Theoretic Computer Science

Contents

1. 数学术语

1.1. 字符串和语言

1.1.1. 字母表

任意非空有穷集合:

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

1.1.2. (字符) 串

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

字符串的 **连接**

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

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

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

1.1.3. 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

1.1.4. 语言

语言 是由字符串组成的集合

几种特殊的语言:

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

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

$$A \subset \Sigma^*$$

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:

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

有两个特殊情况是

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

1.1.5. 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

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

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:

$$\text{SAT} = \{\phi \mid \phi \ \text{ is a satisfiable Boolean expression }\}$$

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 (∧) 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:

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

3SAT 问题:

$$\text{3SAT} = \{\phi \mid \phi \ \text{ is a satisfiable 3-CNF }\}$$

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. $\delta : Q \times \Sigma \to Q$ 是一个 **状态转移函数**, 它定义了在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.

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

这里面, $\hat{\delta}$ 是 δ 的扩展, 它定义了在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:

```python
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 $w = w_1 w_2 ... w_n$, $w_i \in \Sigma$, 如果存在一个状态序列 $r_0, r_1, ..., r_n$, 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 $L^* = \{x_1 x_2 ... x_k \mid k \geq 0, x_i \in L\}$ 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

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

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

一些其他运算:

1. **补 (Complement)**: 如果 L 是正则语言, 则 $\overline{L} = \Sigma^* \setminus L$ 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 $L_1 - L_2 = L_1 \cap \overline{L_2}$ 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. $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ 是一个 **状态转移函数**, 其中 $\mathcal{P}(Q)$ 表示状态集合的幂集. 它定义了在给定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 定义了在给定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 $w = w_1 w_2 ... w_n$, $w_i \in \Sigma$, 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:

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

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

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

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':

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

   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':

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

   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:

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

   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

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

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

6.2.1. 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_1R_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 1.1.4).
- **(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 $L(R)$.

6.2.2. 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 (+)