Advanced Computational Complexity

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU Math 600

1. Space Complexity
   - Space-Bounded Computation
   - PSPACE-Completeness
   - NL-Completeness

Subsection 1

Space-Bounded Computation

Space-Bounded Computation

Definition (Space-Bounded Computation)

Let $S : \mathbb{N} \to \mathbb{N}$ and $L \in \{0,1\}^*$.

- We say that $L \in \mathsf{SPACE}(S(n))$ if there is a constant c and a TM M deciding L, such that at most $c \cdot S(n)$ locations on M's work tapes (excluding the input tape) are ever visited by M's head during its computation on every input of length n.
- We say that $L \in \mathsf{NSPACE}(S(n))$ if there is an NDTM M deciding L that never uses more than $c \cdot S(n)$ nonblank tape locations on length n inputs, regardless of its nondeterministic choices.

- The space bound applies only to the work tape.

Space Constructible Bounds

- A function $S : \mathbb{N} \to \mathbb{N}$ is **space-constructible** if there exists a TM that, given input x computes

$$x \mapsto S(|x|)$$

in $O(S(|x|))$ space.

- We consider only space-constructible bounds $S : \mathbb{N} \to \mathbb{N}$.
- S being space-constructible means the machine "knows" the space bound.
- All common functions are space-constructible.

Space Bounds vs. Time Bounds

- Separating the TM's work tapes from its input tape makes it possible to consider space-bounded machines that use space less than the input length, i.e., such that

$$S(n) < n.$$

- In contrast, for time bounded computation $\mathrm{DTIME}(T(n))$, $T(n) < n$ does not make much sense, since the TM does not have enough time to read the entire input.
- We will require that $S(n) > \log n$, since the input tape has length n, and we would like the machine to at least be able to "remember" the index of the cell of the input tape that it is currently reading.

Space Bounds vs. Time Bounds (Cont'd)

- By construction, a TM can access only one tape cell per step.
- So we have
$$\mathsf{DTIME}(S(n)) \subseteq \mathsf{SPACE}(S(n)).$$
- A SPACE($S(n)$) machine can run for much longer than $S(n)$ steps, since space can be reused.
- Indeed, a space $S(n)$ machine can easily run for as much as $2^{\Omega(S(n))}$ steps.

  Example: Consider the machine that uses work tape of size $S(n)$ to maintain a counter that it increments from 1 to $2^{S(n)-1}$.

  This machine:
  - Uses space $S(n)$;
  - Runs for $2^{\Omega(S(n))}$ steps.

Relations Between Time and Space Bounds

- Any language in $\mathsf{SPACE}(S(n))$ is in $\mathsf{DTIME}(2^{O(S(n))})$.
- The same holds even for languages in $\mathsf{NSPACE}(S(n))$.
- Up to logarithmic terms, these are the only relationships known between the power of space-bounded and time-bounded computation.

Theorem

For every space constructible $S : \mathbb{N} \to \mathbb{N}$,

$$\mathsf{DTIME}(S(n)) \subseteq \mathsf{SPACE}(S(n)) \subseteq \mathsf{NSPACE}(S(n)) \subseteq \mathsf{DTIME}(2^{O(S(n))}).$$

Space Bounds and the Halting Requirement

- In defining nondeterministic space bounds, imposing the additional restriction that the NDTM has to halt and produce an answer on every input, regardless of the sequence of nondeterministic choices, is redundant, provided we focus on $\mathsf{NSPACE}(S(n))$, where $S(n)$ is space-constructible.
- The NDTM can be easily modified to always halt.
- This scan be achieved by keeping a counter and halting if the computation runs for more than $2^{cS(n)}$ steps, for some suitable constant c.

Configuration Graphs

- Let M be a (deterministic or nondeterministic) TM.
- At a particular point during the execution of M, a **configuration** of M consists of:
  - Its state;
  - The contents of all non blank entries of M's tapes;
  - The head positions.
- For a space $S(n)$ TM M and input $x \in \{0,1\}^*$, the **configuration graph of** M **on input** x, denoted $G_{M,x}$, is a directed graph.
- The nodes correspond to all possible configurations of M, where:
  - The input contains x;
  - The work tapes have $\leq S(|x|)$ non blank cells.

Configuration Graphs (Cont'd)

- The configuration graph $G_{M,x}$ has a directed edge from configuration C to configuration C',
$$C \longrightarrow C',$$
if C' can be reached from C in one step according to M's transition function.
  - If M is deterministic, then the graph has out-degree one.
  - If M is nondeterministic, then the graph has out-degree at most two.
- By modifying M to erase all its work tapes before halting, we can assume that there is only a single configuration C_{accept} on which M halts and outputs 1.
- This means that M accepts the input x if and only if there exists a directed path in $G_{M,x}$ from C_{start} to C_{accept}.

Properties of Configuration Graphs

Claim

Let $G_{M,x}$ be the configuration graph of a space-$S(n)$ machine M on some input x of length n. Then:

1. Every vertex in $G_{M,x}$ can be described using $cS(n)$ bits for some constant c (depending on M's alphabet size and number of tapes).

