

理论计算机科学导引
TCS - 毛宇尘老师班

shrike505

目录

一、 问题与编码	2
1.1 编码 (Encoding)	2
1.1.1 prefix-free 编码	3
1.1.2 编码与可数的关系	4
二、 计算模型	4
2.1 布尔电路 (Boolean Circuit)	4
2.1.1 NAND 电路 (NAND Circuit)	6
2.2 计算规模	7
2.2.1 优化效率	8
2.2.2 编码程序	9
2.2.3 更长的输入: Infinite!	10
三、 语言与自动机	10
3.1 DFA 与正则语言	10
3.2 NFA	12
3.3 正则表达式	13
3.3.1 Pumping Theorem	15
3.4 Pushdown Automaton	16
3.5 语法 (Grammar)	18
3.6 上下文无关语言 (Context-Free Language, CFL)	19
四、 图灵机 (Turing Machine)	21
4.1 图灵完备	22
4.1.1 NAND-TM	22
4.1.2 NAND-RAM	24
4.2 通用图灵机 (Universal Turing Machine)	24
4.3 函数的可计算性 (Computability)	24
4.3.1 停机问题	25
4.3.2 Recursion Theorem	27
4.3.3 Godel's Incompleteness Theorem	27
五、 运行时间	27
5.1 TIME Hierarchy Theorem	28
5.2 从有穷函数到无穷函数 (运行空间)	29
5.3 Difficulty Levels of Functions	31

5.3.1	Verifiability	32
5.4	Probability: Randomized Computation	34
5.4.1	Field of Probabilistic Turing Machines	36
5.4.2	Pesudo-random Generators	36

泱泱猩热血， /汨若风袭泉， /起落樱唇际， /往来兰气间。

Reference:

- introtcs.org - 教材 (电子书版)
- <https://fla.cuijiacai.com/> - 南京大学形式语言与自动机课程笔记
- <https://williamhoza.com/teaching/spring2025-intro-to-complexity/> - UChicago: CMSC 28100: Introduction to Complexity Theory

问题与编码

一言以蔽之，它 (TCS) 研究的是问题的上界与下界。

需要界定计算所需要解决的问题，以及计算所需要的设备（模型）。

这一节先规定前者。回顾一些经典的算法或数学上的问题：给定带权重的图 G ，求其中的最短路/其的最小生成树；提供矩阵 A, B ，求其乘积 AB 。这些问题都可以看作一个函数：给定输入，求输出。

与程序设计中的函数强调 implementation (即 How to compute the answer) 相比，这里的函数更多具有数学意义，强调 specification (即 What should the answer be)。

接下来聚焦这些函数的输入，计算机无法理解图、矩阵这些概念，只能理解二进制串 (binary string)，也就是一串又一串的 0 和 1——于是要通过某些编码方式将这些元素编码为 01 串。先定义一个字符表 (Alphabet)： $\Sigma = \{0, 1\}$ ，于是长度为 n 的二进制串的集合可表示为 $\Sigma^n = \Sigma \times \Sigma \times \dots \times \Sigma = \{(a_1, a_2, \dots, a_n) \mid a_i \in \Sigma\}$

特别规定 Σ^0 是长度为 0 的串的集合，这个串用 e 表示，即 $\Sigma^0 = \{e\}$

$\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$ 即为所有长度的二进制串集合。



Q 前缀 (Prefix)

$x = a_1 a_2 \dots a_n, y = b_1 b_2 \dots b_n$ 的拼接 (Concatenation) 为 $xy = a_1 a_2 \dots a_n b_1 b_2 \dots b_n$

x 是 y 的一个前缀 (Prefix)，当对于某些 $z \in \Sigma^*$, $y = xz$

类似的可以定义后缀 (Suffix)，不再赘述

可以将 Σ 中的 0 和 1 换成任意字符，例如 26 字母，方框三角圆，以此组建你自己的 Alphabet!



1.1 编码 (Encoding)

有了最基础的元素 (字符)，将图、矩阵、等等等等计算函数的输入转化为字符串的过程，称为编码，即一个映射 $E : A \rightarrow \{0, 1\}^*$ 。

Q 编码性质

显然，这个映射需要是单射 (injective，在下文中会频繁表示为 one-to-one)，即不同元素的映射结果 (得到的字符串) 必须是不同的。



例子

自然数 $n \in N$ ($\text{parity}(n)$ 是 n 对 2 取余的结果): $NtS(n) = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ NtS(\lfloor \frac{n}{2} \rfloor) \text{ parity}(n) & \text{if } n>1 \end{cases}$

- 亦即 n 的二进制表示
- 自然数对 $(a, b) \in N \times N$, 自然的想法是 a 的编码拼接 b 的编码, 但是会出现编码重复, 并不是单射
- 对于 1110, 可以解释为 (1, 6) 和 (3, 2), 这实质上是因为在计算机读取完前两个 1 时, 并不知道它代表 3 还是一个其他数的前缀

1.1.1 prefix-free 编码

在第二个例子的教训下, 我们需要找到的编码映射是 prefix-free 的, 即对于任何的 $x \neq x'$, $E(x)$ 都不是 $E(x')$ 的前缀。

接下来 myc 老师突然就这个 prefix-free 证明了两个寻找另一种编码的引理, 感觉很突兀。

引理 1.1.1

假设已经存在一个 prefix-free 的编码 $E : A \rightarrow \{0, 1\}^*$, 那么对于编码 $\bar{E} : A^* \rightarrow \{0, 1\}^*$ ($A^* = \bigcup_{n \geq 0} A^n$, 我理解为一个由任意长待映射元素 $(a_i \in A)$ 序列组成的集合, 接下来要找到对这些元素序列的编码), 命 $\bar{E}(a_1 a_2 \dots a_n) = \begin{cases} E(a_1)E(a_2)\dots E(a_n) & \text{if } n \geq 1 \\ e & \text{if } n=0 \end{cases}$, 那么 \bar{E} 是 one-to-one 的。

证明. 假设存在 $(a_1, a_2, \dots, a_n) \neq (b_1, b_2, \dots, b_m)$, 使得 $\bar{E}(a_1 a_2 \dots a_n) = \bar{E}(b_1 b_2 \dots b_m)$, 那么 $E(a_1)E(a_2)\dots E(a_n) = E(b_1)E(b_2)\dots E(b_m)$, 且 $\exists i, s.t. \forall j < i, a_j = b_j$, 且 $a_i \neq b_i$ (即在第 i 个字符前两个元素序列的每个元素都相同)

那么 $E(a_1)E(a_2)\dots E(a_{i-1}) = E(b_1)E(b_2)\dots E(b_{i-1})$, 则 $E(a_i)\dots E(a_n) = E(b_i)\dots E(b_m)$, 那么对于 $E(a_i)$ 和 $E(b_i)$, 要么前者是后者的前缀, 要么后者是前者的前缀, 又考虑到 $a_i \neq b_i$, 则 E 不是 prefix-free 的, 这与题设冲突。 \square

引理 1.1.2

如果存在 one-to-one 的 $E : A \rightarrow \{0, 1\}^*$, 那么存在 prefix-free 的 $E' : A \rightarrow \{0, 1\}^*$, 且使得 $|E'(a)| \leq 2|E(a)| + 2, \forall a \in A$

证明. 将原编码中的 0 映射为 00, 1 映射为 11, 该元素再次编码结束后再添加一个 01, 例如对于 $E(a) = 010$, $E'(a) = 00110001$

这种编码显然有性质 0: 01 不会出现在任何编码的奇数-偶数位置, 即 $\forall k \in N, E'(a_{2k+1})E'(a_{2k+2}) \neq 01$

试证 prefix-free 性: 假设 $E'(a)$ 是 $E'(b)$ 的前缀, 由于 01 标识了 $E'(a)$ 和 $E'(b)$ 的结束, 且由于性质 0, 很明显有 $E'(a) = E'(b)$, 那么 $E(a) = E(b)$, 且 E 是单射, 则 $a = b$, 于是 prefix-free 得证。 \square

结合两个引理可以得到一个结论:

💡 定理

如果存在 one-to-one 的 $E : A \rightarrow \{0, 1\}^*$, 那么便存在 one-to-one 的 $E' : A^* \rightarrow \{0, 1\}^*$

这是很重要的, 对于数学元素, 如果我们可以给数字做编码, 那么就可以编码向量, 再运用一次定理就能编码矩阵, 然后是更高维的张量。

1.1.2 编码与可数的关系

🔥 下面四条等价

1. A 是可数的
2. A 是有限的, 要么存在一个双射 $f : A \rightarrow N$
3. 存在单射 $g : A \rightarrow N$
4. 存在满射 $h : N \rightarrow A$

引理 1.1.3

$\{0, 1\}^*$ 是可数的

证明. 对 $\{0, 1\}^*$ 的元素进行这样的排序: $e, 0, 1, 00, 01, 10, 11 \dots$

即先按长度排序, 内部再按二进制数大小排序。

那么定义从字符串映射到排序后序号的函数 $f : \{0, 1\}^* \rightarrow N: \forall x \in \{0, 1\}^*, f(x) = \begin{cases} 0 & \text{if } x=e \\ 2^{|x|} + (x \text{ 对应的二进制数大小}) & \text{if } |x| \geq 1 \end{cases}$

可理解为组的序号+组内的序号, 这是一个单射, 于是可数。 □



这个引理也引导出一个定理, 即可数性和单射编码的关系:

💡 定理

A 是可数的当且仅当存在单射 $E : A \rightarrow \{0, 1\}^*$

编码的设定完备后, 我们面临的问题 (Problem) 就抽象为了一个从二进制串到二进制串的函数了 (即对输入和输出都做编码)

计算模型

2.1 布尔电路 (Boolean Circuit)

可以可以, 问题 (Problem) 的输入输出已经被我们编码, 可以投诸于计算了——那么, 计算的具体步骤, 或者说方法, 是什么呢?

这一节里探讨的一类问题/函数统称为有限函数 (Finite Function), 其输入输出均为固定长度的一个二进制串, 即 $f : \{0, 1\}^n \rightarrow \{0, 1\}^m$

一个很经典的有限函数计算模型即布尔电路 (Boolean Circuit, 我去, 计算机系统 1), 显然对于与门 (AND) 和或门 (OR) 而言, $n = 2, m = 1$, 非门的 n 为 1, 别的门电路可类比; 另一个例子是 MAJ 函数, 即对于 $n = 3$ 的输入的每一位, 如果 1 占多数, 则输出 1, 否则输出 0

一个『电路』还是太具体了, 不利于更本质的计算理解——将一个布尔电路 C 抽象为一个有向无环图 G , 它包含如下节点 (Nodes):

- n input nodes: 记为 $X[0], X[1], \dots, X[n - 1]$, 均没有入度且出度至少为 1
- s gates: 即逻辑门, 根据种类有不同的度
 - ▶ 一个电路 C 的 size 定义为 $|C| = s$
- m output nodes: 记为 $Y[0], Y[1], \dots, Y[1]$

舒服了, 可以利用门电路模拟从输入到输出的计算过程了: 对于输入长为 n 的 $x \in \{0, 1\}^n$, 令其第 i 位即为 input nodes 中的 $X[i - 1]$, 输出答案 $y \in \{0, 1\}^m$ 同理。

