有穷自动机与正则语言

崔家才
校验：贺卓宇
NJUCS ICS PASCAL

日期：2023 年 9 月 5 日

摘　　要

本文是关于有穷自动机以及正则语言的一篇教程，内容包括形式语言的定义，确定性有穷自动机、非确定性有穷自动机、以及 ε-非确定性有穷自动机的定义和相互转化，正则表达式，正则表达式与有穷自动机的相互转化，正则语言的定义、判定性质和闭包性质。

关键词： 形式语言，有穷自动机，正则表达式，正则语言

1　形式语言

1.1　语言的定义

在形式语言的体系中，我们将**语言（Language）**定义为**字符串（String）**的集合，通常用字母 L 表示。其中，字符串定义为**符号（Symbol）**的序列，例如 cat，dog，house。

一个语言中的所有可能出现的符号是由这个语言的**字母表（Alphabet）**定义的，通常用希腊字母 Σ 表示，它是一个符号的集合。如果 $\Sigma = \{a, b\}$，那么基于这个字母表可以产生字符串：a，ab，abba 等等。

习惯上，我们会用 u，v，w 之类的英文字母中靠后的字母表示字符串，例如 $u = ab$，$v = bbbaaa$，$w = abba$。

1.2　字符串的操作

令 $w = a_1a_2\cdots a_n$，$v = b_1b_2\cdots b_m$，在**表 1**中，我们定义了一些字符串基本操作及其数学表示，以便后续的表述上的方便。

1.3　字母表的操作

定义字母表 Σ 的**星闭包（Star Closure）**为 Σ^*，Σ^* 是从字母表 Σ 产生的所有字符串的集合。比如说对于 $\Sigma = \{a, b\}$，我们有

$$\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, \cdots\}$$

其中，无论 Σ 是什么，都有 $\varepsilon \in \Sigma^*$。

操作	定义
拼接（concatenation）	$wv = a_1a_2\cdots a_nb_1b_2\cdots b_m$
翻转（reversion）	$w^R = a_na_{n-1}\cdots a_2a_1$
长度（length）	$\|w\| = n, \|v\| = m, \|wv\| = \|w\| + \|v\|$
空串（empty string）	$\|\varepsilon\| = 0, \varepsilon w = w\varepsilon = w, \varepsilon^R = \varepsilon$
子串（substring）	s 是 x 的子串当且仅当存在字符串 y, z，使得 $x = ysz$
前缀（prefix）	当 $x = sz$，即 $y = \varepsilon$ 时，s 是 x 的前缀
后缀（suffix）	当 $x = ys$，即 $z = \varepsilon$ 时，s 是 x 的后缀
重复（repetition）	$w^n = \underbrace{ww\cdots w}_{n个w}$，规定 $w^0 = \varepsilon$

表 1: 字符串相关操作的数学定义与表示

由于定义在 Σ 上的语言是 Σ 中的符号组成的字符串的集合，我们不难发现，定义在 Σ 上的语言永远是 Σ^* 的子集。

字母表 Σ 上还有的另一种不那么常用的闭包操作——**正闭包（Positive Closure）**：

$$\Sigma^+ = \Sigma^* - \{\varepsilon\}$$

在上述诸多定义当中，空串是一个容易理解出错的地方，和它相关的三个概念需要读者辨析明白：

- 集合：$\emptyset = \{\} \neq \{\varepsilon\}$
- 集合的势：$|\{\}| = |\emptyset| = 0, |\{\varepsilon\}| = 1$
- 字符串长度：$|\varepsilon| = 0$

有了这些定义和记号之后，我们后续描述语言就方便多了。比如说定义 $L = \{a^nb^n | n \geq 0\}$，我们可以很容易地判断出 $\varepsilon \in L, ab \in L, aabb \in L$，但是 $abb \notin L$。

1.4 语言的操作

语言本身就是字符串的集合，因此集合的所有操作都适用于语言，不过后续我们通常用到的只有交集、并集、差集。

此外，和字符串类似，我们也在**表 2**中定义了一些常用的语言操作，并给出了例子以帮助读者理解。

操作	定义	样例
补集	$\overline{L} = \Sigma^* - L$	$\overline{\{a, ba\}} = \{\varepsilon, b, aa, ab, bb, aaa, \cdots\}$
翻转	$L^R = \{w^R \| w \in L\}$	$\{a^nb^n \| n \geq 0\}^R = \{b^na^n \| n \geq 0\}$
拼接	$L_1L_2 = \{xy \| x \in L_1, y \in L_2\}$	{a, ab, ba}{b, aa} = {ab, aaa, abb, abaa, bab, baaa}
多重拼接	$L^n = \underbrace{LL\cdots L}_{n个L}$，规定 $L^0 = \{\varepsilon\}$	$\{a, b\}^3 = \{a, b\}\{a, b\}\{a, b\} = \{aaa, aab, aba, abb, baa, bab, bba, bbb\}$
星闭包	$L^* = L^0 \cup L^1 \cup L^2 \cup \cdots = \bigcup_{i=0}^{\infty} L^i$	$\{a, bb\} = \{\varepsilon, a, bb, aa, abb, bba, bbbb, aaa, aabb, abba, abbbb, \cdots\}$
正闭包	$L^+ = L^* - \{\varepsilon\}$	$\{a, bb\} = \{a, bb, aa, abb, bba, bbbb, aaa, aabb, abba, abbbb, \cdots\}$

表 2: 语言相关操作的数学定义与表示

其实，字母表本身也可以看成是一个语言，因为我们可以将符号看成是长度为 1 的字符串，从而有 $\Sigma \subseteq \Sigma^*$ 。所以你会发现，语言闭包的定义和字母表闭包的定义本质上说的是同一件事。

不过这个地方有一个定义的微妙之处，比如说 a，它既可以是一个符号，也可以是一个长度为 1 的字符串，到底是哪个含义，是由具体的语境决定的。在习惯上，我们会用 26 英文字母表尾部的，像 $\cdots, w, x, y, z$ 这样的字母来表示字符串；用 26 英文字母表头部的，像 $a, b, c, \cdots$ 这样的字母来表示单个符号。本文后续会沿用这个习惯。

并且，一个符号不一定只能占 1 个“格子”，ab 也可以是一个长得“胖”一些，占两个“格子”的符号，甚至一个苹果，一个香蕉，一个集合都能看成是一个符号，我们在定义中对于符号本身并没有作任何的限制。因此，我们的定义其实是很强大的，在之后的学习中你会逐渐地体会到这一点。

2 有穷自动机

2.1 基本概念

有穷自动机（Finite Automata, FA）是一个形式系统，它只记忆有限的信息量，这些信息量通过它的**状态（State）**表示。当有**输入（Input）**给到有穷自动机的时候，它会相应地改变自己的状态。告诉有穷自动机它应该如何根据自己得到的输入以及自己当前的状态来改变自己之后的状态的一系列规则，被称为**转移函数（Transfer Function）**。

有穷自动机在电路与通信协议的设计和验证方面被广泛使用；也适用于许多的文本处理应用场景（因为它是正则表达式的计算机实现方式）；同时是编译器的一个重要组成部分；另外还可以用来描述事件的简单模式。因此，学习有穷自动机还是很有必要的。

下面我们来看一个简单的打网球的例子，网球比赛的胜利规则有两条：至少要得 4 次分；还要比对手多得两次分。

用 s 表示东道主得分（server wins point），用 o 表示对手得分（opponent wins point），则我们可以用**状态转换图（State Transition Graph）图 1**来描述整个比赛的输赢变化的过程。

图 1中，数字比表示东道主与对手的得分次数比。S^1 表示东道主领先 1 分，O^1 表示对手领先 1 分；S^2 表示东道主胜利，O^2 表示对手胜利。

这个状态转换图也称为**自动机（Automata）**。其中，双环表示**终止状态（Final state）**，也就是比赛结束的时候。

给定一系列的输入，比如说 $sosososososs$ ，从**初始状态（Start State）**开始，根据每个输入符号和当前的输入符号，基于状态转换图**图 1**跳转到下一个状态，不难发现，最终我们到达了 S^2 状态，因此东道主赢了。这一系列的输入称作**输入字符串（Input String）**。如果一个输入字符串可以让一个自动机进入终止状态，我们称这个输入字符串被这个自动机**接受（Accept）**了。因此，终止状态也称为**接受状态（Accepting State）**。