   In particular, $G_{M,x}$ has at most $2^{cS(n)}$ nodes.
2. There is a $\mathrm{O}(S(n))$-size CNF formula $\varphi_{M,x}$ such that, for every two strings C, C',

$$\varphi_{M,x}(C, C') = 1 \quad \text{iff} \quad C \text{ and } C' \text{ encode two neighboring configurations in } G_{M,x}.$$

Properties of Configuration Graphs (Cont'd)

- For Part 1, we note that a configuration is completely described by giving:
  - The state the TM is in;
  - The contents of all work tapes;
  - The positions of the heads.

  Encoding a configuration entails encoding:
  - The snapshot, i.e., state and current symbol read by all tapes;
  - In sequence, the non blank contents of all the work tapes, inserting special "marker" symbols to denote the locations of the heads.

Properties of Configuration Graphs (Cont'd)

- Part 2 is based on the proof of the Cook-Levin Theorem.

  Deciding whether two configurations are neighboring can be expressed as the AND of many checks.

  Each check depends on only a constant number of bits.

  Such checks can be expressed by constant-sized CNF formulas.

  Moreover, the number of variables is proportional to the workspace.

Proof of the Time-Space Bounds Theorem

Theorem

For every space constructible $S : \mathbb{N} \to \mathbb{N}$,

$$\mathrm{DTIME}(S(n)) \subseteq \mathrm{SPACE}(S(n)) \subseteq \mathrm{NSPACE}(S(n)) \subseteq \mathrm{DTIME}(2^{O(S(n))}).$$

- $\mathrm{DTIME}(S(n)) \subseteq \mathrm{SPACE}(S(n)) \subseteq \mathrm{NSPACE}(S(n))$ are obvious.

  So it suffices to show $\mathrm{NSPACE}(S(n)) \subseteq \mathrm{DTIME}(2^{O(S(n))})$.
  - By enumerating over all possible configurations, we can construct the graph $G_{M,x}$ in $2^{O(S(n))}$-time.
  - We then check whether C_{start} is connected to C_{accept} in $G_{M,x}$ using a standard linear time (in the size of the graph) breadth-first search algorithm for connectivity.

Some Space Complexity Classes

Definition (Some Space Complexity Classes)

$$\text{PSPACE} = \bigcup_{c>0} \text{SPACE}(n^c); \qquad \text{L} = \text{SPACE}(\log n);$$

$$\text{NPSPACE} = \bigcup_{c>0} \text{NSPACE}(n^c); \qquad \text{NL} = \text{NSPACE}(\log n).$$

- We can think of PSPACE and NPSPACE as the space analogs of the time complexity classes P and NP, respectively.
- Since time bounds shorter than the input length do not make sense, there are no time analogs for L and NL.

Example

- We show $3\text{SAT} \in \text{PSPACE}$.
- We describe a TM that decides 3SAT in linear space, i.e., $O(n)$ space, where n is the size of the 3SAT instance.
- The machine just uses the linear space to cycle through all 2^k assignments in order, where k is the number of variables.
- Once an assignment has been checked:
    - It can be erased from the work tape;
    - The work tape can then be reused to check the next assignment.
- A similar idea of cycling through all potential certificates applies to any NP language.
- Therefore,

$$\text{NP} \subseteq \text{PSPACE}.$$

Example

- Consider the languages

$$\begin{array}{rcl} \textsc{Even} & = & \{x : x \text{ has an even number of 1s}\}; \\ \textsc{Mult} & = & \{(\llcorner n \lrcorner, \llcorner m \lrcorner, \llcorner nm \lrcorner) : n \in \mathbb{N}\}. \end{array}$$

- We may use:
  - The grade school method for arithmetic;
  - The fact that a log space machine has enough space to keep a counter up to n.
- In that way, we can check that both Even and Mult are in L.

The Directed Path Problem

- Consider the problem

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

- We show that $\mathrm{PATH} \in \mathsf{NL}$.
- If there is a path from s to t, then there is one of length at most n.
- A nondeterministic machine can take a "nondeterministic walk" starting at s.
- It always maintains the index of the vertex it is at.
- Moreover, it uses nondeterminism to select a neighbor of this vertex to go to next.

The Directed Path Problem (Cont'd)

- The machine accepts iff the walk ends at t in at most n steps, where n is the number of nodes.
- If the nondeterministic walk has run for n steps already and t has not been encountered, the machine rejects.
- The work tape only needs to hold $O(\log n)$ bits of information at any step:
  - The number of steps that the walk has run for;
  - The identity of the current vertex.
- It is open whether PATH is also in L.
- PATH is NL-complete.
- So the problem of whether PATH is in L is equivalent to whether or not $L = NL$.
- The restriction of PATH to undirected graphs is known to be in L.

Space Hierarchy Theorem

- Analogously to time-bounded classes, there is also a hierarchy theorem for space bounded computation.

