Indian Association for the Cultivation of Science
(Deemed to be University under *de novo* Category)

Master's/Integrated Master's-PhD Program/ Integrated Bachelor's-Master's Program/PhD Course

Theory of Computation II: COM 5108

Lecture III

Instructor: Goutam Biswas *Autumn Semester 2023*

1 Space Complexity

We shall consider *workspace-bounded computation* of Turing machines. In this case the model is slightly different. The input tape is *read-only* and the space is measured in terms of the space used by the work-tape.

1.1 PSPACE, NPSPACE, L and NL

Definition 1. A language $L \subseteq \Sigma^*$ is in **DSPACE**$(f(n))$, if there is a *proper* function $s : \mathbb{N}_0 \to \mathbb{N}_0$ so that there is a DTM M, that decides L, using at most $cf(n)$ $(O(f(n)))$ cells of the work tape for all but finite number of input $x \in \Sigma^*$ (Σ is often $\{0, 1\}$), where $n = |x|$ and c is a constant.

Similarly **NSPACE**$(f(n))$ is defined when the Turing machine is nondeterministic.

The space of the read-only input tape is not included in the space usage. So it makes sense to talk about *sub-linear* of space $(f(n) < n)$ bounded language classes. But $f(n)$ should be greater than $\log n$.

The *configuration* of a machine with *read-only* input tape consists of (i) position of the head on the input tape $(n = |x|)$, all possible configurations of the work tape using $f(n)$ tape cells $(c^{f(n)}, c = |\Gamma|)$, head positions on the work-tape $(f(n))$, state of the machine $(q = |Q|)$. The total number of configurations is

$$qnf(n)c^{f(n)} = 2^{\log_2 q} \times 2^{\log_2 n} \times 2^{\log_2 f(n)} \times 2^{f(n)\log_2 c} = 2^{(q+\log_2 n+\log_2 f(n)+f(n)\log_2 c)} = 2^{O(f(n)},$$

if $f(n) \geq \log n$.

Definition 2. A function $f : \mathbb{N}_0 \to \mathbb{N}_0$ is *proper* if it is *space constructible* i.e. $f(n)$ is at least $O(\log n)$, and there is a DTM that computes $f(|x|)$ using $O(f(|x|)$ space, on input x.

We already know the following relations among the time and space complexity classes. For any *space* and *time-constructible* function $f : \mathbb{N}_0 \to \mathbb{N}_0$,

$$\text{DTIME}(f(n)) \subseteq \text{DSPACE}(f(n)) \subseteq \text{NSPACE}(f(n)) \subseteq \text{DTIME}(2^{O(f(n))}).$$

For polynomial functions we further know that

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} = \mathbf{NPSPACE} \subseteq \mathbf{EXP}.$$

Definition 3. We define two classes of languages using sublinear space where $f(n) = \log n$.

$$\mathbf{L} = \text{DSPACE}(\log n),\ \mathbf{NL} = \text{NSPACE}(\log n).$$

Example 1. All regular languages are in DSPACE(O(1)). As it is not necessary to use any work-tape space for its recognition.
Example 2.

$$L_1 = \{0^n 1^n :\ n \in \mathbb{N}_0\}.$$

Our DTM M has a *read-only* input tape and a *read-write* work tape. It works as follows:
M: input x

1. Scan the input to see that there is no 0 after 1. If there is, *reject*.
2. Maintain a counter on the *work-tape* and count the number of 0's as a binary numeral.
3. Decrement the counter with the number of 1's.
4. If the count is non-zero and all 1's are counted, or the count is zero but a few more 1's are left, then *reject*, otherwise *accept*.

The counter will take $O(\log n)$ work space, so $L_1 \in \mathbf{L}$.
Exercise 1. $L_2 = \{0^n 1^n 2^n :\ n \in \mathbb{N}\}$ is in $\mathbf{L}$.
Example 3.

$$\text{MULT} = \{< a, b, a \times b >:\ a, b \in \mathbb{N}_0\} \in \mathbf{L}.$$

The algorithm depends on the fact that the i^{th} bit of the result of n by n bit multiplication can be calculated as follows.

$$\left(c + \sum_{j=max(0,i+1-n)}^{min(i,n-1)} a_j b_{i-j} \right) \mod 2,$$

Example 4.

9	8	7	6	5	4	3	2	1	0 ($Bits$)
					1	0	1	1	0 (a)
				×	1	1	0	1	1 (b)
				+	1	**0**	1	1	0
			+	1	0	**1**	1	0	
		+	0	0	0	**0**	0		
	+	1	0	1	1	**0**			
+	1	0	1	1	0				
1	0	0	1	0	1	**0**	0	1	0 (result)
0	1	1	1	10	1	**1**	1	0	0 (carry)