能够被自动机 A 接受的字符串的集合称为 A 的语言，记为 $L(A)$ ，比如说记**图 1**所定义的自动机为 $Tennis$ ，则 $L(Tennis) =$ 能够决出胜者的输赢字符串的集合。对于同样的一组状态和转移函数来说，指定不同的终止状态，就会有不同的自动机，也就能够表示不同的语言了。

直观来讲，有穷自动机的基本要素有：

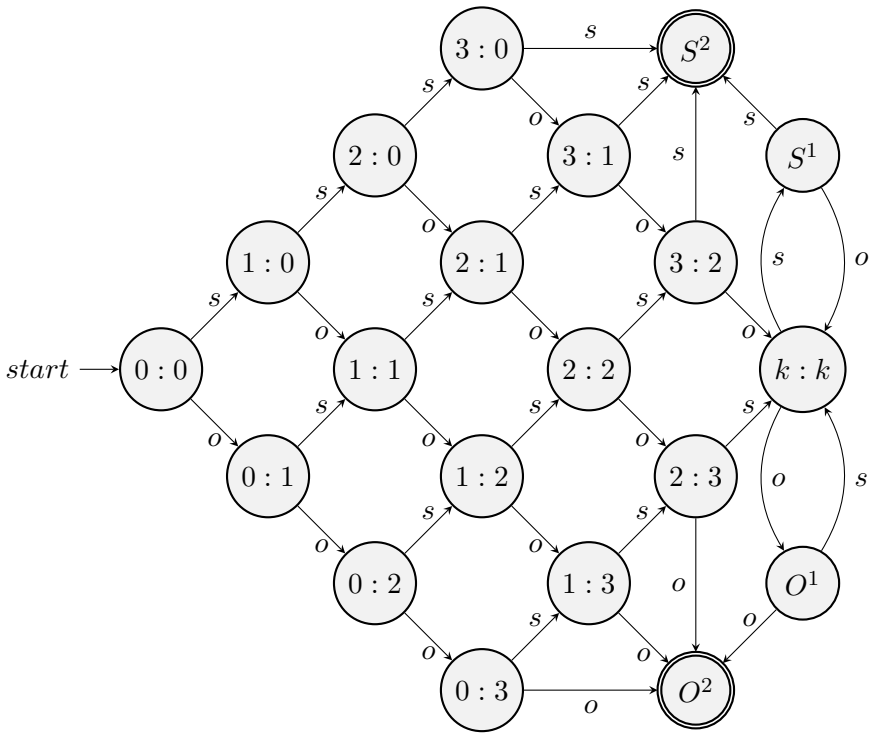


图 1: 网球比赛状态转换图

- 有限个状态（**图 1**上的结点）;
- 状态间的转移函数（**图 1**上的边）;
- 一个初始状态（**图 1**上 start 箭头指向的结点）;
- 一个或者多个终止状态（**图 1**上的双环结点）。

到此为止，你应该对于有穷自动机有了一些基本的认知，下面我们来看一些具体类别的有穷自动机及其形式定义和性质。

2.2 确定性有穷自动机

2.2.1 定义

基于有穷自动机的基本概念，我们可以很容易得给出确定性有穷自动机的形式化定义。

定义 2.1. 定义**确定性有穷自动机（Deterministic Finite Automaton, DFA）**为一个五元组：

$$(Q, \Sigma, \delta, q_0, F)$$

其中，Q 是一个有限的状态集合；Σ 是一个输入字母表；$\delta \subseteq Q \times \Sigma \times Q$ 是一个转移函数；q_0 是初始状态，$q_0 \in Q$，F 是终止状态的集合，$F \subseteq Q$。

转移函数 δ 接受两个参数：一个状态和一个输入符号。$\delta(q, a) = p$ 意味着 DFA 在处于状态 q 时接收输入符号 a 会进入状态 p。

为了书写简洁，后续我们用缩写 DFA 来表示确定性有穷自动机。

在定义2.1中需要注意的是 δ 是一个**全函数（Total Function）**，也就是对于任何状态下的任何输入，δ 总能给出下一个状态。只不过有的时候，我们不会把所有输入下的输出给显式地定义

出来，这种情况下，我们约定，如果没有显式定义 $\delta(q,a)$ ，缺省有 $\delta(q,a)=dead\notin F$ 。即存在一个不需要显示定义的**死状态（Dead State）**，以保证任何情况下 DFA 都能正常运行。

以之前网球比赛的 DFA 为例，我们在**图 2**中给出其带有死状态的版本，之后的状态转换图中就不再显示地画出死状态了。

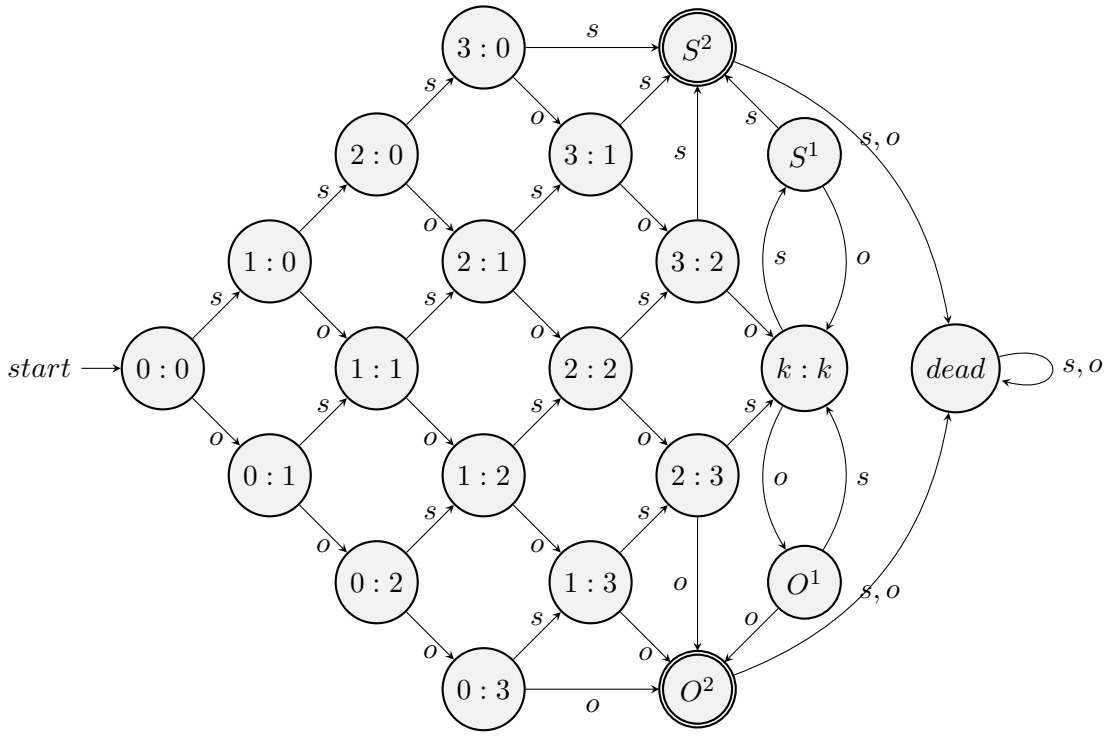


图 2: 含死状态的网球比赛 DFA 状态转换图

在定义2.1中，转移函数 δ 每次调用只能表示一次跳转，如果我想要表示从 q 状态开始，接受输入字符串 $sososo$ 之后进入状态 p ，需要写作 $\delta(\delta(\delta(\delta(\delta(\delta(q,s),o),s),o),s),o)=p$ ，这样表示是极其麻烦的。因此我们下面来定义一个扩展的转移函数，来简化这种繁琐的表示。

定义 2.2. 递归定义 DFA 的**扩展转移函数（Extended Transfer Function）**为

$$\hat{\delta}(q,w)=\begin{cases}q & \text{if } w=\varepsilon\\ \delta(\hat{\delta}(q,x),a) & \text{if } w=xa\end{cases}$$

其中，w 是一个字符串，a 是一个输入符号，wa 表示这两者的拼接，在拼接过程中将 a 视为长度为 1 的字符串。

习惯上，我们并不区分书写 δ 和 $\hat{\delta}$ ，统一都用 δ 表示。至于其含义到底是定义2.1中的转移函数还是定义2.2中的扩展转移函数，由具体语境下的参数类型决定。这样的话，后续我们在表示上会方便很多。

