

Theoretic Computer Science

Tianlin Pan

October 28, 2025

Contents

1	数学术语	3
1.1	字符串和语言	3
1.2	可满足性问题	4
2	定理和证明	5
2.1	零知识证明	5
3	确定性有穷自动机	6
3.1	有穷自动机的定义	6
3.2	有穷自动机识别的语言	6
3.3	有穷自动机的模拟运行	6
4	正则语言	8
4.1	计算的形式化定义	8
4.2	正则语言的定义	8
4.3	非正则语言的例子	8
4.4	正则运算	8
5	非确定性有穷自动机	10
5.1	形式化定义	10
5.2	NFA 的状态转移	10
5.3	NFA 的计算过程	10
5.4	NFA 的“猜想”行为	11
5.5	ϵ -NFA 的转换	11
5.6	NFA 和 DFA 的等价性	12
6	正则表达式	14
6.1	引子	14
6.2	正则表达式的形式化定义	14
6.3	运算的优先级	15
6.4	有穷自动机和正则表达式	15
6.5	正则表达式代数定律	18
6.6	Arden 引理	19
6.7	正则表达式的应用	20
7	泵引理 (Pumping Lemma)	21
7.1	引子	21
7.2	引理内容	21
7.3	证明思路	21
7.4	使用泵引理证明语言非正则	21
8	极小化 DFA	22

1 数学术语

1.1 字符串和语言

1.1.a 字母表

任意非空有穷集合:

- $\Sigma = \{0, 1\}$
- $\Sigma = \{a, b, c, \dots, x, y, z, \text{space}\}$

1.1.b (字符) 串

字母的有穷序列 - $x = 01001, w = \text{madam}$

字符串的 **连接**

- $xw = 01001\text{madam}$
- $xx = x^2 = 0100101001$

串的 **长度** - $|x| = |w| = 5 \implies |xw| = 10$

空串 $|\varepsilon| = 0, x^0 = \varepsilon$.

1.1.c 子串和子序列

子串 要求是连续的片段 - ada 是 madam 的子串

子序列 则不要求连续 - mdm 是 madam 的子序列

1.1.d 语言

语言 是由字符串组成的集合

几种特殊的语言:

$$\begin{aligned}\Sigma^* &= \{x \mid x \in \Sigma \text{ and } |x| \text{ is finite}\} \\ \Sigma^+ &= \{x \mid x \in \Sigma \text{ and } |x| \text{ is finite and } |x| > 0\} \\ \Sigma' &= \{x \mid x \in \Sigma \text{ and } |x| \text{ is infinite}\}\end{aligned}\tag{1}$$

因此, 任何由有限长度字符串组成的语言 A 都是 Σ^* 的一个子集

$$A \subset \Sigma^*\tag{2}$$

空语言 是一个不包含任何字符串的语言, 用符号 \emptyset 表示 **空串语言** 是一个包含唯一一个字符串的语言, 而这个唯一的字符串是**空串**, 符号为 $\{\varepsilon\}$.

语言的连接:

$$AB = \{xy \mid x \in A \text{ and } y \in B\} \quad (3)$$

有两个特殊情况是

- $\{\varepsilon\}A = A\{\varepsilon\} = A$
- $\emptyset A = A\emptyset = A$

1.1.e 标准序

排序方法

- **先短后长**: 如果两个字符串的长度不同, 那么短的字符串排在前面
- **等长逐位比较**: 如果两个字符串的长度相同, 那么就从第一个字符开始, 逐个字符地比较它们的顺序

$$\Sigma_2^* = \{\varepsilon, a, b, c, \dots, x, y, z, aa, ab, ac, \dots ax, ay, az, ba, \dots, bz, \dots, za, \dots, zz, aaa, \dots, zzz, aaaa, \dots\} \quad (4)$$

1.2 可满足性问题

是否存在一种对变量的赋值方式 (比如把某个变量设为真或假), 能让整个公式的结果为真. 如果存在, 那么这个公式就是**可满足的**.

SAT 定义为所有可满足的布尔公式的集合:

$$\text{SAT} = \{\phi \mid \phi \text{ is a satisfiable Boolean expression} \} \quad (5)$$

CNF 公式 也就是 **合取范式**. 一个布尔公式如果是由若干个子句通过与 (\wedge) 连接起来的, 那么它就是一个 CNF 公式. 特别地, 如果一个 CNF 公式中的所有字句都有三个文字, 那么这个公式就被称为 **3-CNF 公式**. 例如:

$$(x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (x_3 \vee \bar{x}_5 \vee x_6) \wedge (x_3 \vee \bar{x}_6 \vee x_4) \quad (6)$$

3SAT 问题:

$$3\text{SAT} = \{\phi \mid \phi \text{ is a satisfiable 3-CNF} \} \quad (7)$$

2 定理和证明

2.1 零知识证明

零知识证明是一种协议, 它允许一个人向另一个人证明某个陈述是真实的, 而无需透露任何关于这个陈述的额外信息. 可以把它想象成一个谜题, **证明者**知道谜底, 他要让**验证者**相信他确实知道谜底, 但同时又不能把谜底告诉对方.