Let $n = 5$, $i = 0, 1, \cdots, 9$. Let us consider the bit-3 and bit-5 of the result.i

r_3: $i = 3$, the lower value of $j = \max\{0, i+1-n\} = \max\{0,-1\} = 0$ to $j = \min\{i, n-1\} = \min\{3, 5-1\} = 3$.
So the result bit is

$$\left(c + \sum_{j=0}^{3} a_j b_{i-j}) \mod 2 = (1 + a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0) \bmod 2 = (1+0+0+1+0) \right) \bmod 2 = 0.$$

r_5: $i = 5$, the lower value of $j = \max\{0, i+1-n\} = \max\{0,1\} = 1$ to $j = \min\{i, n-1\} = \min\{5, 5-1\} = 4$.
So the result bit is

$$\left(c + \sum_{j=1}^{4} a_j b_{i-j}) \mod 2 = (1 + a_1b_4 + a_2b_3 + a_3b_2 + a_4b_1) \bmod 2 = (1+1+1+0+1) \right) \bmod 2 = 0.$$

where c is the carry from the previous stage. The value $i = 0, \cdots, 2n-1$, where both a and b are n bit numbers. The next stage carry is generated as

$$\lfloor (c + \sum_{j=max(0,i+1-n)}^{min(i,n-1)} a_j b_{i-j})/2 \rfloor.$$

The number of bits to be stored in work tape for computation is $O(\log n)$. So MULT $\in$ **L**.
Example 5.

$$\text{PATH} = \{< G, s, d >: \ G \text{ is a directed graph with a path from } s \text{ to } d\}.$$

Let $|V(G)| = k$. The nodes can be numbered with $\lceil \log_2 k \rceil$ bits and the value of k can be stored in $\lceil \log_2 k \rceil$ bits. An NTM N starting from the start node s nondeterministically chooses the next nodes on the path from s to d, if there is one. The next node number created by N is checked with d. The path length cannot be more than $k-1$. So the computation stops after generation of at most $k-1$ nodes.

It is not necessary to store the complete path (that requires $O(k \log k)$ space). The work tape contains total node count k, current node number and the next node number. This can be computed in $O(\log n)$ work space by a nondeterministic Turing machine. So PATH $\in$ **NL** .

It is unknown whether PATH $\in$ **L**. But we shall prove that PATH is **NL** complete.

1.2 PSPACE

It is known that **P** $\subseteq$ **NP** $\subseteq$ **PSPACE**. But no stronger result is known e.g. whether **P** or **NP** are proper subsets. People believe that they are, but no proof.

Definition 4. A language L is **PSPACE**-*hard* if for every $L' \in$ **PSPACE**, L' is polynomial time Karp reducible to L, $L' \leq_P L$. A language L is **PSPACE**-*complete* if it is **PSPACE**-*hard* and also belongs to **PSPACE**.

The following language is **PSPACE**-*complete*.

$$L_{\mathbf{PSPACE}} = \{< M, x, 1^n >: \ M \text{ accepts } x \text{ in space } n\}$$

Any language $L \in$ **PSPACE** is Karp reducible to $L_{\mathbf{PSPACE}}$. Let M be a deterministic Turing machine decides L in polynomial space $p(n)$, where n is the length of the input. The reduction function maps $x \mapsto < M, x, 1^{p(|x|)} >$, $\forall x \in \{0,1\}^*$.

Compare it with the **NP**-*complete* language

$$L_{\mathbf{NP}} = \{< M, x, 1^n, 1^t >: \exists w \in \{0,1\}^n \text{ such that the TM } M \text{ accepts } < x, w > \text{ in } t \text{ steps}\}.$$

We are now going to look for a more useful language known to be **PSPACE**-complete.

Definition 5. A *quantified Boolean formula (QBF)* is defined inductively as follows:

1. Boolean constants *true* (1) and *false* (0) are QBFs.
2. Boolean variables $x_1, x_2, \cdots$ are QBF.
3. If f_1 and f_2 are Boolean formulas and x_i is a Boolean variables, then $(f_1 \vee f_2)$, $(f_1 \wedge f_2)$, $\neg f_1$, $\exists x_i f_1$ and $\forall x_i f_1$ are QBFs.

We use appropriate associativity and precedence conventions to avoid some of the parentheses. The scope of a quantifier is as usual. In general a QBF looks as follows:

$$Q_1 x_1 \cdots Q_n x_n \phi(x_1, \cdots, x_n),\ n \geq 0,$$

where Q_i is either an *existential* ($\exists$) or a *universal* ($\forall$) quantifier and x_i's are boolean variables that take values over { *true*, *false*}.
A QBF is called *closed* if all variables are quantified. A *closed* QBF can be evaluated to *true* (1) or *false* (0). A variable is said to be *free* if it is not quantified. An QBF with k free variables is a map from $\{0,1\}^k \to \{0,1\}$.
Example 6. The *closed* QBF $\exists x_1 \forall x_2 (x_1 \vee \overline{x_2})$ is *true* as $x_1 = 1$ makes the formula always *true*. But the *closed* formula $\exists x_1 \forall x_2 (x_1 \wedge \overline{x_2})$ is *false*.
The *open* formula $\forall x_2 (x_1 \vee \overline{x_2})$, where x_1 is the *free* variable, is a map from $\{0,1\} \to \{0,1\}$, $\forall x_2 (0 \vee \overline{x_2}) \mapsto 0$, $\forall x_2 (1 \vee \overline{x_2}) \mapsto 1$.