举个例子，假设我们有 $\delta(A,0)=A,\delta(A,1)=B,\delta(B,0)=A,\delta(B,1)=C,\delta(C,0)=C,\delta(C,1)=C$ ，则 $\delta(B,011)$ 可以如下计算：

$$\delta(B,011)=\delta(\delta(B,01),1)=\delta(\delta(\delta(B,0),1),1)=\delta(\delta(A,1),1)=\delta(B,1)=C$$

2.2.2 DFA 的表示

除了我们形式化地将 $(Q, \Sigma, \delta, q_0, F)$ 全都列出来之外，DFA 还有两种看起来更加直观的表示方法：状态转换图和**转移表（Transition Table）**。

就像在**图 1**中画的那样，在状态转换图中，我们用**结点（Node）**表示状态；用**弧线（Arc）**表示转移函数，从状态 q 到状态 p 的弧上面标有所有能够让 DFA 从状态 q 跳转到状态 p 的输入符号；用标有 $start$ 的箭头指向初始状态；用**双圆（Double Circle）**表示终止状态。比如说**图 3**是一个可以识别输入串中是否含有 ing 的 DFA。

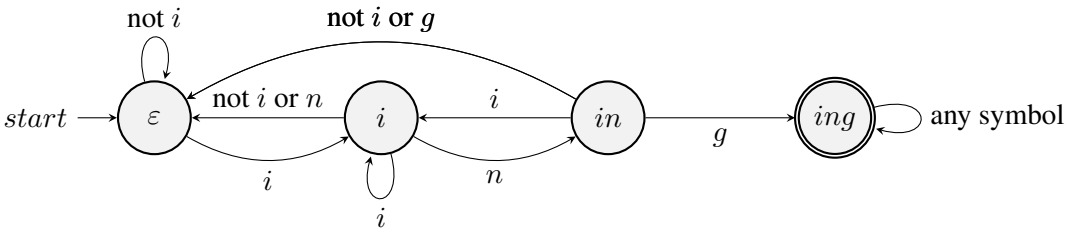


图 3: 识别英语中 ing 的 DFA

在转移表中，我们用行头（每一行的开头）表示状态，用列头（每一列的开头）表示输入符号，用箭头指向初始状态，用星号标记终止状态集。表中的每一个表项表示转移函数在行头状态和列头输入符号作用下的结果，比如说处于 q 行 a 列的状态 p 表示 $\delta(q, a) = p$ 。

		0	1
→	A	A	B
*	B	A	C
*	C	C	C

表 3: 转移表样例

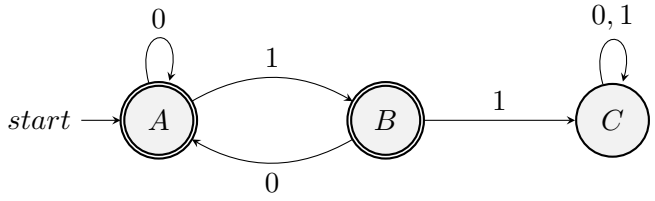


图 4: 与转移表**表 3**等价的状态转换图

表 3是一个转移表样例，**图 4**是与之等价的状态转换图，它们表示的是同一个 DFA。通常，状态数比较多的时候，用转移表描述 DFA 更加方便；状态数比较少的时候，用状态转换图描述 DFA 更加直观；在书写严格数学证明的时候，用定义2.1来描述 DFA 更加严谨。

2.2.3 DFA 的语言

一个自动机能够接收字符串输入，并根据最终状态是否是终止状态判断接受还是拒绝。对于一个 DFA 来说，从状态转换图上来看，它所能接受的所有字符串就是状态转换图中从初始状态到终止状态的所有路径所代表的字符串的集合。

定义 2.3. 对于一个 DFA $A = (Q, \Sigma, \delta, q_0, F)$ ，称字符串 w 被 A **接受**，如果

$$\delta(q_0, w) \in F$$

记 A 能接受的所有字符串组成的集合为 **DFA A 的语言**，记为 $L(A)$ 。形式化表述为：

$$L(A) = \{w | \delta(q_0, w) \in F\}$$

从定义2.3中，我们会发现，除了通过集合的描述来定义语言之外，给出一个自动机也能够定义一种语言。

下面我们举一个例子，来帮助读者熟悉一下定义2.3以及自动机相关证明的书写风格。请读者思考：**图 4**所描述的 DFA，它所定义的语言是什么?

应该不难看出，**图 4** DFA ，不妨记为 A ，A 所定义的语言是：由 0 和 1 组成的，不含连续 1 的所有字符串构成的集合。即

$$L(A) = \{w | w \in \{0,1\}^* \wedge w \text{ 不含两个连续的1}\}$$

下面我们来证明这个结论。

证明. 记 $L = \{w | w \in \{0,1\}^* \wedge w \text{ 不含两个连续的1}\}$ ，**图 4**中的 DFA 为 A ，我们将会分两部分来证明：$L = L(A)$ 。

先证 $L(A) \subseteq L$ ，即 $\forall w \in L(A), w \in L$ ，也就是说：如果字符串 w 被 DFA A 接受，那么 w 中没有连续的 1。下面通过对 w 的长度进行归纳来证明如下两个结论：

1. 如果 $\delta(A, w) = A$ ，那么 w 没有连续的两个 1 且不以 1 结尾。
2. 如果 $\delta(A, w) = B$ ，那么 w 没有连续的两个 1 且以单个 1 结尾。

基础情况：$|w| = 0$ ，即 $w = \varepsilon$ 时，结论 1 显然成立；因为 $\delta(A, \varepsilon) = A \neq B$ ，所以结论 2 也成立（假命题蕴含任何命题）。

归纳假设：假设结论 1，2 对于长度比 $|w| \geq 1$ 短的输入字符串都成立。

归纳步骤：下证结论 1，2 对于 w 也成立，由于 $|w| \geq 1$ ，不妨设 $w = xa$ ，其中 $a \in \{0,1\}$ 。

1. 如果 $\delta(A, w) = A$ ，根据**图 4**定义的转移函数，$\delta(A, x)$ 一定是 A 或者 B ，且 a 一定是 0 ，由归纳假设可知，x 没有两个连续的 1 ，从而 $w = xa = x0$ 也没有两个连续的 1 且不以 1 结尾。
2. 如果 $\delta(A, w) = B$ ，根据**图 4**定义的转移函数，$\delta(A, x)$ 一定是 A 且 a 一定是 1 ，由归纳假设可知 x 没有两个连续的 1 且不以 1 结尾，从而 $w = xa = x1$ 也没有两个连续的 1 且以单个 1 结尾（因为 x 的末尾不是 1 ）。

于是，只要一个字符串 w 被 DFA A 接受，等价于 $\delta(A, w) = A$ 或者 $\delta(A, w) = B$，那么 w 一定没有两个连续的 1。

再证 $L \subseteq L(A)$ ，即 $\forall w \in L, w \in L(A)$ ，也就是说：如果 w 没有两个连续的 1 ，那么它会被**图 4**给出的 DFA A 所接受。

反证法。假设 w 没有被 DFA 接受，由于只有 C 不是终止状态，我们有 $\delta(A, w) = C$ 。根据转移函数，我们可以将 w 写作 $w = x1y$ ，因为从初始状态 A 走到 C 必须先经过 $\delta(B, 1) = C$ 这条转移，其中 $\delta(A, x) = B$ 。类似的，根据 $\delta(A, x) = B$ ，我们可以将 x 写作 $x = z1$ ，因为从初始状态 A 走到 B 的最后一步只能是 $\delta(A, 1) = B$ 。因此 $w = z11y$ ，这与 w 没有两个连续的 1 是矛盾的。

综上，我们有 $L(A) \subseteq L \wedge L \subseteq L(A)$ ，从而 $L = L(A)$ 得证。

□

2.2.4 正则语言

定义 2.4. 如果一个语言 L 可以由某个 DFA A 定义，即 $L = L(A)$ ，则称语言 L 是**正则的（Regular）**，是一个**正则语言（Regular Language）**。所有正则语言组成的集合称为**正则语言类（Regular Languages）**，记为 RE ，则

$$RE = \{L | L = L(A), A \text{ is a DFA}\}$$

定义2.4是正则语言的形式化定义，下面我们通过几个例子先简单了解一下正则语言，到本文的最后，你会对正则语言有一个十分透彻的掌握。

在此之前，我们已经见过了不少的 DFA，它们所定义的语言都是正则语言。所以，我们先看一个不是正则语言的例子。

定理 2.1. $L = \{0^n 1^n | n \geq 1\}$ 不是正则语言。

证明. L 其实就是形如连续 n 个 0 紧接着连续 n 个 1 的字符串组成的集合，下面我们来证明这个语言不是正则的，即不存在任何 DFA 能够定义这个语言。

反证法，假设存在一个有 m 个状态的 DFA A ，使得 $L = L(A)$ 。考虑输入字符串 $0^m 1^m \in L$ 所产生的状态变化（左侧字母表示状态，右侧表示剩余的输入）：

$$S_0 0^m 1^m \to S_1 0^{m-1} 1^m \to \cdots \to S_m 1^m \to \cdots \to S_{2m-1} 1 \to S_{2m}$$

