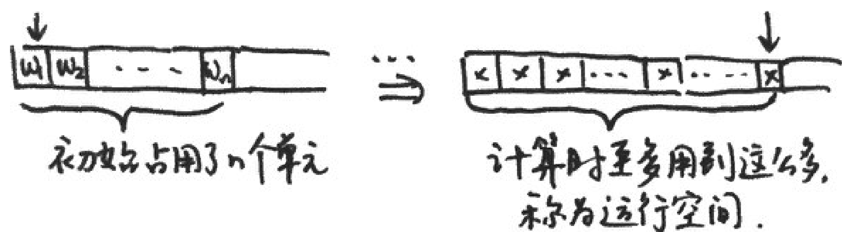
CHAPTER 7

讨论完时间，紧接着讨论空间。有了先前的积累，我们便能循径前行，适当加快步伐。

def 运行空间。

设M是一台TM，且从不迷途。(即M是一台判定器。) M在输入w后的整个计算流程中占用存储单元的峰值，称为M在w下的运行空间。



类似地，设N是一台NTM，且从不在任何分支迷途。N在输入w后，所有分支下占用存储单元的峰值，称为N在w下的运行空间。

remark. 我们在定义中特别强调TM/NTM必须不迷途，因为Chapter 6, 7 考虑的均是可判定语言的精细刻画，而那些会迷途的机器在此无讨论价值。

def 空间复杂度.

设M是一台TM/NTM. 定义函数 $S:\mathbb{N}\to\mathbb{N}$

$$S(n) := \max_{w:|w|=n} (M\text{在}w\text{下的运行空间})$$

称 $S(n)$ 为 M 的空间复杂度.

当然，空间复杂度仍是就一台特定机器而言的。把具体机器抽离，便有了下面的语言类。

def SPACE$(f(n))$语言类.

$\text{SPACE}(f(n)) := \{$语言A $\mid \exists$TM $M: \mathcal{L}(M)=A$ 且M的空间复杂度为 $S(n)=O(f(n))\}$

def NSPACE$(f(n))$语言类

$\text{NSPACE}(f(n)) := \{$语言A $\mid \exists$NTM $N: \mathcal{L}(N)=A$ 且N的空间复杂度为 $S(n)=O(f(n))\}$

我们先来思考一番时间复杂度与空间复杂度的内在联系。任给一台TM M，设其时间复杂度为$T(n)$，空间复杂度为$S(n)$，二者是否有关联？显然，由于TM每步至多只能向右移动一格，所以读写头所能到达的最远距离即为$T(n)$，故 $S(n)\leq T(n)$。另一方面，在$S(n)$这么多空间内，所能出现的所有可能格局数为$|Q|\cdot|\Gamma|^{S(n)}\cdot S(n)$，这意味着，一旦$T(n)$大于该数，则必有重复，导致TM迷途。因而，我们只能有

$$T(n)\leq |Q|\cdot|\Gamma|^{S(n)}\cdot S(n)=2^{O(S(n))}$$

综上，我们有

$$S(n)\leq T(n)=2^{O(S(n))}$$

（对NTM也成立）

它直接导出如下引理。

Lemma 1 对任意$f(n)$均有

$$\mathrm{TIME}(f(n))\subseteq \mathrm{SPACE}(f(n))\subseteq \mathrm{TIME}(2^{O(f(n))})$$

$$\mathrm{NTIME}(f(n))\subseteq \mathrm{NSPACE}(f(n))\subseteq \mathrm{NTIME}(2^{O(f(n))})$$

空间与时间最大的差别在于：空间可以重复利用，而时间一旦流逝即不复返。是故，用少量空间也可完成许多困难的任务。比如，3SAT便能在线性空间内判定，即 $3SAT\in \mathrm{SPACE}(n)$；（证明起来没有难度，略。）而我们目前尚未找到3SAT的多项式时间解法。

初接触空间复杂性时，一定不要与时间混为一谈（虽然二者有不等式关系）。例如，要证明 $A\in \mathrm{SPACE}(f(n))$ 时，只须构造符合空间限制的TM即可，它完全不必在时间意义下高效。

Theorem 2 (Savitch) 对于$f(n)\geq \log n$，

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

proof.

我们将证明 $\forall A\in \mathrm{NSPACE}(f(n))$ 均有 $A\in \mathrm{SPACE}(f^2(n))$.

对任何 $A \in NSPACE(f(n))$，存在NTM N：
$\mathcal{L}(N)=A$ 且 N 的空间复杂度 $S_N(n)=O(f(n))$.
由Lemma 1 知 N 的时间复杂度 $T_N(n)=2^{O(f(n))} \le 2^{d \cdot f(n)}$

现在，我们希望构造出TM M：$\mathcal{L}(M)=A$ 且 M 的空间复杂度 $S_M(n)=O(f^2(n))$（也就是不太大）。换句话说，我们想用较小的空间开销来去除不确定性，模拟 N 的行为。

回顾我们此前是如何用TM模拟NTM的：我们用一个队列来记录NTM当前所有可能的格局，开展「广度优先搜索」。不幸的是，这种方法在时间上高效、空间上低效。

于是，我们必须优先考虑充分利用空间。为此，我们引入了分治思想，开发出以下算法：

```
FindPath(c1, c2, t):
    ~~for each c: |c|≤f(n) do~~
    if (t=1) then
        if (c1⇒c2 or c1=c2) return true
        else return false
    for each c: |c|≤f(n) do
        if (FindPath(c1, c, t/2) and FindPath(c, c2, t/2)) then return true
        ~~else return false~~
    return false
```

该算法回答的是：N 是否能在 t 步以内由格局 c_1 转移至格局 c_2。有几点值得说明：

1° "⇒"指的是 N 中的推导关系（定义见Chap3）

2° 这个算法非常慢，因为它要利用循环枚举所有可能的「中继格局」c.

3° 这个算法空间利用率高。递归深度仅有 $\log t$ 层，而每一层仅需保留变量 c_1、c_2、t、c、$t/2$ 以及两个true/false，总计消耗空间代价是 $(\log t)\cdot O(f(n))$.

4° 这个算法是确定性的，可用TM实现的。（用TM模拟栈及递归调用）.