A QBF is not restricted to *prenex normal form*. Quantifiers may appear within the body of the formula. But such formula can be convert to *prenex normal form* using the following equivalences: $\forall x \phi(x) \equiv \neg\neg \forall x \phi(x) \equiv \neg \exists x \neg \phi(x)$, $\psi \vee \exists x \phi(x) \equiv \exists x (\psi \vee \phi(x))$, $\psi \wedge \forall x \phi(x) \equiv \forall x (\psi \wedge \phi(x))$, where ψ does not have any free x. If ψ has a free variable x, then we can rename the bound variable in $\forall x \phi(x)$.
Example 7. The question of *satisfiability* of $\phi(x_1, \cdots, x_n)$ is equivalent to the question of truth value of the QBF $\exists x_1 \cdots \exists x_n \phi(x_1, \cdots, x_n)$. The formula $\phi(x_1, \cdots, x_n)$ is SAT if and only if $\exists x_1 \cdots \exists x_n \phi(x_1, \cdots, x_n)$ is *true*.

Similarly the question of validity of $\phi(x_1, \cdots, x_n)$ is equivalent to the truth value of $\forall x_1 \cdots \forall x_n \phi(x_1, \cdots, x_n)$. The formula $\phi(x_1, \cdots, x_n)$ is a tautology (in TAUT) if and only if $\forall x_1 \cdots \forall x_n \phi(x_1, \cdots, x_n)$ is true.
The language TQBF is defined as follows:

$$\text{TQBF} = \{\psi :\ \psi \text{ is a } \textit{closed} \text{ and } \textit{true} \text{ QBF}\}.$$

Theorem 1. The language TQBF is **PSPACE**-*complete*.
Proof: First we prove that TQBF is in **PSPACE**. Let the size of the formula ψ be m and there are n variables.

If $(n = 0)$, the formula contains Boolean constants, and it can be evaluated in $O(m)$ time and space.

Let $s(n, m)$ be the space required to evaluate a formula of n variables and of length m.

If we initialize the first variable x_1 with 0, we get a new formula $\psi[x_1 \leftarrow 0]$ of $(n-1)$ variables. Similarly, if we initialize x_1 with 1, we get another formula $\psi[x_1 \leftarrow 1]$ of $(n-1)$ variables.

So we have the following recursive procedure to evaluate the truth value of a QBF.

eval($Q_i x_i \cdots Q_n x_n \phi(v_1, \cdots, v_{i-1}, x_i, \cdots, x_n)$, i)
  if $i = n$ evaluate the constant Boolean expression.
    $b_1 =$ eval$(Q_{i+1} \cdots Q_n x_n \phi(v_1, \cdots, v_{i-1}, 0, x_{i+1}, \cdots, x_n), i+1)$,
    $b_2 =$ eval$(Q_{i+1} \cdots Q_n x_n \phi(v_1, \cdots, v_{i-1}, 1, x_{i+1}, \cdots, x_n), i+1)$,
  if $Q_i = \exists$ *then return* $b_1 \vee b_2$,
  if $Q_i = \forall$ *then return* $b_1 \wedge b_2$.

The recursive procedure reuses the space after the first recursive call. So the recurrence relation for space requirement is

$$s(n, m) = s(n-1, m) + O(m)$$

The depth of recursion is the number of variables n. Solving the recurrence we get, $s(n, m) = \overbrace{O(m) + \cdots + O(m)}^{n} = O(mn)$. So the space requirement is a polynomial of the size of the formula.

Now we show that TQBF is **PSPACE**-*hard*, every language $L \in$ **PSPACE** is polynomial time reducible to TQBF. Let the language L be decided by a Turing machine M, space-bounded by the polynomial n^k for some constant k. There is a polynomial time reduction from L to TQBF. An input x is mapped to a QBF ϕ so that ϕ is *true* (in TQBF) if and only if M accepts x.

Unfortunately in this case a construction like Cook-Levin does not work. If the input length is n and the polynomial space n^k is used for computation, the number of configurations are $2^{O(n^k)}$. A Cook-Levin style formula representing this computation is too long to generate in polynomial time.

The solution uses a technique similar to the proof of Savitch's theorem. It reuses the space. The computation is divided into half and the same formula is used with different sets of parameters using universal quantifier. Following is an over simplified example:

$$\exists m(\phi(x, m) \wedge \phi(m, y))$$

can be coded as

$$\exists m \forall p \forall q(((p = x \wedge q = m) \vee (p = m \wedge q = y)) \wedge \phi(p, q))$$

In our case $\phi(x, y)$ is a long formula with large number of variables. Two instances of the formula is replaced by one formula and a sequence of universal quantifiers over the set of variables.

Equality of boolean variables can be expressed as a boolean formula $a = b$ is equivalent to $((\overline{a} \vee b) \wedge (a \vee \overline{b}))$. The universal quantifications will run over a

long list of variables corresponding to the cells, states and symbols of the TM M. But they are linear in space used.