观察前 m 次状态转移，产生了 $S_0 \sim S_m$ 共 $m+1$ 种状态，而 DFA A 只有 m 个状态，根据鸽笼原理，至少有一个状态出现了两次，即存在 $0 \leq i < j \leq m$ ，使得 $S_i = S_j$ ，于是有

$$S_0 0^m 1^m \to \cdots \to S_i 0^{m-i} 1^m \to \cdots \to S_j 0^{m-j} 1^m \to \cdots \to S_{2m}$$

考虑输入字符串 $0^{m-j+i} 1^m$ ，基于先前的状态变化序列，我们有

$$S_0 0^{m-j+i} 1^m \to \cdots \to S_i 0^{m-j+i-i} 1^m = S_i 0^{m-j} 1^m = S_j 0^{m-j} 1^m \to \cdots \to S_{2m}$$

于是，$0^{m-j+i} 1^m \in L(A)$ ，但是 $0^{m-j+i} 1^m \notin L$ ，这与 $L = L(A)$ 矛盾。因此，不存在使得 $L = L(A)$ 的 DFA A 。

□

此外，$L = \{w | w \in \{(,)\}^* \wedge w$中的括号是平衡的$\}$ 这个语言也不是正则语言。一个直观的感觉是，DFA 并不具备“计数”变量的能力。

当然，也有很多语言是正则的，比如说 $L = \{w | w$**是一个浮点数的字符串表示**$\}$ 这个语言是正则的，因为它可以用正则表达式书写，后续我们会具体学习正则表达式，并且证明它和 DFA 的等价性，这里暂时按下不表。

再比如说，$L = \{w | w \in \{0,1\}^* \wedge 23 | w\}$ ，这里判断是否 $23 | w$ 时将 w 视为一个二进制无符号整数，ε 视为 0。L 也是一个正则语言。

证明. 下面通过构造可以定义 L 的 DFA 来证明 L 是一个正则语言。

因为任意一个数模 23 只会有 23 种有限的情况，我们可以用 23 个状态来表示，记为 $0, 1, 2, \cdots, 22$ ，唯一的初始状态和终止状态就是 0 。下面构造转移函数即可，我们递归地给出转移函数的定义：

基础情况：$\delta(0,\varepsilon)=0$

递归情况：假设 $\delta(0,w)=i$，则：$w0$ 表示 $2i$，取 $\delta(i,0)=(2i \bmod 23)$；$w1$ 表示 $2i+1$，取 $\delta(i,1)=(2i+1 \bmod 23)$。

从而我们构造出了能够定义 L 的 DFA $A=(\{0,1,2,\cdots,22\},\{0,1\},\delta,0,\{0\})$，其中 δ 的定义如上所述，因此 L 是一个正则语言。

（此处如果严谨的话还需要补充证明 $L=L(A)$，但这很容易看出来，笔者就不再多费笔墨了）

□

2.3 非确定性有穷自动机

2.3.1 定义

之前已经有了定义 DFA 的经验，这里我们先不加引导地给出非确定性有穷自动机的相关定义，然后再通过例子加以阐释。

定义 2.5. 定义**非确定性有穷自动机（Nondeterministic Finite Automaton, NFA）**为一个五元组：

$$(Q,\Sigma,\delta,q_0,F)$$

其中，Q,Σ,q_0,F 的定义同定义2.1。$\delta\subseteq Q\times\Sigma\times 2^Q$ 是一个转移函数，它接受两个参数：一个状态和一个输入符号，并返回一个状态的集合，即 $\delta(q,a)\subseteq Q$。

$\delta(q,a)=\{p_1,p_2,\cdots,p_k\},k\geq 0$ 意味着 NFA 处于状态 q 时接收输入符号 a 可以进入状态 $p_1,p_2,\cdots,p_k$ 中的任意一个。

为了书写简洁，后续我们直接用缩写 NFA 来表示非确定性有穷自动机。

定义 2.6. 递归定义 NFA 的**扩展转移函数** $\hat{\delta}$ 为

$$\hat{\delta}(q,w)=\begin{cases}\{q\} & \text{if } w=\varepsilon\\ \bigcup\limits_{p\in\hat{\delta}(q,x)}\delta(p,a) & \text{if } w=xa\end{cases}$$

其中，w 是一个字符串，a 是一个输入符号，wa 表示这两者的拼接，在拼接过程中将 a 视为长度为 1 的字符串。

和定义2.2类似，习惯上，不区分书写 δ 和 $\hat{\delta}$，统一都用 δ 表示，至于具体含义是定义2.5中的转移函数还是2.6中的扩展转移函数，根据具体语境下的参数类型来决定。

定义 2.7. 称一个字符串 w 被一个 NFA $A=(Q,\Sigma,\delta,q_0,F)$ **接受**，如果

$$\delta(q_0,w)\cap F\neq\emptyset$$

一个 **NFA A 的语言**就是 A 所能接受的所有字符串的集合，记为 $L(A)$，形式化表述即为

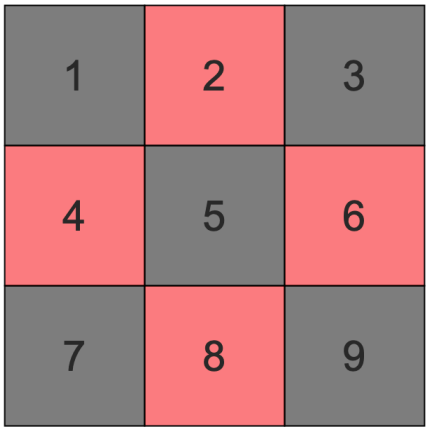
$$L(A)=\{w|\delta(q_0,w)\cap F\neq\emptyset\}$$

下面我们来稍微解释一下定义2.5、2.6、2.7。从运行状态的角度来看，NFA 可以同一时刻处于多种状态之中，因为当有一个符号输入的时候，它可以跳转到一个由转移函数定义的状态集

合中的任意状态。因为 NFA 在每个时刻处于哪个状态其实是不确定的，所以称之为非确定性有穷自动机。其中，当我们谈论时刻的时候，指的是 NFA 消耗掉输入字符串中的每一个符号的时刻。

NFA 的每次状态跳转都可能有多种选择，因此哪怕是同一个输入，对每个输入符号每次状态的选择组成的序列不同，NFA 也会跳转到不同的状态。但是根据定义2.7，只要在该输入下有任意一组状态选择的序列能够使得 NFA 从 q_0 跳转到 F 中的任何一个状态，我们就认为这个输入被接受了。

因此，直观上来看，虽然转移函数给出了多个跳转目标，但是我们可以等价地认为，对于每一种输入，NFA 一定会一直选择最“正确”的那个目标，直至走到终止状态；除非在该输入下无论怎么选择每一次跳转的状态，都无法到达终止状态，这个时候 NFA 就会拒绝这个输入，也就是说这个输入并不在 NFA 所定义的语言当中。



(a) 棋盘

		r	b
→	1	2, 4	5
	2	4, 6	1, 3, 5
	3	2, 6	5
	4	2, 8	1, 5, 7
	5	2, 4, 6, 8	1, 3, 7, 9
	6	2, 8	3, 5, 9
	7	4, 8	5
	8	4, 6	5, 7, 9
*	9	6, 8	5

(b) NFA 的转移表

图 5: NFA 棋盘样例的示意图和转移表

最后我们用一个例子来结束对于 NFA 定义的阐释。考虑**图 5a**中的双色棋盘，定义一个 NFA 如下：将棋子的位置视为棋盘的状态，输入是以 $\{r, b\}$ 为字母表的字符串，其中 r 表示移动到相邻的红色格子中，b 表示移动到相邻的黑色格子中。初始状态是棋子在左上角的情况，即状态 1；终止状态是棋子在右下角的情况。根据如上定义，我们可以给出这个 NFA 的转移表**表 5b**。