Theorem (Space Hierarchy Theorem)

If f, g are space-constructible functions satisfying $f(n) = o(g(n))$, then

$$\mathsf{SPACE}(f(n)) \subsetneq \mathsf{SPACE}(g(n)).$$

- The proof is analogous to the proof of the Time Hierarchy Theorem.
- One has a universal TM using only a constant factor of space overhead (as opposed to the logarithmic factor in time overhead).
- Thus, in the present context, the logarithmic term of the Time Hierarchy Theorem is not needed.

Subsection 2

PSPACE-Completeness

PSPACE-Hardness and PSPACE-Completeness

- We do not know if P $=$ PSPACE.
- Since NP $\subseteq$ PSPACE, equality would imply P $=$ NP.
- Recall $L \leq_p L'$ means that L is polynomial-time reducible to L'.
- We use reductions to present some complete problems for PSPACE.

Definition (PSPACE-Hard and PSPACE-Complete Language)

A language L' is PSPACE-**hard** if for every $L \in$ PSPACE, $L \leq_p L'$.
If, in addition, $L' \in$ PSPACE then L' is PSPACE-**complete**.

- If any PSPACE-complete language is in P then so is every other language in PSPACE.
- Equivalently, if PSPACE $\neq$ P, then a PSPACE-complete language is not in P.
- Intuitively speaking, a PSPACE-complete language is the "most difficult" problem of PSPACE.

A First PSPACE-Complete Language

- Define the language

$$\textsc{SpaceTmSat} = \{\langle M, w, 1^n \rangle : \text{DTM } M \text{ accepts } w \text{ in space } n\}.$$

Theorem

SpaceTmSat is PSPACE-complete.

- First, we reason that SpaceTmSat is in PSPACE.

  There exists a universal Turing machine using only a constant factor of space overhead.

  We use this machine to simulate M on input w for n steps.

  Note that a machine using only space n can run in time at most 2^{cn}, for some fixed constant c.

A First PSPACE-Complete Language (Cont'd)

- We next show hardness.
  Suppose that $L \in$ PSPACE.
  Thus, there exists a DTM M, running in n^k space that decides L.
  The polynomial time reduction
  $$f : L \to \textsc{SpaceTmSat}$$
  maps an input x for L to an instance
  $$f(x) = \langle M, x, 1^{|x|^k} \rangle$$
  of $\textsc{SpaceTmSat}$.
  Now $\llcorner M \lrcorner$ and k are constant.
  So $f(x)$ can be constructed in polynomial time, given x.
  Finally, we have
  $$\begin{array}{lll} x \in L & \text{iff} & M \text{ accepts } x \text{ in space } |x|^k \\ & \text{iff} & \langle M, x, 1^{|x|^k} \rangle \in \textsc{SpaceTmSat}. \end{array}$$

Quantified Boolean Formulae

Definition (Quantified Boolean Formula)

A **quantified Boolean formula** (**QBF**) is a formula of the form

$$Q_1 x_1 Q_2 x_2 \cdots Q_n x_n \varphi(x_1, x_2, \ldots, x_n),$$

where:

- Each Q_i is one of the two quantifiers $\forall$ or $\exists$;
- $x_1, \ldots, x_n$ range over $\{0, 1\}$;
- φ is a plain (unquantified) Boolean formula.

The quantifiers $\forall$ and $\exists$ have their standard meaning of "for all" and "exists".

Prenex Normal Form

- The definition restricts attention to quantified Boolean formulae in **prenex normal form** (all quantifiers appear to the left).
- A general quantified Boolean formulae (where the quantifiers can appear elsewhere) may be transformed into an equivalent formula in prenex form in polynomial time.
- Unlike in the case of the SAT and 3SAT, we do not require that the inner unquantified formula φ is in CNF or 3CNF form.
- This choice is also not important, since we can in polynomial time transform a general quantified Boolean formula to an equivalent formula where the inner unquantified formula is in 3CNF form.
- Finally, note that, since all the variables of a QBF are bound by some quantifier, the QBF is always either true or false.

Example

- Consider the formula

$$\forall x \exists y (x \wedge y) \vee (\overline{x} \wedge \overline{y}).$$

- The formula says

"for every $x \in \{0, 1\}$, there is a $y \in \{0, 1\}$, that is equal to x".

- Informally written,

$$\forall x \exists y (x = y).$$

- This formula is true.
- Switching the second quantifier to $\forall$ gives

$$\forall x \forall y (x \wedge y) \vee (\overline{x} \wedge \overline{y}).$$

- This formula is false.

Example

- SAT asks to decide, given a Boolean formula φ, with n free variables $x_1, \ldots, x_n$, whether or not φ has a satisfying assignment.
- I.e., whether there exist $x_1, \ldots, x_n \in \{0, 1\}$, such that

$$\varphi(x_1, \ldots, x_n)$$

  is true.
- Equivalently, SAT asks whether the quantified Boolean formula

$$\psi = \exists x_1 \cdots \exists x_n \varphi(x_1, \ldots, x_n)$$

  is true.