最后，我们来构造 M："输入 w 时，

1° 计算 $n:=|w|$ 及 $f(n)$，作为全局变量

2° 令 $C_{start} := q_0 w$，其中 q_0 是 N 的初始状态。

3° 枚举所有含有 q_{accept} 的格局 C_{accept}（有 $2^{O(f(n))}$ 个）

(1) 调用 FindPath ($C_{start}, C_{accept}, 2^{df(n)}$)

(2) 如果返回 true，则接纳；否则继续。

4° 若此时尚未接纳，则拒绝。"

因为前面我们已说过 N 的时间复杂度为 $T_N(n) \le 2^{d \cdot f(n)}$，所以 N 若接纳 w 则必须在 $2^{d \cdot f(n)}$ 步以内完成，故

$$N \text{接纳} w \iff M \text{接纳} w$$

即 $\mathcal{L}(M) = \mathcal{L}(N) = A$.

另外，M 的空间复杂度为

$$S_M(n) = \underbrace{O(f(n))}_{\text{存储 } f(n)\text{、}C_{start}\text{、}C_{accept} \text{ 所需}} + \underbrace{\log 2^{df(n)} \cdot O(f(n))}_{\text{FindPath 所需}}$$

$$= O(f^2(n))$$

因此，存在 TM M: $\mathcal{L}(M) = A$ 且 $S_M(n) = O(f^2(n))$，

故 $A \in SPACE(f^2(n))$. ∎

remark. 证明中忽略了一个细节：计算 $f(n)$ 真的只花费 $O(f^2(n))$ 空间吗？未必。为解决之，我们可以猜测 $f(n)$ 的值。具体而言，把第1°步改为

1° 假设 $f(n) = 1, 2, 3, \ldots$

(1) 在当前假设下，反复调用 FindPath (C_{start}, C, $2^{df(n)}$)，统计有多少个格局 C 可由 C_{start} 到达

(2) 如果统计结果与上一次假设相同，则意味着 $f(n)$ 的值正是上一次假设的值。（因为增加 $f(n)$ 已不能让 N 到达更多格局了）

Theorem 2 揭示了空间具有极强的复用性。对比上一章的 Theorem 1, 2，可看出时间与空间的差异。

def PSPACE 语言类与 NPSPACE 语言类

$$PSPACE := \bigcup_{k \in \mathbb{N}} SPACE(n^k)$$

$$NPSPACE := \bigcup_{k \in \mathbb{N}} NSPACE(n^k)$$

由 Theorem 2 直接推得

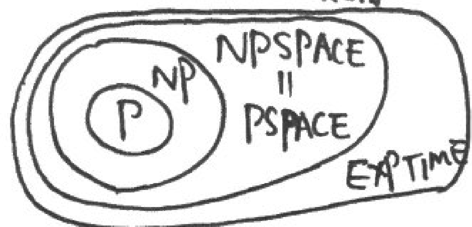
Theorem 3 PSPACE = NPSPACE.

对目前为止讨论过的重要语言类作一小结:

$$P \subseteq NP \subseteq NPSPACE = PSPACE \subseteq EXPTIME$$

（$\subseteq$ 由 lemma1，$=$ 由 theorem3，$\subseteq$ 由 lemma1）

(其中 $EXPTIME := \bigcup_{k \in \mathbb{N}} TIME(2^{n^k})$)



而后面我们会证明 $P \subset EXPTIME$，因此上面的链条中至少有一个「$\subseteq$」是真包含关系。可惜人们目前未有进展。

下面我们通过两个具体例子把握概念。

e.g.1 $ALL_{DFA} := \{\langle D \rangle \mid D$是DFA且$\mathcal{L}(D) = \Sigma^*\}$

首先，ALL_{DFA}是可判定的，而且$ALL_{DFA} \in P$。这是因为我们只须检查由q_0可达的所有状态是否均$\in F$即可，线性时间即可判定。

其次，上述方法消耗的空间也是线性的，故 $ALL_{DFA} \in PSPACE$。（当然由 $ALL_{DFA} \in P$ 可直接推出这一点，但我们这里是在强调空间的概念）

e.g.2 $ALL_{NFA} := \{\langle N \rangle \mid N$是NFA且$\mathcal{L}(N) = \Sigma^*\}$

因为我们总可将NFA转成DFA，所以ALL_{NFA}是可判定的。但请注意：这不意味着$ALL_{NFA} \in P$。这是因为，我们目前了解到的转换方法将造成对应DFA的状态数 $= 2^{O(\text{NFA的状态数})}$。因此，「先转成DFA再解决」的思路无法说明$ALL_{NFA} \in P$，而只能说明$ALL_{NFA} \in EXPTIME$.

接下来，让我们推进一步，说明$ALL_{NFA} \in NPSPACE = PSPACE$。依然，我们不能希求「先转换再操作」，否则空间上不合要求。我们惟有直接对NFA开刀。先设计如下的**非确定**算法（即NTM）:

NotAll(S, d):

//S是N目前所处状态集，d是深度

1° 若 $d > 2^{|Q|}$ 则拒绝，否则继续

2° 若 $\forall q \in S: q \notin F$ 则接纳，否则继续

3° 对每个 $a \in \Sigma_\varepsilon$ 均产生一个分支：

(1) $S' :=$ 状态集S读到a以后转移到的状态集

(2) NotAll$(S', d+1)$

我们断言，NotAll$(\{q_0\}, 0)$ 接纳（也即是存在一条分支接纳）当且仅当 $\mathcal{L}(N) \neq \Sigma^*$。这是因为

$\mathcal{L}(N) \neq \Sigma^* \iff \exists w \in \Sigma^*: w \notin \mathcal{L}(N)$

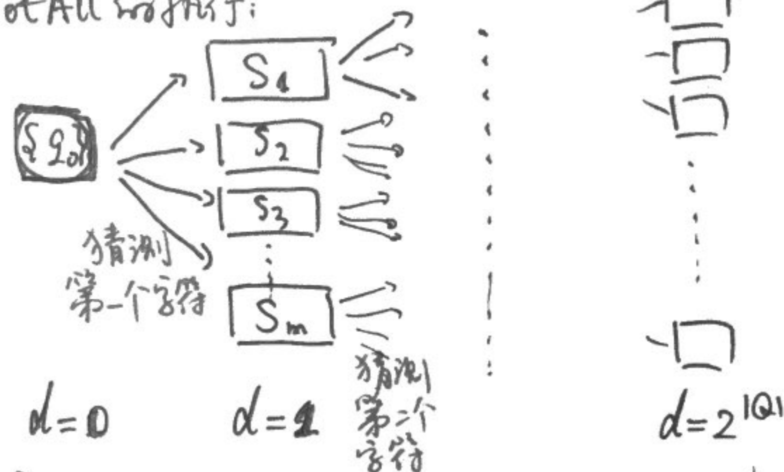
$\iff \exists w \in \Sigma^*$: N在所有计算流程下均拒绝w