如果接受输入 rbb，这个 NFA 的状态变化情况如**图 6**所示，其中每一列表示接收了对应输入的时刻 NFA 可能处在的状态。最终这个 NFA 可能处在五种状态中，$\delta(1, rbb) = \{5, 1, 3, 7, 9\}$。并且，只有当状态变化恰好是 $1 \to 4 \to 5 \to 9$ 这条路径的时候才能走到终止状态。

根据定义以及我们对于定义的解释，只要有成功接受的可能，NFA 就一定会选择最对的路径，也就是 NFA 一定会选择 $1 \to 4 \to 5 \to 9$，从而接受 rbb 这个输入字符串。

相信看到这里，你会有一种感觉：NFA 看起来是比 DFA“强”的。毕竟，NFA 的定义相比于 NFA 来说给了更多的自由度：我们不仅可以在一定范围内去选择下一步的状态是什么，并且 NFA 一定会选择最正确的那条路径，除非不存在任何正确的路径。

不过这其实是一种错觉，NFA 和 DFA 在语言的表达能力上是等价的，他们所能够定义的语言组成的集合都是正则语言类。下一小节，我们将会证明这一点。

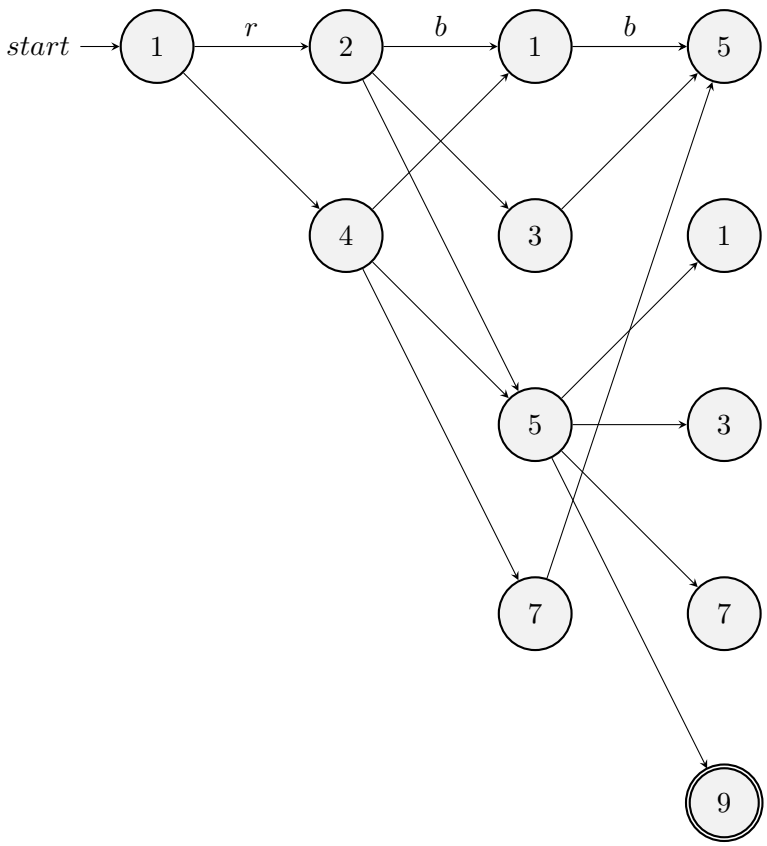


图 6: NFA 棋盘接收输入 rbb 时的状态变化（注意这并不是 NFA 的状态转换图）

2.3.2 NFA 与 DFA 的等价性

定理 2.2. *DFA 和 NFA 在语言的表达能力上是等价的，NFA 的语言也是正则语言。具体地，*

- *能够用 DFA 定义的语言，一定存在某个 NFA，可以定义出相同的语言；*
- *能够用 NFA 定义的语言，一定存在某个 DFA，可以定义出相同的语言。*

证明. 我们只需要证明 DFA 和 NFA 之间可以相互转化就可以了。

首先，DFA 可以转化成 NFA 是显然的，因为 DFA 就相当于是转移函数总返回单元集的 NFA。

其次，我们证明 NFA 可以转化成 DFA，通过给出一种构造方法来证明。任意给定一个 NFA，可以通过如下的方式构造一个定义相同语言的 DFA，该方法称为**子集构造（Subset Construction）**。

假设原 NFA 为 $N = (Q, \Sigma, \delta_N, q_0, F_N)$，基于 N 构造 DFA $D = (2^Q, \Sigma, \delta_D, \{q_0\}, F_D)$。其中状态集 2^Q 是 Q 的幂集，终止状态集 F_D 为

$$F_D = \{F \in 2^Q | F \cap F_N \neq \emptyset\}$$

转移函数 δ_D 为

$$\delta_D(\{q_1, q_2, \cdots q_k\}, a) = \bigcup_{i=1}^{k} \delta_N(q_i, a)$$

下面证明 $L(N) = L(D)$。

先证对于任意的字符串 w，$\delta_N(q_0, w) = \delta_D(\{q_0\}, w)$。对 $|w|$ 进行归纳：

基础情况：$w = \varepsilon$ 时，$\delta_N(q_0, \varepsilon) = \delta_D(\{q_0\}, \varepsilon) = \{q_0\}$ 显然成立。

归纳假设：假设结论对于长度比 w 更短的字符串都成立。

归纳步骤：令 $w = xa$ ，根据 δ_N 和 δ_D 的定义，有

$$\delta_N(q_0, w) = \bigcup_{q \in \delta_N(q_0, x)} \delta_N(q, a)$$

$$\delta_D(\{q_0\}, w) = \delta_D(\delta_D(\{q_0\}, x), a) = \bigcup_{q \in \delta_D(\{q_0\}, x)} \delta_N(q, a)$$

根据归纳假设，$\delta_N(q_0, x) = \delta_D(\{q_0\}, x)$ ，因此 $\delta_N(q_0, w) = \delta_D(\{q_0\}, w)$ 。

于是，对于任意的字符串 w ，

$$w \in L(N) \Leftrightarrow \delta_N(q_0, w) \cap F_N \neq \emptyset \Leftrightarrow \delta_D(\{q_0\}, w) \cap F_N \neq \emptyset \Leftrightarrow \delta_D(\{q_0\}, w) \in F_D \Leftrightarrow w \in L(D)$$

从而 $L(N) = L(D)$ 。

综上，DFA 和 NFA 在语言的表达能力上是等价的，NFA 的语言也是正则语言。

□

在上述证明中需要再阐释一下的是，DFA D 状态集中的元素是 NFA N 状态集中元素的集合。比如说如果 p, q 是 NFA 的两个状态，那么 $\{p, q\}$ 就是 DFA 中的一个状态，只不过这个状态是一个集合而已。在定义2.1中，并没有限制状态集 Q 中的元素不可以是集合。

此外，虽然在证明中我们定义了 D 的状态集为 2^Q ，但其实并不是每个状态都是可达的，会有很多孤立的状态，因为 δ_D 有可能返回空集。只不过因为它们并不影响数学证明，为了书写方便，我们并没有强行将这些孤立节点剥离掉。但是，在编程实践中，为了节约资源，将 NFA 转化成 DFA 的时候，一般是不保留从初始状态不可达的状态的。

对于具体的例子来说，做 NFA 到 DFA 的转换只需要从初始状态 $\{q_0\}$ 开始，根据原 NFA 的转换表，求从 $\{q_0\}$ 可达的传递闭包就行了。转移表**表 5b**所定义的 NFA 对应的 DFA 的转移表为**表 4**，这个 DFA 的状态集为 $\{\{1\}, \{2,4\}, \{5\}, \{2,4,6,8\}, \{1,3,5,7\}, \{1,3,7,9\}, \{1,3,5,7,9\}\}$ ，初始状态为 $\{1\}$ ，终止状态集为 $\{\{1,3,7,9\}, \{1,3,5,7,9\}\}$ ，转移函数如**表 4**所示。

		r	b
→	$\{1\}$	$\{2,4\}$	$\{5\}$
	$\{2,4\}$	$\{2,4,6,8\}$	$\{1,3,5,7\}$
	$\{5\}$	$\{2,4,6,8\}$	$\{1,3,7,9\}$
	$\{2,4,6,8\}$	$\{2,4,6,8\}$	$\{1,3,5,7,9\}$
	$\{1,3,5,7\}$	$\{2,4,6,8\}$	$\{1,3,5,7,9\}$
*	$\{1,3,7,9\}$	$\{2,4,6,8\}$	$\{5\}$
*	$\{1,3,5,7,9\}$	$\{2,4,6,8\}$	$\{1,3,5,7,9\}$