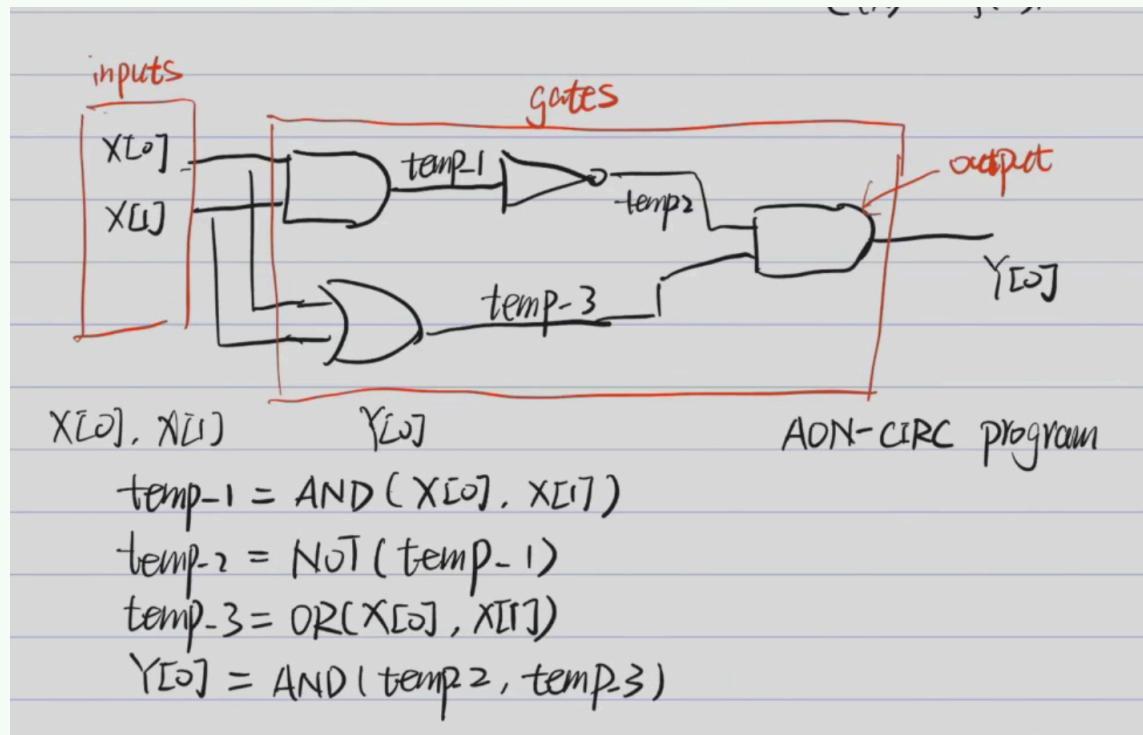
定义 2.1.1

记 $C(X) = (Y[0], Y[1], \dots, Y[m - 1])$, 如果 $\forall x \in \{0, 1\}^n, C(x) = f(x)$, 则称电路 C 计算了函数 f .



现在还亟待一种对计算过程的书面化描述, 即, 这个电路的每一步, 信号 (或者说数据) 流过每一个门时得到了什么中间结果? 于是 [Anonjae](#)—[Anon-Circ](#) AON-Circ (AND-OR-NOT Circuit) Program 登场了。

定义 2.1.2 (AND-OR-NOT Circuit Program)



对于图中的电路，将其每个中间逻辑门的输出保存为一个 temp 变量，便得到了下方的多行 (Lines)，这就是 AON-Circ Program

不难发现每一行对应一个逻辑门的计算过程，于是 Program 的行数与其对应电路中逻辑门的个数相同。此时 Program 的行数即为电路的 size

AON-C Program 计算了某个函数的定义与上方电路 C 计算函数类似，不再赘述。

实际上程序和电路这种对应关系就是等价的。

定理 2.1.3

一个函数可被一个有 s 个逻辑门的布尔电路计算，当且仅当它可以被 s 行的 AON-C Program 计算。

2.1.1 NAND 电路 (NAND Circuit)

接下来介绍一种门：NAND，即 NOT(AND)，易知 AON 三者都可以只用 NAND 实现，于是可以搭建一个仅由 NAND 门构成的电路。

NAND Circuit	\Leftrightarrow	AON Circuit
s gates	\rightarrow	$\leq 2s$ gates, for NAND decomposes to NOT(AND)
$\leq 3s$ gates, for NAND(NAND(a,a),NAND(b,b))	$\equiv \leftarrow$	s gates

相类似的，有 NAND-Circ Program，其每一行都形如 `foo = NAND(bar,blah)`。

定理 2.1.4

Boolean Circuit, AON-Circ Program, NAND-Circ Program, NAND Circuit 四者的转化只需要 s 的常数倍（即 $\Theta(s)$ ）

**定理 2.1.5**

$\forall n, m, f : \{0, 1\}^n \rightarrow \{0, 1\}^m$, 都存在一个布尔电路计算这个函数, 其含有 $O(m \cdot n \cdot 2^n)$ 个逻辑门。

证明. 对于输出的某一位 $Y[j]$, 其计算情况可枚举如下表:

$X[0]$	\cdots	$X[n - 1]$	$Y[j]$
0	\cdots	0	some value
0	\cdots	1	some value
\cdots	\cdots	\cdots	\cdots
1	\cdots	1	some value

写出 $Y[j]$ 的具体计算式, 用析取范式表示: $Y[j] = (\dots \wedge \dots \wedge \dots) \vee (\dots \wedge \dots \wedge \dots) \vee \dots \vee (\dots \wedge \dots \wedge \dots)$, 其中共有 2^n 个括号, 对应上表中的 2^n 行, 每一行通过合取计算出一种情况下的 $Y[j]$, 再析取得到 $Y[j]$ 的具体表达; 每个括号中共有 n 项, 对应 $X[0]$ 到 $X[n - 1]$ 本身或取反再进行合取, 于是得到 $n \cdot 2^n$

而 Y 的长度为 m , 于是得到 $O(m \cdot n \cdot 2^n)$



满足上述定理, 即可以计算任意函数的电路, 称为通用 (universal) 的; 要判断一个函数集合 (化成的电路) 是否是通用的, 只需要判断其是否能计算 NAND。

2.2 计算规模

可以可以, 已经可以用电路/程序计算某个函数/问题了, 那么对于某个函数, 我们需要多少个门的电路, 多少行的程序, 这是可以估量的吗? 由上面的定理似乎已经有了上界: $O(m \cdot n \cdot 2^n)$ (myc 剧透: 其实可以把数量打到 $O(\frac{m \cdot 2^n}{n})$)。我们来从 NAND 电路入手。

 ADD Function

试设计电路程序，计算 $\text{ADD} : \{0, 1\}^{2n} \rightarrow \{0, 1\}^{n+1}$, $\text{ADD}(x_0, \dots, x_{2n-1}) = x_0 \dots x_{n-1} + x_n \dots x_{2n-1}$

解.

```
def ADD(X[0], ..., X[2n-1]):
    Result = [0] * (n+1)
    Carry = [0] * (n+1)
    for i in range(n):
        Result[i] = XOR(Carry[i], XOR(X[i], X[i+n]))
        Carry[i+1] = MAJ(Carry[i], X[i], X[i+n])
    Result[n] = Carry[n]
    return Result
```

XOR 和 MAJ 函数只需要常数行的 NAND-Circ Program 实现，经过 n 次循环，于是该加法函数的规模（行数）即为 $O(n)$

这是很好的，计算规模与输入串的长度成正比。

 MUL Function

乘法基于加法，摆了。

规模可以不断优化: $O(n^2) \rightarrow O(n^{\log_2 3}) \rightarrow$ even better.

 LOOKUP Function

试设计电路程序，计算 $\text{LOOKUP} : \{0, 1\}^{2^k+k} \rightarrow \{0, 1\}$, 具体而言，输入分为两段: 2^k 位作为表， k 位视作一个最大可表达 $2^k - 1$ 的二进制数 i ，输出为表中的第 i 位。

解. 利用归纳思想， $k = 1$ 时：

```
def LOOKUP_1(X[0],X[1],i[0]):
    if i[0] == 0:
        Y[0] = X[0]
    else:
        Y[0] = X[1]
```

对于 k 时的情形

```
def LOOKUP_k(X[0],X[1],...,X[2^k-1],i[0],...,i[k-1]):
    if i[0] == 0: # 根据剩下的 i 查 X 的前半段
        Y[0] = LOOKUP_(k-1)(X[0],...,X[2^(k-1)-1],i[1],...,i[k-1])
    else:
        Y[0] = LOOKUP_(k-1)(X[2^(k-1)],...,X[2^k-1],i[1],...,i[k-1])
```

于是有规模 $\begin{cases} L(k) = C + 2L(k-1), \\ L(1) = O(1) \end{cases}$, 解得 $O(2^k)$

2.2.1 优化效率

现在来探讨如何将某函数 (n input, m output) 计算的规模从 $O(m \cdot n \cdot 2^n)$ 优化至 $O(\frac{m \cdot 2^n}{n})$ 。

我们用上面介绍过的 LOOKUP 函数来理解，对于如下函数计算的过程真值表：

$X[0]$	\cdots	$X[n - 1]$	$Y[j]$
0	\cdots	0	some value
0	\cdots	1	some value
\cdots	\cdots	\cdots	\cdots
1	\cdots	1	some value

进行计算时可以看作是将 $X[0] - X[n - 1]$ 作为输入的二进制数索引 i ，原函数输出 Y 的某一位 $Y[j]$ 在所有情况的 X 下的输出列作为输入的表（有 2^n 个取值可能性，即上表中的 2^n 行），从而计算 $Y[j]$ 确定取得的值）

例如对如下函数 g 的真值表：

$X[0]$	$X[1]$	$Y[0]$
0	0	1
0	1	0
1	0	1
1	1	1

命 $G_0 = 1, G_1 = 0, G_2 = 1, G_3 = 1$ （即函数输出的 2^2 中情况），那么 $g(X[0], X[1]) = \text{LOOKUP}_2(G_0, G_1, G_2, G_3, X[0], X[1])$

接下来分析这种方法的效率：对于抽出输出的 2^n 种情况，每种 0 或 1 利用 ONE/ZERO 函数便可用常数行实现，于是共有 $O(2^n)$ 行；而 LOOKUP 函数的规模也为 $O(2^n)$ ，于是计算 $Y[j]$ 的规模为 $O(2^n)$ ；而 Y 有 m 位，于是总规模优化为 $O(m \cdot 2^n)$

⚠ Incomplete Section

接下来是如何将规模进一步优化到 $O(\frac{m \cdot 2^n}{n})$ ，但我没听咋懂 myc 的讲解，于是 skip 了

2.2.2 编码程序

现在考虑将 Circ-Program 编码为字符串形式：就一个有 s 行的 NAND-Circ Program 而言，其每一行都是 `foo = NAND(bar,blah)` 的形式，因此只需要考虑编码其中所有的变量（易知涉及的变量个数不超过 $3s$ 个），且每一行都可以看作是一个三元组 (triple)：`(foo, bar, blah)`，于是共得到 s 个三元组。

（以 $3s$ 个变量为例）将这些变量记作 $0, 1, 2, 3, \dots, n - 1, n, \dots, n + m - 1, \dots, 3s - 1$ ，其中前 n 个为对应函数的 input 变量 ($X[0], X[1], \dots, X[n - 1]$)，中间 m 个为 output 变量 ($Y[0], Y[1], \dots, Y[m - 1]$)，其余为中间变量。

于是每一个变量都可以编码为一个长度为 $\lceil \log(3s) \rceil$ (hey where does this come from) 的二进制串，只需要按照程序顺序拼接每个三元组中变量的编码，便得到了一个长度为 $3s \cdot \lceil \log(3s) \rceil$ 的二进制串。

编码完成了——但是，为什么要做这件事，难道说？没错，有了字符串编码的程序，我们甚至能将这个程序作为另一个程序的输入 🤖，例如下面这个例子。



EVAL Function

函数 $\text{EVAL}_{s,m,n} : \{0,1\}^{3s[\log(3s)]+n} \rightarrow \{0,1\}^m$ 是对一个 NAND-Circ Program p (含 s 行, 计算 $f : \{0,1\}^n \rightarrow \{0,1\}^m$) 和一个输入 $x \in \{0,1\}^n$ 的计算 (或者说, 运行), 其输入的前 $3s \cdot [\log(3s)]$ 位为 p 的编码, 后 n 位为 input x , 输出为 $f(x)$.