3 确定性有穷自动机

3.1 有穷自动机的定义

一个确定性有穷自动机 (Deterministic Finite Automaton, DFA) 是一个五元组 $(Q, \Sigma, \delta, q_0, F)$, 其中

1. Q 是一个有限 **状态** 集合.
2. Σ 是一个有限 **输入符号** 集合, 称为 **字母表**.
3. $\delta: Q \times \Sigma \rightarrow Q$ 是一个 **状态转移函数**, 它定义了给定当前状态和输入符号的情况下, 自动机将转移到哪个状态.
4. $q_0 \in Q$ 是 **初始状态**.
5. $F \subseteq Q$ 是一个 **接受状态** 集合.

3.2 有穷自动机识别的语言

一个 DFA 可以识别某些语言. 给定一个输入字符串, 自动机从初始状态 q_0 开始, 根据输入字符串的每个符号和状态转移函数 δ 依次转移状态. 如果在处理完输入字符串后, 自动机停在一个接受状态 F 中的某个状态, 那么该字符串被认为是被该自动机接受的.

$$L(M) = \{w \in \Sigma^* \mid \text{the DFA } M \text{ accepts } w\} = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}$$

这里面, $\hat{\delta}$ 是 δ 的扩展, 它定义了给定初始状态和输入字符串的情况下, 自动机将转移到哪个状态.

3.3 有穷自动机的模拟运行

如果使用 Python 来模拟一个 DFA 的运行, 可以按照以下步骤进行:

```
class DFA:
    def __init__(self, s, a, t, q0, f):
        self.states = s # 状态集合
        self.alphabet = a # 输入符号集合
        self.transition = t # 状态转移函数
        self.start_state = q0 # 初始状态
        self.accept_states = f # 接受状态集合

    def run(self, input_string):
        current_state = self.start_state
        for symbol in input_string:
            if symbol in self.alphabet:
                current_state =
self.transition.get((current_state, symbol), None)
```

```
        if current_state is None:
            break
    else:
        break
    return current_state in self.accept_states
```

4 正则语言

4.1 计算的形式化定义

对于一个有穷自动机 $M = (Q, \Sigma, \delta, q_0, F)$ 和输入串 $w = w_1 w_2 \dots w_n, w_i \in \Sigma$, 如果存在一个状态序列 r_0, r_1, \dots, r_n , 使得

1. $r_0 = q_0$ (初始状态)
2. $r_{i+1} = \delta(r_i, w_{i+1})$ 对于 $0 \leq i < n$ (状态转移)
3. $r_n \in F$ (接受状态)

则称 M 接受输入串 w (接受计算). M 接受的所有字符串的集合称为 M 识别的语言, 记为 $L(M)$.

4.2 正则语言的定义

如果一个语言 L 中的所有字符串都可以被某个有穷自动机 M 接受, 则称 L 是一个 **正则语言 (Regular Language)**. 即存在一个有穷自动机 M 使得 $L = L(M)$.

4.3 非正则语言的例子

通常来说, 一个非正则语言 (即不能被任何有穷自动机识别的语言) 需要 **存储** 的能力 (requires memory). 例如, 语言 $L = \{a^n b^n \mid n \geq 0\}$ 包含所有形式为 a 的若干个后跟同样数量的 b 的字符串. 这个语言不是正则的, 因为有穷自动机无法记住它已经读了多少个 a , 以确保它读了相同数量的 b .

4.4 正则运算

正则语言在以下运算下是封闭的:

1. **并 (Union)**: 如果 L_1 和 L_2 是正则语言, 则 $L_1 \cup L_2$ 也是正则语言.
2. **连接 (Concatenation)**: 如果 L_1 和 L_2 是正则语言, 则 $L_1 \circ L_2 = \{xy \mid x \in L_1, y \in L_2\}$ 也是正则语言.
3. **克林闭包 (Kleene Star)**: 如果 L 是正则语言, 则 $L^* = \{x_1 x_2 \dots x_k \mid k \geq 0, x_i \in L\}$ 也是正则语言.

注意这里克林闭包是一元运算 (unary operation), 而并和连接是二元运算 (binary operations). 且克林闭包包含了空串 ($k = 0$ 的特殊情况), 即 $\varepsilon \in L^*$.

