Lesson 4: The class NL and PSPACE

Theme: Some classical results on the class **NL** and **PSPACE**.

1 Classical results on the class NL

We recall the notion of *log-space reduction.* Let $F : \Sigma^* \to \Sigma^*$ be a function. We say that F is computable in logarithmic space, if there is a 3-tape DTM $\mathcal{M}$ such that on input word w, it works as follows.

- Tape 1 contains the input word w and its content never changes.
- There is a constant c such that $\mathcal{M}$ uses only $c \log |w|$ space in tape 2.
- The head in tape 3 can only "write" and move right, i.e., once it writes a symbol to a cell, the content of that cell will not change.

Tape 1 is called the *input tape*, tape 2 the *work tape* and tape 3 the *output tape.*

Definition 4.1 A language L is log-space reducible to another language K, denoted by $L \leqslant_{\log} K$, if there is a function $F : \Sigma^* \to \Sigma^*$ computable in logarithmic space such that for every $w \in \Sigma^*$, $w \in L$ if and only if $F(w) \in K$.

Remark 4.2 The relation $\leqslant_{\log}$ is transitive in the sense that if $L_1 \leqslant_{\log} L_2$ and $L_2 \leqslant_{\log} L_3$, then $L_1 \leqslant_{\log} L_3$.

Definition 4.3 Let K be a language.

- K is **NL**-*hard*, if for every language $L \in$ **NL**, $L \leqslant_{\log} K$.
- K is **NL**-*complete*, if $K \in$ **NL** and K is **NL**-hard.

Define the following language PATH.

$$\mathsf{PATH} \stackrel{\mathsf{def}}{=} \{(G, s, t) : G \text{ is } \textit{directed} \text{ graph and there is a path in } G \text{ from vertex } s \text{ to vertex } t\}$$

Theorem 4.4 PATH *is* **NL**-*complete.*

Theorem 4.5 (Savitch 1970) NL $\subseteq$ DSPACE$[\log^2 n]$.

To prove Theorem 4.5, it suffices to show that PATH $\in$ DSPACE$[\log^2 n]$. See Appendix A.

Theorem 4.6 (Immerman 1988 and Szelepcsényi 1987) NL = coNL.

To prove Theorem 4.6, we consider the complement language of PATH:

$$\overline{\mathsf{PATH}} \stackrel{\mathsf{def}}{=} \{(G, s, t) : G \text{ is } \textit{directed} \text{ graph and there is } \textit{no} \text{ path in } G \text{ from vertex } s \text{ to vertex } t\}$$

Note that $\overline{\mathsf{PATH}}$ is **coNL**-complete. To prove Theorem 4.6, it suffices to show that $\overline{\mathsf{PATH}} \in$ **NL**. See Appendix B.

2 Classical results on the class PSPACE

Definition 4.7 Let K be a language.

- K is **PSPACE**-*hard*, if for every language $L \in$ **PSPACE**, $L \leqslant_p K$.
- K is **PSPACE**-*complete*, if $K \in$ **PSPACE** and K is **PSPACE**-hard.

Quantified Boolean formulas (QBF) are formulas of the form:

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

where each $Q_i \in \{\forall, \exists\}$ and $\varphi(x_1, \ldots, x_n)$ is a Boolean formula with variables $x_1, \ldots, x_n$.

The intuitive meaning of each Q_i is as follows.

- $\forall x \ \psi$ means that for all $x \in \{\mathsf{true}, \mathsf{false}\}$, ψ is true.
- $\exists x \ \psi$ means that there is $x \in \{\mathsf{true}, \mathsf{false}\}$ such that ψ is true.

We define the problem TQBF:

TQBF	
Input:	A QBF φ.
Task:	Return true, if φ is true. Otherwise, return false.

As usual, it can be viewed as a language TQBF $\stackrel{\text{def}}{=} \{\psi : \psi \text{ is a true QBF}\}$. Note also that the usual Boolean formula can be viewed as a QBF, where each Q_i is $\exists$. Thus, TQBF is a more general problem than SAT.

Theorem 4.8 (Stockmeyer and Meyer 1973) TQBF *is* **PSPACE**-*complete.*

Theorems 4.9 and 4.10 below are the polynomial space analog of Theorem 4.5 and 4.6, respectively. In fact, they can be easily generalized to the so called *time* and *space constructible functions.* See Appendix C.

Theorem 4.9 (Savitch 1970) $\textsc{Nspace}[n^k] \subseteq \textsc{Dspace}[n^{2k}]$.

Theorem 4.10 (Immerman 1988 and Szelepcsényi 1987) $\textsc{Nspace}[n^k] = \text{co}\textsc{Nspace}[n^k]$.

Note that Theorem 4.9 implies **PSPACE** = **NPSPACE** = **coNPSPACE**.

Appendix

A Proof of Theorem 4.5

Algorithm 1 below decides the language PATH.

Algorithm 1

Input: (G, s, t), where G is a directed graph and s and t are two vertices in G.
Task: ACCEPT iff there is a path in G from s to t.

```
Let n be the number of vertices in G.
ACCEPT iff Check_G(s, t, ⌈log n⌉) = true.
```

It uses Procedure $\textsc{Check}_G$ defined below.