定理 2.2.1 (EVAL 的规模)

$\forall s, n, m, \exists$ NAND-Circ Program $U_{s,n,m}$ 计算函数 $\text{EVAL}_{s,m,n}$, 且其规模为 $O(s^2 \cdot \log s)$



2.2.3 更长的输入: Infinite!

更一般的计算方法无疑是针对任意长度输入的函数 $f : \{0,1\}^* \rightarrow \{0,1\}^*$, 然而这是无法利用已经讲述过的布尔电路/程序来计算的 (回顾: 我们的方法是, 使用一个真值表, 记录每组输入 $X[0] - X[n-1]$ 下 $Y[m]$ 的取值)

定理 2.2.2

$$\text{For every } F : \{0,1\}^* \rightarrow \{0,1\}^*, BF = \begin{cases} F(x)_i & \text{if } i < |F(x)|, b=0 \\ 1 & \text{if } i < |F(x)|, b=1 \\ 0 & \text{if } i \geq |F(x)| \end{cases}$$

使用 BF 计算 F 只需要遍历 0 到 $|F(x)| - 1$ 即可, 反过来只需要取每一位



语言与自动机

3.1 DFA 与正则语言

显然一个布尔函数 $f : \{0,1\}^* \rightarrow \{0,1\}$ 和这样一个集合对应: $A_f = \{x \in \{0,1\}^* \mid f(x) = 1\}$, 这个集合称为语言 (Language); 此时显然有 $f(x) = 1$ 当且仅当 $x \in A_f$

于是计算函数的值 $f(x) = ?$ 与判断 $x \in ? A_f$ 是等价的。

于是来研究语言的性质 (自己觉得突兀不 (流汗

考虑累加 x 的每一位再模二的 XOR 算法, 可得到以下流程图:

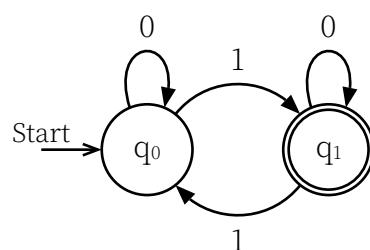


图 1 假设 q_1 为末态, 表示输出为 1

定义 3.1.1

这被称作一个确定性有限自动机 (Deterministic Finite Automaton, DFA)，它可以看作一个四元组 (K, s, F, δ) ，其中：

- K - 一组状态的有限集合
- $s \in K$ - 初始状态
- $F \subseteq K$ - 末态（接受状态）的集合
- $\delta : K \times \{0, 1\} \rightarrow K$ - 状态转移函数



假设有输入字符串 $x_0x_1\dots x_{n-1}$, DFA 的计算过程为： $s_0 = s, s_1 = \delta(s_0, x_0), s_2 = \delta(s_1, x_1), \dots, s_n = \delta(s_{n-1}, x_{n-1})$, 然后判断 $s_n \in F$, 如果是则接受 (accept), 否则拒绝 (reject)。

定义 DFA 计算某个函数的方法： M computes f if M accepts x 当且仅当 $f_x = 1$

同时定义 DFA Decides 某个语言的方法： M decides A_f if M accepts x 当且仅当 $x \in A_f$

被 DFA 决定的语言称为正则语言 (Regular Language)。

定理 3.1.2

某台 DFA M 可判断的语言有且仅有一个，即 $L(M) = \{x \in \{0, 1\}^* \mid M \text{ accepts } x\}$



练习. 证明空集， $\{0, 1\}^*$ ， $\{e\}$ ， $\{w \in \{0, 1\}^* \mid w \text{ 有一个字串是 } 101 \text{ (何意味)}\}$ 是正则语言
解.

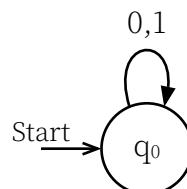


图 2 空集，使可接受集为空即可

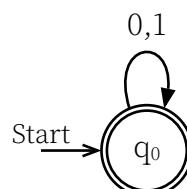


图 3 $\{0, 1\}^*$ ，使所有状态均为可接受状态即可，这里简洁起见就弄一个状态

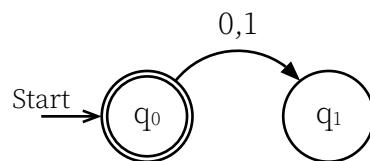


图 4 $\{e\}$ ，使初始状态为唯一可接受状态即可

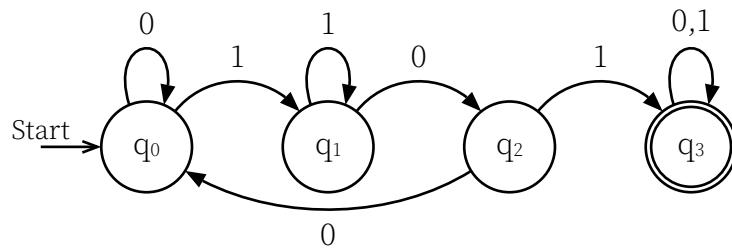


图 5 $\{w \in \{0,1\}^* \mid w \text{ 有一个字串是 } 101\}$, 设计状态转移使得读到连续的 101 时进入可接受状态即可

定理 3.1.3

如果 A 和 B 都是正则的, 那么 $A \cup B$ 是正则的。(对新 DFA 的每一个成员进行探究即可)



3.2 NFA

| 你和雪有一个共同点。

定义 3.2.1

不确定性有限自动机 (Nondeterministic Finite Automaton, NFA), 如下是其与 DFA 的区别:

- 状态转移函数中, 下一个状态可以有多个
- 读入空串也可导致状态转移

可见是在状态转移时有所不同; 实际上这里的状态转移比“函数”要更一般, 即关系。

$$N = \{K, s, F, \Delta(\text{transition relation})\}, \Delta \subseteq K \times \{0, 1, e\} \times K$$

NFA 做计算的过程也比 DFA 宽松一些: 能读入字符串输入的全部并且“有一条路可以走通并走到可接收状态”(可理解为并行计算的)即可。



练习. 设计 NFA, 计算 $W = \{w \in \{0,1\}^* \mid w \text{ 的倒数第二位是 } 1\}$

解.

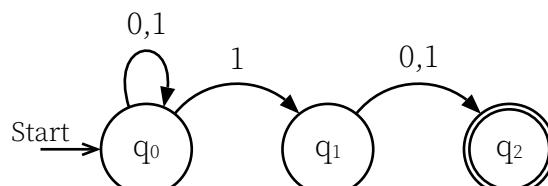


图 6 计算 $W = \{w \in \{0,1\}^* \mid w \text{ 的倒数第二位是 } 1\}$

NFA 看起来可以猜测出一条正确的路径, 似乎比 DFA 更强大一些, 但实际上两者是等价的。

定理 3.2.2

DFA 能判断一个语言等价于 NFA 也能判断这个语言。

证明. DFA 本身就是一台特殊的 NFA, 这个方向是显然的。

反过来的证明, 想法是让 DFA 模拟 NFA 的 tree-like 计算过程。下面先研究一个 NFA 的情况

对于 $p \in K$, 定义 $E(p) = \{p\} \cup \{q \mid (p, e, q) \in \Delta\}$ (即 p 和不读入 symbol 即可到达的状态组成的集合); 假设该 NFA 从某层 (tree-like arch) 状态集合 Q 接收一个 0 到达下一层 Q' , 那么 $Q' = \bigcup_{q \in Q} \bigcup_{p \in (q, 0, p)} E(p)$

接下来用 DFA 模拟这个过程。很明显其每个元素有如下表示:

- 状态集合 $K' = \{Q \mid Q \subseteq K\}$
- 初始状态 $s' = E(s)$
- accepting states $F' = \{Q \mid Q \subseteq K, Q \cap F \neq \emptyset\}$
- transition function $\delta(Q, a) = \bigcup_{q \in Q} \bigcup_{p \in (q, a, p)} E(p)$

□

因此正则语言也可以使用 NFA 来判断。

**定理 3.2.3**

如果 A 和 B 都是正则的, 那么 AB (拼接, Concatenation) 是正则的。(构造一个新 DFA, 其状态为原两个 DFA 状态的笛卡尔积, 且仅当两个状态均为可接受状态时新状态才为可接受状态)

证明. 总有一个 A 和 B 的交界处, 其前面的字串用 NFA_A 判断, 后面的字串用 NFA_B 判断。交界处由 NFA “猜测”, 通过 e 切换。

□

**定理 3.2.4**

如果 A 是正则的, 那么 $A^* = \{a_1 a_2 \dots a_k : k \geq 0 \text{ and } a_i \in A\}$ (即从 A 中选取若干串做拼接) 是正则的。

证明. 对于 NFA_A , 每次跑完一个字串后利用 “猜测” 功能返回其初态。

□



3.3 正则表达式

每种正则语言可以用正则表达式 (Regular Expression, RE) 表示:

定义 3.3.1

- Base Case: $0, 1, \emptyset$ are REs, 分别对应 $L(0) = \{0\}, L(1) = \{1\}, L(\emptyset) = \emptyset$
- 如果 R 和 S 是 REs, 那么以下也是 REs:
 - $(R \cup S)$, 对应 $L(R \cup S) = L(R) \cup L(S)$
 - (RS) , 对应 $L(RS) = L(R)L(S)$ (recall it is concatenation)
 - (R^*) , 对应 $L(R^*) = (L(R))^*$
 - 括号书写时很麻烦, 记录一个算数优先级 precedence: $*$ > concatenation > \cup



例子

Language	RE
$\{e\}$	\emptyset^*
$\{w \in \{0, 1\}^* : w \text{ starts with } 0 \text{ and culminates with } 1\}$	$0(0 \cup 1)^*1$
$\{w \in \{0, 1\}^* : w \text{ 至少包含两个 } 0\}$	$(0 \cup 1)^*0(0 \cup 1)^*0(0 \cup 1)^*$

定理 3.3.2

语言是正则的当且仅当其可以被某个正则表达式表示。

练习. 画出 $(01 \cup 0)^*$ 对应的 NFA。

解.

RE	NFA
0	
1	
01	
$\{01 \cup 0\}^*$	

证明. 上面的练习可以总结出正则表达式向 NFA 的转化。从 NFA 向正则表达式的转化采用状态消除法 (State Elimination Method):

给定 NFA $N = (K, s, F, \Delta)$, 由以下步骤得到正则表达式 R :

- 将 N 转化为 N'
 - N' 中没有状态指向其的 initial state (新建一个初态, 通过 e transition 指向原先的初态)
 - N' 的 accepting state 只有一个, 且该状态不指向任何状态 (新建一个 accepting state, 使原先的若干 accepting states 通过 e transition 指向它)
- 删除 N' 中的非初态非末态状态, 直到只剩下初态和末态; 删除方法遵循下述规则 (q 为将被删除的状态) :

NFA Component	RE (by State Elimination)
	ab
	ac^*b
	$a(c \cup d)^*b$

□

上述的状态删除法实质上是一个动态规划问题的求解。(我觉得不考)

3.3.1 Pumping Theorem

现在要证明某个语言是正则的, 可以利用 DFA/NFA/RE 三种等价的方法证明, 但要证明某个语言不是正则的, 就比较困难了 (需要说明所有的 FA/RE 都无法表示该语言?)。

于是介绍 Pumping Theorem, 其说明了正则语言的一个性质: 其中有些 Pattern 是可以重复的。

定义 3.3.3 (Pumping Theorem)

设 A 是一个正则语言, 那么存在一个整数 (Pumping length) $p \geq 1$, 使得对于任意字符串 $s \in A$ 且 $|s| \geq p$, 都可以将 s 分解为 $s = xyz$, 满足:

1. $\forall i \geq 0, xy^i z \in A$ (重复 Pattern y 任意次, 字符串仍在语言中)
2. $|y| \geq 1$ (可重复的 Pattern 非空)
3. $|xy| \leq p$ (可以在前 p 个字符中找到该 Pattern)

证明. 由于 A 正则, 于是存在 DFA M decides A , 命 $p = \text{number of states in } M$, 即状态总数

对于 $w \in A, |w| \geq p$, 考虑其被 DFA 计算的过程: $q_0 \rightarrow q_1 \rightarrow \dots \rightarrow q_{n-1} \rightarrow q_n$, q_0, q_n 分别为初态和接受状态, 由于要处理 w 的至少 p 位, 必然有 $n \geq p$, 即计算经过的状态数 至少为 p ; 与此同时, 不同状态的总数只有 p 个, 于是由鸽巢原理, 在经过的状态中, 一定存在 i, j , 使得 $0 \leq i < j \leq p$ 且 $q_i = q_j$

于是在状态 q_0 到 q_j 部分处理的字符串记为 x , 在 q_i 到 q_j 部分处理的字符串记为 y , 在 q_j 到 q_n 部分处理的字符串记为 z , 则有 $s = xyz$.