例子:

设 $A = \{\text{Good}, \text{Bad}\}$, $B = \{\text{Boy}, \text{Girl}\}$, 则

- $A \cup B = \{\text{Good}, \text{Bad}, \text{Boy}, \text{Girl}\}$
- $A \circ B = \{\text{GoodBoy}, \text{GoodGirl}, \text{BadBoy}, \text{BadGirl}\}$

- $A^* = \{\varepsilon, \text{Good}, \text{Bad}, \text{GoodGood}, \text{GoodBad}, \text{BadGood}, \text{BadBad}, \text{GoodGoodGood}, \dots\}$

一些其他运算:

1. **补 (Complement):** 如果 L 是正则语言, 则 $\bar{L} = \Sigma^* \setminus L$ 也是正则语言.
2. **交 (Intersection):** 如果 L_1 和 L_2 是正则语言, 则 $L_1 \cap L_2$ 也是正则语言.
3. **差 (Difference):** 如果 L_1 和 L_2 是正则语言, 则 $L_1 - L_2 = L_1 \cap \bar{L}_2$ 也是正则语言.
4. **对称差 (Symmetric Difference):** 如果 L_1 和 L_2 是正则语言, 则 $L_1 \oplus L_2 = (L_1 - L_2) \cup (L_2 - L_1) = (L_1 \cup L_2) - (L_1 \cap L_2)$ 也是正则语言.

5 非确定性有穷自动机

5.1 形式化定义

一个非确定性有穷自动机 (Nondeterministic Finite Automaton, NFA) 是一个五元组 $(Q, \Sigma, \delta, q_0, F)$, 其中

1. Q 是一个有限 **状态** 集合.
2. Σ 是一个有限 **输入符号** 集合, 称为 **字母表**.
3. $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q)$ 是一个 **状态转移函数**, 其中 $\mathcal{P}(Q)$ 表示状态集合的幂集. 它定义了在给当前状态和输入符号的情况下, 自动机可以转移到哪些状态. 注意这里允许 ε 转移, 即自动机可以在不读取任何输入符号的情况下转移状态.
4. $q_0 \in Q$ 是 **初始状态**.
5. $F \subseteq Q$ 是一个 **接受状态** 集合.

5.2 NFA 的状态转移

NFA 的状态转移函数 δ 定义了在给当前状态和输入符号的情况下, 自动机可以转移到哪些状态. 具体来说, 对于每个状态 $q \in Q$ 和输入符号 $a \in \Sigma \cup \{\varepsilon\}$, $\delta(q, a)$ 是一个状态集合, 表示自动机可以从状态 q 通过读取输入符号 a 转移到的所有可能状态. 这意味着在某个状态下, 自动机可能有多个选择, 包括不读取任何输入符号而直接转移到另一个状态 (通过 ε 转移).

5.3 NFA 的计算过程

给定一个 NFA $M = (Q, \Sigma, \delta, q_0, F)$ 和输入串 $w = w_1 w_2 \dots w_n$, $w_i \in \Sigma$, NFA 的计算过程可以描述如下:

1. **初始状态**: 计算从初始状态 q_0 开始, 包括所有通过 ε 转移可以到达的状态集合. 记为 $E(q_0)$.
2. **状态转移**: 对于输入串的每个符号 w_i (从 $i = 1$ 到 n), 计算当前状态集合 S_{i-1} (初始时为 $E(q_0)$) 通过读取符号 w_i 后可以到达的所有状态集合, 记为 S_i :

$$S_i = \bigcup_{q \in S_{i-1}} \delta(q, w_i) \quad (9)$$

然后, 计算 S_i 中所有状态通过 ε 转移可以到达的状态集合, 记为 $E(S_i)$.

3. **接受状态**: 在处理完输入串 w 后, 如果最终状态集合 S_n (包括通过 ε 转移可以到达的状态) 中包含至少一个接受状态 F 中的状态, 则称 NFA **接受** 输入串 w .

5.4 NFA 的“猜想”行为

NFA 的计算过程可以看作是对所有可能的状态路径进行“猜测”。在每个状态, NFA 可以选择多个可能的转移, 包括通过 ε 转移跳过某些输入符号. 这种非确定性使得 NFA 能够在某种程度上“并行”处理多个计算路径, 从而提高了对复杂语言的识别能力.

比如说, 如果我们要设计一个 NFA 来识别语言

$$L = \{x \mid x \text{ the third-to-last character of } x \text{ is } 1\} \quad (10)$$

且 $\Sigma = \{0, 1\}$, 我们可以设计如下的 NFA:

- 状态集合: $Q = \{q_0, q_1, q_2, q_3\}$
- 输入符号集合: $\Sigma = \{0, 1\}$
- 初始状态: q_0
- 接受状态: $F = \{q_3\}$
- 状态转移函数 δ :

当前状态	输入符号	下一个状态
q_0	0	q_0
q_0	1	q_0, q_1
q_1	0	q_2
q_1	1	q_2
q_2	0	q_3
q_2	1	q_3

可以看到, 在状态 q_0 读取到输入符号 **1** 时, NFA 可以选择留在 q_0 (继续读取更多的符号), 也可以转移到 q_1 (表示已经找到了一个可能的倒数第三个字符). 这种“猜想”行为使得 NFA 能够有效地识别符合条件的字符串.