表 4: 与转移表**表 5b**所描述的 NFA 定义相同语言的 DFA 的转移表

2.4 带空转移的非确定性有穷自动机

2.4.1 定义

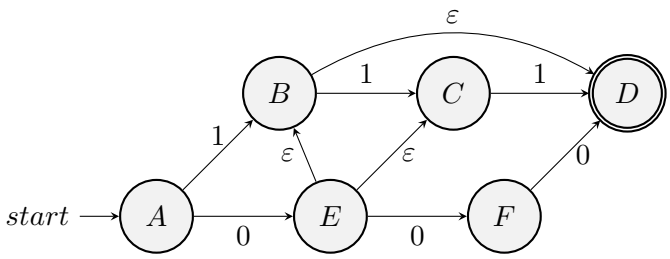
定义 2.8. 定义**带空转移的非确定性有穷自动机（ε-NFA）**为一个五元组：

$$(Q, \Sigma, \delta, q_0, F)$$

其中，Q, Σ, q_0, F 的定义同定义2.1。$\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times 2^Q$ 是一个转移函数，它接受两个参数：一个状态和一个输入符号或者 ε，并返回一个状态的集合，即 $\delta(q, a) \subseteq Q$，$\delta(q, \varepsilon) \subseteq Q$。

δ 基本含义同定义2.5。$\delta(q, \varepsilon) = \{p_1, p_2, \cdots, p_k\}, k \geq 0$ 意味着 NFA 处于状态 q 时可以不消耗输入而进入状态 $p_1, p_2, \cdots, p_k$ 中的任意一个。并且，必然有 $q \in \delta(q, \varepsilon)$。

图 7是一个简单的 ε-NFA 的状态转换图及其转移表的示例。



(a) ε-NFA 状态转换图

		0	1	ε
→	A	$\{E\}$	$\{B\}$	$\{A\}$
	B	$\emptyset$	$\{C\}$	$\{B, D\}$
	C	$\emptyset$	$\{D\}$	$\{C\}$
	D	$\emptyset$	$\emptyset$	$\{D\}$
*	E	$\{F\}$	$\emptyset$	$\{B, C, E\}$
	F	$\{D\}$	$\emptyset$	$\{F\}$

(b) ε-NFA 的转移表

图 7: 同一个 ε-NFA 的状态转换图和转移表

定义 2.9. 定义状态 q 的**状态闭包（Closure of States）**是从状态 q 开始通过接收一个或者若干个 ε 能够到达的状态集合，记为 $CL(q)$。定义状态集合 Q 的闭包为

$$CL(Q) = \bigcup_{q \in Q} CL(q)$$

例如，在图 7所描述的 ε-NFA 中，有 $CL(A) = \{A\}, CL(E) = \{B, C, D, E\}$。

定义 2.10. 定义 ε-NFA 的**扩展转移函数为**：

$$\hat{\delta}(q, w) = \begin{cases} CL(q) & \text{if } w = \varepsilon \\ \bigcup\limits_{p \in \hat{\delta}(q,x)} CL(\delta(p, a)) & \text{if } w = xa \end{cases}$$

直观上，$\hat{\delta}(q, x)$ 应该是从状态 q，通过代表的字符串是 x 的路径所能到达的所有状态的集合；其中，每一步如果有 ε 边都可以选择直接跳转，当然，也可以选择不走 ε 边。比如说，在图 7所描述的 ε-NFA 中，我们有 $\hat{\delta}(A, \varepsilon) = CL(A) = \{A\}$，$\hat{\delta}(A, 0) = CL(\{E\}) = \{B, C, D, E\}$，$\hat{\delta}(A, 01) = CL(\{C, D\}) = \{C, D\}$。和之前 DFA、NFA 扩展转移函数类似，我们习惯上不区分书写 δ 和 $\hat{\delta}$，统一写作 δ，具体含义根据具体语境下参数的类型决定。

定义 2.11. 称一个字符串 w 被一个 ε-NFA $A = (Q, \Sigma, \delta, q_0, F)$ **接受**，如果

$$\delta(q_0, w) \cap F \neq \emptyset$$

一个 **ε-NFA A 的语言**就是 A 所能接受的所有字符串的集合，记为 $L(A)$，形式化表述为

$$L(A) = \{w|\delta(q_0, w) \cap F \neq \emptyset\}$$

看到这里，你可能又会产生种感觉：ε-NFA 看起来是比 NFA 更“强”的。毕竟 ε-NFA 的定义在 NFA 的基础上放松了“空转移”这个更多的自由度，并且依旧保持着 NFA 一定会选择最正确的路径这一“非确定性”的性质。

不过和之前一样，这依旧是一种错觉，和定理2.2类似，ε-NFA 和 NFA 在语言的表达能力上是等价的，ε-NFA 所能够定义的语言组成的集合依旧是正则语言类。下一小节，我们将会证明这一点。

2.4.2 ε-NFA 与 NFA 的等价性

定理 2.3. *NFA 和 ε-NFA 在语言的表达能力上是等价的，ε-NFA 的语言也是正则语言。具体地，*

- *能够用 NFA 定义的语言，一定存在某个 ε-NFA，可以定义出相同的语言；*
- *能够用 ε-NFA 定义的语言，一定存在某个 NFA，可以定义出相同的语言。*

证明. 我们只需要证明 NFA 和 ε-NFA 之间可以相互转化就可以了。

首先，NFA 可以转化成 ε-NFA 是显然的，因为 NFA 就相当于是一个不含 ε 边的 ε-NFA 。

其次，我们证明 ε-NFA 可以转化成 NFA。考虑任意 ε-NFA $E = (Q, \Sigma, \delta_E, q_0, F_E)$，构造 NFA $N = (Q, \Sigma, \delta_N, q_0, F_N)$ 。其中

$$\delta_N(q, a) = \bigcup_{p \in CL(q)} \delta_E(p, a)$$
$$F_N = \{q \in Q|CL(q) \cap F_E \neq \emptyset\}$$

下面证明 $L(E) = L(N)$ 。

先证对于任意的字符串 w ，$\delta_E(q_0, w) = CL(\delta_N(q_0, w))$ 。对 $|w|$ 进行归纳：

基础情况：$w = \varepsilon$ 时，$\delta_E(q_0, \varepsilon) = CL(q_0) = CL(\{q_0\}) = CL(\delta_N(q_0, \varepsilon))$ ，结论显然成立。

归纳假设：假设结论对于长度小于 $|w| \geq 1$ 的字符串都成立。

归纳步骤：令 $w = xa$ ，根据 δ_E 和 δ_N 的定义，有

$$\delta_E(q_0, w) = \bigcup_{p \in \delta_E(q_0, x)} CL(\delta_E(p, a))$$
$$\delta_N(q_0, w) = \delta_N(\delta_N(q_0, x), a) = \bigcup_{p \in CL(\delta_N(q_0, x))} \delta_E(p, a)$$

根据归纳假设，$\delta_E(q_0, x) = CL(\delta_N(q_0, x))$，所以

$$\begin{aligned} CL(\delta_N(q_0, w)) &= CL(\bigcup_{p \in CL(\delta_N(q_0, x))} \delta_E(p, a)) \\ &= \bigcup_{p \in CL(\delta_N(q_0, x))} CL(\delta_E(p, a)) \\ &= \bigcup_{p \in \delta_E(q_0, x)} CL(\delta_E(p, a)) \\ &= \delta_E(q_0, w) \end{aligned}$$

即 $\delta_E(q_0, w) = CL(\delta_N(q_0, w))$ 。

于是，对于任意的字符串 w ，

$$w \in L(E) \Leftrightarrow \delta_E(q_0, w) \cap F_E \neq \emptyset \Leftrightarrow CL(\delta_N(q_0, w)) \cap F_E \neq \emptyset \Leftrightarrow \delta_N(q_0, w) \cap F_N \neq \emptyset \Leftrightarrow w \in L(N)$$

从而 $L(E) = L(N)$ 。

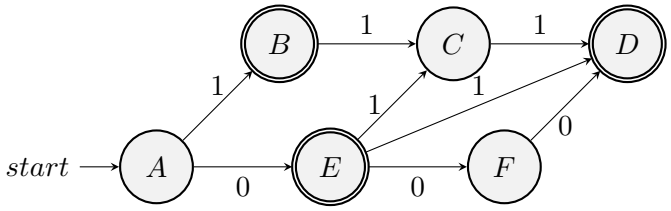
综上，NFA 和 ε-NFA 在语言的表达能力上是等价的，ε-NFA 的语言也是正则语言。

