logarithmic space is just large enough to solve interesting problems.
  - It has attractive mathematical properties such as robustness.
- Since pointers into the input may be represented in logarithmic space, log-space algorithms correspond to a fixed number of input pointers.
- Example: The language $A = \{0^k 1^k : k \geq 0\}$ is a member of L. We described a Turing machine that decides A by zigzagging back and forth across the input, crossing off the 0s and 1s as they are matched. That algorithm uses linear space to record which positions have been crossed off. It can be modified to use only log space:
  The log space TM for A cannot cross off the 0s and 1s because input tape is read-only. Instead, the machine counts the number of 0s and 1s in binary on the work tape. The only space required is that used to record the two counters. In binary, each counter uses only logarithmic space. Hence the algorithm runs in $O(\log n)$ space and $A \in$ L.

A Language in Nondeterministic Log Space

- Recall the language

$$\textsc{Path} = \{\langle G, s, t\rangle : G \text{ is a directed graph that has a directed path from } s \text{ to } t\}.$$

- We saw that PATH is in P.
- The algorithm provided uses linear space. It is not known whether PATH can be solved in logarithmic space deterministically.
- We have a nondeterministic log space algorithm for PATH:
  - Start at node s and nondeterministically guess a node from s.
  - Record only the position of the current node on the work tape.
  - Then nondeterministically select the next node from among those pointed at by the current node.
  - Repeat this action until:
    - Node t is reached, whence accept; or
    - m steps have been completed and reject, where m is the number of nodes in the graph.

Thus, PATH is in NL.

Configurations of a Machine

- The claim that any $f(n)$ space bounded Turing machine also runs in time $2^{O(f(n))}$ is no longer true for very small space bounds.
  Example: A Turing machine that uses $O(1)$ (i.e., constant) space may run for n steps.

Definition (Machine Configuration)

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 $n2^{O(f(n))}$.
  If M has c states and g tape symbols, the number of strings that can appear on the work tape is $g^{f(n)}$. The input head can be in one of n positions and the work tape head can be in one of $f(n)$ positions. So, total number of configurations of M on w is $cnf(n)g^{f(n)}$, or $n2^{O(f(n))}$.

Time vs Space and Savitch's Theorem for Log Space

- We focus on space bounds $f(n)$ that are at least $\log n$.
- The claim that the time complexity of a machine is at most exponential in its space complexity remains true for such bounds: We have $n2^{O(f(n))}$ is $2^{O(f(n))}$ when $f(n) \geq \log n$.
- Savitch's theorem shows that we can convert nondeterministic TMs to deterministic TMs and increase the space complexity $f(n)$ by only a squaring, provided that $f(n) \geq n$.
- Savitch's theorem can be extended for sublinear space bounds down to $f(n) \geq \log n$.

  The proof is identical to the original, except that we use Turing machines with a read-only input tape and instead of referring to configurations of N we refer to configurations of N on w. Storing a configuration of N on w uses $\log\left(n2^{O(f(n))}\right) = \log n + O\left(f(n)\right)$ space. If $f(n) \geq \log n$, the storage used is $O\left(f(n)\right)$. The remainder of the proof remains the same.

Subsection 6

NL-Completeness

L and NL

- The PATH problem is known to be in NL but not known to be in L. PATH is thought not to belong to L, but no proof exists.
- More generally, no problem in NL exists that is known to be outside L.
- Analogous to the question of whether P = NP we have the question of whether L = NL.
- We can exhibit certain languages that are NL-complete, i.e., that are, in a certain sense, the most difficult languages in NL.
- If L and NL are different, all NL-complete languages cannot be in L.
- We define an NL-complete language to be one which is in NL and to which any other language in NL is reducible.
- One cannot use polynomial time reducibility because all problems in NL are solvable in polynomial time, so every two problems in NL except $\emptyset$ and Σ^* polynomial time reduce to one another.
- Since polynomial time reducibility is too strong to differentiate problems in NL, we use log space reducibility.

Log Space Reducibility and NL-Completeness

Definition (Log Space Reducibility)

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 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. We call f a **log space computable function**.

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.

Definition (NL-Complete Language)

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

1. $B \in \mathrm{NL}$;
2. Every A in NL is log space reducible to B.