如果我们使用 DFA 来识别同样的语言, 我们则必须要为最后三位可能的所有组合 (000, 001, 010, 011, 100, 101, 110, 111) 设计状态, 这会导致状态数量的指数级增长.

5.5 ε -NFA 的转换

一个 ε -NFA 是一种特殊的 NFA, 它允许在不读取任何输入符号的情况下进行状态转移 (即通过 ε 转移). 这种能力使得 ε -NFA 在某些情况下更容易设计和理解.

我们可以通过以下步骤将一个 ε -NFA 转换为一个等价的 NFA:

1. **计算 ε -闭包:** 对于每个状态 $q \in Q$, 计算其 ε -闭包 $E(q)$, 即从状态 q 出发, 通过任意数量的 ε 转移可以到达的所有状态集合 (包含 q 自己).
2. **定义新的状态转移函数:** 对于每个状态 $q \in Q$ 和输入符号 $a \in \Sigma$, 定义新的状态转移函数 δ' :

$$\delta'(q, a) = \bigcup_{p \in E(q)} \delta(p, a) \quad (11)$$

这表示从状态 q 出发, 通过 ε 转移到达的所有状态 p ,

然后读取输入符号 a 后可以到达的所有状态集合.

3. **定义新的接受状态:** 定义新的接受状态集合 F' :

$$F' = \{q \in Q \mid E(q) \cap F \neq \emptyset\} \quad (12)$$

这表示如果从状态 q 出发, 通过 ε 转移可以到达至少一个接受状态, 则 q 也是一个接受状态.

4. **构造新的 NFA:** 最终, 我们得到一个新的 NFA $M' = (Q, \Sigma, \delta', q_0, F')$, 它与原始的 ε -NFA 等价, 即它们识别相同的语言.

5.6 NFA 和 DFA 的等价性

如果两个自动机 M_1 和 M_2 识别相同的语言, 即 $L(M_1) = L(M_2)$, 则称它们是 **等价** 的.

DFA 显然是 NFA 的一个特例, 所以要证明 NFA 和 DFA 的等价性, 只需要证明对于任意一个 NFA, 都存在一个等价的 DFA.

我们可以通过以下步骤将一个 NFA 转换为一个等价的 DFA:

1. **状态集合:** 新的 DFA 的状态集合 Q' 是原始 NFA 的状态集合 Q 的幂集, 即 $Q' = \mathcal{P}(Q)$. 这意味着每个新的状态都是原始 NFA 的状态的一个子集.
2. **初始状态:** 新的 DFA 的初始状态 $q_{0'}$ 是原始 NFA 的初始状态 q_0 的 ε -闭包, 即 $q_{0'} = E(q_0)$.
3. **接受状态:** 新的 DFA 的接受状态集合 F' 包含所有包含至少一个原始 NFA 接受状态的子集, 即 $F' = \{S \subseteq Q \mid S \cap F \neq \emptyset\}$.
4. **状态转移函数:** 新的 DFA 的状态转移函数 δ' 定义如下:

$$\delta'(S, a) = \bigcup_{q \in S} \delta(q, a) \quad (13)$$

这表示从状态集合 S 出发, 读取输入符号 a 后可以到达的所有状态集合.

通过上述步骤, 我们可以构造出一个新的 DFA $M' = (Q', \Sigma, \delta', q_0', F')$, 它与原始的 NFA 等价, 即它们识别相同的语言.

推论: 一个语言是正则的, 当且仅当它可以被某个 NFA 识别. 即

$$L \text{ is regular} \Leftrightarrow \exists \text{ NFA } M \text{ s.t. } L = L(M) \quad (14)$$

子集构造法 (Subset Construction): 上述将 NFA 转换为 DFA 的方法称为子集构造法, 因为新的 DFA 的状态是原始 NFA 状态的子集.

6 正则表达式

6.1 引子

我们在算术中可以用运算符 (如 $+$, $-$, \times , \div) 来构造表达式, 类似地, 可以用正则运算符来构造描述语言的表达式, 称为正则表达式. 也就是说, 正则表达式的值是一个语言.

正则表达式能定义所有的正则语言, 反之亦然. 因此, 正则表达式和有穷自动机 (DFA/NFA) 是等价的.

6.2 正则表达式的形式化定义

6.2.a 归纳定义法

一个正则表达式 (Regular Expression, RE) 是通过以下规则递归定义的:

1. 基本表达式:

- **(empty set)** \emptyset 是一个正则表达式, 它表示空语言.
- **(empty string)** ε 是一个正则表达式, 它表示只包含空串的语言.
- **(literal character)** 对于每个符号 $a \in \Sigma$, a 是一个正则表达式, 它表示只包含字符串 a 的语言.