□

从 ε-NFA 到 NFA 的转换的核心思路就是将空转移用普通转移来代替，也就是说如果从状态 q 到状态 p 需要通过 ε 和 1 两条边，那么我们直接用一条从 q 到 p 的带有标签 1 的边来代替就可以了，从而我们能够将所有的 ε 边去掉。

有一个细节是由于我们的转移函数总是先取闭包，再转移，即总是将相邻的 εa 两条边用 a 代替，我们最终会缺少一次闭包操作，因为 ε-NFA 中实际的路径（用边表示）应该是形如 $\varepsilon, a_1, \varepsilon, a_2, \cdots, \varepsilon, a_k, \varepsilon$ 的。因此，如果终止状态不是 a_k 的目标状态，而是 a_k 的目标状态通过 ε 边到达的其他状态，这种情况会被转移函数给漏掉。解决方案是我们同时修改了终止状态集，将原本 ε-NFA 中终止状态集的闭包作为转化后 NFA 的终止状态集，这样一来，最后的一次 ε 跳转就不需要转移函数来完成了。

具体地，读者可以自行尝试将**图 7**所定义的 ε-NFA 转化成 NFA。结果如**图 8**所示。



(a) NFA 状态转换图

		0	1
→	A	$\{E\}$	$\{B\}$
*	B	$\emptyset$	$\{C\}$
	C	$\emptyset$	$\{D\}$
*	D	$\emptyset$	$\emptyset$
*	E	$\{F\}$	$\{C, D\}$
	F	$\{D\}$	$\emptyset$

(b) NFA 的转移表

图 8: 与**图 7**中 ε-NFA 等价的 NFA

到此为止，有穷自动机的大部分内容就介绍完毕了。DFA，NFA 和 ε-NFA 在语言的表达能力上是等价的，它们所能定义的语言的集合是相同的，都是正则语言。NFA 和 ε-NFA 在定义上给了更多的自由度——一定会猜中最对的路径、空转移，这使得在使用 NFA 和 ε-NFA 进行设计的时候状态数可以相较于 DFA 来说成指数级地减少。但是，在计算机的世界当中不存在“非确定性”这件事，计算机运行的每一步都是确定的，因此一定会猜中这件事情是无法直接实现的，只有 DFA 可以被实现，不过好在我们已经证明了 NFA 和 ε-NFA 都可以转化成 DFA。

3 正则表达式

3.1 定义

与有穷自动机不同，**正则表达式 (Regular Expression, RE)** 通过代数式的方式来描述语言。并且，正则表达式恰好能够描述正则语言，我们之后会证明它和有穷自动机在语言表达能力上

的等价性。

定义 3.1. 记一个正则表达式 E 的语言为 $L(E)$ 。递归地定义**正则表达式**如下：

基础情况：

1. 如果 a 是一个符号，那么 a 是一个正则表达式，且 $L(a) = \{a\}$ ；
2. ε 是一个正则表达式，且 $L(\varepsilon) = \{\varepsilon\}$ ；
3. $\emptyset$ 是一个正则表达式，且 $L(\emptyset) = \emptyset$ 。

递归情况：

1. 如果 E_1 与 E_2 是正则表达式，那么 $E_1 + E_2$ 也是一个正则表达式，且
$$L(E_1 + E_2) = L(E_1) \cup L(E_2)$$
2. 如果 E_1 与 E_2 是正则表达式，那么 E_1E_2 也是一个正则表达式，且
$$L(E_1E_2) = L(E_1)L(E_2)$$
3. 如果 E 是一个正则表达式，那么 E^* 也是一个正则表达式，且
$$L(E^*) = L(E)^*$$

正则表达式是由三种基本表达式（符号、ε、$\emptyset$ ）以及三种基本运算（并集、拼接、星闭包）组成的。我们可以使用括号来指定运算的优先级；默认情况下，星闭包优先级最高，其次是拼接，再其次才是并集。下面是一些正则表达式的例子：

$L(01) = \{01\}$

$L(01 + 0) = \{01, 0\}$

$L(0(1 + 0)) = \{01, 00\}$

$L(0^*) = \{0, 00, 000, 0000, \cdots\}$

$L((0 + 10)^*(\varepsilon + 1)) = \{w | w \in \{0, 1\}^* \wedge w$ 不含两个连续的1$\}$

并且，我们不难发现，正则表达式的运算和代数运算类似，满足一些运算律。并操作有点像加法，满足交换律和结合律；拼接操作有点像乘法，满足结合律，但是不满足交换律，因为符号内容相同但是符号顺序不同的字符串被认为是两个不同的字符串；$\emptyset$ 相当于代数运算中的 0 ，满足 $R + \emptyset = R, R\emptyset = \emptyset R = \emptyset$ ；ε 相当于代数运算中的 1 ，满足 $\varepsilon R = R\varepsilon = R$ 。

在编程实践中，我们使用的正则表达式其实是一个扩展版的正则表达式，有除了上述三种运算以外的其他运算。但是，这些其他运算仅仅只是一些方便的语法糖而已，定义3.1中的三种运算——并集、拼接、星闭包——才是最基本的，其他任何运算都可以通过这三种运算来实现——就好像逻辑表达式中的各种复杂的逻辑运算符都可以使用与或非来实现一样。比如说，在一般编程环境中表示十进制整数（多位数不以 0 开头）的正则表达式`0|[1-9][0-9]*`，用定义3.1表述为 $0 + (1 + 2 + \cdots + 9)(0 + 1 + \cdots + 9)^*$ 。

正则表达式所能够定义的语言依旧是正则语言，为了证明这一点，下一节我们会讨论正则表达式与有穷自动机的等价性。

3.2 正则表达式与有穷自动机的等价性

定理 3.1. 正则表达式和有穷自动机在语言的表达能力上是等价的。正则表达式的语言也是正则语言。具体地，

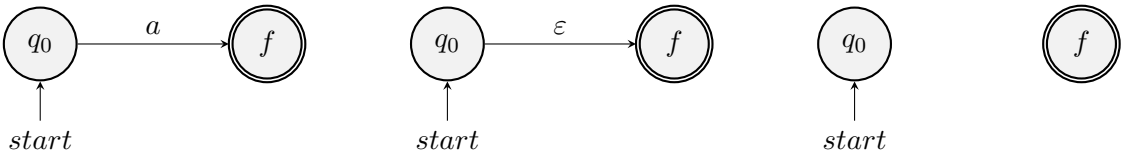
- 能够用正则表达式定义的语言，一定存在某个有穷自动机，可以定义出相同的语言；
- 能够用有穷自动机定义的语言，一定存在某个正则表达式，可以定义出相同的语言。

证明. 我们只需要证明有穷自动机可以和正则表达式相互转化即可。由于在定理2.2和定理2.3中已经证明了三种有穷自动机之间可以相互转化，我们只需要任意选用其中一种就可以了。

首先证明任意正则表达式 E 总是可以转化只有一个终止状态的 ε-NFA N ，对 E 中的操作符数量进行归纳，给出递归的构造方法并说明等价性。

基础情况：当 E 不含操作符时

1. $E = a$ ，构造 $N = \{\{q_0, f\}, \{a\}, \{(q_0, a, \{f\})\}, q_0, \{f\}\}$ ，如**图 9a**，$L(N) = L(E)$ 显然成立；
2. $E = \varepsilon$ ，构造 $N = \{\{q_0, f\}, \emptyset, \{q_0, \varepsilon, \{f\}\}, q_0, \{f\}\}$ ，如**图 9b**，$L(N) = L(E)$ 显然成立；
3. $E = \emptyset$ ，构造 $N = \{\{q_0, f\}, \emptyset, \emptyset, q_0, \{f\}\}$ ，如**图 9c**，$L(N) = L(E)$ 显然成立。



(a) 正则表达式 a 的 ε-NFA　　**(b)** 正则表达式 ε 的 ε-NFA　　**(c)** 正则表达式 $\emptyset$ 的 ε-NFA

图 9: 不含操作符的基础正则表达式对应的 ε-NFA

归纳假设：假设操作符个数比 E 少的所有正则表达式 E_1, E_2 都可以等价转化成

$$N_1 = \{Q_1, \Sigma_1, \delta_1, q_1, \{f_1\}\}, N_2 = \{Q_2, \Sigma_2, \delta_2, q_2, \{f_2\}\}$$