$\overset{\star}{\iff} \exists w \in \Sigma^*, |w| < 2^{|Q|}$: N在所有计算流程下均拒绝w

$\iff$ NotAll$(\{q_0\}, 0)$

从而我们得以判定 $N \in \overline{ALL_{NFA}}$。

NotAll的执行：



显然，每条分支仅需存储当前的S和d就够了（无需记录父亲的S和d），故运行空间与|Q|成正比。故 $\overline{ALL_{NFA}} \in$ NPSPACE = PSPACE，因此 $ALL_{NFA} \in$ PSPACE。

那么，$ALL_{NFA} \in$ NP。目前尚不得知。

由例子我们看见，似乎NP是PSPACE的真子集，因为 $ALL_{NFA} \in$ PSPACE 但也许 $\notin$ NP。事实上，人们也倾向于相信 NP $\subset$ PSPACE，因为空间可复用，而「多项式时间」的限制比「多项式空间」的限制强许多。

既然如此，我们就有理由单独为PSPACE语言类定义其中「最难的语言」。

def. NP下的归约关系 $\leq_{NP}$

$\leq_{NP} := \{ (A, B) \in 全体语言类^2 \mid$ 可由在多项式时间内判定B的NTM，构造出在多项式时间内判定A的NTM $\}$

def PSPACE完全。

若 $B \in PSPACE$，且 $\forall A \in PSPACE$ 有 $A \leq_{NP} B$，则称B是PSPACE完全的。

（若B未必属于PSPACE，但 $\forall A \in PSPACE$ 有 $A \leq_{NP} B$，则称B是PSPACE难的。）

remark. 乍一看，拿「时间层面的关系」去度量「空间层面的类」，是很奇怪的。但是回想我们的初衷便得解释：「$\leq_p$」是为了架起P与NP的桥梁，而「$\leq_{NP}$」是为了架起NP与PSPACE的桥梁；为此，必须得「缺什么给什么」。

Proposition 4 若B是PSPACE完全的，且 $B \in NP$，则 $PSPACE = NP$.

Lemma 5,6 设A与B是两门语言。若存在变换 $f: \Sigma^* \to \Sigma^*$，f可用TM在多项式时间内实现，且 $\forall w \in \Sigma^*$，$w \in A \Leftrightarrow f(w) \in B$ ($w \in A \Leftrightarrow f(w) \notin B$)，那么 $A \leq_{NP} B$.

这也不过是上一章引理的转写。注意引理的条件还可适当放宽，但当前形式足够我们使用。

下面，我们介绍一个PSPACE完全的语言——TQBF。可视其为SAT/3SAT的扩展。

在SAT中，我们考虑的是命题逻辑公式，如 $(x_1 \vee \overline{x_2}) \wedge x_3 \vee (x_4 \wedge \overline{x_1})$。在TQBF中，我们引入量词「$\exists$」及「$\forall$」，例如

$$\exists x_1 \forall x_2 \forall x_3 \exists x_4 \left((x_1 \wedge \overline{x_2}) \vee (x_3 \vee x_4) \right)$$

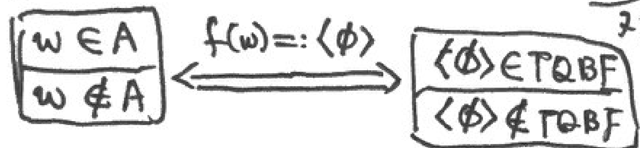
注意论域 $D = \{0, 1\}$，而且我们不允许谓词，所以仍与一阶谓词逻辑有差别。

TQBF := {⟨φ⟩ | φ是仅含「∃」与「∀」量词及若干(约束)变元、∧、∨、¬ 的前束范式，且φ为真}

比如说，$\langle \forall x_1 \exists x_2 (x_1 \vee (\overline{x_1} \wedge x_2)) \rangle \in$ TQBF.

Theorem 7 TQBF是PSPACE完全的。

proof. 我们利用Lemma 5，说明(对任意语言A∈PSPACE)存在变换f:

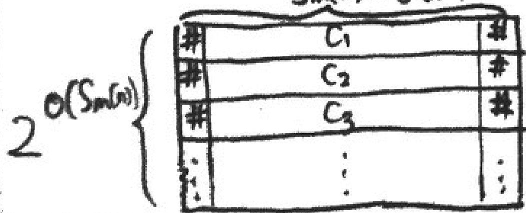


且f可用TM在多项式时间内实现。

因为A是任意的，所以，与Cook-Levin Theorem一样，我们的f只能利用A的抽象性质，即A∈PSPACE，亦即∃TM M: $L(M)=A$ 且M的空间复杂度 $S_M(n)=O(n^k)$。

有可能直接仿照Cook-Levin的方法做吗？且看M的计算流程(引表)有多大吧：

$S_M(n)=O(n^k)$



这意味着，照搬Cook-Levin的方法需引入 $2^{O(n^k)}$ 多个变量，从而f不可能在多项式时间内做完。

为解决，我们借用Savitch的(二分)思想，自顶向下地构造φ。大致的思路是

$$\phi := \exists C_{accept}\, \psi(C_{start}, C_{accept}, 2^{d\cdot n^k})$$

而 $\psi(C_1, C_2, t) := \exists C\, \psi(C_1, C, t/2) \wedge \psi(C, C_2, t/2)$

$= \exists C, \forall (C_3, C_4) \in \{(C_1, C), (C, C_2)\}: \psi(C_3, C_4, t/2)$

底层 $\psi(C_1, C_2, 1) := \begin{cases} 1, & C_1 \Rightarrow C_2 \\ 0, & \text{否则} \end{cases}$