2. 复合表达式:

- **(concatenation)** 如果 R_1 和 R_2 是正则表达式, 则 $R_1 R_2$ 是一个正则表达式, 它表示语言的连接 $L(R_1)L(R_2)$ (见 [Section 1.1.d](#)).
- **(alternation)** 如果 R_1 和 R_2 是正则表达式, 则 $R_1 + R_2$ 是一个正则表达式, 它表示语言的并 $L(R_1) \cup L(R_2)$ (见 [Section 4.4](#)).
- **(Kleene star)** 如果 R 是一个正则表达式, 则 R^* 是一个正则表达式, 它表示语言的克林闭包 $L(R)^*$ (见 [Section 4.4](#)).

正则表达式 R 所表示的语言记为 $L(R)$.

6.2.b 一些正则表达式的例子

假设字母表 $\Sigma = \{0, 1\}$, 则以下是一些正则表达式及其对应的语言:

1. $0^*10^* = \{w \mid w \text{ has exactly one } 1\}$. 这是因为 0^* 表示任意数量的 0 (包括零个), 因此 0^*10^* 表示一个 1 前后可以有任意数量的 0.
2. $\Sigma^*1\Sigma^* = \{w \mid w \text{ contains at least one } 1\}$.
3. $\Sigma^*001\Sigma^* = \{w \mid w \text{ contains } 001 \text{ as a substring}\}$.

4. $1^*(01^+)^* = \{w \mid \text{every } 0 \text{ in } w \text{ is followed by at least one } 1\}$. 注意这里为了方便起见, 我们用 R^+ 表示 RR^* .
5. $(\Sigma\Sigma)^* = \{w \mid \text{the length of } w \text{ is even}\}$.
6. $01 + 10 = \{01, 10\}$
7. $0\Sigma^*0 + 1\Sigma^*1 + 0 + 1 = \{w \mid w \text{ starts and ends with the same symbol}\}$
8. $(0 + \varepsilon)(1 + \varepsilon) = \{\varepsilon, 0, 1, 01\}$
9. $1^*\emptyset = \emptyset$. 注意空集连接任何语言仍然是空集.
10. $\emptyset^* = \{\varepsilon\}$. 星号运算把该语言中的任意个字符串连接在一起, 得到运算结果中的一个字符串. 如果该语言是空集, 星号运算能把 0 个字符串连接在一起, 结果就是空串.

6.3 运算的优先级

正则表达式中的运算符有不同的优先级, 其优先级从高到低依次为:

1. 克林闭包 (*)
2. 连接
3. 并 (+)

6.4 有穷自动机和正则表达式

就描述语言的能力而言, 有穷自动机 (DFA/NFA) 和正则表达式是等价的. 具体来说, 任何的正则表达式都能够转换成能识别它所描述语言的有穷自动机, 反之亦然.

6.4.a 如果一个语言可以用正则表达式描述, 那么它是正则的.

考虑如下的证明思路:

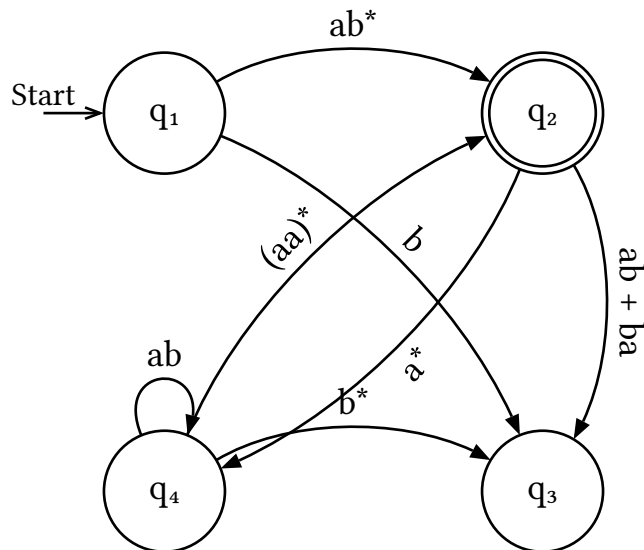
1. 由于 NFA 和 DFA 是等价的, 因此只需要证明对于任意一个正则表达式 R , 都存在一个等价的 NFA M 使得 $L(R) = L(M)$.
2. 对于每个基本正则表达式, 我们可以直接构造一个等价的 NFA:
 - 对于 \emptyset , 构造一个 NFA $M = (\{q_0\}, \Sigma, \delta, q_0, \emptyset)$, 其中 δ 是一个空函数. 该 NFA 没有接受状态, 因此它识别的语言是空集.
 - 对于 ε , 构造一个 NFA $M = (\{q_0\}, \Sigma, \delta, q_0, \{q_0\})$, 其中 δ 是一个空函数. 该 NFA 的初始状态也是接受状态, 因此它识别的语言只包含空串.