Log Space and Log Space Reducibility

- If one language is log space reducible to another language already known to be in L, the original language is also in L:

Theorem

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

- A tempting approach to get a log space algorithm for A would be to:
  - first map its input w to $f(w)$, using the log space reduction f;
  - then apply the log space algorithm for B.

  However, the storage required for $f(w)$ may be too large to fit within the log space bound.

A Valid Proof

Theorem

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

- A's machine M_A computes individual symbols of $f(w)$ as requested by B's machine M_B.
  In the simulation, 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 $f(w)$. Doing so may require occasional recomputation of parts of $f(w)$ and so is inefficient in its time complexity. The advantage of this method is that only a single symbol of $f(w)$ needs to be stored at any point, in effect trading time for space.

If L $\neq$ NL no NL-Complete Language is in L

Corollary

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

- Clearly, L $\subseteq$ NL.
- Suppose $A \in$ NL. Let $B \in$ L be NL-complete. Then, $A \leq_{\mathrm{L}} B$, whence, by the theorem, $A \in$ L. Thus, NL $\subseteq$ L.

Searching in Graphs

Theorem

PATH is NL-complete.

- We saw that PATH is in NL. Thus, it suffices to show that PATH is NL-hard, i.e., that every language A in NL is log space reducible to PATH.

  We construct a graph that represents the computation of the nondeterministic log space Turing machine for A. The reduction maps a string w to a graph whose nodes correspond to the configurations of the NTM on input w. One node points to a second node if the corresponding first configuration can yield the second configuration in a single step of the NTM. Hence, the machine accepts w whenever some path from the node corresponding to the start configuration leads to the node corresponding to the accepting configuration.

NL-Completeness of PATH

- We give a log space reduction from any A in NL to PATH. Suppose the 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 that contains a path from s to t if and only if M accepts w.
  - The nodes of G are the configurations of M on w.
  - For configurations c_1 and c_2 of M on w, (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.
  - Node s is the start configuration of M on w.
  - Machine M is modified to have a unique accepting configuration, which is designated to be node t.
- This mapping reduces A to PATH:
  - If M accepts, some branch of its computation accepts. Thus, a path from the start configuration s to the accepting configuration t in G exists.
  - Conversely, if some path exists from s to t in G, some computation branch accepts when M runs on input w. Thus, M accepts w.

Reduction from A to PATH is in Log Space

- The reduction operates in log space: To see this, we describe a log space transducer which, on input w, outputs a description of G. The description comprises two lists: G's nodes and G's edges.
  - Listing the nodes is easy because each node is a configuration of M on w and can be represented in $c \log n$ space for some constant c. The transducer sequentially goes through all possible strings of length $c \log n$, tests whether each is a legal configuration of M on w, and outputs those that pass the test.
  - The transducer lists the edges similarly. Log space is sufficient for verifying that a configuration c_1 of M on w can yield configuration c_2 because the transducer only needs to examine the actual tape contents under the head locations given in c_1 to determine that M's transition function would give configuration c_2 as a result. The transducer tries all pairs (c_1, c_2) in turn to find which qualify as edges of G. Those that do are added to the output tape.

Consequences for the Complexity Class Hierarchy

Corollary

NL $\subseteq$ P.

- The theorem shows that any language in NL is log space reducible to PATH. A Turing machine that uses space $f(n)$ runs in time $n2^{O(f(n))}$. Thus, a reducer that runs in log space also runs in polynomial time. Therefore, any language in NL is polynomial time reducible to PATH, which in turn is in P. Every language that is polynomial time reducible to a language in P is also in P, which completes the proof.
- Despite the restrictiveness of log space reducibility, it is sufficient for many reductions:
  - The reduction of any PSPACE problem to TQBF, presented earlier, may be computed using only log space. Thus, TQBF is PSPACE-complete with respect to log space reducibility.
  - This conclusion is important in view of NL $\subsetneq$ PSPACE. This separation with log space reducibility implies that TQBF $\notin$ NL.

Subsection 7

NL Equals coNL

NL = coNL

- Recall that the classes NP and coNP are generally believed to be different.
- At first glance, the same appears to hold for the classes NL and coNL, but, in fact, NL equals coNL.

Theorem

NL = coNL.