但以上的描述不甚详尽，没有点明许多关键的雷区。最最关键的，是我们如何用0-1变量来编码格局。策略来自于Cook-Levin：用变量 $1_{i,a}$ 来表明格局的第i个字符是否为a。

于是，式子「$\psi(C_1, C_2, t) := \exists C, \forall (C_3, C_4) \in \{(C_1, C), (C, C_2)\}\ \psi(C_3, C_4, t/2)$」便要写作

$$\psi(C_1, C_2, t) := \underbrace{\exists \mathbb{1}^{(t)}_{1,\cdot} \exists \mathbb{1}^{(t)}_{2,\cdot} \exists \mathbb{1}^{(t)}_{3,\cdot} \cdots \exists \mathbb{1}^{(t)}_{S_M(n),\cdot}}_{\text{一共 } S_M(n)\times|\Gamma| \text{ 个，编码 } C}$$

$$\underbrace{\forall \mathbb{1}'^{(t)}_{1,\cdot} \forall \mathbb{1}'^{(t)}_{2,\cdot} \cdots \forall \mathbb{1}'^{(t)}_{S_M(n),\cdot}}_{\text{一共 } S_M(n)\times|\Gamma| \text{ 个，编码 } C_3} \underbrace{\forall \mathbb{1}''^{(t)}_{1,\cdot} \forall \mathbb{1}''^{(t)}_{2,\cdot} \cdots \forall \mathbb{1}''^{(t)}_{S_M(n),\cdot}}_{\text{同前，编码 } C_4}$$

[((C₁的0-1变量与C₃的全等 ∧ C的0-1变量与C₄全等) ∨ (C的…与C₃全等 ∧ C₂的…与C₄全等)] → $\psi(C_3, C_4, t/2)$.

可以看见，$\psi(C_1, C_2, t)$ 相当于在低层 $\psi(C_3, C_4, t/2)$ 的前面附加了长度为 $O(n^k)$ 的头部。头部中定义出的变量 $\mathbb{1}'^{(t)}_{\cdot,\cdot}$ 与 $\mathbb{1}''^{(t)}_{\cdot,\cdot}$ 传递给了 $\psi(C_3, C_4, t/2)$，作进一步的展开。到了底层，$\psi(C_1, C_2, 1)$ 取得的 C_1、C_2 之中无非包含的是上面一层定义出来的0-1变量，共 $O(n^k)$ 个。$\psi(C_1, C_2, 1)$ 通过M的转移函数了直接对其进行约束。

综上，ϕ 每展开一层，就新增 $O(n^k)$ 长度。因一共展开了 $\log 2^{dS(n)} = O(n^k)$ 层，故 ϕ 的总长为 $O(n^{2k})$，构造成功。■

remark. 为了简便，我们在证明中省略了每层对变量 $\mathbb{1}^{(t)}$、$\mathbb{1}'^{(t)}$、$\mathbb{1}''^{(t)}$ 的部分约束（比如要求一个格子只能放一个字符），而是把这约束等价地挪至底层。请思考其中的合理性。

有了「第一个」PSPACE完全的语言，要找出其它的PSPACE完全语言就简单了。下面我们介绍两个与博弈密切相关的语言，并证明其PSPACE完全性。

在一场非输即赢的二人博弈中，双方轮番上阵，对一个系统做变换。如果某方变换完毕以后系统进入了「终结状态」，则该方告胜，反方告负。

为了简便，我们总是假定甲方先行，且博弈过程中双方共计行 m 步（m 是定值）。设 F 是「终结状态」的集合，那么，所谓「甲方必胜」指的是

$$\exists f_1 \forall f_2 \exists f_3 \forall f_4 \cdots \exists f_m [f_m f_{m-1} \cdots f_2 f_1 S \in F]$$

（S：初状态）

意即：甲方存在一种变换 f_1，使得乙方无论以何种变换 f_2 应对，甲方都存在进一步的变换 f_3，使得乙方无论以何种变换 f_4 应对，……，甲方都存在「致胜一步」f_m。
（严格来说，还应要求 f_2、f_3、…f_n 是「合法」的，此处为了简便将其省略）
类似地，「乙方必胜」指的是

$$\forall f_1 \exists f_2 \forall f_3 \exists f_4 \cdots \forall f_m [f_m f_{m-1} \cdots f_2 f_1 S \notin F]$$

即：无论甲方以何种变换 f_1 开局，乙方总能找到应对 f_2，使得甲方无论以何种变换 f_3，乙方总能找到应对 f_4，……，使得甲方最终无法胜利（即告负）。

注意，「甲方必胜」与「乙方必胜」的含义并非甲/乙无论怎么走都能坐等胜利，而是说甲/乙有某种极聪明的「策略」，依策略行进则高枕无忧。

不难由逻辑式中看出，「甲方必胜」与「乙方必胜」恰好互相反。换言之，甲方必胜 $\Leftrightarrow$ 非乙方必胜

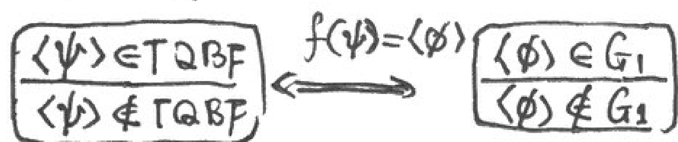
有了上述背景知识，让我们来考虑一个具体的博弈情景。假设 ϕ 是一个仅含变元 $x_1, \ldots, x_m$ 的命题公式，如

$$\phi = (x_1 \wedge x_2 \vee (x_3 \to \neg x_4)) \wedge (\neg x_2 \vee \neg x_3)$$