- 对于每个符号 $a \in \Sigma$, 构造一个 NFA $M = (\{q_0, q_1\}, \Sigma, \delta, q_0, \{q_1\})$, 其中 $\delta(q_0, a) = \{q_1\}$ 且 δ 在其他情况下是空函数. 该 NFA 识别的语言只包含字符串 a .
3. 对于复合正则表达式, 我们可以通过归纳假设来构造等价的 NFA:
 - 对于 $R_1 R_2$, 假设存在等价的 NFA M_1 和 M_2 分别识别 R_1 和 R_2 . 我们可以构造一个新的 NFA M , 通过将 M_1 的所有接受状态与 M_2 的初始状态通过 ε 转移连接起来. 这样, M 识别的语言就是 $L(R_1)L(R_2)$.
 - 对于 $R_1 + R_2$, 假设存在等价的 NFA M_1 和 M_2 分别识别 R_1 和 R_2 . 我们可以构造一个新的 NFA M , 通过添加一个新的初始状态 $q_{0'}$ 和两个 ε 转移, 分别从 $q_{0'}$ 指向 M_1 和 M_2 的初始状态. 这样, M 识别的语言就是 $L(R_1) \cup L(R_2)$.
 - 对于 R^* , 假设存在一个等价的 NFA M 识别 R . 我们可以构造一个新的 NFA M' , 通过添加一个新的初始状态 $q_{0'}$ 和一个新的接受状态 $q_{f'}$. 然后, 添加两个 ε 转移, 一个从 $q_{0'}$ 指向 M 的初始状态, 另一个从 M 的所有接受状态指向 $q_{f'}$. 另外, 添加一个 ε 转移从 $q_{0'}$ 指向 $q_{f'}$, 以允许接受空串. 这样, M' 识别的语言就是 $L(R)^*$.
 4. 通过上述步骤, 我们可以递归地构造出一个等价的 NFA M 对应于任意的正则表达式 R , 从而证明了如果一个语言可以用正则表达式描述, 那么它是正则的.

6.4.b 如果一个语言是正则的, 则可以用正则表达式描述它

考虑如下的证明思路:

1. 由于 NFA 和 DFA 是等价的, 因此只需要证明对于任意一个 DFA M , 都存在一个等价的正则表达式 R 使得 $L(R) = L(M)$.
2. 该过程需要引入一种称为 **广义非确定型有穷自动机** (Generalized Non-deterministic Finite Automaton, GNFA) 的中间模型. GNFA 与 NFA 类似, 但它的状态转移函数允许使用正则表达式作为标签, 而不仅仅是单个符号或 ε . 如下图所示:



3. GNFA 还满足以下条件:

- GNFA 只有一个初始状态, 它有到所有其他状态的转移, 但没有任何进入它的转移.
- GNFA 只有一个接受状态, 它有从所有其他状态的转移, 但没有任何离开它的转移.
- 除了初始状态和接受状态之外, 每对状态之间恰好有一条单向转移 (可能标签为 \emptyset).

4. 考虑 DFA \rightarrow GNFA 的构造过程: 给定一个 DFA $M = (Q, \Sigma, \delta, q_0, F)$, 我们可以构造一个等价的 GNFA $M' = (Q', \Sigma, \delta', q_{0'}, q_{f'})$, 其中:

- 状态集合: $Q' = Q \cup \{q_{0'}, q_{f'}\}$, 其中 $q_{0'}$ 是新的初始状态, $q_{f'}$ 是新的接受状态.
- 输入符号集合: Σ 与原始 DFA 相同.
- 状态转移函数 δ' 定义如下:
 - 对于每个状态 $q \in Q$, 添加一个从 $q_{0'}$ 到 q 的转移, 标签为 ε .
 - 对于每个接受状态 $f \in F$, 添加一个从 f 到 $q_{f'}$ 的转移, 标签为 ε .
 - 对于每对状态 $(p, q) \in Q \times Q$, 如果存在一个输入符号 $a \in \Sigma$ 使得 $\delta(p, a) = q$, 则添加一个从 p 到 q 的转移, 标签为 a ; 如果有多个这样的输入符号, 则标签为它们的并 (用 $+$ 连接); 如果没有这样的输入符号, 则标签为 \emptyset .

5. 考虑 GNFA \rightarrow 正则表达式 的转换过程: 给定一个 GNFA $M' = (Q', \Sigma, \delta', q_{0'}, q_{f'})$, 我们可以通过逐步消除状态来构造一个等价的正则表达式 R :

- 选择一个非初始状态且非接受状态 $r \in Q' \setminus \{q_{0'}, q_{f'}\}$.
- 对于每对状态 $(p, q) \in Q' \times Q'$ (包括 $p = r$ 或 $q = r$), 更新状态转移函数 δ' 以反映消除状态 r 的影响:

- ▶ 如果存在从 p 到 r 的转移, 标签为 R_1 , 从 r 到 r 的转移, 标签为 R_2 , 以及从 r 到 q 的转移, 标签为 R_3 , 则添加一个从 p 到 q 的新转移, 标签为 $R_1 R_2^* R_3$.
 - ▶ 如果存在从 p 到 r 的转移, 标签为 R_1 , 以及从 r 到 q 的转移, 标签为 R_3 , 则添加一个从 p 到 q 的新转移, 标签为 $R_1 R_3$.
 - ▶ 如果存在从 p 到 q 的转移, 标签为 R_4 , 则更新该转移的标签为 $R_4 + R_1 R_2^* R_3$ (如果同时存在上述情况).
 - 删除状态 r 及其所有相关的转移.
 - 重复上述步骤直到只剩下初始状态和接受状态.
6. 最终, 当只剩下初始状态 q_0' 和接受状态 $q_{f'}$ 时, 状态转移函数 δ' 中从 q_0' 到 $q_{f'}$ 的转移标签就是所求的正则表达式 R .

6.5 正则表达式代数定律

6.5.a 正则表达式的等价性

如果两个正则表达式定义了相同的语言, 则称这两个正则表达式是等价的.

例如, 下面两个表达式都表示交替的 0 和 1 的串的集合

- $(\varepsilon + 1)(01)^*(\varepsilon + 0)$
- $(01)^* + (10)^* + 1(01)^* + 0(10)^*$

6.5.b 单位元和零元

1. 单位元: $R + \emptyset = R, R\varepsilon = \varepsilon R = R$
2. 零元: $R\emptyset = \emptyset R = \emptyset, R + \Sigma^* = \Sigma^*$

6.5.c 正则表达式的代数定律

1. 交换律: $R_1 + R_2 = R_2 + R_1$
2. 结合律: $(R_1 + R_2) + R_3 = R_1 + (R_2 + R_3), (R_1 R_2) R_3 = R_1 (R_2 R_3)$
3. 分配律: $R_1 (R_2 + R_3) = R_1 R_2 + R_1 R_3, (R_1 + R_2) R_3 = R_1 R_3 + R_2 R_3$

6.5.d 幂等律

1. $R + R = R$
2. $R \cap R = R$

6.5.e 克林闭包的性质

1. $\varepsilon \in L(R^*)$
2. $(R^*)^* = R^*$
3. $\emptyset^* = \varepsilon, \varepsilon^* = \varepsilon$
4. $R^+ = RR^* = R^*R$

$$5. R^* = R^+ + \varepsilon$$

6.6 Arden 引理

6.6.a 内容

设 P 和 Q 是正则表达式, 则关于正则表达式 X 的方程 $X = Q + XP$ 的解为 $X = QP^*$. 如果 $\varepsilon \notin L(P)$, 则该解是唯一的.

6.6.b 使用 Arden 引理把 NFA 转换为正则表达式

前提假设:

1. 有穷自动机的状态转移图中不包含 ε 转移.
2. 有穷自动机仅含有一个初始状态

转换步骤:

1. 对于有穷自动机的每个状态 q_i , 定义一个正则表达式 R_i 表示从初始状态出发到达状态 q_i 的所有字符串的集合.
2. 根据状态转移图, 对每个状态 q_i , 写出一个关于 R_i 的方程. 该方程的形式为:

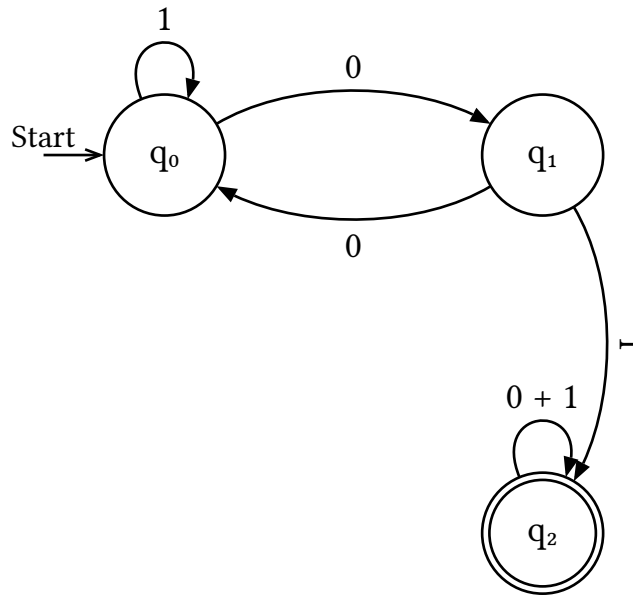
$$R_i = \sum_j R_j a_{ji} + b_i \quad (15)$$

其中, a_{ji} 是从状态 q_j 到状态 q_i 的输入符号 (如果存在多条边, 则用 $+$ 连接), b_i 是从初始状态直接到达状态 q_i 的输入符号 (如果存在多条边, 则用 $+$ 连接; 如果没有, 则为 \emptyset).