Procedure $\textsc{Check}_G$

Input: (u, v, k) where u and v are two vertices in G, and k is an integer $\geqslant 0$.
Task: Return true, if there is a path in G of length $\leqslant 2^k$ from u to v. Otherwise, return false.

```
if k = 0 then
    return true iff (u = v or (u, v) is an edge in G).
for all vertex x in G do
    b := Check_G(u, x, k − 1).
    if b = true then
        b := Check_G(x, v, k − 1).
        if b = true then
            return true.
return false.
```

Note that when computing $\textsc{Check}_G(u, x, k-1)$ and $\textsc{Check}_G(x, v, k-1)$, Procedure $\textsc{Check}_G$ can use the same space. Thus, it uses only $O(k \log n)$ space. Since k is initialized with $\lceil \log n \rceil$, **Algorithm 1** uses $O(\log^2 n)$ space in total.

B Proof of Theorem 4.6

Consider the following algorithm.

Algorithm No-path

Input: (G, s, t) where G is directed graph and s and t are two vertices in G.
Task: There is an accepting run iff there is *no* path in G from s to t.

```
m := the number of vertices in G reachable from s.
{Note: This value m is computed with Procedure Count-Vertex_G below.}
for all vertex x in G do
    Guess if x is reachable from s.
    if the guess is “yes” then
        m := m − 1.
        Guess a path from s to x.
        if it is not possible to guess such a path then REJECT.
        if there is such a path and x = t then REJECT.
ACCEPT iff m = 0.
```

The number of vertices reachable from s can be computed with Procedure COUNT-VERTEX$_G$ defined below.

Procedure COUNT-VERTEX$_G$

Input: u where u is a vertex in G.

Task: Return the number of vertices in G reachable from vertex u, where the number is written in binary form.

1: Let n be the number of vertices in G.
2: $m := 1 +$ the outdegree of u.
3: {Note: m is initialized with the number of vertices reachable from u in $\leqslant 1$ steps.}
4: **for** $i = 2, \ldots, n$ **do**
5:     $m' := 0$.
6:     **for all** vertex x in G **do**
7:         Guess if there is a path from u to x with length $\leqslant i$.
8:         **if** the guess is "yes" **then**
9:             Verify it by guessing such a path (of length $\leqslant i$).
10:             $m' := m' + 1$.
11:         **if** the guess is "no" **then**
12:             Verify that indeed there is no such a path (of length $\leqslant i$).
13:     $m := m'$.
14:     {Note: On each iteration, m is the number of vertices reachable from u in $\leqslant i$ steps.}
15: **return** m

The verification in Line 12 above is done with the following procedure.

Procedure VERIFY$_G$

Input: (u, x, m, i) where u and x are vertices in G, $i \geqslant 2$ is an integer and m is the number of vertices in G reachable from u in $\leqslant i - 1$ steps.

Task: Verify that x is not reachable from u in $\leqslant i$ steps.

1: $\ell := m$.
2: **for all** vertex y in G **do**
3:     Guess if there is a path from u to y with length $\leqslant i - 1$.
4:     **if** the guess is "yes" **then**
5:         $\ell := \ell - 1$.
6:         Guess a path (of length $\leqslant i - 1$) from u to y.
7:         Verify that the edge (y, x) does not exist in G.
8: Verification is complete iff $\ell = 0$.

Note that if any of the verification in Lines 9 and 12 in Procedure COUNT-VERTEX$_G$ and Line 7 in Procedure VERIFY$_G$ fails, the whole algorithm rejects immediately.

The correctness of Procedure COUNT-VERTEX$_G$ can be established by induction on i. The correctness of Algorithm NO-PATH follows immediately from COUNT-VERTEX$_G$.

C Time and space constructible functions

Definition 4.11 Let $T : \mathbb{N} \to \mathbb{N}$ be a function.

- We say that T is *time constructible*, if for every n, $T(n) \geqslant n$ and there is a DTM that on input 1^n computes $1^{T(n)}$ in time $O(T(n))$.

- We say that T is *space constructible*, if there is a DTM that on input 1^n computes $1^{T(n)}$ in space $O(T(n))$.

Intuitively, when we say that $\mathcal{M}$ runs in time/space $O(T(n))$, where T is time/space constructible function, we can assume that on input word w, $\mathcal{M}$ first "computes" the amount of time/space needed to decide w, before going on to process w.

Theorems 4.9 and 4.10 can be easily generalized to space constructible functions as follows.

Theorem 4.12 *Let $f : \mathbb{N} \to \mathbb{N}$ be space constructible function such that $f(n) \geqslant \log n$, for every n.*

- **(Savitch 1970)** $\textsc{Nspace}[f(n)] \subseteq \textsc{Dspace}[f(n)^2]$.
- **(Immerman 1988 and Szelepcsényi 1987)** $\textsc{Nspace}[f(n)] = \text{co}\textsc{Nspace}[f(n)]$.

D Hardness via log space reduction

In our definition of hardness for **NP**, **coNP** and **PSPACE**, we require that the reduction is polynomial time reduction. It is also common to define hardness by insisting the reduction is log-space reduction. That is, we can define K as **NP**-hard by insisting $L \leqslant_{\log} K$, for every $L \in \mathbf{NP}$, rather than $L \leqslant_p K$. Similarly, for **coNP** and **PSPACE**.

Most **NP**-, **coNP**- and **PSPACE**-complete problems are known to remain complete even under log-space reduction, including SAT, 3-SAT and TQBF.

- SAT and 3-SAT are **NP**-complete under log-space reduction.
- TQBF is **PSPACE**-complete under log-space reduction.