Negation of a Formula

- Note that the negation of the formula

$$Q_1 x_1 \cdots Q_n x_n \varphi(x_1, x_2, \ldots, x_n)$$

  is the same as

$$Q'_1 x_1 \cdots Q'_n x_n \neg\varphi(x_1, x_2, \ldots, x_n),$$

  where:
  - Q'_i is $\exists$ if Q_i was $\forall$;
  - Q'_i is $\forall$ if Q_i was $\exists$.
- The switch of $\exists$ to $\forall$ in the case of SAT gives instances of TAUTOLOGY.
- We saw that TAUTOLOGY is a coNP-complete language.

True Quantified Boolean Formulae

Definition (True Quantified Boolean Formulae)

$\mathrm{TQBF} = \{\psi : \psi$ is a true quantified Boolean formula$\}$.

Theorem

TQBF is PSPACE-complete.

- We first show $\mathrm{TQBF} \in$ PSPACE.
  Let
  $$\psi = Q_1 x_1 Q_2 x_2 \cdots Q_n x_n \varphi(x_1, x_2, \ldots, x_n)$$
  be a quantified Boolean formula with n variables.
  We denote the size of φ by m.
  We design a recursive algorithm A that decides the truth of ψ in $\mathrm{O}(n + m)$ space.
  We solve the slightly more general case, where φ may include the constants 0 ("false") and 1 ("true")

TQBF in in PSPACE

- Suppose, first, $n = 0$ (there are no variables).

  Then the formula contains only constants and can be evaluated in $\mathrm{O}(m)$ time and space.

  Suppose, next, that $n > 0$.

  For $b \in \{0, 1\}$, denote by

  $$\psi\restriction_{x_1=b}$$

  the modification of ψ where Q_1 is dropped and all occurrences of x_1 are replaced by b.

  The algorithm A works as follows.

  - If $Q_1 = \exists$, output 1 iff $A(\psi\restriction_{x_1=0})$ or $A(\psi\restriction_{x_1=1})$ outputs 1.
  - If $Q_1 = \forall$, output 1 iff both $A(\psi\restriction_{x_1=0})$ and $A(\psi\restriction_{x_1=1})$ output 1.

  It is clear that A returns the correct answer on any formula ψ.

Space Complexity of A

- Let $s_{n,m}$ denote the space A uses on formulas with n variables and description size m.

  Space, unlike time, is a reusable resource.

  The crucial point is that both recursive computations

  $$A(\psi\restriction_{x_1=0}) \quad \text{and} \quad A(\psi\restriction_{x_1=1})$$

  can run in the same space.

  After computing $A(\psi\restriction_{x_1=0})$, the algorithm A:
  - Needs to retain only the single bit of output from that computation;
  - Can reuse the rest of the space for the computation of $A(\psi\restriction_{x_1=1})$.

  Suppose A uses $O(m)$ space to write $\psi\restriction_{x_1=b}$ for its recursive calls.

  Then we get that

  $$s_{n,m} = s_{n-1,m} + O(m).$$

  This recursive equation yields

  $$s_{n,m} = O(n \cdot m).$$

PSPACE-Hardness of TQBF

- We now show that, for every $L \in \mathsf{PSPACE}$, $L \leq_p \mathrm{TQBF}$.

  Let M be a machine that decides L in $S(n)$ space and let $x \in \{0,1\}^n$.

  We show how to construct a quantified Boolean formula χ of size $\mathrm{O}\left(S(n)^2\right)$, such that

  $$\chi \text{ is true} \quad \text{iff} \quad M \text{ accepts } x.$$

  Let $m = \mathrm{O}\left(S(n)\right)$ denote the number of bits needed to encode a configuration of M on length n inputs.

  We have shown that there is a Boolean formula $\varphi_{M,x}$, such that, for every two strings $C, C' \in \{0,1\}^m$,

  $$\varphi_M(C, C') = 1 \quad \text{iff} \quad C \text{ and } C' \text{ encode two adjacent configurations in the configuration graph } G_{M,x}.$$

PSPACE-Hardness of TQBF (Cont'd)

- We will use $\varphi_{M,x}$ to come up with a polynomial-sized quantified Boolean formula ψ that has:
  - Polynomially many variables bound by quantifiers;
  - Two unquantified variables,

  such that, for every $C, C' \in \{0,1\}^m$,

  $$\psi(C, C') \text{ is true} \quad \text{iff} \quad C \text{ has a directed path to } C' \text{ in } G_{M,x}.$$

  By plugging in the values C_{start} and C_{accept} to ψ we get a quantified Boolean formula χ that is true iff M accepts x.

  We define the formula ψ inductively.

Construction of the Quantified Boolean Formula

- We let $\psi_i(C, C')$ be true if and only if there is a path of length at most 2^i from C to C' in $G_{M,x}$.

  Note that $\psi = \psi_m$ and $\psi_0 = \varphi_{M,x}$.