3. 对于每个接受状态 q_f , 使用 Arden 引理求解关于 R_f 的方程, 得到 R_f 的正则表达式.
4. 最终, 语言 $L(M)$ 可以表示为所有接受状态的正则表达式的并:

$$L(M) = \sum_{q_f \in F} R_f \quad (16)$$

例子: 考虑如下的 DFA:



1. 定义正则表达式: R_0, R_1, R_2 分别表示从初始状态 q_0 出发到达状态 q_0, q_1, q_2 的所有字符串的集合.
2. 写出方程:
 - 对于状态 q_0 : $R_0 = R_01 + R_10 + \varepsilon$
 - 对于状态 q_1 : $R_1 = R_00$
 - 对于状态 q_2 : $R_2 = R_11 + R_20 + R_21$
3. 求解方程:
 - 从第二个方程得到 $R_1 = R_00$.
 - 将 R_1 代入第一个方程, 得到 $R_0 = R_01 + R_000 + \varepsilon = R_0(1 + 00) + \varepsilon$. 根据 Arden 引理, 得到 $R_0 = \varepsilon(1 + 00)^* = (1 + 00)^*$.
 - 将 R_1 代入第三个方程, 得到 $R_2 = R_001 + R_2(0 + 1)$. 根据 Arden 引理, 得到 $R_2 = R_001(0 + 1)^* = (1 + 00)^*01(0 + 1)^*$.
4. 语言 $L(M)$ 可以表示为 R_2 : $L(M) = R_2 = (1 + 00)^*01(0 + 1)^*$.

6.7 正则表达式的应用

6.7.a grep 工具

grep 是一个在类 Unix 系统中广泛使用的命令行工具, 用于在文件中搜索符合特定模式的文本行.

例子:

- 搜索包含 “error” 的行: `grep "error" filename.txt`
- 使用正则表达式搜索以 “a” 开头, 后跟任意数量的字符, 然后是 “b” 的行: `grep "^a.*b$" filename.txt`
- 忽略大小写搜索 “warning”: `grep -i "warning" filename.txt`

7 泵引理 (Pumping Lemma)

7.1 引子

DFA 的状态数量是有限的, 因此当输入字符串足够长时, DFA 必须在某些状态之间循环 (即鸽巢原理), 而重复路径对应的子串就可以任意重复或删除而不影响字符串是否被接受. 泵引理正是基于这一观察, 它为正则语言提供了一个 **必要条件**.

7.2 引理内容

设 L 是一个正则语言, 则存在一个整数 $p \geq 1$ (称为泵长度), 使得对于任意字符串 $s \in L$ 且 $|s| \geq p$, 都可以将 s 分割为三个子串 $s = xyz$, 满足以下条件:

1. $|y| \geq 1$ (即子串 y 非空)
2. $|xy| \leq p$ (即子串 xy 的长度不超过泵长度 p)
3. 对于所有整数 $i \geq 0$, 字符串 $xy^iz \in L$ (即通过重复子串 y 任意次数, 生成的新字符串仍然属于语言 L)

7.3 证明思路

设 L 是一个正则语言, 则存在一个识别 L 的 DFA $M = (Q, \Sigma, \delta, q_0, F)$, 其中 $|Q| = n$ (状态数量). 取泵长度 $p = n$. 对于任意字符串 $s \in L$ 且 $|s| \geq p$, 在处理字符串 s 时, DFA 必须经过至少 $p + 1$ 个状态 (包括初始状态). 根据鸽巢原理, 在这 $p + 1$ 个状态中, 至少有两个状态是相同的. 设这两个相同的状态分别在处理字符串 s 的第 k 个和第 m 个字符后到达, 其中 $0 \leq k < m \leq |s|$. 将字符串 s 分割为三个子串:

- $x = s[0 : k]$ (从第 0 个字符到第 $k - 1$ 个字符)
- $y = s[k : m]$ (从第 k 个字符到第 $m - 1$ 个字符)
- $z = s[m : |s|]$ (从第 m 个字符到最后一个字符)

由于在处理子串 y 时, DFA 从状态 q 回到状态 q , 因此对于任意整数 $i \geq 0$, 处理字符串 xy^iz 时, DFA 仍然会到达接受状态 (因为它与处理字符串 s 时经过的状态路径相同). 因此, $xy^iz \in L$. 这证明了泵引理的内容.

7.4 使用泵引理证明语言非正则

要使用泵引理证明一个语言 L 非正则, 我们可以采用反证法. 假设 L 是正则的, 则根据泵引理, 存在一个泵长度 $p \geq 1$. 然后, 我们需要找到一个字符串 $s \in L$ 且 $|s| \geq p$, 使得对于任意的分割 $s = xyz$ 满足泵引理的条件, 都存在一个整数 $i \geq 0$ 使得 $xy^iz \notin L$. 这就与泵引理的结论矛盾, 因此 L 非正则.

8 极小化 DFA