由于 $q_i = q_j$, 也就是说处理 y 的状态转换部分可以重复无数次; 而 $|xy| = j - i \leq p$, 且 $|y| = j - i \geq 1$, 于是满足 Pumping Theorem 的要求。 \square

练习. 说明 $\{0^n 1^n : n \geq 0\}$ 不是正则的。

证明. 假设其是正则的, 则存在 pumping length p 。

考虑字符串 $w = 0^p 1^p$, 它应当可以被分为 $w = xyz$ 且满足三条性质。

由于 $|xy| \leq p$, 于是仅有可能 $xy = 0^k, k \leq p$, 且 y 不为空串, 那么 $y = 0^k$

但这样一来, $xy^i z = 0^{k-k'+ik'} 1^p$, 显然对于某些 i , 新得到的字符串不在该语言中, 与第一条性质矛盾。 \square

3.4 Pushdown Automaton

我们已经介绍过 DFA, NFA, Regex 三种等价的计算模型, 现在做出的考虑是: 向其中添加一些新的 feature, 以计算更多的函数, 决定更多的语言。

比方说, 在 DFA/NFA 中, 状态切换时只是单纯地转移到下一个状态, 但并没有“记忆”已经处理过的信息; 如果除了 FA 的四元组外, 能有一个数据结构来存储一些信息, 或许能计算更多的函数。(这篇文章介绍了一个 PDA 在实际游戏开发中的例子: <https://zhuanlan.zhihu.com/p/575664217>)

定义 3.4.1 (PDA)

Pushdown Automaton(PDA) = NFA + stack

PDA 是一个四元组 $P = (K, s, F, \Delta)$ 其中 K, s, F 和 NFA 中的定义一样, $\Delta \subseteq (K \times \{0, 1, e\} \times \{0, 1\}^*) \times (K \times \{0, 1\}^*)$ (其中第一部分的 $\{0, 1\}^*$ 表示栈顶将被 pop 出来的字符串, 第二部分的 $\{0, 1\}^*$ 表示将被 push 到栈内的字符串) (何意味?)

e.g. 对于 $(q, a, \gamma, p, \beta) \in \Delta$, 表示 PDA 在状态 q 下读入字符 a , 且栈顶为 γ 时, 可以转移到状态 p , 并将栈顶的 γ pop 出, push 入 β 。

似乎这么看来, 每次进行 transition 时, 都需要从栈里 pop 出一些东西, 再 push 入一些东西, 但实则可以用形如 $(q, a, \gamma, p, \beta\gamma)$ 的 relation 来表示“只 push 入而不 pop 出”的操作。



由此可以看出, PDA 中状态的转换通常还伴随着栈的操作, 而不仅仅只是先前两种 FA 那样的状态转换; 因此方便起见, 定义一个新的“配置”(Configuration), PDA 通过 relation 在不同的配置之间进行转换, 这个配置也称为即时描述 (Instantaneous Description, ID)。

这个配置相当于一种新的“状态”, 思考一下, 其与什么有关?

- 当前 PDA 所处的状态 $q \in K$
- 剩余未处理的输入字符串 $w \in \{0, 1\}^*$
- 栈内当前的字符串 $\alpha \in \{0, 1\}^*$, 栈顶在左侧

定义 3.4.2 (Configuration of PDA)

A Configuration ID $C \in K \times \{0, 1\}^* \times \{0, 1\}^*$, i.e. $C = (q, w, \alpha)$



如果一个 PDA P 从配置 C_1 只通过一步转移到配置 C_2 , 则记作 $C_1 \xrightarrow{P} C_2$; 对于不限制次数的转移, 记作 $C_1 \xrightarrow{*} C_2$ 。

定义 3.4.3 (Decision of PDA)

PDA P accepts string $w \in \{0, 1\}^*$ if $\exists q \in F$ such that $(s, w, e) \xrightarrow{*} (q, e, e)$, 即走到可接受态 & 栈空 & 输入处理完

$$L(P) = \{w \in \{0, 1\}^* \mid P \text{ accepts } w\}, P \text{ decides } L(P)$$

$L(P)$ 称为上下文无关语言 (Context-Free Language, CFL)。





找出下面 CFL 的 PDA

1. $L = \{w \in \{0,1\}^* \mid \text{in } w, \text{ number of 0s is equal to number of 1s}\}$
 - $K = F = \{q\}$
 - $s = q$ (只有一个状态作为初态和可接受态)
 - $\Delta = \{$
 - $((q, 0, e), (q, 0)), \# \text{ push 0 onto empty stack when reading 0}$
 - $((q, 0, 0), (q, 00)), \# \text{ push another 0 onto stack when reading 0 and stack top is 0}$
 - $((q, 0, 1), (q, e)), \# \text{ pop 1 from stack when reading 0 and stack top is 1}$
 - $((q, 1, e), (q, 1)), \# \text{ push 1 onto empty stack when reading 1}$
 - $((q, 1, 1), (q, 11)), \# \text{ push another 1 onto stack when reading 1 and stack top is 1}$
 - $((q, 1, 0), (q, e)), \# \text{ pop 0 from stack when reading 1 and stack top is 0}$
 - }
 - 易知，每次在栈顶为 1/0 的情况下读到 0/1 就会抵消掉栈顶的 1/0，于是要达成最终栈空的条件，必然需要 0 和 1 数量相等
2. $L = \{ww_{\text{Reversal}} \mid w \in \{0,1\}^*\}$
 - $K = \{l, r\}$, l 表示正在读取左半部分, r 表示正在读右半部分
 - $s = l$
 - $F = \{r\}$
 - $\Delta = \{$
 - $((l, 0, e), (l, 0)), \# \text{ Push whatever read while we are on the former half}$
 - $((l, 1, e), (l, 1)), \# \text{ Push whatever read while we are on the former half}$
 - $((l, e, e), (r, e)), \# \text{ NFA guess when we reach the middle and transit to latter}$
 - $((r, 0, 0), (r, e)), \# \text{ Pop out what's there, if we read the same on the latter half}$
 - }

3.5 语法 (Grammar)

接下来学习一个全新的概念：语法 (Grammar)。先前的三种 FA 都是用于判断某个字符串是否属于某个语言，而语法则用于生成 (generate) 某个语言中的字符串；主要介绍上下文无关语法 (Context-Free Grammar, CFG)。

定义 3.5.1 (CFG)

A CFG $G = \{V, S, R\}$, where:

- V is a finite set of symbols, including 0,1
- $S \in V - \{0,1\}$: start symbol
- $R \in (V - \{0,1\}) \times V^*$: rules (must be finite)
 - ▶ 为表示方便，一般将规则写作 $A \rightarrow \alpha$, 其中 $A \in V - \{0,1\}, \alpha \in V^*$

” 引述

为了推导出一个 CFG 所描述的语言中的字符串，我们从起始符号开始，不断地将变量 A 用它的某一个产生式的体来替代，从而得到所有的字符串。这里， A 的产生式指的是以 A 为头的产生式。

— https://fla.cuijiacai.com/04-cfg/#_4-1-2-%E5%BD%A2%E5%BC%8F%E5%8C%96%E5%AE%9A%E4%B9%89

为了解析 rules 的推导过程，定义推导（Derivation）：

定义 3.5.2 (Derive)

if $A \rightarrow \alpha \in R$, then $\beta A \gamma \Rightarrow \beta \alpha \gamma$, 其中 $\beta, \gamma \in V^*$

对于若干次的推导，类似的，表示为 $\stackrel{*}{\Rightarrow}$



接下来就可以定义 CFG 生成语言的方法： G generates $w \in \{0, 1\}^*$ if $S \stackrel{*}{\Rightarrow} w \& L(G) = \{w \in \{0, 1\}^* \mid G \text{ generates } w\}$



Find CFG for these languages

- $\{w \in \{0, 1\}^* \mid w = w_{\text{Reversal}}\}$, 即回文句

- 考虑到 $w = 0u0$ or $1u1$ where $u = u_{\text{Reversal}}$
- R: $S \rightarrow e | 1 | 0 | 1S1 | 0S0$

3.6 上下文无关语言 (Context-Free Language, CFL)

大的来了：上面两小部分讲述的 PDA 和 CFG 其实是等价的计算模型！即一个语言被某台 PDA 决定当且仅当它可以被某个 CFG 生成。

定理 3.6.1

CFG \Leftrightarrow PDA



证明.

1. 试证 $\text{CFG} \Rightarrow \text{PDA}$

- Given CFG $G = (V, S, R)$, 构造 PDA $P = (K, p, F, \Delta)$, where:
 - $K = \{p, q\}$
 - $F = \{q\}$
 - $\Delta = \{(p, e, e), (q, S)\}, \# \text{ Push start symbol onto empty stack}$
 - $((q, e, A), (q, \alpha))$ for each $A \rightarrow \alpha \in R$, $\# \text{ Replace non-terminal A on top of stack with alpha}$
 - $((q, a, a), (q, e))$ for each $a \in \{0, 1\}$, $\# \text{ Pop out what's on stack if it matches the read input}$
 - }

2. 试证 $\text{PDA} \Rightarrow \text{CFG}$

- Given PDA $P = (K, s, F, \Delta)$, 先将其转化为 simple 的。

定义 3.6.2 (Simple PDA)

A PDA $P = (K, \Delta, s, F)$ is simple, if

- $|F| = 1$ (可接受态只有 1 个)
- $\forall ((p, a, \alpha), (q, \beta)) \in \Delta$, either $\alpha = e$ & $|\beta| = 1$, or $|\alpha| = 1$ & $\beta = e$ (每次 transition 要么只 push 一个 symbol, 要么只 pop 一个 symbol)

定理 3.6.3 (PDA => simple PDA)

这里介绍通过以下几个步骤如何将 PDA 转化为 simple PDA:

- 1) if $|F| > 1$, 回忆小节 3.3 部分从 NFA 转为 RE 的 proof, 创造一个新的可接受态 f , $\forall q \in F$, 创造一个新的 transition $((q, e, e), (f, e))$, 然后令 $F = \{f\}$ 即可。
- 2) for $((p, a, \alpha), (q, \beta)) \in \Delta$,
 - if $|\alpha| \geq 1 \& |\beta| \geq 1$, split the transition into two via adding a intermediate state r : $((p, a, \alpha), (r, \beta))$ and $((r, e, e), (q, e))$, then we go to next cases
 - if $|\alpha| > 1 \& \beta = e$, split the pop operation into $|\alpha| - 1$ steps via adding intermediate states $r_1, r_2, \dots, r_{|\alpha|-1}$: $((p, a, \alpha[0]), (r_1, e)), ((r_1, e, \alpha[1]), (r_2, e)), \dots, ((r_{|\alpha|-1}, e, \alpha[|\alpha|-1]), (q, e))$
 - if $\alpha = e \& |\beta| > 1$, split the push operation just like above
 - if $\alpha = \beta = e$, also split into 2 via adding an intermediate state r : $((p, a, 0), (r, e))$ and $((r, e, e), (q, 0))$ (push 0 再 pop 出来)