Following is the construction Given c_1 and c_2, two sets of variables corresponding to two configurations and a number $t > 0$, we construct a QBF formula $\phi_{c_1,c_2,t}$ such that if we assign truth values of two actual configurations to c_1 and c_2, the formula $\phi_{c_1,c_2,t}$ is *true* if and only if M can go from c_1 to c_2 in at most t steps. The formula $\phi = \phi_{c_s,c_a,T}$, where c_s corresponds to the start configuration with x as the input ($|x| = n$), c_a to an *accepting* configuration, $T = 2^{dn^k}$ is the bound on the number of configurations of M on an input of length n. The formula $\phi = \phi_{c_s,c_a,T} \in$ TQBF if and only if M accepts x.

As in the case of Cook-Levin theorem, a formula encodes the content of cells of a configuration. Several variables are associated with each cell $c[i,j]$. The variable $x_{i,j,p}$ is *true* if $p \in \Sigma \cup Q$ is in $c[i,j]$. Each configuration has n^k cells. Boolean values of $l \times (n^k) = O(n^k)$ variables represent a configuration, where $l = |\Sigma \cup Q|$.

If $t = 1$, then either $c_1 = c_2$ or c_2 is reached in one step from c_1. The equality is expressed by writing a boolean expression. It says that each of the variables representing c_1 has the same boolean value as the corresponding variable representing c_2.
In the second situation, values of variables of each triple[1] of c_1's cells ($3l$ variables) should yield the values of variables of the corresponding triple of c_2's cells.

If $t > 1$ (we take t to be power of 2 for ease of presentation), $\phi_{c_1,c_2,t}$ is constructed recursively. The obvious solution that comes to mind is as follows. But it does not work.

$$\phi_{c_1,c_2,t} = \exists c'(\phi_{c_1,c',t/2} \wedge \phi_{c',c_2,t/2}).$$

The symbol c' represents a configuration. But the actual meaning of $\exists c'$ is the existence of $l = O(n^k)$ variables that encodes the configurationc' i.e. $\exists x'_1 \cdots \exists x'_l$.

The machine M can go from c_1 to c_2 in at most t steps, if some intermediate configuration c' exists such that M can go from c_1 to c' in at most $\frac{t}{2}$ steps and from c' to c_2 in at most $\frac{t}{2}$ steps.

But now we need to construct two formulas $\phi_{c_1,c',t/2}$ and $\phi_{c',c_2,t/2}$, and though the number of steps are cut to half $(t/2)$, the length of formula is doubled! Finally the length of the formula will be exponentially large as $2^{\log 2^{dn^k}} = 2^{O(n^k)}$.

We use '$\forall$' quantifier along with the '$\exists$' quantifier as follows to solve the problem.

$$\phi_{c_1,c_2,t} = \exists c' \forall (c_3, c_4) \in \{(c_1, c'), (c', c_2)\}(\phi_{c_3,c_4,t/2}).$$

By introducing new variables representing the configurations c_3 and c_4, allows us to fold the two parts of the original sub-formula $\phi_{c_1,c',t/2}$ and $\phi_{c',c_2,t/2}$ into a single formula.

The meaning of $\forall (c_3, c_4) \in \{(c_1, c'), (c', c_4)\}$ is that the variables of c_3 and c_4 can take the values of the variables of c_1 and c' or the values of the variables of c' and c_4 respectively. And in both the cases $\phi_{c_3,c_4,t/2}$ is true.

Next we calculate the size of the formula. $\phi_{c_s,c_a,T}$ where $T = 2^{df(n)}$. At each level of recursion the new variables and equality formula added is linear in

[1]The notion of triple window was presented in the proof of Cook-Levin Theorem.

the size of configurations The depth of recursion is $\log(2^{df(n)}) = O(f(n))$. So the size of the formula is $O(f^2(n))$. QED.

The proof of the theorem does not require that the machine M should be deterministic. It actually proves that TQBF is also a complete problem for NPSPACE. This implies that $PSPACE = NPSPACE$.

It is known that a problem in **NP** has a short certificate that can be verified in polynomial time. A quantified statement is viewed as a *two player* game with *perfect information*. Both players have access to what is known to the other. The game of chess and a configuration of the chess-board is an example of such game. The notion of **PSPACE**-*complete* problem is related to the existence of a *winning strategy* for player 1. It amounts to say that there is a first move for the player-1 such that for all moves of the player-2, there is a second move of the player1, $\cdots$, such that at the end the player-1 wins.

In case of a problem in **NP**, finding a witness requires a search through the space of certificates. But the length of a certificate is bounded by a polynomial. In case of a problem in **PSPACE**, the search is for a winning strategy on the game tree. An encoding of a winning strategy of player-1 may be exponentially long as it depends on all possible (at least 2) moves of player-2. .

1.3 QBF-Game

The "play ground" of a QBF game is a closed QBF $\exists x_1 \forall x_2 \cdots Q x_k \phi(x_1, x_2, \cdots, x_k)$ with k variables. $Q = \exists$ if k is odd and it is $\forall$ if k is even. Two players E and A, alternately make moves by picking values of the corresponding variables. The player E picks the values of $x_1, x_3, \cdots$, and the player A picks the values of $x_2, x_4, \cdots$.

At the end the values picked up by two players are used to evaluate $\phi(x_1, x_2, \cdots, x_k)$. The first player wins if $\phi(x_1, \cdots, x_{2n})$ is *true* for the chosen values of the Boolean variables. Similarly, the second player wins if it is *false*.