  The crucial observation is that there is a path of length at most 2^i from C to C' if and only if there is a configuration C'' with:

  - A path of length at most 2^{i-1} from C to C'';
  - A path of length at most 2^{i-1} from C'' to C'.

  This suggest defining ψ_i by

  $$\psi_i(C, C') = \exists C'' \psi_{i-1}(C, C'') \wedge \psi_{i-1}(C'', C').$$

  But this definition of $\psi_i(C, C')$ is not satisfactory.

  ψ_i's size is at least twice the size of ψ_{i-1}.

  By a simple induction, ψ_m has size about 2^m, which is too large.

Construction of the Quantified Boolean Formula (Cont'd)

- Instead, we use additional quantified variables to save on description size:

$$\begin{array}{rcl}\psi_i(C,C') & = & \exists C''\forall D^1\forall D^2((D^1=C\wedge D^2=C'') \\ & & \vee\,(D^1=C''\wedge D^2=C'))\Rightarrow\psi_{i-1}(D^1,D^2).\end{array}$$

  Now

$$\text{size}(\psi_i)\leq\text{size}(\psi_{i-1})+\mathrm{O}\left(m\right).$$

  Hence, $\text{size}(\psi_m)\leq\mathrm{O}\left(m^2\right)$.

  The two definitions of ψ_i are logically equivalent.

  We can convert the final formula to prenex form in polynomial time.

PSPACE and NPSPACE

- The proof of the theorem uses the notion of a configuration graph.
- Moreover, it does not require this graph to have out-degree one.
- Thus, it can be applied to show that TQBF is not just hard for PSPACE but in fact even for NPSPACE!
- But we know that $\mathrm{TQBF} \in$ PSPACE.
- So this implies that PSPACE $=$ NPSPACE.
- This is quite surprising, since our intuition is that the corresponding classes for time (P and NP) are different.

Savitch's Theorem

Theorem (Savitch's Theorem)

For any space-constructible $S : \mathbb{N} \to \mathbb{N}$, with $S(n) \geq \log n$,

$$\mathsf{NSPACE}(S(n)) \subseteq \mathsf{SPACE}(S(n)^2).$$

- Let $L \in \mathsf{NSPACE}(S(n))$ be a language decided by a TM M, such that for every $x \in \{0,1\}^n$:
  - The configuration graph $G = G_{M,x}$ has at most $M = 2^{O(S(n))}$ vertices;
  - Determining whether $x \in L$ is equivalent to determining whether C_{accept} can be reached from C_{start} in this graph.

Proof of Savitch's Theorem

- We describe a recursive procedure $\text{REACH?}(u, v, i)$ that returns:
  - "YES", if there is a path from u to v of length at most 2^i;
  - "NO", otherwise.

  Again, the main observation is that there is a path from u to v of length at most 2^i iff there is a vertex z such that:
  - There exists an at most 2^{i-1} long path from u to z;
  - There exists an at most 2^{i-1} long path from z to v.

  On inputs u, v, i, REACH? does the following:
  - It enumerates over all vertices z (at a cost of $O(\log M)$ space);
  - Output "YES" if it finds one z, such that:
    - $\text{REACH?}(u, z, i-1) =$"YES";
    - $\text{REACH?}(z, v, i-1) =$'YES".

Proof of Savitch's Theorem (Cont'd)

- Although the algorithm makes n recursive invocations, it can reuse the space in each of these invocations.

  Let $s_{M,i}$ be the space complexity of $\textsc{Reach?}(u, v, i)$ on an M vertex graph.

  Then
  $$s_{M,i} = s_{M,i-1} + O(\log M).$$

  Hence,
  $$s_{M,\log M} = O\left(\log^2 M\right) = O\left(S(n)^2\right).$$

  Finally, observe that C_{accept} is reachable from C_{start} iff it can be reached via a path of length at most M.

PSPACE-Completeness vs. NP-Completeness

- Recall that the central feature of NP-complete problems is that a yes answer has a short certificate.
- The key concept for PSPACE-complete problems seems to be that of a winning strategy for a two-player game with perfect information.
- E.g., in chess two players alternately make moves, and the moves are made on a board visible to both, hence the term perfect information.
- More explicitly, Player 1 has a winning strategy iff:
  - There is a first move for Player 1, such that:
  - For every possible first move of Player 2:
  - There is a second move of Player 1, such that:

    ⋮

  such that at the end Player 1 wins.

PSPACE-Completeness vs. NP-Completeness (Cont'd)

- Deciding whether or not the first player has a winning strategy seems to require searching the tree of all possible moves.
- This is reminiscent of NP, for which we also seem to require exponential search.
- The crucial difference is the lack of a short "certificate" for the statement "Player 1 has a winning strategy".
- The only certificate we can think of is the winning strategy itself.
- As noticed, such a strategy requires exponentially many bits to even describe.