- 接下来把 simple PDA 转化为 CFG: Given simple PDA $P = (K, s, \{f\}, \Delta)$, 构造 CFG $G = (V, S, R)$
 - 先定义 V : symbol set, 由于期望中, G 要生成的语言就是 PDA 所决定的, 而 P 决定一个语言又是对其中所有的 $w \in \{0, 1\}^*$ 进行从初态到可接受态的转换, 因此 V 中需要包含所有可能的 state pairs: $(p, q) \in K \times K$, 由于 CFG 处理的是 symbol, 需要将这个 pair 表示为 symbol, 即 $V = \{0, 1\} \cup \{A_{pq} \mid (p, q) \in K \times K\}$
 - 接下来先不着急继续寻找其他两个元素的定义, 想一想我们的目标: 将这些符号 A_{pq} produce 出某些 $w \in \{0, 1\}^*$, 并且令其与“ w 也可通过 P 的判断得到”这个陈述是等价的。严谨阐述就是 $A_{pq} \xrightarrow{*} w$ for some $w \in \{0, 1\}^* \Leftrightarrow w \in \left\{ u \in \{0, 1\}^* \mid (p, u, e) \xrightarrow{P} (q, e, e) \right\}$
 - 显然我们会发现一个很特殊的 symbol A_{sf} , 它对应的语言正是 PDA P 决定的语言, 考虑到 CFG 做的本职工作就是把 S 阐释为一种语言, 因此使这两种语言相同, 令 $S = A_{sf}$ 即可。





构造 PDA/CFG 的小技巧

- 来自助教复习课
- 一般而言，题目都形如：证明某个语言 $L = \{w \in \{0, 1\}^* \mid \text{some property on } w\}$ 是上下文无关的。解题过程就是构造判定/生成这种语言的 PDA/CFG。
- 可以先写出语言中串的特殊形式，例如 $w = u\square 101u^r\square$ ，其中方框表示为任意串；然后套用下面的模板：
- 任意串对应的操作：
 - PDA：不修改栈，利用非确定性只随便读字符且不改变状态： $\Delta + = \{(q, 0, e), (q, e), ((q, 1, e), (q, e))\}$
 - CFG： $A \rightarrow 1A|0A|e$

图灵机 (Turing Machine)



合味道

我要提一个以著名计算机科学家名字命名的，一个屈居于以初代校长为名的扫码学院下的神秘班级了。

尽管 PDA/CFG 已经比 FA 强大了不少，但它们仍然无法计算所有的函数/决定所有的语言，例如， $\{0^n 1^n 0^n \mid n \geq 0\}$ ，于是，隆重端出图灵机 (Turing Machine, TM)。



引述

The “granddaddy” of all models of computation is the Turing machine.

— introtcs.org

想象一根从一端开始无限长的纸带 (Tape)，上面划分为一个个方格 (Cell)，每个方格中可以写入一个 symbol；同时有一个读写头 (Head) 可以在纸带上左右移动，并且可以读出/写入当前所在方格的 symbol：Symbol 有四种：1, 0, \triangleright , \emptyset ，最后两种分别是纸带的开头和表示该 Cell 为空的字符

该纸带还附带一个状态控制器 (State Controller)，读写头每到一个 Cell 时，根据当前状态和该 Cell 中的 symbol，可以完成下面两个操作：

- Write a symbol to the current Cell
- Move the head one cell to the left or right/Stay/Halt

定义 4.1 (Turing Machine)

A Turing Machine $M = (K, \Sigma, s, \delta)$, where:

- K is a finite set of states
- $s \in K$ is the initial state
- Σ is a finite set of symbols, $\{0, 1, \triangleright, \emptyset\} \subseteq \Sigma$
- δ is “Transition Function”
 - $\delta : K \times \Sigma \rightarrow K \times \Sigma \times \{\text{MoveLeft}, \text{MoveRight}, \text{Stay}, \text{Halt}\}$, 即根据当前状态和当前 Cell 中的 symbol, 输出下一状态、写入的 symbol、读写头的操作。



直接来看该模型是怎么接收字符串的: 对于 input $x = x_0x_1\dots x_{n-1}$, TM 先将纸带的前 n 个 Cell 写为 $\triangleright x_0x_1\dots x_{n-1}$ ($T[0] = \triangleright, T[1] = x_0, T[2] = x_1, \dots, T[n] = x_{n-1}$), 其余 Cell 均为 \emptyset , 接下来记初态 $s = 0$, 且开始从 \triangleright 后的 $i = 0$ 位置开始读取字符串, 重复以下几步:

- 根据当前 Cell 的内容和当前状态 q , 通过 $\delta(q, T[i]) = (q', a, \text{Direction})$ 得到下一状态 q' 、写入 symbol a 和读写头的操作。(对于开头的 \triangleright , 要求其唯一且不可覆盖, 即 $a == \triangleright$ 当且仅当 $T[i] == \triangleright$)
- 写入 Symbol, 转移状态: $T[i] = a, q = q'$
- switch Direction
 - if Direction = MoveRight, $i = i + 1$
 - if Direction = MoveLeft, $i = \max(0, i - 1)$
 - if Direction = Stay, $i = i$
 - if Direction = Halt, stop

💡 提示

停机后, TM 会输出一个东西: $M(x) = T[1]T[2]\dots T[i]$ where $T[i+1]$ 是第一个 \emptyset 。

如果接受完了 input 还没有停机, 则记 $M(x) = \perp$

定义 4.2 (Computation via TM)

TM M computes function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ if $\forall x \in \{0, 1\}^*, M(x) = f(x)$

注意这要求了 TM 必须对所有输入停机; 可被 TM 计算的函数称为可计算函数 (Computable Function)。



4.1 图灵完备

使用一些常见的编程语言都可以实现上面介绍的 TM 计算步骤, 这些语言被称为图灵完备 (Turing Complete) 的语言, 即其可以完成图灵机做的事情。

现在从另一个方向开展研究: 图灵机可以做到这些个编程语言能完成的任务吗?

4.1.1 NAND-TM

就一种名叫 NAND-TM 的编程语言, 其可以看作先前介绍的 NAND-CIRC 的扩展: 允许使用数组 (Array) 存储任意长度的字符串, 并且允许使用循环 (循环次数与输入长度有关, 不一定为常数的 Loop) 来处理这些数组。

- NAND-TM 中有两种数据类型:

- i : integer value
- other: boolean value
- 有两种变量类型:
 - scalar: boolean variable
 - array: array of boolean variables (can be infinite long)
 - All arrays are accessed only and simultaneously by i
- 其输入 X 和 输出 Y 都是 array 类型的变量
- 程序最后一行是 MODANDJUMP(a,b) : modify i and jump back to the first line.
 - if $a = 1$ and $b = 1$, $i = i + 1$
 - if $a = 0$ and $b = 1$, $i = i - 1$
 - if $a = 1$ and $b = 0$, $i = i$
 - if $a = 0$ and $b = 0$, halt
- 所有变量（除输入 X 外）初始均为 0

于是在程序每轮执行中，对 $A[i], B[i] \dots$ 等进行 NAND 操作，并且通过 MODANDJUMP 来控制 i 的变化，进行循环处理。

定理 4.1.1 (TM & NAND-TM)

TM \Leftrightarrow NAND-TM，并且其元素有如下对应关系

TM Element	NAND-TM Element
finite states	scalar variables
tape	array variables
head position	integer variable i

证明.

- 左推右:
 - 假设有 TM $M = (K, \Sigma, s, \delta)$, 构造 NAND-TM 程序 P :
 - 使用 $\lceil \log_2(|K|) \rceil$ 个 scalar 变量来表示 TM 的状态；使用 $\lceil \log_2(|\Sigma|) \rceil$ 个 array 变量来表示 TM 的 tape；两个 scalar 变量表示读写头的四个操作（可编码为 00, 01, 10, 11）
 - 那么 relation $\delta : K \times \Sigma \rightarrow K \times \Sigma \times \{\text{MoveLeft}, \text{MoveRight}, \text{Stay}, \text{Halt}\}$ 转化为了 function $\delta' : \{0, 1\}^{\lceil \log_2(|K|) \rceil + \lceil \log_2(|\Sigma|) \rceil + 2} \rightarrow \{0, 1\}^{\lceil \log_2(|K|) \rceil + \lceil \log_2(|\Sigma|) \rceil + 2}$, 这是一个有限函数，也就是说，可以被一个 NAND-CIRC program 计算。
 - 记其计算得到的结果最后两位为 a, b , 则在程序最后加上 MODANDJUMP(a,b) , 即得到与原 TM 对应的 NAND-TM。
- 右推左:
 - 假设有 NAND-TM 程序 P , 构造 TM
 - k 个 scalar 变量, 对应 TM 的 2^k 个状态
 - l 个 array 变量, 那么字符集数量 $|\Sigma| = 2^l$
 - 对于 $f : \{0, 1\}^{k+l} \rightarrow \{0, 1\}^{k+l}$ 的这个 NAND-TM Program, 可转化为 $g : \{0, 1\}^{k+l} \rightarrow \{0, 1\}^{k+l} \times \{\text{MoveLeft}, \text{MoveRight}, \text{Stay}, \text{Halt}\}$ 的 TM Transition Function



再加一点语法糖:

- GOTO 语句: `GOTO L`, 表示跳转到 `line` 为 L 的行继续执行。
- Multi-Indexing: 允许使用多个 index 来访问不同 array 的不同位置
- Multi-Dimensional Arrays: 允许使用多维数组

4.1.2 NAND-RAM

TM 可以再次进化为 RAM, 实际上现代编程语言一般都基于 RAM 模型。

纸带记为 Memory, 每个 Cell 可以存储一个整数而非一个字符, 并且有 k 个寄存器 (Register), 每个寄存器可以存储一个 integer value。

1. 可以 Write/Read Memory Cell, 并且是指哪就能访问哪, 不需要读写头
2. 可以对寄存器进行加减乘除等算术运算
3. 可以根据判断结果进行下一步操作

同样, RAM 也对应一种 NAND-RAM 编程语言, 其有 integer 变量 (有最大值) 和 index 访问的 array; 还有许多算术运算指令。

可以证明 $\text{NAND-TM} \Leftrightarrow \text{NAND-RAM}$

定理 4.1.2 (Church-Turing Thesis)

图灵机是终极的计算模型。



4.2 通用图灵机 (Universal Turing Machine)

引述

The universal machine/program - “one program to rule them all” .