Example 8. Consider the formula

$$\exists x_1 \forall x_2 \exists x_3 (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2}) \wedge (x_2 \vee \overline{x_3}).$$

The player E has a winning strategy: $x_1 = 0, x_3 = \overline{x_2}$.
If we change the formula in the following way,

$$\exists x_1 \forall x_2 \exists x_3 (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2}) \wedge (\overline{x_2} \vee \overline{x_3}),$$

then the player A has a winning strategy with $x_2 = 1$.

If we have sequence of existential or universal quantifiers, e.g.

$$\exists x_1 x_2 \forall x_3 x_4 x_5 \cdots Q x_k \phi(x_1, \cdots, x_k).$$

The E player will choose values of x_1, x_2 and A player will choose values x_3, x_4, x_5 etc.

$QBF-GAME = \{< \phi >:$ player E has a winning strategy in the game of $\phi\}$.

This essentially means that QBF-GAME is **PSPACE**-*complete*.

1.4 L, NL and coNL

Here we talk about languages decided in sub-linear space. Important language classes are in **L** and **NL**. A natural question is why do we consider *logspace* when we consider sub-linear complexity classes. One reason is that logspace is just large enough to solve many interesting problems. An n bit input can be addressed using $\lceil \log n \rceil$ bits. It seems logspace is almost a necessity. These classes also have nice properties.

A *logspace* TM model is slightly different. It has a read-only input tape, read/write work-tape, and there may be a write-only output tape where bits of the computed function can be written sequentially. The number of cells used in the work-tape is considered as the measure space complexity.
Our earlier claim of $2^{O(f(n))}$ running time of a $f(n)$ space bounded machine is not valid if $f(n) < \log n$, as it takes n number of steps to read the input.
In general a configuration of an $f(n)$ space bounded machine has information about state, input head position, work-tape head position and the content of the work-tape. If there are g tape symbols, then the maximum number of configurations can be $snf(n)g^{f(n)}$, where s is the number of states. This is equal to $n2^{O(f(n))} = 2^{\log n + O(f(n))}$. Which is equal to $2^{O(f(n))}$ if $f(n) \geq \log n$.

The notion of complete problem in logspace is similar to our earlier concept. But a polynomial time Karp reduction is not useful in case of language class **P**, **L** or **NL**. The reduction cost obscures the resource bound of the complexity class. Every problem in **P**, **NL** or **L** are decidable in polynomial time. Every language $A \neq \emptyset$ and $\neq \Sigma^*$ is a *complete* problem under polynomial time reduction.
Example 9. Let $A \in \mathbf{NL}$ and neither $L = \emptyset$, nor $L = \Sigma^*$. So there is a string $x_i \in L$ and $x_o \notin L$. Let B be any language in **NL**. So B has a polynomial time decider (DTM) M. We define the mapping as follows: for all $x \in \Sigma^*$

$$x \mapsto \begin{cases} x_i & \text{if } M \text{ accepts } x, \\ x_o & \text{if } M \text{ does not accept } x. \end{cases}$$

This makes A an **NL**-*complete* language under Karp reduction.

The reduction function is restricted to *logspace* to make complete problems more meaningful in case of **P, L** and **NL**. But there is a problem of composition of two such functions. Following is a definition of a computable function in *logspace*.

Definition 6. A function $f : \mathbb{N}_0 \to \mathbb{N}_0$ is *logspace* computable, if the input is given on a *read-only* tape, and the output is obtained on a *write-only* output tape. The amount of space used on the read-write work-tape is $O(\log |x|)$, where x is the input.
This gives a natural bound on the size of the value of $f(x)$. The length of $f(x)$ is bounded by a polynomial i.e. $|f(x)| \leq |x|^c$, where c is a constant.

Let f and g be two functions computable in logspace. But to compute $g(f(x))$ we cannot store the intermediate value of $f(x)$, as its length may be polynomial in $|x|$. So to compute $g \circ f$ in logspace, it is necessary to compute bits of $f(x)$, $f(x)_i$, $1 \leq i \leq |f(x)|$ in logspace as and when required.

From the composition point of view, $f : \{0,1\}^* \to \{0,1\}^*$ is *logspace* computable if every bit of $f(x)$ can be computed using *logspace* i.e. the function is implicitly computable in *logspace*.

In terms of language this can be stated as follows. The language $L_f = \{< x, i >:\ f(x)_i = 1\}$ and the language $L'_f = \{< x, i >:\ 1 \leq i \leq |f(x)|\}$ are in **L**. In other words we have a logspace bounded Turing machine that computes $< x, i > \mapsto f(x)_i$, $1 \leq i \leq |f(x)|$.

Definition 7. A language A is *logspace reducible* to a language B, $A \leq_l B$, if there is a *logspace* computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$, $x \in A$ if and only if $f(x) \in B$.

We say that a language A is **NL**-*complete* if it is in **NL** and all languages in **NL** are *logspace reducible* to A. If an **NL**-*complete* problem is in **L**, then **L** = **NL**.

Following proposition shows that *logspace computable* functions are closed under composition and have expected properties of reducibility.

Proposition 1. Let $A, B, C \subseteq \{0,1\}^*$.