The QBF game

- The "board" for the QBF game is a Boolean formula φ whose free variables are $x_1, x_2, \ldots, x_{2n}$.
- The two players alternately make moves, which involve picking values for $x_1, x_2, \ldots$, in order.
  - Player 1 will pick values for the odd-numbered variables $x_1, x_3, x_5, \ldots$;
  - Player 2 will pick values for the even-numbered variables $x_2, x_4, x_6, \ldots$.
- We say Player 1 **wins** iff at the end $\varphi(x_1, x_2, \ldots, x_{2n})$ is true.
- In order for Player 1 to have a **winning strategy**, he must have a way to win for all possible sequences of moves by Player 2.
- Equivalently, Player 1 has a winning strategy iff

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \cdots \forall x_{2n} \varphi(x_1, x_2, \ldots, x_{2n}),$$

  which is just saying that this quantified Boolean formula is true.
- Thus, deciding whether Player 1 has a winning strategy for a given board in the QBF game is PSPACE-complete.

Subsection 3

NL-Completeness

Introducing Logspace Reductions

- We consider problems that are complete for NL.
- To study whether or not NL = L, we cannot use polynomial-time reductions, since $\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P}$.
- The reduction should not be more powerful than the weaker class, which is L.
- We use instead logspace reductions, which, as the name implies, are computed by a deterministic TM running in logarithmic space.
- A subtle issue is that a logspace machine might not even have the memory to write down its output.
- As a solution, we require that the reduction should be able to compute any desired bit of the output in logarithmic space.
- I.e., the reduction f is implicitly computable in logarithmic space, in the sense that there is an $O(\log|x|)$-space machine that, on input $\langle x, i\rangle$, outputs $f(x)_i$, provided that $i \leq |f(x)|$.

Logspace Reductions and NL-Completeness

Definition (Logspace Reduction and NL-Completeness)

- A function $f : \{0,1\}^* \to \{0,1\}^*$ is **implicitly logspace computable** if:
  - f is polynomially bounded, i.e., there is some c, such that

    $$|f(x)| \leq |x|^c, \quad \text{for every } x \in \{0,1\}^*;$$

  - The languages

    $$\begin{aligned} L_f &= \{\langle x, i\rangle : f(x)_i = 1\}, \\ L'_f &= \{\langle x, i\rangle : i \leq |f(x)|\} \end{aligned}$$

    are in L.

Logspace Reductions and NL-Completeness (Cont'd)

Definition (Logspace Reduction and NL-Completeness Cont'd)

- A language B is **logspace reducible** to language C, denoted $B \leq_\ell C$, if there is a function $f : \{0,1\}^* \to \{0,1\}^*$, such that:
  - f is implicitly logspace computable;
  - For every $x \in \{0,1\}^*$,

$$x \in B \quad \text{iff} \quad f(x) \in C.$$

- We say that C is NL-**complete** if:
  - C is in NL;
  - For every $B \in \mathrm{NL}$, $B \leq_\ell C$.

- Another way to think of logspace reductions is to imagine that the reduction is given a separate "write-once" output tape, on which it can either write a bit or move to the right, but never move left or read the bits it wrote down previously.

Properties of Logspace Reductions

Lemma

1. If $B \leq_\ell C$ and $C \leq_\ell D$, then $B \leq_\ell D$.
2. If $B \leq_\ell C$ and $C \in \mathsf{L}$, then $B \in \mathsf{L}$.

- We show that if f, g are two functions that are logspace implicitly computable, then so is the function h where $h(x) = g(f(x))$.

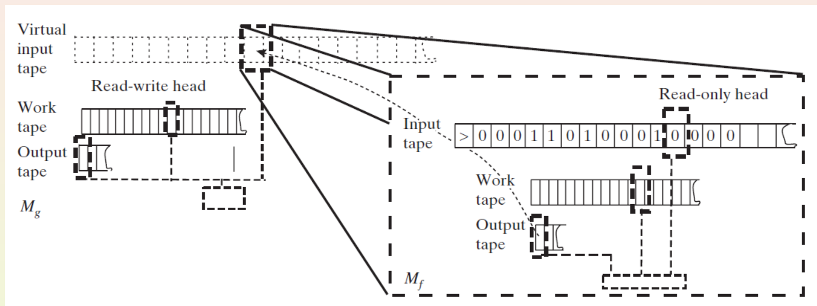
  Then the two parts are proved as follows.
  - For Part 1, let f be the reduction from B to C and g the reduction from C to D.
  - For Part 2, let f be the reduction from B to C and g be the characteristic function of C, i.e., $g(y) = 1$ iff $y \in C$.

  Let M_f be the logspace machine that computes $x, i \mapsto f(x)_i$;

  Let M_g be the logspace machine that computes $y, j \mapsto g(y)_j$.

The Machine M_h

- We construct a machine M_h that, given input x, j, with $j \leq |g(f(x))|$, outputs $g(f(x))_j$.



Machine M_h will pretend that:

- It has an additional (fictitious) input tape on which $f(x)$ is written;
- It is merely simulating M_g on this input.

The true input tape has x, j written on it.