- We show that $\overline{\text{PATH}}$ is in NL. Since PATH is NL-complete, this shows that every problem in coNL is also in NL.
- The NL algorithm M for $\overline{\text{PATH}}$ must have an accepting computation whenever the input graph G does not contain a path from s to t.
- We first tackle an easier problem: Let c be the number of nodes in G that are reachable from s. We assume that c is provided as an input to M and show how to use c to solve $\overline{\text{PATH}}$. Later we show how to compute c.

The Simplified Problem

- Given G, s, t and c, the machine M operates as follows:
  - One by one, M goes through all m nodes of G and nondeterministically guesses whether each one is reachable from s.
  - Whenever a node u is guessed to be reachable, M attempts to verify this guess by guessing a path of length m or less from s to u.
    - If a computation branch fails to verify this guess, it rejects.
    - In addition, if a branch guesses that t is reachable, it rejects.
  - Machine M counts the number of nodes that have been verified to be reachable.
  - When a branch has gone through all of G's nodes, it checks that the number of nodes that it verified to be reachable from s equals c, the number of nodes that actually are reachable.
    - It rejects if not.
    - Otherwise, this branch accepts.
- In summary:

  M nondeterministically selects exactly c nodes reachable from s, not including t, and proves that each is reachable from s by guessing the path. The remaining nodes, including t, are not reachable, so it can accept.

The Original Problem: Set Up and Main Idea

- We show how to calculate c, the number of nodes reachable from s.
- We describe a nondeterministic log space procedure, such that at least one computation branch has the correct value for c and all other branches reject.
- For each $i = 0, \ldots, m$, we define A_i to be the collection of nodes that are at a distance of i or less from s.
  - $A_0 = \{s\}$;
  - $A_i \subseteq A_{i+1}$;
  - A_m contains all nodes that are reachable from s.
- Let c_i be the number of nodes in A_i.
- We describe a procedure that calculates c_{i+1} from c_i. Repeated application of this procedure yields the desired value of $c = c_m$.

The Original Problem: Computing c_{i+1} from c_i

- To calculate c_{i+1} from c_i, the algorithm goes through all the nodes of G, determines whether each is a member of A_{i+1}, and counts the members.
  - To determine whether a node v is in A_{i+1}, we use an inner loop to go through all the nodes of G and guess whether each node is in A_i.
    - Each positive guess is verified by guessing the path of length at most i from s.
    - For each node u verified to be in A_i, the algorithm tests whether (u, v) is an edge of G. If it is an edge, v is in A_{i+1}.
    - Additionally, the number of nodes verified to be in A_i is counted.
  - At the completion of the inner loop:
    - If the total number of nodes verified to be in A_i is not c_i, all A_i have not been found, so this computation branch rejects.
    - If the count equals c_i and v has not yet been shown to be in A_{i+1}, we conclude that it is not in A_{i+1}.

Then we go on to the next v in the outer loop.

The Formal Algorithm I

The algorithm for PATH: Let m be the number of nodes of G.
M: On input $\langle G, s, t \rangle$:

1. Let $c_0 = 1$.
2. For $i = 0$ to $m - 1$:
    3. Let $c_{i+1} = 1$.
    4. For each node $v \neq s$ in G:
        5. Let $d = 0$.
        6. For each node u in G:
        7. Nondeterministically either perform or skip these steps:
        8. Nondeterministically follow a path of length at most i from s and reject if it does not end at u.
        9. Increment d.
        10. If (u, v) is an edge of G, increment c_{i+1} and go to Stage 5 with the next v.
        11. If $d \neq c_i$, then reject.

The Formal Algorithm II

12. Let $d = 0$.
13. For each node u in G:
    14. Nondeterministically either perform or skip these steps:
        15. Nondeterministically follow a path of length at most m from s and reject if it does not end at u.
        16. If $u = t$, then reject.
        17. Increment d.
18. If $d \neq c_m$, then reject. Otherwise, accept.

This algorithm only needs to store u, v, c_i, c_{i+1}, d, i, and a pointer to the head of a path, at any given time. Hence, it runs in log space.

State of the Art in Class Relationships

- Our present knowledge of the relationships among several complexity classes is as follows:

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

- We do not know whether any of these containments is proper, although we will show that $\mathsf{NL} \subsetneq \mathsf{PSPACE}$.
- Consequently, either $\mathsf{coNL} \subsetneq \mathsf{P}$ or $\mathsf{P} \subsetneq \mathsf{PSPACE}$ must hold.
- The current belief is that all these containments are proper.