1. If $A \leq_l B$ and $B \leq_l C$, then $A \leq_l C$.
2. If $A \leq_l B$ and $B \in \mathbf{L}$, then $A \in \mathbf{L}$.

Proof: We first prove that if $f, g : \{0,1\}^* \to \{0,1\}^*$ are implicitly computable in logspace, then so is $h = g \circ f$.

Let M_f and M_g be the logspace machines computing $< x, i > \mapsto f(x)_i$ and $< y, j > \mapsto g(y)_j$ respectively, where $1 \leq i \leq |f(x)|$ and $1 \leq j \leq |g(x)|$. We construct the machine M_h that computes $< x, k > \mapsto g(f(x))_k$ in logspace, where $1 \leq k \leq |g(f(x))|$.

M_h is computing the k^{th} bit of $g(f(x))$ by simulating M_g on $f(x)$. We assume that M_g wants to work on the i^{th} bit of $f(x)$. M_h saves i on its work-tape, that requires $\log |f(x)|$ space.

But then $f(x)_i$ is not actually available, and is to be computed using M_f. So, M_h suspends M_g, saves the configuration of M_g on the work tape. It requires $O(\log |f(x)|)$ space (M_g is a logspace machine and its potential input is $f(x)$). M_h simulates M_f on the input $< x, i >$, where x is available from the actual input and i is available from the work tape of M_h. Running of M_f requires $O(\log |x|)$ space. Once the bit $f(x)_i$ is obtained, the computation of M_f is discarded and M_g is simulated for one more step on $f(x)_i$.

The total space used by M_h is (i) space for the index i of $f(x)$, (ii) space for the computation of M_f on $< x, i >$, (iii) space to save the configuration of M_g on $< f(x), j >$ etc. The lengths of $f(x)$ and $g(f(x))$ are polynomial bounded. So the total space is (i) $O(\log |f(x)|)$, (ii) $O(\log |x|)$, (iii) $O(\log |f(x)| + \log |g(f(x))|)$ which is equal to $O(\log |x|)$ as f, g are bounded by polynomial.

Once it is known that the composition of implicitly logspace computable functions are possible, the proof of the first part is simple. The logspace computable function f reduces A to B and the logspace computable function g reduces B to C. So, the logspace computable function $h = g \circ f$ reduces A to C.

For the second part we observe that B is in **L** implies that the characteristic function χ_B of B is logspace computable. So, $\chi_A = \chi_B \circ f$, the characteristic function of A is also logspace computable. QED.

We have already argued that PATH is in **NL**. Now we prove that it is **NL**-*hard*, and so is **NL**-*complete*.

Proposition 2. PATH is **NL**-*hard*.

Proof: Let the NTM N decides a language A in space $O(\log n)$. We need an *implicitly logspace computable function* f that reduces A to PATH.

Let $x \in \{0,1\}^*$ and $|x| = n$. Let $f(x)$ be the configuration graph $G_{N,x}$ which has $2^{O(\log n)}$ nodes, along with the start configuration C_{start} and the accept configuration C_{accept} i.e.

$$x \mapsto < G_{N,x}, C_{start}, C_{accept} > .$$

$x \in A$ if and only if there is a path from C_{start} to C_{accept} in the graph $G_{N,x}$.

A configurations of N can be stored in $O(\log n)$ space. The reduction machine will essentially produce a list of configurations, the set of vertices of $G_{N,x}$ and a list of edges (C_i, C_j) of $G_{N,x}$, where C_i, C_j are vertices of $G_{N,x}$ and $C_i \vdash_M C_j$. Each configuration is of length $k \log n$, where $n = |x|$ and k is a constant. The reduction machine sequentially generates a strings of length $k \log n$, tests whether the generated string encodes a valid configuration of N, and output it. It also generates all pairs of configurations C_i and C_j, checks whether C_j is reachable from C_i in one step of N. If yes, it outputs the pair (C_i, C_j). This can also be done in $O(\log n)$ space. QED.

Corollary 2. $\mathbf{NL} \subseteq \mathbf{P}$.

Proof: Let $A \in \mathbf{NL}$. So $A \leq_l PATH$ takes $2^{O(\log n)} = O(n^k)$ steps. It is also known that $PATH \in \mathbf{P}$. So A can be decided in polynomial time. QED.

The class **NP** has a *verifier* based definition. A similar definition is there for **NL** where the certificate is of polynomial length but each bit of it can be read only once.

Definition 8. A language A is in **NL**, if there is a deterministic logspace bounded verifier V with a special *read once* certificate tape and a polynomial $p : \mathbb{N}_0 \to \mathbb{N}_0$, such that for all $x \in \{0,1\}^*$, $x \in L$ if and only if $\exists w \in \{0,1\}^{p(|x|)}$ s.t. V accepts $< x, w >$ in logspace $(O(\log |x|))$, where x is on the read-only input tape and w is on the *read once* certificate tape.

We argue that the definitions of **NL** using a nondeterministic logspace machine is equivalent to the logspace deterministic verifier based definition, where the verifier uses a polynomial length read only once certificate.