— introtcs.org

现在已经有了最强最厉害的计算模型了, 我们希望找到一台能够完成所有任务的图灵机 (这被称为 Universality), 而非对每个函数都寻找一个专用的 TM。

命这样的一台图灵机为 $U(M, x) = \begin{cases} M(x) & \text{if } M \text{ is encoding of a TM} \\ M, x & \text{Otherwise} \end{cases}$

为了证明 U 的存在性, 需要先说明如何对图灵机 ($M = (K, \Sigma, s, \delta)$) 进行编码:

- 状态集 $K \rightarrow 0, 1, \dots, |K| - 1$, 命编码为 0 的为初态 s
- 字符集 $\Sigma \rightarrow 0, 1, \dots, |\Sigma| - 1$
- 转移函数 $\delta : K \times \Sigma \rightarrow K \times \Sigma \times \{\text{MoveLeft}, \text{MoveRight}, \text{Stay}, \text{Halt}\}$, 表示为五元组的集合 $\{(CurrentState, CurrentSymbol, NextState, WrittenSymbol, Direction)\}$, 这集合里的元组数量为 $|K| \times |\Sigma|$, 对每个元组中的每个元素进行编码即可。

em, 然后对于任意的输入 x , 只需要就上面的编码方式进行 relation 的查询, 即可利用输入的图灵机进行计算; 也就是说, U 是存在的。

4.3 函数的可计算性 (Computability)

现在, 不止有最强的计算模型, 我们甚至有 rule them all 的通用图灵机了——那么, 世界上所有的问题, 都可以被图灵机解决吗?

定理 4.3.1 (Uncomputability of some functions)

存在一个布尔函数 $F : \{0, 1\}^* \rightarrow \{0, 1\}$, 其无法被任何图灵机计算。

证明.

- 理论上由如下两点即可证明
 - set of boolean functions: uncountable
 - set of Turing machines: countable (because TMs can be coded)
- 尝试构造该 $F: F(x) = \begin{cases} 0 & \text{if } x \text{ is encoding of a TM } M \text{ and } M(x)=1 \\ 1 & \text{otherwise} \end{cases}$
- 即 $F(x) = 0$ 时, 有一 TM 将自己的编码作为自己的输入且输出 1
- 则 $\forall \text{ TM } M$, 记其编码为 m
 - 如果 $M(m) = 1$, 则 $F(m) = 0$
 - 如果 $M(m) \neq 1$, 则 $F(m) = 1$
- 由于 M 并不是在所有相同的输入下都和 F 有相同的输出, 那么 F 无法被 M 计算, 即无法被任何图灵机计算。

□



呃呃, 这会不会有种钦定的感觉? 感觉这个函数是专门找出来驳倒图灵机似的——然而, 确实有很多有价值的、但是被发现是不可计算的函数。

4.3.1 停机问题

定义 4.3.2 (Halting Problem)

Given a TM M and input $x \in \{0, 1\}^*$, does M halt on x ?

该问题对应函数为 $\text{Halt}(M, x) = \begin{cases} 1 & \text{if } M \text{ is a TM that halts on } x \\ 0 & \text{otherwise} \end{cases}$



定理 4.3.3 (Uncomputability of Halt)

$\text{Halt}(M, x)$ is uncomputable.

证明. 在小节 4.3 部分已经构造出了一个不可计算的函数 F , 因此利用反证法, 试证下面这个陈述成立: “如果 $\text{Halt}(M, x)$ 可被某个 TM M_{Halt} 计算, 那么 F 也可计算”。

即: 利用已知的神秘 Oracle TM M_{Halt} 来计算 F 。

这台 TM M_F 的工作流程如下:

1. 对输入 x , 先利用 M_{Halt} 计算 $\text{Halt}(x, x)$
2. if $\text{Halt}(x, x) = 0$, 说明 TM M_x 在输入 x 时不会停机/ x 根本就不是一台 TM 的编码, 于是 $F(x) = 1$
3. if $\text{Halt}(x, x) = 1$, 说明 TM M_x 在输入 x 时会停机, 于是
 - 1) run x on TM M_x until it halts
 - 2) if $M_x(x) == 1$, 由 F 的定义可知 $F(x) = 0$; else $F(x) = 1$

综上, TM M_F 可以计算 F , 与 F 不可计算矛盾, 因此 $\text{Halt}(M, x)$ 也是不可计算的。 □



证明停机问题无法计算后, 可以通过其证明更多不可计算的函数/问题了。

 Halt Zero

Given a TM M , does M halt on input 0?

该问题对应函数 $\text{HALTONZERO}(M) = \begin{cases} 1 & \text{if } M \text{ is a TM that halts on 0} \\ 0 & \text{otherwise} \end{cases}$

证明. 类似上一个定理证明中采用的反证法, 只需要证明: “如果这个函数可计算, 那么停机问题也可计算”(由于已知 Halt 是不可计算的, 那么可以直接得到这个函数的不可计算性)。

类似的, 假设神秘 Oracle TM $M_{\text{HALTONZERO}}$ 可以计算 $\text{HALTONZERO}(M)$, 那么可以构造 TM M_{HALT} 来计算停机问题, 具体而言, 通过下面几步证明:

1. 构造 TM N 使得 M 在 x 上停机当且仅当 N 在输入 0 上停机, 其中 M, x 是 M_{HALT} 的输入
 - 1) N 是这样的: 忽略输入, 直接 run M on input x ; 易验证 M 在 x 上停机当且仅当 N 在 0 上停机
2. 运行 $M_{\text{HALTONZERO}}(N)$ 就可以判断 N 是否在输入 0 上停机; 且 $M_{\text{HALT}}(M, x) = M_{\text{HALTONZERO}}(N)$, 于是可以计算停机问题。
3. 但由于停机问题不可计算, 因此 HALTONZERO 也是不可计算的。

□

 Zero Function

Given a TM M , is $M(x) = 0 \forall x \in \{0, 1\}^*$?

该问题对应函数 $\text{ZEROFUNC}(M) = \begin{cases} 1 & \text{if } M \text{ is a TM that computes the zero function} \\ 0 & \text{otherwise} \end{cases}$

证明. 依旧, 假设 M_{ZERO} 可以计算 $\text{ZEROFUNC}(M)$, 那么试构造 TM M_{HALT} 来计算停机问题:

1. 构造 TM N 使得 M 在 x 上停机当且仅当 N 计算 ZERO 函数
 - 1) N : run M on input x ; return 0.
2. Run $M_{\text{ZEROFUNC}}(N)$ 来判断 N 是否计算 ZERO 函数; 且 $M_{\text{HALT}}(M, x) = M_{\text{ZEROFUNC}}(N)$, 于是可以计算停机问题。
3. 但由于停机问题不可计算, 因此 ZEROFUNC 也是不可计算的。

□

上面两个例子中采用的证明过程是类似的: 构造新问题涉及的中间件 TM N 使得 $\text{HALT}(M, x) = \text{TargetTM}(N)$, 即从停机问题转换为目标问题的图灵机, 这被称为归约 (**reduction**)。

定义 4.3.4 (Reduction)

A Reduction from F to G is a **computable** function $R : \{0, 1\}^* \rightarrow \{0, 1\}^*$ such that $\forall x \in \{0, 1\}^*, F(x) = G(R(x))$.

引理 4.3.4

If G is computable, then F is computable.

\Leftrightarrow

If F is uncomputable, then G is uncomputable.



Rice's Theorem

Given a TM M , does M have property P ? (P 是一个布尔函数)

该问题对应函数 $P(M) = \begin{cases} 1 & \text{if } M \text{ is a TM that has property } P \\ 0 & \text{otherwise} \end{cases}$

Rice's theorem: 如果 P 是 semantic & non-trivial 的, 那么 P 是不可计算的。

两台图灵机 M_1, M_2 是 functionally equivalent 的, 如果 $\forall x \in \{0, 1\}^*, M_1(x) = M_2(x)$ 。

一个性质 $P : \{0, 1\}^* \rightarrow \{0, 1\}$ 是 semantic 的, 如果对于任意两台 functionally equivalent 的图灵机 M_1, M_2 , 有 $P(M_1) = P(M_2)$

一个性质 P 是 trivial 的, 如果 P 是 constant function

证明. https://introtcs.org/public/lec_08_uncomputability.html#ricethmsec

□

4.3.2 Recursion Theorem

定义 4.3.6 (Recursion Theorem)

For any TM T that takes input $\langle M \rangle$ (the encoding of a TM) and x , there exists a TM R such that $\forall x \in \{0, 1\}^*, R(x) = T(\langle R \rangle, x)$



4.3.3 Gödel's Incompleteness Theorem

我赌不考

哦对了, 接下来的部分是本课程最混沌的环节

运行时间

在纠结完“所有问题的可计算性”后, 接下来我们着手于“可计算问题的计算效率”——即运行时间 (Running Time)。

定义 5.1 (Running Time of a TM)

$T : N \rightarrow N$ is a function and the Running Time of TM M is $T(n)$ if

- 对于任意充分大的 n 和任意长度为 n 的输入 $x \in \{0, 1\}^n$, TM M 都在 $T(n)$ 步内停机

注意: $T(n)$ 只界定了该图灵机的运行时间上界, 可以理解为 $T(n)$ 实际上表示的是 DS 课程里那个 $O(T(n))$ 。

有了运行时间, 就可以得到某个问题/函数的计算时间, 并据此对函数进行分类。

定义 5.2 (满足计算时间为 $T(n)$ 的函数集合)

$\text{Time}_{\text{TM}}(T(n)) =$

{boolean function F: F is computable by some TM with running time $T(n)$ }

比如说, 显然有 $\text{Time}_{\text{TM}}(10 \times n^3) \subseteq \text{Time}_{\text{TM}}(2^n)$

通过图灵机的运行时间界定函数, 在图灵机离现代语言差距太远的情况下是很没有意义的, 考虑到在小节 4.1 中提到的 $\text{TM} \Leftrightarrow \text{NAND-TM} \Leftrightarrow \text{NAND-RAM}$, 而 NAND-RAM Program 的运行更类似现代语言, 下面定义 NAND-RAM Program 的运行时间:

定义 5.3 (Running Time of a NAND-RAM Program and a function set)

$T : N \rightarrow N$ is a function and the Running Time of NAND-RAM Program P is $T(n)$ if

- 对于任意充分大的 n 和任意长度为 n 的输入 $x \in \{0, 1\}^n$, NAND-RAM Program halts on input x after executing $T(n)$ lines.

$\text{TIME}_{\text{RAM}}(T(n))$

= {boolean function F: F is computable by some NAND-RAM Program with running time $T(n)$ }

事实上 “等价” 的关系是 roughly 的, 有如下定理:

定理 5.4 (Time_{TM} vs Time_{RAM})

For any function $T : N \rightarrow N$ where $T(n) \geq n$,

$$\text{TIME}_{\text{TM}}(T(n)) \subseteq \text{TIME}_{\text{RAM}}(10 \times T(n)) \subseteq \text{TIME}_{\text{TM}}(T(n)^4)$$

证明可以通过对比 TM 和 NAND-RAM Program 的运行步骤数/行数来完成。

接下来的 TIME 全都默认为 TIME_{RAM} 。

哦对了, 这里还有两个比较特别的函数类: 在多项式时间内可计算的函数类 $\mathcal{P} = \bigcup_{c \in N} \text{TIME}(n^c)$ 和在指数时间内可计算的函数类 $\mathcal{EXP} = \bigcup_{c \in N} \text{TIME}(2^{n^c})$ 。显然有 $\mathcal{P} \subsetneq \mathcal{EXP}$ 。

5.1 TIME Hierarchy Theorem

先来一个在之后的证明中会用到的美妙工具:

定理 5.1.1 (Running Time of universal NAND-RAM Program)

我们已经知道有了通用图灵机 $U(M, x)$, 可以模拟任意图灵机 M 在输入 x 上的计算过程并得到相同的结果, 即 $U(M, x) = M(x)$; 类似能自然的想到肯定也存在(由于 TM 和 NAND-RAM Program 是等价的) 通用 NAND-RAM Program $U_{\text{RAM}}(P, x)$, 使得 $U_{\text{RAM}}(P, x) = P(x)$ 。可以证明 (课上没讲怎么证明) 如果 P 的运行时间为 $T(n)$, 那么 U_{RAM} 的运行时间为 $a|P|^b T(n)$, 其中 a 和 b 是神秘常数。

接下来来个有关 Running Time 的定理, 如果时间长了, 能计算的函数必然多了。

定理 5.1.2 (TIME Hierarchy Theorem)