甲乙两方轮番对 $x_1, x_2, \cdots, x_m$ 赋值0或1。m 轮结束后，若 ϕ 的真值为1，则

甲胜，否则乙胜。

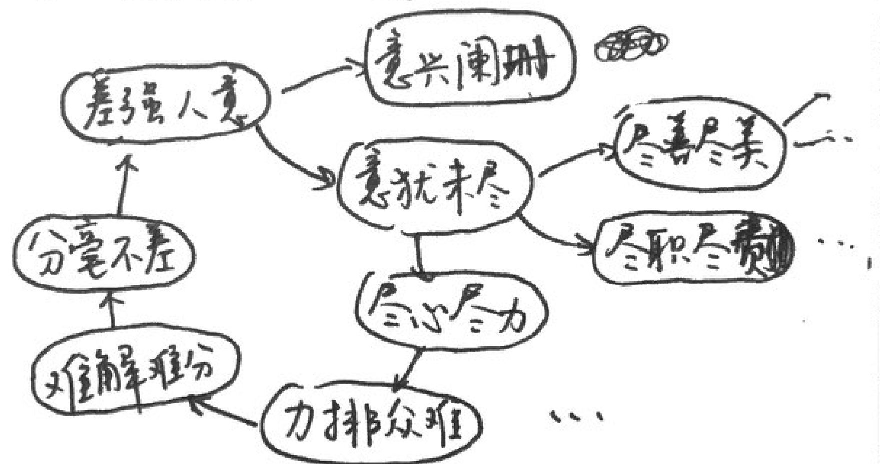
我们考察语言 $G_1 ::= \{\langle\phi\rangle \mid$ 在上述博弈中，给定 ϕ，甲方必胜$\}$，那么我们很容易发现 G_1 与 TQBF 的内在关系。事实上，$TQBF \leq_{NP} G_1$。



给定 $\langle\psi\rangle \in TQBF$，我们首先把 ψ 中的 $\forall$ 量词与 $\exists$ 量词变成一样多。具体做法是：若 $\forall$ 多了，则补上若干「$\exists x_i$」，其中 x_i 是 ψ 中没出现过的变元；若 $\exists$ 多了，处理方式类似。显然这种处理不会改变 ψ 的真假。

设改完以后 $\psi = Q_m x_m Q_{m-1} x_{m-1} \cdots Q_1 x_1 \phi$，我们断言 $\langle\phi\rangle \in G_1 \Leftrightarrow \psi$ 为真。这是因为，即使 $Q_m, \cdots, Q_1$ 不是严格的「$\exists$」、「$\forall$」交错，也不会影响我们对于「甲方必胜」的定义（思考题）。是故，$TQBF \leq_{NP} G_1$.

下面，我们考察一个更复杂的博弈：成语接龙。将接龙游戏抽象出来，博弈双方无非是在一张有向图上行走：



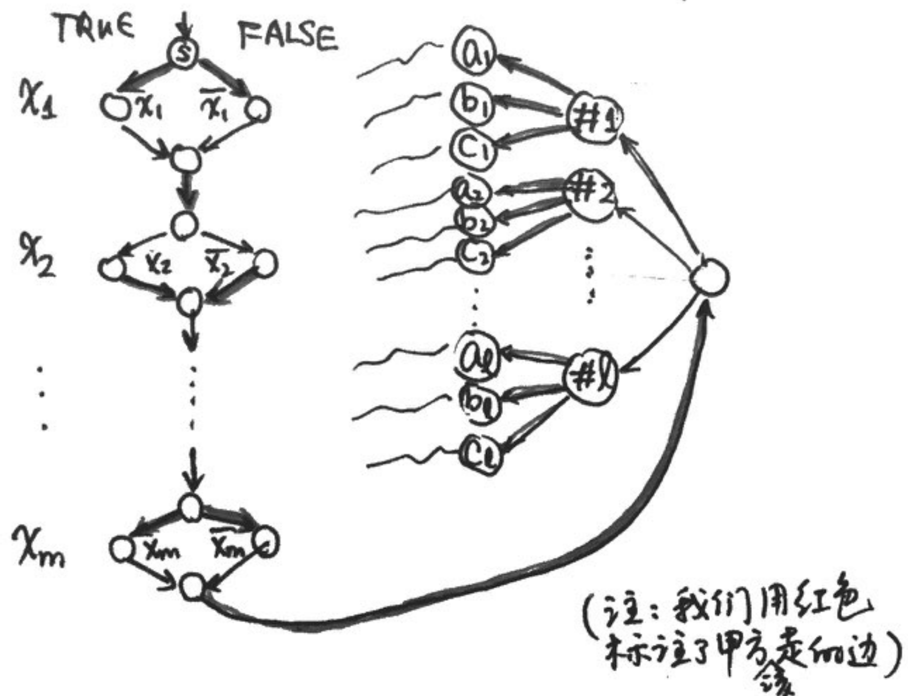
我们要求成语不能重复；当甲方走后，乙方无处可接时，甲胜。问甲是否必胜？换言之，我们考察的是如下语言

$G_2 ::= \{\langle G, s\rangle \mid$ 从图 G 的 s 点出发，甲方必胜$\}$

$G_2 \in PSPACE$ 的证明留作习题。我们只在此说明 G_2 是 PSPACE 完全的。只需证明 $G_1 \leq_{NP} G_2$ 即可。

给定公式 ϕ，我们先把它转换成 3-cnf（即形如 $(\cdot \vee \cdot \vee \cdot) \wedge (\cdot \vee \cdot \vee \cdot) \wedge \cdots$）。转换以后，设 $\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_l \vee b_l \vee c_l)$，其中任何 a_i, b_i, c_i 都是变量 $x_1/x_2/\cdots/x_m$ 或其反。

接下来，我们仿照 Chap.6 Theorem7 (HAMCYCLE) 的做法，生成如下的图 G



其中，若 $a_i = x_j$，则 $(a_i) \to (x_j)$；若 $a_i = \overline{x_j}$，则 $(a_i) \to (\overline{x_j})$。$b_i$、$c_i$ 的处理类似。

现在，我们来说明 ϕ 下甲方必胜 $\Leftrightarrow$ (G,s) 下甲方必胜。

1° ($\Rightarrow$) 因 ϕ 下甲方必胜，故甲总能找到一种对 x_1、x_3、$\cdots$、x_m 的指派策略，使 $\phi=1$，也就是每个子句 $(a_i \vee b_i \vee c_i)$ 均为真。

在博弈 (G,s) 中，甲方一开始循着 ϕ 的策略行走。若在 ϕ 中应给 x_j 赋 1，则在 G 中行经 x_j 对应菱形时走左边；否则走右边。当甲行至 G 的右半部分以后，无论乙选了哪个 (#i)，甲总是走使 $(a_i \vee b_i \vee c_i)$ 为真的那个 $(a_i)/(b_i)/(c_i)$，如此一来，乙总是陷入无路可走的境地。