If A is decided by an **NL** machine N, then the number of nondeterministic choices is bounded by some polynomial. The sequences of these choices can be used as a certificate for the deterministic logspace verifier.

If there is a logspace verifier V for A that uses a polynomial length read-only-once certificate, then we have the following **NL** machine:

N: input x

1. Non-deterministically guess a witness bit and simulate the logspace verifier.

2. A log-space verifier cannot remember the entire history of certificate bits as it is of polynomial length. In other words, the verifier reads the bit only once.

The class **coNL** is defined in a natural way. If $A \in \mathbf{NL}$, then $\overline{L} \in \mathbf{coNL}$. So

$$\overline{PATH} = \{< G, s, d >: \text{ there is no path from } s \text{ to } d \text{ in } G\}.$$

is in **coNL**. In fact it is **coNL** complete. The reduction function that reduces a language $A \in \mathbf{L}$ to PATH will also reduce $\overline{A} \in \mathbf{coNL}$ to $\overline{PATH}$.

An interesting result ([NI], [RS]) is that the class **NL** = **coNL**.

$\overline{PATH} \in \mathbf{NL} \Rightarrow \overline{PATH} \leq_l PATH$. $\overline{PATH} \in \mathbf{NL} \Rightarrow PATH \in \mathbf{coNL}$. For all $\overline{A} \in \mathbf{coNL} \overline{A} \leq_l \overline{PATH} \leq_l PATH$. So by composition of reduction, all $\overline{A} \in \mathbf{coNL}$ is in **NL**. So all $A \in \mathbf{NL}$ is also in **coNL**.

The language PATH can be decided by a nondeterministic Turing machine in $\log n$ space. It was shown by Immerman, Szelepcsényi that the number of reachable (unreachable = $m-$ reachable) nodes from a given node s in a graph G can be counted by a nondeterministic Turing machine using $\log n$ space.

The algorithm for counting the number of reachable nodes from a given node s is as follows.

- Let $m = |V(G)|$ and s be the start node.
- The set $A_i \subseteq V(G)$, $i = 0, \cdots, m-1$ is the collection of all nodes that are at a distance $\leq i$ from s. The basis is $A_0 = \{s\}$, $A_i \subseteq A_{i+1}$, $0 \leq i < m$, and A_{m-1} is total collection of reachable nodes of $V(G)$ from s.
- Let the size of A_i be c_i. We have $c_0 = 1$ and c_{m-1} is the total number of reachable nodes of $V(G)$ from s.

We calculate c_{i+1} from c_i starting from c_0. The elements of A_i are computed, but they cannot be stored as that will exceed logspace limit.

We compute and store c_{i+1}. At the beginning of the i^{th} iteration the value of c_i is known and it occupies logspace. At the end of iteration c_{i+1} is available and c_i is discarded.

- Following computation takes place for $i = 0, \cdots, m-2$.
- The algorithm goes through every $v \in V_G \setminus \{s\}$, and checks whether $v \in A_{i+1}$. The value of c_{i+1} is accumulated.
- To test whether $v \in A_{i+1}$, an inner loop goes through every $u \in V_G$. There is a non-deterministic guess whether $u \in A_i$. If that be the case and there is an edge (u, v), then $v \in A_{i+1}$.
- If u is guessed to be in A_i, the non-deterministic algorithm for PATH is used to enumerate a path of length $\leq i$. If the last node is not u, the branch of computation '**No**'-halt. If the last node is u and $(u, v) \in E_G$, then $v \in A_{i+1}$ and c_{i+1} is incremented.
- A local counter lc is maintained to keep track of the count of elements of A_i. At the end, if $c_i \neq lc$, the branch of computation '**No**'-halt. Note that c_i is already known.

The question is why does the following simple minded non-deterministic machine will not work, where N_{PATH} is a non-deterministic logspace machine that decides $PATH$. Does it have something to do with function computation by a nondeterministic machine or space?

C: Input $< G, s >$

1. $c \leftarrow 1$
2. **for** every $u \in V(G) \setminus \{s\}$ **do**
3. if N_{PATH} *accepts* $< G, s, u >$, then $c \leftarrow c + 1$
   // N_{PATH} gets G and s from the input tape of C and u from its work tape.
4. **return** c.

Theorem 3. (Immerman, Szelepcsényi) $\overline{PATH} \in \mathbf{NL}$.
Proof: Let G be a graph with m nodes. We know how to calculate the number of nodes reachable from s in G. Let it be equal to c. We wish to design a non-deterministic logspace bounded Turing machine M that given the input $< G, s, d >$ and c will *accept*, if there is no path from s to d.

- For every node of $u \in V(G)$, the machine M nondeterministically guesses whether u is reachable from s. If the guess is that u is 'reachable', it does the following:
- If $u = d$, the computation branch of the non-deterministic machine comes to *reject halt*.
- Otherwise, a path from s to u is guessed using the non-deterministic algorithm for PATH of length at most $m - 1$.
- If there is a verification error, the computation branch comes to *reject halt*.
- Otherwise the computation continues with the next u.
- A local counter of reachable nodes is maintained. At the end if the count is less than the known count c, the branch comes to *reject halt*.
- Otherwise it is accepted.

We put two parts of the algorithm together.