For every nice function $T : N \rightarrow N$, 在函数集合 $\text{TIME}(T(n) \log(n)) \setminus \text{TIME}(T(n))$ 中都存在一个布尔函数 F 。

即, 将函数的计算时间从 $T(n)$ 提升到 $T(n) \log(n)$, 必定可以多计算至少一个函数 F 。

证 明 . 固 定 一 个 nice 函 数 $T : N \rightarrow N$, 定 义 $\text{HALT}_T(P, x) = \begin{cases} 1, & \text{if } |P| \leq \log \log|x| \text{ and } P \text{ is a NAND-RAM Program which halts on } x \text{ in } 100T(|P| + |x|) \text{ steps} \\ 0, & \text{otherwise} \end{cases}$

这是一个 bounded halting problem (里面的 100 和 $\log \log|x|$ 可以调, 此处为了方便而取这么奇怪的值), 即计算程序 P 是否能在 $100T(|P| + |x|)$ 这么长的时间内停机

接下来证明: 这个函数 HALT_T 就是我们想要的 F 。即证:

- $\text{HALT}_T \in \text{TIME}(T(n) \log(n))$
- $\text{HALT}_T \notin \text{TIME}(T(n))$

证明 $\text{HALT}_T \in \text{TIME}(T(n) \log(n))$:

- 直接写一个程序/函数来计算 HALT_T :
 1. if $|P| > \log \log|x|$, return 0 # 花费 $O(|P|)$ 时间
 2. 计算 $T_0 = T(|P| + |x|)$ # 花费 $O(T(|P| + |x|))$ 时间
 3. Simulate P on input x for $100T_0$ steps # 即利用 Universal NAND-RAM Program 来模拟 P 的运行, 花费 $a|P|^b 100T_0$ 时间
 4. if P halts during the simulation, return 1; else return 0

$$\text{Total Time} = O(|P| + T(|P| + |x|) + a|P|^b 100T_0) = O(|P|^b T_0)$$

由 于 $|P| \leq \log \log|x| \leq \log \log(|P| + |x|)$, 那 么 $O(|P|^b T_0) \leq O(T(|P| + |x|)(\log \log(|P| + |x|))^b) = o(T(|P| + |x|) \log(|P| + |x|))$, 命 $n = |P| + |x|$ 即证。

证明 $\text{HALT}_T \notin \text{TIME}(T(n))$:

我他妈真懒得写了

□



5.2 从有穷函数到无穷函数 (运行空间)

Ω 提示

在计算模型章节中, 有穷函数 (布尔函数) 与 NAND-CIRC 程序是一一对应的, 且前者的规模由其中的逻辑门数量决定, 后者的规模由其中的行数决定。刚刚介绍的无穷函数与 NAND-

RAM 程序也是一一对应的，且后者的规模由更抽象的 #steps 决定。回忆如果（计算布尔函数 F 的）程序有 S 行，那么其规模为 S ，记为 $F \in \text{SIZE}(S)$ 。

现在我们试图从运行时间开始探究无穷函数的规模，这可以从构建其与有穷函数的联系入手：

对于 $F \in \{0, 1\}^* \rightarrow \{0, 1\}$ 规定 $F_{\uparrow n} : \{0, 1\}^n \rightarrow \{0, 1\}$ where $F_{\uparrow n}(x) = F(x)$ for all $x \in \{0, 1\}^n$ 。

定义如下两点：

- $F_{\uparrow n} \in \text{SIZE}(T(n))$, 如果计算 $F_{\uparrow n}$ 的布尔电路所需的逻辑门数量为 $T(n)$ (That is, 计算其的 NAND-CIRC program 的行数最多为 $T(n)$)
- $F \in \text{SIZE}(T(n))$, 如果 $\forall n, F_{\uparrow n} \in \text{SIZE}(T(n))$

💡 提示

注意, $F \in \text{SIZE}(T(n))$ 并不代表 F 可以被某个图灵机计算。

一个自然的想法是比对一下 $F \in \text{SIZE}(T(n))$ 和 $F \in \text{TIME}(T(n))$ 的异同：前者是，对于每个可能的输入大小，都用 $T(n)$ 行程序运算，后者是，图灵机计算 $T(n)$ 步后停机；难道说，这两者是一个意思吗？

哈哈，不是这样的。

$$\begin{aligned} F \in \text{Size}(T(n)) &\Leftrightarrow \forall n, \exists \text{NAND-CIRC}, \forall x \in \{0, 1\}^n, F(x) \text{ can be} \\ &\quad \text{program } P \qquad \qquad \qquad \text{computed by } f \\ &\quad \leq T(n) \text{ line} \\ F \in \text{TIME}(T(n)) &\Leftrightarrow \exists \text{NAND-TM}, \forall n, \forall x \in \{0, 1\}^n, F(x) \text{ can be computed} \\ &\quad \text{program } P \qquad \qquad \qquad \text{in } T \end{aligned}$$

图 15 Difference between Size and Time

可以看到，二者定义最大的区别是 $\forall n$ 和 \exists NAND-CIRC/TM 的先后顺序，即 Size 对某个 n ，对于固定长度为 n 的输入都可以有不同的机器/程序来计算，而 Time 则要求对特定的一台机器/程序，对它而言所有长度的输入都可以计算。

误会解除了，那么， $\text{SIZE}(T(n))$ 和 $\text{TIME}(T(n))$ 这两个函数集合之间有什么大小关系吗？还真有！你可以看到，由于前者中的函数对于每个 n ，都有各自的独属的神秘程序来计算，而后者中的函数只能用一台（或一些）神秘机器来计算，即，要加入后者集合，对函数的要求更高，因此其中的函数更少： $\text{TIME}(T(n)) \subseteq \text{SIZE}(T(n))$

接下来趁着 SIZE 的余味，再介绍一个函数类： $P_{/\text{poly}} = \bigcup_{c \in N} \text{SIZE}(n^c)$ ，由 TIME 和 SIZE 的关系可知 $\mathcal{P} \subsetneq P_{/\text{poly}}$ （真包含是因为，后者中存在一些不可计算的函数）。

下面是一个属于 $P_{/\text{poly}}$ 但不属于 \mathcal{P} 的函数例子：

定义 5.2.1 (Unary Halting Function)

$UH(x) = \text{HALTONZERO}(S(|x|))$; 其中 $S : N \rightarrow N$ 将输入的数转化为二进制表示再删去最高位的 1 最后返回。

UH is uncomputable, 证明通过把 HALTONZERO 归约到 UH 完成。



5.3 Difficulty Levels of Functions

我们有了一堆从里到外包着的 TIME 函数类 (由其 running time $T(n)$ 划分界限); 然而现实是, 没有可行的方法来判断, 某个函数究竟属于哪一层, 即, 无法判断某个问题究竟有多难。

定义 5.3.1 (3 SAT Problem)

Given a 3 CNF formula φ , φ 可满足吗?

该问题对应函数 $3SAT(\varphi) = \begin{cases} 1 & \text{if } \varphi \text{ is satisfiable} \\ 0 & \text{otherwise} \end{cases}$



定义 5.3.2 (01 EQ Problem)

给定一系列线性等式 (方程), 问该方程组是否有可行解

该问题对应函数 $01EQ(A, B) = \begin{cases} 1 & \text{if } A \text{ has a row equal to a row in } B \\ 0 & \text{otherwise} \end{cases}$



定义 5.3.3 (Subset Sum)

Given n integers x_1, x_2, \dots, x_n and a target integer B , is there a subset of $\{x_1, x_2, \dots, x_n\}$ that sums to B ?



这三个问题就是典型的看起来很难但是没法用 TIME 划分难度的函数。退而求其次, 我们尝试找一个很难的函数, 然后证明这些函数都比那个函数简单, 或者说差不多难度。

有关“证明”, 考虑到我们之前用过的归约 (Reduction) 工具, 其可以判断不同函数之间的难度, 但是那个难度指的是“是否可计算”, 而现在我们划分难度用的是 Running Time, 因此需要对归约进行改进, 加一些条件。

定义 5.3.4 (Polynomial-time Reduction)

Polynomial-time Reduction 相比于前面的 Reduction 只多了一个条件: R 是可以在多项式时间内计算的。此时 $F(x) = G(R(x))$ 也记作 $F \leq_P G$ 。



引理 5.3.5 (Polynomial-time Reduction Lemma)

If $F \leq_P G$ and $G \in \mathcal{P}$, then $F \in \mathcal{P}$.

If $F \leq_P G$ and $G \leq_P H$, then $F \leq_P H$.



接下来对上面提到的三个问题试着规约到一些已知的神秘函数。

定理 5.3.6

$$3SAT \leq_P 01EQ$$

证明. https://introtcs.org/public/lec_12_NP.html#reducing-3sat-to-zero-one-and-quadratic-equations □

$$3SAT \leq_P \text{Subset Sum}$$

证明. https://introtcs.org/public/lec_12_NP.html#the-subset-sum-problem □



5.3.1 Verifiability

在可计算性之外，这里介绍一个可验证性。关注上面的三个问题 (3SAT, 01EQ, Subset Sum)，如果说它们可以被计算，即 {存在一串赋值使得 φ (from 3SAT) 可被满足，使得 01EQ 的方程组成立，使得 Subset Sum 的子集和为 B }，那么我们该如何说明，这些赋值，或者说解，是什么？答案显而易见：直接把这些 x_i 的赋值写出来就是了，然后对于 3SAT，直接把赋值代入 φ 看是否成立；对于 01EQ，直接把对应的行列写出来看是否相等；对于 Subset Sum，直接把解加起来看是否等于 B 。并且，如上的这些操作都是多项式时间内可以完成的，我们说这些问题都是多项式时间可验证的。

定义 5.3.7 (Polynomial-time Verifiability)

$F : \{0, 1\}^* \rightarrow \{0, 1\}$ 是多项式时间内可验证的，如果存在一个 Running Time 是 Polynomial 的 TM V 使得 $\forall x \in \{0, 1\}^*, F(x) = 1 \Leftrightarrow \exists t \in \{0, 1\}^*, s.t. \begin{cases} V(x, t) = 1 \\ |t| \leq |x|^a \end{cases}$

V 又称作验证机 (Verifier)， t 称作证据 (Proof/Certificate)。

可以这么理解， $F(x) \equiv 1$ 也就说明这是一个“真”的 statement，那么存在一个长度合适的证据 t 能够被多项式时间验证机 V 调用，完成证明（得到输出 1）（证明的过程就类似上面举例的那些，什么代入赋值比较，什么把解加和比较），且如果能够证明，那么 $F(x) = 1$ 也是 genuine 的。满足这个性质的函数就是多项式时间可验证的函数。



来个例子：如果 F 是 3SAT，那么 $V(x, t)$ 是这样的（作为证据，很明显 t 应当是一组赋值；且 x 应当是一个 3-CNF formula）：

1. if x 不是一个 3-CNF formula 或者 t 不是一组合法的 assignment, return 0
 2. else, 把 t 代入 x 算一下，如果 x 在这个 assignment 下成立, return 1; else return 0
- 显然 V has a polynomial running time，并且 V 输出 1 确实和 3SAT 可被满足等价。

接着，隆重介绍迄今最神秘的函数类：

$\mathcal{NP} = \{\text{boolean function } F: F \text{ is polynomial-time verifiable}\}$ 。

引理 5.3.8

$$\mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{EXP}$$

证明. 先证明前半部分。

$\forall F \in \mathcal{P}$ 都存在一台图灵机 M 在多项式时间内计算 F , 那么目标是构造一台验证机 V 。

idea: 注意验证机最显著的特点就是它要在 Proof 的协助下对 x 输出 1 (即进行验证), 而我们现在的 M 直接能扛住计算这个任务 (而根本不需要 Proof)。因此, 我们可以让验证机 $V(x, t)$ 忽略 Proof:

1. Run M on input x # Polynomial Time
2. return $M(x)$ # Constant Time

于是得到多项式时间验证机 V , 那么 $F \in \mathcal{NP}$, 那么 $\mathcal{P} \subseteq \mathcal{NP}$ 。

接下来证明后半部分。

$\forall F \in \mathcal{NP}$ 都存在一台验证机 V 在多项式时间内验证 F , 那么目标是构造一台图灵机 M 。

idea: 枚举所有可能的 Proof t , 并利用验证机 V 来验证 $F(x)$ 是否为 1:

$M(x)$:

1. enumerate all strings t where $|t| \leq |x|^a$ (formally, $\forall t \in \{0, 1\}^*$ with $|t| \leq |x|^a$) # Exponential Time, specifically $O(2^{|x|^a})$
 - 1) Run $V(x, t)$ # Polynomial Time
 - 2) if **some** $V(x, t) = 1$, return 1 # Constant Time
2. return 0 # Constant Time

于是得到 Exponential Time 计算机 M , 那么 $F \in \mathcal{EXP}$, 那么 $\mathcal{NP} \subseteq \mathcal{EXP}$ 。

□



并且我们已经知道 $\mathcal{P} \subsetneq \mathcal{EXP}$, 那么上面两个包含关系至少有一个是真包含——就这了, 我们只能得到这么多了, 对于 \mathcal{P} 是否等于 \mathcal{NP} , 以及 \mathcal{NP} 是否等于 \mathcal{EXP} , 我们无从得知。

❓ Conjecture

如今的猜测是: $\mathcal{P} \neq \mathcal{NP}$ 且 $\mathcal{NP} \neq \mathcal{EXP}$, 如果相等的话, 那么计算和验证的难度就相等了, 这是反直觉的。

——至少, 证明 $\mathcal{P} = \mathcal{NP}$ 是很难的一件事, 在下面的 Theorem 中我们会提到: 这件事的难度堪比证明 $3SAT \in \mathcal{P}$ 。

先来点开胃小定义。

定义 5.3.9 (Prelude: NP-Completeness & NP-Hardness)

A function $F : \{0, 1\}^* \rightarrow \{0, 1\}$.

如果 $\forall G \in \mathcal{NP}, G \leq_P F$, 那么 F 是 NP-Hard 的, 即满足所有 NP 问题都可以被规约到它。

如果 F 既是 NP-Hard 的又 $\in \mathcal{NP}$, 那么 F 是 NP-Complete 的 (即 F 在 NP 问题中是最难的那一档)。



由 NPC 的定义可立刻得到下面的引论：

引理 5.3.10

F 是一个 NPC 问题，那么如果 $F \in \mathcal{P}$ ，那么 $\mathcal{P} = \mathcal{NP}$ 。



接下来这节课智云没有画面（只能赌这块不考了）😅 其实有画面我也觉得世界上没有人能听得懂呃呃

Cook-Levin Theorem: 3SAT is NP-Complete

proof: <https://zhuanlan.zhihu.com/p/73234959>

 **Ordreality**
写的很好，完全理解了
2020-10-24

 **酷勒米**
太好了，细看
2020-04-10

图 16 理解在哪？

定理 5.3.11

$3SAT \in \mathcal{P} \Leftrightarrow \mathcal{P} = \mathcal{NP}$



5.4 Probability: Randomized Computation

哎呦我去，这不是我最擅长、最喜欢、荣誉课里最宜人的概率论（括弧 H 括回）吗？

其实我这一章本来打算直接不学了，myc 搞这胡加新东西呢。但是考虑到上面的 Cook-Levin Theorem 智云没画面，还是在这记点什么，聊以自慰。

对某些问题，使用随机的计算方法，在允许出错概率不超过某个值的情况下，可以以更低的时间复杂度来完成计算。

现在要如何将随机性 apply 到已经介绍过的那些计算模型上呢？下面考虑 polynomial-running-time 的 NAND-TM Program：

```
? = NAND(?, ?)
...
? = NAND(?, ?)
MODANDJUMP(?, ?) # total num of lines is poly(|x|)
```

给普通的 NAND-TM Program 添加一条新指令：`? = RAND()` (`RAND` 函数等概率返回 0 或 1，然后将其赋值给某个变量)，得到 RNAND-TM Program。

相对应的，我们熟知的图灵机可以根据当前状态 p 和读入 symbol a 来唯一确定下一状态 q 、写入 symbol b 和读写头移动方向 d ，即 $\delta(p, a) = (q, b, d)$ ；那么现在我们可以让图灵机的转移函

数变为 $\delta(p, a) = \begin{cases} (q_1, b_1, d_1) \\ (q_2, b_2, d_2) \end{cases}$, 即对于某个状态和读入 symbol, 可以有两个以等概率选取的转移结果, 然后通过掷硬币来决定采用哪个转移结果, 得到 Probabilistic Turing Machine。

有了 PTM, 接下来可以定义函数类了:

$$\text{BPP} = \left\{ F : \{0, 1\}^* \rightarrow \{0, 1\} \mid \exists \text{ an RNAND-TM Program } P \right. \\ \left. \text{s.t. } \forall x \in \{0, 1\}^*, P \text{ halts within } a|x|^b \text{ steps and } \Pr(P(x) = F(x)) \geq \frac{2}{3} \right\} \quad (1)$$

即, 计算该类函数的概率图灵机在输入长度的多项式时间内停机, 且输出正确结果的概率至少为 $\frac{2}{3}$ 。这里的 $\frac{2}{3}$ 其实可以取任何在 $(\frac{1}{2}, 1)$ 之间的数值, 下面这个引理说明了原因: 只要是大于 $\frac{1}{2}$ 的概率, 就可以通过一种办法放大这个数值直到逼近 1.

引理 5.4.1 (Amplification)

$\forall F : \{0, 1\}^* \rightarrow \{0, 1\}$, 如果存在一个多项式时间的随机算法(即 RNAND-TM Program) P 使得 $\forall x \in \{0, 1\}^*$, $\Pr(P(x) = F(x)) \geq p$ for some $p \in (\frac{1}{2}, 1)$, 那么一定存在另一个多项式时间的随机算法 Q 使得 $\forall x \in \{0, 1\}^*$ and for every $n \in \mathbb{N}$, $\Pr(Q(x) = F(x)) \geq 1 - \frac{1}{2^{n^m}}$ 。

证明. Q :

1. run P on input x for $2k$ times
2. return the majority value among these $2k$ outputs

这样构造 Q , 我们可以获悉: 如果 Q 失败, 那么超过半数(即至少 k 次) P 的输出都是错误的。注意到每次跑 P 都是一次独立重复试验, 那么有

$$\begin{aligned} \Pr(Q \text{ Fails}) &= \Pr(P \text{ Fails at least } k \text{ times}) \\ &= \sum_{i=k}^{2k} \Pr(P \text{ Fails } i \text{ times}) \\ &= \sum_{i=k}^{2k} C_{2k}^i p^{2k-i} (1-p)^i \\ &\leq \sum_{i=k}^{2k} C_{2k}^i p^k (1-p)^k \left(\text{since } p > \frac{1}{2} \text{ and } i \geq k \right) \quad (2) \\ &= p^k (1-p)^k \sum_{i=k}^{2k} C_{2k}^i \\ &\leq p^k (1-p)^k 2^{2k} \\ &= (4p(1-p))^k \\ &\xrightarrow{k \rightarrow \infty} 0 \left(\text{since } 1 > p > \frac{1}{2} \text{ and thus } 4p(1-p) < 1 \right) \end{aligned}$$

令 $k = \frac{n^m}{-\log_2(4p(1-p))}$ 即得到 $\Pr(Q \text{ Fails}) < \frac{1}{2^{n^m}}$, 那么 $\Pr(Q(x) = F(x)) \geq 1 - \frac{1}{2^{n^m}}$ 。 \square



Another aspect of BPP: 现在, 以一个全新的角度考虑 RNAND-TM Program 的那条新指令: `? = RAND()`, 其在每次执行该函数时都随机生成一个随机的 0 或 1. 但我们也一样操作: 在整个程序开始前先生成一个随机 01 串 r , 然后每次执行 `? = RAND()` 时都从 r 中依次取出一个 bit 来使用。这种方法和之前是等价的, 可以看作是某个更大的程序同时接受了原先的输入 x 和一个随机字符串 r 作为输入然后进行计算; 于是得到 BPP 的另一个定义:

$$\begin{aligned} \text{BPP} = & \left\{ F : \{0,1\}^* \rightarrow \{0,1\} \mid \exists \text{ a polynomial-running-time TM } G \right. \\ & \left. s.t. \forall x \in \{0,1\}^*, \Pr_{r \in \{0,1\}^{a|x|^b}}(G(xr) = F(x)) \geq \frac{2}{3} \right\} \end{aligned} \quad (3)$$

这里限定了 r 的长度为 $a|x|^b$, 也是因为规定 RNAND-TM Program 在多项式时间内运行, 那么其最多执行多项式次数的 `RAND()` 指令, 因此只需要这么长的随机串就够了。

5.4.1 Field of Probabilistic Turing Machines

接下来看看 BPP 和之前介绍过的函数类之间的关系。

首先, 由于计算 BPP 函数的 RNAND-TM Program 的 Running Time 是 Polynomial 的, 那么计算 \mathcal{P} 函数的 NAND-TM Program 就是其的一种特殊情况 (不用 RAND 指令), 于是

$$\mathcal{P} \subset \text{BPP} \quad (4)$$

其次, 对于某个函数 $F \in \text{BPP}$, 我们尝试去除其中的随机性 (Derandomization), 从而得到一个确定性的计算方法 (Deterministic Algorithm) (这个过程也被称为 Brute-Force Derandomization):

- 令 M 为一台计算 F 的 Probabilistic TM, 其输入长度为 n , 时间复杂度为 $T(n^k)$, 错误率不超过 $\frac{1}{3}$;
- 构造图灵机 M' :
 1. 输入 $x \in \{0,1\}^n$
 2. 对于所有可能的随机字符串 $r \in \{0,1\}^{n^k}$
 - 1) 运行 M on input (x, r)
 - 2) 记录 M 的输出
 3. 输出出现次数最多的结果

『所有可能的』说明这个算法的时间复杂度是 2^{n^k} , 即 $F \in \mathcal{EXP}$, 于是

$$\text{BPP} \subset \mathcal{EXP} \quad (5)$$

目前 (迄 2025 年 12 月 27 日) 0 人知道上面这两个包含关系是否为真包含; 注意到 Brute-Force Derandomization 中主要影响结果复杂度的就是 r 的长度, 如果能证明只需要 $|r| \leq \log|x|$ 这么长的随机串就够用, 那就能说明 $\mathcal{P} = \text{BPP}$ 了——可惜人们还在探索如何证明。

回顾: $\mathbf{P}_{/\text{poly}}$ 是指对于固定长度的输入, 都有规模为输入长度多项式的布尔电路来计算的函数类, 有

$$\text{BPP} \subset \mathbf{P}_{/\text{poly}} \quad (6)$$

证明通过对 BPP 中的函数输入固定长度后 Derandomization, 最后可以得到存在一个长度固定的 r , 固定该 r (这一步可看作是把 r hardcode 进一个 NAND-TM 电路中) 后使用 TM 计算输入 (xr) 即可。(这东西又叫作 Adleman's Theorem)

还有一个引理: 如果 $\mathcal{P} = \mathcal{NP}$ 那么 $\text{BPP} = \mathcal{P}$ 。

5.4.2 Pesudo-random Generators

这一节如果考的话就真过了——已放弃。

于是, 理论计算机科学导引的故事就这么结束了……吗?

一定有更好的办法。