2° ($\Leftarrow$) 等价于证 ϕ 下乙方必胜 $\Rightarrow$ (G,s) 下乙方必胜。论证同上。

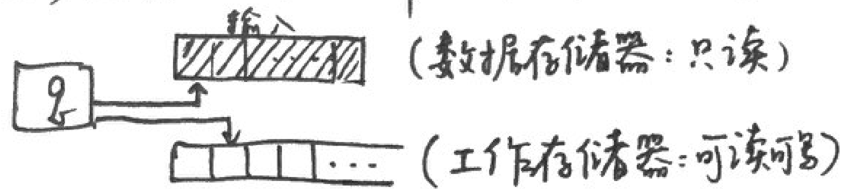
于是 $G_1 \leq_{NP} G_2$，又 G_1 是 PSPACE 完全的，故 G_2 也是 PSPACE 完全的。

APPENDIX
The Log Space

截至目前，「不足线性」的空间复杂度是不可想象的——光是输入就需要耗费线性空间，又怎么可能谈「亚线性」空间呢？但是，如果我们稍稍更改一下规定，不把输入计在内的话，「亚线性」的空间复杂度则是可能的。

研究亚线性的空间复杂度有何意义？我们可以这么想：一台计算机的内存容量（即可供「工作」的容量）为 4GB，而它却希望处理存档在硬盘中的、规模为 100GB 的数据。只有使用亚线性空间的算法，才可能达成这个目标。

def. 写入受限的图灵机：一台双带图灵机，只不过第一个存储器只允许读，不允许写，且其读写头不许超出输入范围。



转移函数 $\delta: Q\times\Gamma^2 \to Q\times\Gamma\times\{L,R\}^2$.

def 写入受限的非确定图灵机：同上，只是引入非确定性。转移函数 δ:

$$Q\times\Gamma^2 \to 2^{Q\times\Gamma\times\{L,R\}^2}$$

remark. 本附录中，为了简单起见，分别用 TM / NTM 来称呼二者。

二者的时间复杂度定义与以前类似，只是空间复杂度仅仅考虑工作存储器。详细的定义我们就不给出了；让我们赶紧进入正题。

def L和NL语言类.

$L := SPACE(\log n)$

$NL := NSPACE(\log n)$

e.g. $\{0^n 1^n 2^n \mid n \in \mathbb{N}_0\} \in L$

（只需维护二进制计数器即可）

（注意到该语言并非CFL，可见虽然我们对空间作了很强限制，但TM的判定能力依旧~~[illegible]~~不弱）

e.g. $PATH := \{\langle G, s, t\rangle \mid$ 图G中有一条从s到t的路径$\}$

$PATH \in NL$.

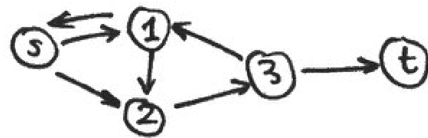
思路：要判定G中是否有$s \leadsto t$的路径，也就是判定G中是否有$s \leadsto t$的、长度小于$|V|$的路径。可是$|V|$可能与输入长度n差不多大，所以我们不能指望在$O(\log n)$的空间内存储整条路径，从而不可能开展正常的DFS。

为了解决这个问题，可引入不确定性。在搜索过程中，我们只记录当前处在哪个节点，不记录此前曾到达哪些节点，走了至多$|V|$步即告终止。当然，由于我们仿佛得了健忘症，很有可能出现原地打转的情形。可是，只要有一种情形能使s到t，我们就成功了。

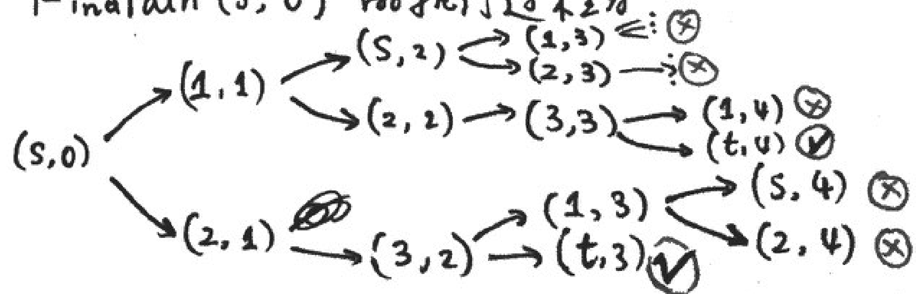
```
FindPath(u, step):
    if u=t then accept
    if step≥|V| then reject
    foreach (u,v)∈E do
        产生一条新分支并调用 FindPath(v, step+1)
```

注意：产生新分支是一个「永不返回」的过程。v将在新分支中覆盖u；step同理。

以下图为例.



FindPath(s, 0) 的执行过程为



也许你留意到，Savitch Theorem 本质上也是寻路过程；它利用二分法减少了所需空间。能否把它移植过来，解决PATH呢？恐怕不行。我们至多只能说明

$$PATH \in SPACE(\log^2 n)$$

却无法说明

$$PATH \in SPACE(\log n) = L.$$

具体原因留作思考。

与 $P \overset{?}{=} NP$ 一样，$L \overset{?}{=} NL$ 也是计算复杂性理论中的未解之谜。人们猜测 $L \subset NL$，即NL语言类中的某些问题确实难以在确定的对数空间内解决。于是，我们便有动机去定义NL之中「最难判定」的一类语言——NL完全的语言。

def 对数空间归约关系 $\leq_L$.

$\leq_L := \{(A,B) \in 全体语言类^2 \mid$ 能够通过B的、对数空间的判定器(TM)构造A的、对数空间的判定器(TM)$\}$

def NL完全的. 若 $B \in NL$，且 $\forall A \in NL$ 均有 $A \leq_L B$，则称B是NL完全的。

下面的定义与引理是证明 $\leq_L$ 关系的工具。

def 可由TM在对数空间内实现的函数.