M : Input $< G, s, d >$

1. $c_0 = 1$ /* $A_0 = \{s\}$ - counting of reachable nodes start */
2. **for** $i = 0$ **to** $m - 2$ /* Compute c_{i+1} from c_i
3. $c_{i+1} = 1$ /* $s \in A_{i+1}$ */
4. For each $v \in V(G) \setminus \{s\}$
5. $lc = 0$ /* recounting $|A_i|$ */
6. For each $u \in V(G)$
7. Nondeterministically guess whether u is reachable and perform 8-10 or skip
8. Nondeterministically guess a path from s of length $\leq i$
9. **if** the last node is u, **then** $lc = lc + 1$ /* size of $|A_i|$ incremented */
   else *reject*
10. **if** $(u, v) \in V(G)$, **then** { $c_{i+1} = c_{i+1} + 1$ and **goto** 5 }
    /* size of A_{i+1} is incremented */

11. **if** $lc \neq c_i$, *reject* - /* wrong guess for the elements of A_{i-1} */
/* counting of reachable nodes ends */
12. $lc = 0$ /* counting of A_{m-1} starts */
13. For each $u \in V(G)$
14. Nondeterministically whether u is reachable and perform 15-17 or skip
15. Nondeterministically guess a path of length $\leq m$ from s
if the last node of the path is not u, *reject*.
16 **if** $u = d$, **then** *reject*
16 $lc = lc + 1$
17. **if** $lc \neq c_{m-1}$, **then** *reject*,
18. **else** *accept*

The algorithm needs to store $m, u, v, c_i, c_{i+1}, d, i$, and s. So the only logspace is required. QED.

This shows that

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

From the *hierarchy theorem* it is known (we have not yet proved) that **L** is a proper subset of **PSPACE** and **P** is a proper subset of **EXP**. It is also known that **NL** is a proper subset of **PSPACE** (we have not proved yet). So, either **NL** is a proper subset of **P** or **P** is a proper subset of **PSPACE**. But nothing is known.

Proposition 3. 2SAT is in **NL**.

Proof: If we prove that $\overline{2SAT} \in \mathbf{NL}$, then $2SAT \in \mathbf{coNL}$. But then $\mathbf{NL} = \mathbf{coNL}$, so $2SAT \in \mathbf{NL}$.
If a 2CNF formula is unsatisfiable, then a nondeterministic logspace machine guesses a variable x and path from x to $\neg x$ and back. QED.

Proposition 4. 2SAT is **NL**-*complete*.

Proof: We reduce $\overline{PATH}$ to 2SAT in logspace. Given a graph $< G, s, d >$ the 2SAT formula ϕ is designed as follows:

1. For every vertex $v \in V(G)$, there is a variable x_v. This variable is true if there is a path from s to v in G.
2. For every edge $(u, v) \in F(G)$, a clause $(\overline{x_u} \vee x_v)$ is there in ϕ. Note that $(\overline{x_u} \vee x_v)$ is logically equivalent to $(x_u \Rightarrow x_v)$.
3. There are two more clauses, (x_s) and $(\overline{x_d})$.

We claim that $\phi \in 2SAT$ if and only if $< G, s, d > \in \overline{PATH}$.
If there is no path from s to d, then the formula is satisfiable by assigning *true* to all variables corresponding to the nodes reachable from s. If u is reachable from s and there is an edge (u, v), then $x_v \leftarrow$ *true*. It satisfies the clause $(\overline{x_u} \vee x_v)$. If u is not reachable from s, $x_u \leftarrow$ *false*, and that satisfies the clause $(\overline{x_u} \vee x_v)$. Finally, $x_d \leftarrow$ *false*, makes $(\overline{x_d})$ *true*.
In the other direction, if there is a path $s, u_1, \cdots, u_k, d$ in G, we have the following clauses in ϕ:

$$(x_s) \wedge (\overline{x_s} \vee x_{u_1}) \wedge \cdots \wedge (\overline{x_{u_k}}, x_d) \wedge (\overline{x_d}).$$

Following assignment is necessary to satisfy it.

$$\{x_s, x_{u_1}, \cdots, x_{u_k}, x_d, \overline{x_d}\} \leftarrow \textit{true}.$$

But that is impossible. QED.

References

[MS] *Theory of Computation* by Michael Sipser, (3rd. ed.), Pub. Cengage Learning, 2007, ISBN 978-81-315-2529-6.

[CHP] Computational Complexity by *Christos H Papadimitriou*, Pub. Addision-Wesley, 1994, ISBN 0-201-53082-1.

[NI] N Immerman, *Nondeterministic space is closed under complementation*, SIAM J. Comput. 17(5):935-938, 1988.

[SABB] *Computational Complexity, A Modern Approach* by Sanjeev Arora & Boaz Barak, Pub. Cambridge University Press, 2009, ISBN 978-0-521-42426-4.

[RS] R Szelepcsényi, *The method of forcing for nondeterministic automata*, Bulletin of the European Association for Theoretical Computer Science, 33:96-100, Oct. 1987.

[WJS] W J Savitch, *Relationship between nondeterministic abd deterministic tape complexities*, in J. Comput. Syst. Sci., 4:177-192, 1970.