The Machine M_h (Cont'd)

- M_h always maintains on its work tape the index, say i, of the cell on the fictitious tape that M_g is currently reading.

  This requires only $\log |f(x)|$ space.

  To compute for one step, M_g needs to know the contents of $f(x)_i$. At this point:
  - M_h temporarily suspends its simulation of M_g (copying the contents of M_g's work tape to a safe place on its own work tape);
  - Invokes M_f on inputs x, i to get $f(x)_i$.

  Then it resumes its simulation of M_g using this bit.

  The total space M_h uses is

  $$\mathrm{O}\left(\log |g(f(x))| + s(|x|) + s'(|f(x)|)\right).$$

  Since $|f(x)| \leq \mathrm{poly}(x)$, this expression is $\mathrm{O}(\log |x|)$.
- By the lemma, an NL-complete language is in L iff NL $=$ L.

NL-Completeness of Directed Paths

- Recall the language

$$\textsc{Path} = \{\langle G, s, t\rangle : \text{vertex } t \text{ can be reached from } s \text{ in the directed graph } G\}.$$

Theorem

$\textsc{Path}$ is NL-complete.

- We have seen that PATH is in NL.

  Let L be any language in NL.

  Let M be a machine that decides L in space $\mathrm{O}(\log n)$.

  We describe a logspace implicitly computable function f that reduces L to $\textsc{Path}$.

NL-Completeness of PATH (Cont'd)

- Let X be an input of size n.

  $f(x)$ will be the configuration graph $G_{M,x}$.

  Its nodes are all possible $2^{O(\log n)}$ configurations of the machine on input x, along with the start configuration C_{start} and the accepting configuration C_{accept}.

  In this graph there is a path from C_{start} to C_{accept} iff M accepts x.

  We have

  $$f(x) = \langle G_{M,x}, C_{\text{start}}, C_{\text{accept}} \rangle.$$

  The graph $G_{M,x}$ is represented as usual by an adjacency matrix.

  It contains 1 in the $\langle C, C' \rangle$th position iff there is an edge from C to C' in $G_{M,x}$.

NL-Completeness of PATH (Cont'd)

- We show that this matrix can be computed by a logspace reduction.

  To this end, we describe a logspace machine that can compute any desired bit in it.

  Suppose the input is $\langle C, C' \rangle$.

  A deterministic machine can examine C, C' and check whether C' is one of the at most two configurations that can follow C according to the transition function of M.

  The space requirement is

$$O\left(|C| + |C'|\right) = O\left(\log|x|\right).$$

Read-Once Certificates

- We provide an alternative definition of the class NL using **certificates** instead of nondeterministic TMs.
- A subtle issue is that, since a certificate may be polynomially long, a logspace machine may not have the space to store it.
- Thus, the certificate-based definition of NL assumes that the certificate is provided to the logspace machine on a separate tape that is "**read once**".
- This means that the machine's head on the tape can only sweep the tape from left to right once, and, thus, never read the same bit of the certificate twice.
- At each step, the machine's head on that tape can either stay in place or move to the right.
- Read-once access to bits in a certificate is just an alternative way to view nondeterministic choices.

Certificate Definition of NL

Definition (Alternative Definition of NL)

A language L is in NL if there exists a deterministic TM M (called the **verifier**), with an additional special read-once input tape, and a polynomial $p : \mathbb{N} \to \mathbb{N}$, such that:

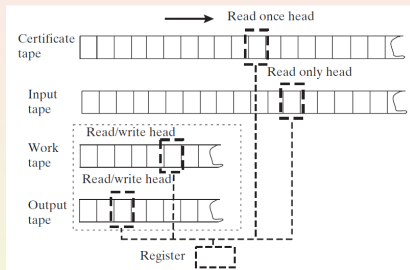
- For every $x \in \{0,1\}^*$,

$$x \in L \quad \text{iff} \quad (\exists u \in \{0,1\}^{p(|x|)})(M(x,u) = 1),$$

  where $M(x,u)$ denotes the output of M, where x is placed on its input tape and u is placed on its special read-once tape;
- M uses at most $O(\log |x|)$ space on its read-write tapes, for every input x.

Certificate Definition of NL (Illustration)

- Below we give the certificate view of NL.



- Suppose, in the above scenario, we remove the read-once restriction and allow the TM's head to move back and forth on the certificate, and read each bit multiple times.
- Then the class changes from NL to NP.

The Complement of PATH and coNL

- We define coNL as the set of languages that are complements of NL languages.
- A simple example of a coNL language is $\overline{\text{PATH}}$, the complement of the PATH language.
- A decision procedure for $\overline{\text{PATH}}$ must accept the tuple $\langle G, s, t \rangle$ when there is no path from s to t in the graph.
- It is easy see that $\overline{\text{PATH}}$ is not only in coNL, but is in fact coNL-complete.
- That is, every coNL language is logspace reducible to $\overline{\text{PATH}}$.
- Unlike in the case of PATH, there is no natural certificate for the nonexistence of a path from s to t.
- Thus, researchers believed that $\overline{\text{PATH}} \notin \text{NL}$.
- The Immerman-Szelepcsényi Theorem showed the opposite holds.