设 $f: \Sigma^* \to \Sigma^*$。如果存在一台三带TM M，其三个存储器分别为输入、工作区、输出，且

1° M能由输入w计算出 $f(w)$ 并打印在输出存储器上

2° 打印过程中输出头总是向右走

3° M的空间复杂度为 $O(\log n)$

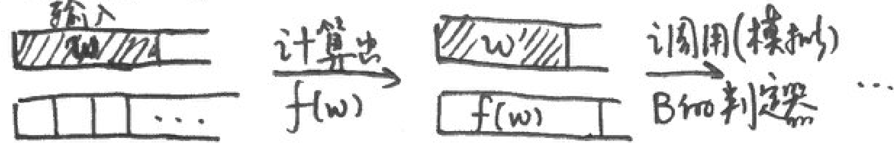
那么称 f 是可由TM在对数空间内实现的。

Lemma 1 设A与B为两门语言。如果存在某个可由TM在对数空间内实现的函数 f，满足 $\forall w \in \Sigma^*$，$w \in A \iff f(w) \in B$ (*)，则 $A \leq_L B$.

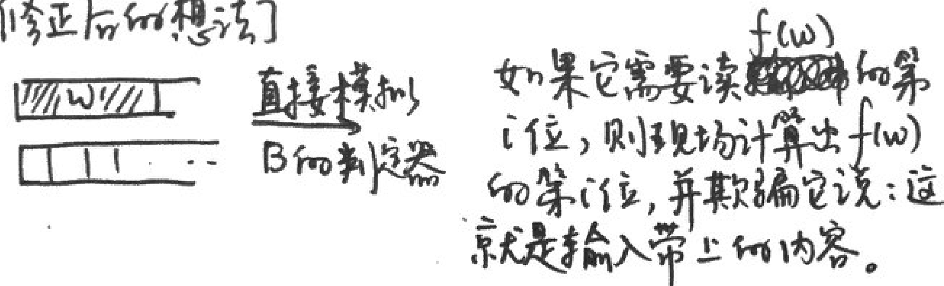
proof. 乍一看，似乎只要先计算出 $f(w)$，再去调用B的判定器就可以了。实则不然。虽然计算 $f(w)$ 的工作仅需花费对数空间，但是存储 $f(w)$ 呢？说不定要花费多项式空间。（因为上面定义中并没有对输出存储器的空间作限制）

为此，我们取个巧：不把$f(w)$存储下来，而是「随叫随到」，需求哪一位就算哪一位。由于定义中有假定2°，所以$f(w)$中的每一位之间没有依赖关系，因而「随叫随到」是切实可行的。

[开始的想法]



[修正后的想法]



（B的判定器自以为输入存储器上的内容就是$f(w)$。但事实上，这是假象）

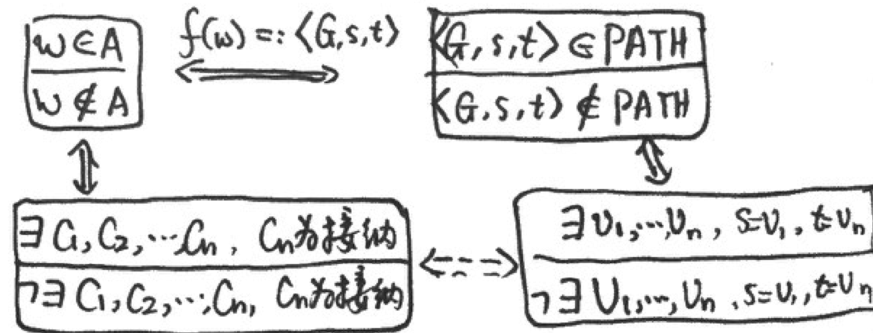
落到实处，设M是B的判定器，而T是实现f的对数空间内TM。我们可修改T使之只输出$f(w)$中的特定某一位。设 T_i := 只输出$f(w)$第i位的TM。我们可在M的每个转移处都「嵌入」某个T_i，以营造「读到$f(w)$第i位」的假象。■

Theorem 2

PATH是NL完全的。

proof.

前面已说明PATH ∈ NL，下面只需证 $\forall A \in NL$，$A \leq_p PATH$。这只需证明：存在Lemma 1中所说的函数f.



对任何给定的语言$A \in NL$，设N_A是一台能在对数空间内判定A的NTM。首先，我们强令N_A在进入接纳格局以前把存储器归零（即回到初始的样式）。这么做可以保证N_A的接纳格局是唯一的。设N_A的空间复杂度为$S(n) \leq d \cdot \log n$.

然后，$\forall w \in \Sigma^*$，我们来生成 $f(w) =: \langle G, s, t \rangle$.

$s :=$ 初始格局
$t :=$ 接纳格局 ，$G = (V, E)$.

V 由所有长度 $\le d \cdot \log n$ 的格局构成
E 中的边代表格局之间具有推导关系。

注意，在编码格局时，输入存储器上的内容(w)不必出现(否则就是冗余的)。只需记录输入头的位置即可。

很显然，$w \in A \iff \langle G, s, t \rangle \in PATH$.

> Problem 这么看来，w 似乎根本没出现在 $\langle G, s, t \rangle$ 中。果真如此吗？

接下来是证明的关键：f 是可由 TM 在对数空间内实现的。

首先，强令 N_A 在接纳以前归零是人工完成的，与 f 无关，不予考虑。

其次，d 也是人工得到的。$d \cdot \log n$ 是可在对数空间内计算并存储的。

其三，s 和 t 长度都 $\le d \cdot \log n$，可在对数空间内计算与输出。

最后，$G = (V, E)$ 可分两步输出：

1° 枚举所有长度 $\le d \cdot \log n$ 的字符串 c，检查 c 是否为格局，若是则输出之。

2° 枚举所有长度 $\le d \cdot \log n$ 的字符串对 (c_1, c_2)，检查 c_1 与 c_2 是否为格局，且 $c_1 \Rightarrow c_2$。若是，则输出 (c_1, c_2)。

■

def. coNL 语言类

$$coNL := \{\overline{A} \mid A \in NL\} = \{A \mid \overline{A} \in NL\}$$

Theorem 3 (*Immerman–Szelepcsényi*)

$$NL = coNL$$

proof. 如果我们证明了 $\overline{PATH} \in NL$，那么便蕴涵 $NL = coNL$。为什么呢？回忆 Theorem 2 的证明，我们说 $\forall A \in NL$，都存在可在对数空间内实现的函数 f，

使得 $\forall w,\ w\in A \iff f(w)\in PATH$，因此 $\forall w,\ w\in \overline{A} \iff f(w)\in \overline{PATH}$。如果证明了 $\overline{PATH}\in NL$，那么便能借助 f 构造出在对数空间内判定 $\overline{A}$ 的 NTM。这样一来，$\forall A\in NL$，均有 $\overline{A}\in NL$，从而 $NL\subseteq coNL$。复用这一结论，$\forall A\in coNL \Rightarrow \overline{A}\in NL \Rightarrow \overline{\overline{A}}\in NL \Rightarrow A\in NL$，从而 $coNL\subseteq NL$。于是，$coNL=NL$。

下面我们便来证明 $\overline{PATH}\in NL$。

$\overline{PATH}=\{\langle G,s,t\rangle \mid$ G中没有从 s 到 t 的路径$\}$

一上来就思考它显得有些困难，我们不妨先作个弊，假定我们已知 $c :=$ 从 s 出发总共可到达多少个顶点（包括 s 自己），使问题得以简化。

既然 c 已知，那么只要 $R\subseteq V$ 满足 (1) $|R|=c$ 且 (2) $\forall v\in R: s\leadsto v$，那么我们就可断言 R 正是由 s 可达的顶点集。那么，$s\not\leadsto t \iff t\notin R$。

受此启发，我们有了如下思路：

1° 生成一个顶点集 $R\subseteq V$

2° 验证是否 $|R|=c$

3° 验证是否 $\forall v\in R: s\leadsto v$

4° 验证是否 $t\notin R$.

当然切不可忘记我们只有对数空间，是故依照 1°–4° 的次序执行是不足取的。我们必须动态地生成 R，边生成边验证，验证完的内容便立即丢弃，不必存储——惟有如此方得在对数空间内完成任务。

size := 0

for $i=1\ldots|V|$ **do**

(1) 创建一个新分支。新分支「猜想」$v_i\in R$，而旧分支则「猜想」$v_i\notin R$。（当然，猜想之正确性亟待验证）新分支执行以下内容，而旧分支跳过以下内容直接进入下一轮。

(2) 按照 PATH 中介绍的方法（非确定地）验证是否 $s\leadsto v_i$。（没有猜对路径的分支自行拒绝了，好比消亡一样；猜对路径的分支犹存，继续下一步）

(3) size++（意即 $v_i\in R$ 猜对了，记录一下）

(4) 若 $t=v_i$ 则拒绝；否则继续循环.

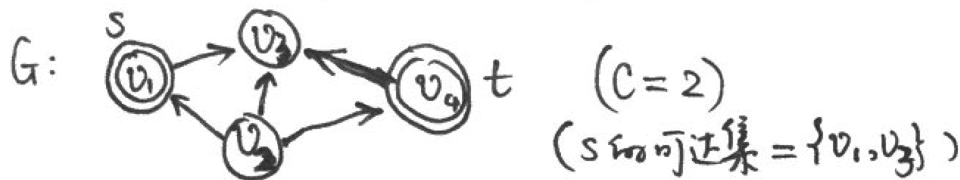
if size=c then

接纳（因为前面的循环成功地猜对了可达集，而且还验证了 $t \notin R$）

else

拒绝（因为前面的循环并未成功地猜出可达集，换言之猜出的 $R \neq$ 可达集）

下面的图示也许有助于理解.



i=1
size=0
猜$v_1 \in R$
猜$v_1 \notin R$
不确定地寻路 $s \rightsquigarrow v_1$
这两个情形其实是完全一样的，但没有关系。

i=2
size=1
size=1
size=0
猜$v_2 \in R$
略
猜$v_2 \in R$
不确定地寻路 $s \rightsquigarrow v_2$
(找不到路，全部清除，说明猜错了)

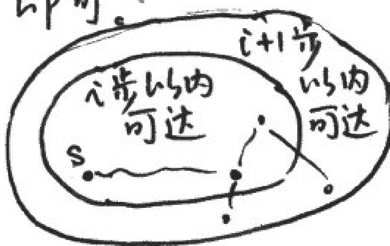
i=3
size=1 ……
size=0
略

i=4
size=2
size=1
size=0
略

收尾工作
size=2
接纳！

剩下的问题是：如何得到 c？其实，计算手段和前面极为类似。

定义 C_i := 由S出发，经过不超过 i 步所能到达的顶点个数。显然 $C_0 = 1$，$C_n = C$。我们只须找出递推 C_i 的方法即可。



设 A_i := 由S出发，i步以内可达的顶点集。（定义它只是为了讨论方便，并不影响算法。）那么 $C_i = |A_i|$。若已知 A_i，能否得到 A_{i+1} 呢？很简单：

② 枚举 $v_j \in V$。若 $\exists u \in A_i : u \leadsto v_j$，则 $v_j \in A_{i+1}$，否则 $v_j \notin A_{i+1}$。

可是我们只有 c_i，没有 A_i，那还可能得到 c_{i+1} 吗？当然可以，只要把 A_i 猜出来即可。这与前面已知 c 猜可达集是完全类似的。

我们直接给出下面的非确定算法：

```
c := 1  (即 c_0)
for i = 1 ... |V| do  (递推|V|次得 c_|V|)
    c' := 0
    for j = 1 ... |V| do  (枚举顶点)
        cnt := 0, found = false
        for k = 1 ... |V| do  (猜想 A_i)
            (1) 创建一个新分支。新分支「猜想」v_k ∈ A_i，
                而旧分支则相反。前者执行以下内容，
                后者跳过以下内容直接进入下一轮
            (2) 修改PATH中的方法，非确定地验证
                是否存在长度 ≤ i 的 s ⇝ v_k 的路径。
            (3) cnt++
            (3) if (v_k, v_j) ∈ E then found := true
        if cnt ≠ c_i then reject
        if found then c'++
```

这段程序最终可能产生成千上万条完全相同的分支，但不要紧，把之前开发的算法接在后头，总能达成想要的效果。■

NL = coNL 的结果是很令人吃惊的。与之类似的还有 coNPSPACE = NPSPACE（因为 NPSPACE = PSPACE，而后者对补操作封闭）。

大略地说，凡是能用 NTM 在对数/多项式空间内判定的「存在性问题」，其对偶的「任意性问题」同样能用 NTM 在对数/多项式空间内判定。人们认为这可能是空间复杂性有别于时间复杂性的性质之一。