The Immerman-Szelepcsényi Theorem

Theorem (Immerman-Szelepcsényi Theorem)

$\overline{\text{PATH}} \in \mathsf{NL}$.

- It suffices to exhibit a $O(\log n)$-space verification algorithm A, such that, for every n-vertex graph G and vertices s and t, there exists a polynomial certificate u, such that

$$A(\langle G, s, t \rangle, u) = 1 \quad \text{iff} \quad t \text{ is not reachable from } s \text{ in } G,$$

  where A has only read-once access to u.

  For simplicity, we identify G's vertices with $\{1, \ldots, n\}$.

  Let C_i be the set of vertices that are reachable from s in G within at most i steps.

The Immerman-Szelepcsényi Theorem (Cont'd)

- Membership in C_i is easily certified.

  For every $i \in [n]$ and vertex v, the sequence of vertices $v_0, v_1, \ldots, v_k$ along the path from s to v, where $k \leq i$, is a certificate.

  Its size is at most polynomial in n.

  The algorithm can check the certificate using read-once access by verifying that:

  (1) $v_0 = s$;
  (2) For $j > 0$, there is an edge from v_{j-1} to v_j;
  (3) $v_k = v$;
  (4) Using simple counting, the path ends within at most i steps.

  Next, we use C_i-membership certifiability to design two more types of certificates.

Process of Certifying that $t \notin C_n$

- We design the following certificates:
  1. A certificate that a vertex v is not in C_i, assuming the verifier has already been told (i.e., convinced about) the size of C_i.
  2. A certificate that $|C_i| = c$, for some number c, assuming the algorithm has already been convinced about the size of C_{i-1}.

  Note that $C_0 = \{s\}$ (and the verifier knows this).

  So we can provide the second kind of certificate to the verifier iteratively to convince it of the sizes of the sets $C_1, \ldots, C_n$.

  Now C_n is just the set of all vertices reachable from s.

  The verifier has been convinced of $|C_n|$.

  So we can use the first kind of certificate to convince the verifier $t \notin C_n$.

Certifying that v is not in C_i, given $|C_i|$

1. Certifying that v is not in C_i, given $|C_i|$:

   The certificate is simply the list of certificates to the effect that u is in C_i, for every $u \in C_i$, sorted in ascending order of vertices.

   The verifier checks that:

   (1) Each certificate is valid;
   (2) The vertex u for which a certificate is given is indeed larger than the previous vertex;
   (3) No certificate is provided for v;
   (4) The total number of certificates provided is exactly $|C_i|$.

- If $v \notin C_i$, then the verifier will accept the above certificate;
- If $v \in C_i$ there will not exist $|C_i|$ certificates that vertices $u_1 < u_2 < \cdots < u_{|C_i|}$ are in C_i, where $u_j \neq v$, for every j.

Certifying that v is not in C_i, given $|C_{i-1}|$

- Certifying that v is no in C_i, given $|C_{i-1}|$:

  The certificate is the list of $|C_{i-1}|$ certificates to the effect that $u \in C_{i-1}$, for every $u \in C_{i-1}$, in ascending order.

  The algorithm checks everything as before except that in Step (3) it verifies that no certificate is given for v or for a neighbor of v.

  Now $v \in C_i$ if and only if, there exists $u \in C_{i-1}$, such that one of the following holds:

  - $u = v$;
  - u is a neighbor of v in G.

  So the procedure will not accept a false certificate by the same reasons as above.

Certifying that $|C_i| = c$, given $|C_{i-1}|$

2. Certifying that $|C_i| = c$, given $|C_{i-1}|$:

   We have described how to give, for any vertex v, certificates to the effect that $v \in C_i$ or $v \notin C_i$ (whichever is true).

   The certificate that $|C_i| = c$ will consist of n certificates for each of the vertices 1 to n in ascending order.

   For every vertex u, there will be an appropriate certificate depending on whether $u \in C_i$ or not.

   The verifier will verify all the certificates and count the vertices that have been certified to be in C_i.

   If this count is equal to c, the verifier accepts.

NPSPACE and coNPSPACE

- Using the notion of the configuration graph, the proof of the Immerman-Szelepcsényi Theorem may be modified to show

Corollary

For every space constructible $S(n) \geq \log n$,

$$\text{NSPACE}(S(n)) = \text{coNSPACE}(S(n)).$$

Space-Bounded and Time-Bounded Complexity Classes

- Based on our understanding of the relations between the various space-bounded and time-bounded complexity classes, we get

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

- The hierarchy theorems imply that

$$\mathsf{L} \subsetneq \mathsf{PSPACE} \quad \text{and} \quad \mathsf{P} \subsetneq \mathsf{EXP}.$$

- So we know that at least some of these inclusions are strict.
- However, we do not know which ones.
- Most researchers believe that all of these inclusions are strict.