满足 $L(E_1) = L(N_1), L(E_2) = L(N_2)$ 。在后续图示时，我们采用如**图 10**的状态转换图画法表示归纳假设中的 ε-NFA 。其中灰色的椭圆表示子自动机的区域，该区域内没有画出的结点都不会有来自外部的弧，也不会有弧去往外部；用两个结点表示整“团”子自动机的“起始”状态和“终止”状态。

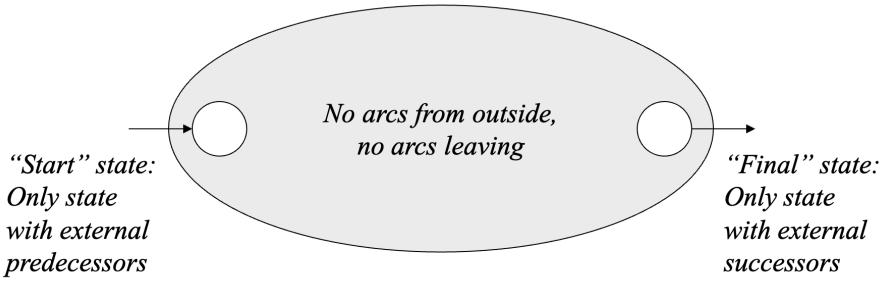


图 10: 基于子自动机构造新自动机时对于子自动机的画法

归纳步骤：下面证明 E 也可以转化成只有一个终止状态的 ε-NFA 。

1. $E = E_1 + E_2$ ，构造

   $$N = \{Q_1 \cup Q_2 \cup \{q_0, f\}, \Sigma_1 \cup \Sigma_2, \delta_1 \cup \delta_2 \cup \{(q_0, \varepsilon, \{q_1, q_2\}), (f_1, \varepsilon, \{f\}), (f_2, \varepsilon, \{f\})\}, q_0, \{f\}\}$$

   如**图 11**，不难发现 $L(E) = L(E_1 + E_2) = L(E_1) \cup L(E_2) = L(N_1) \cup L(N_2) = L(N)$ 。具体

地，可以参照定理2.2和定理2.3的证明方式对输入字符串的长度进行归纳来证明 $L(N_1) \cup L(N_2) = L(N)$ ，这里不再赘述。

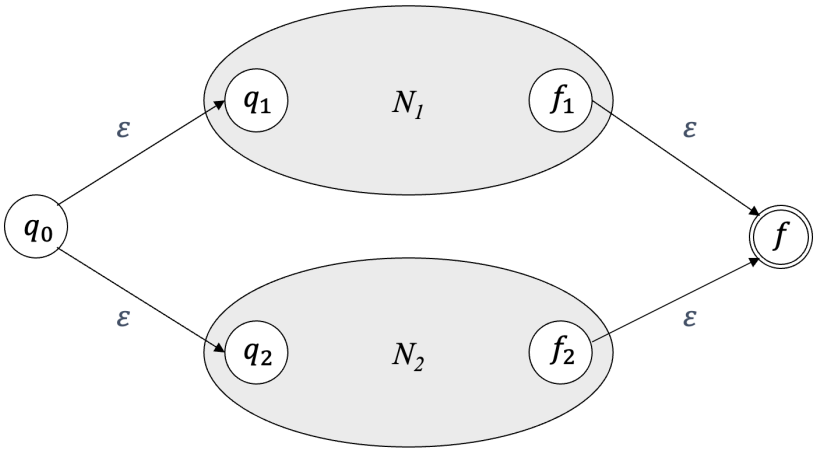


图 11: 基于 E_1, E_2 的 ε-NFA N_1, N_2 来构造 $E = E_1 + E_2$ 的 ε-NFA N

2. $E = E_1E_2$ ，构造

$$N = \{Q_1 \cup Q_2, \Sigma_1 \cup \Sigma_2, \delta_1 \cup \delta_2 \cup \{(f_1, \varepsilon, \{q_2\})\}, q_1, \{f_2\}\}$$

如**图 12**，不难发现 $L(E) = L(E_1E_2) = L(E_1)L(E_2) = L(N_1)L(N_2) = L(N)$ 。

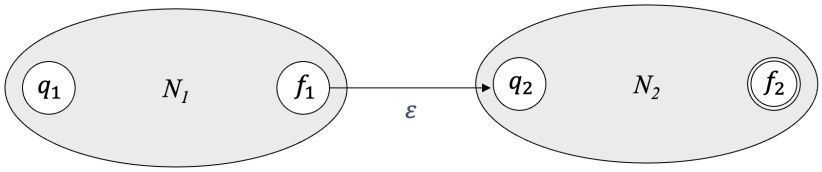


图 12: 基于 E_1, E_2 的 ε-NFA N_1, N_2 来构造 $E = E_1E_2$ 的 ε-NFA N

3. $E = E_1^*$ ，构造

$$N = \{Q_1 \cup \{q_0, f\}, \Sigma_1, \delta_1 \cup \{(q_0, \varepsilon, \{q_1, f\}), (f_1, \varepsilon, \{q_1, f\}), q_0, \{f\}\}$$

如图**图 13**，不难发现 $L(E) = L(E_1^*) = L(E_1)^* = L(N_1)^* = L(N)$ 。

到此为止，我们证明了任意的正则表达式 E 都可转化成一个 ε-NFA N 使得 $L(E) = L(N)$ ，因此能够被正则表达式定义的语言一定是正则语言。

下面我们来证明任意的 DFA $D = (Q, \Sigma, \delta, q_0, F)$ 总是可以转化成正则表达式 E 。假设 $|Q| = n$ ，不失一般性，设 $Q = \{1, 2, \cdots, n\}$ 。定义 k-路径为 DFA 状态转换图上的一条路径，且路径上除起点和终点外的任何结点最大不超过 k 。其中，对于一个 k-路径的起点和终点有多大、是谁，不作任何限制，只要除此之外不存在超过 k 的状态结点即可。于是我们可以明显地看到，D 中所有的路径都是 n-路径，因为 n 已经是整个 DFA 中最大的状态了。

由于 DFA 所定义的语言就是 DFA 中起点为初始状态，终点在终止状态集中的所有 n-路径所代表的字符串的集合，如果我们能够证明每个起点为 i ，终点为 j 的 k-路径都可以转化成正

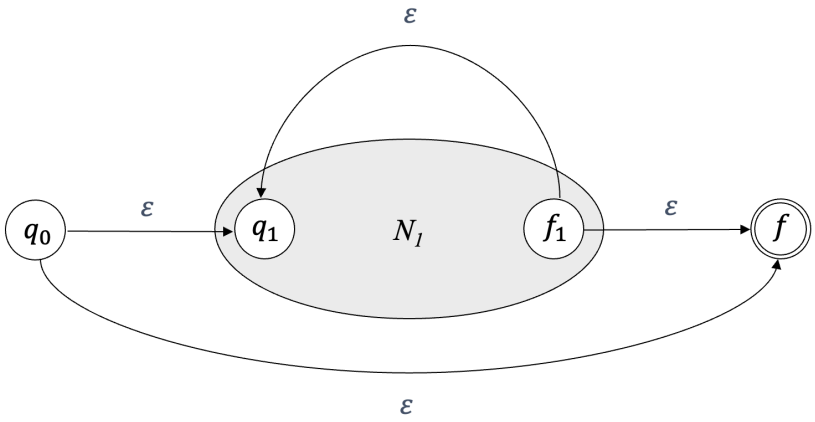


图 13: 基于 E_1 的 ε-NFA N_1 来构造 $E = E_1^*$ 的 ε-NFA N

则表达式 R_{ij}^k ，则显然有

$$L(D) = \bigcup_{i=q_0, j\in F} L(R_{ij}^n) = L(\sum_{i=q_0, j\in F} R_{ij}^n)$$

从而我们就构造出了和 DFA D 定义相同语言的正则表达式 $E = \sum_{i=q_0, j\in F} R_{ij}^n$ 。

因此，我们最后只需要证明每个起点为 i ，终点为 j 的 k-路径都可以转化成正则表达式 R_{ij}^k 就可以了，具体地，我们将通过对 k 进行归纳的方式给出 R_{ij}^n 的构造。

基础情况：$k = 0$ 时，R_{ij}^0 就是所有从 i 到 j 的弧上的输入符号的并集，因为 0-路径上不能有任何中间结点，形式化表述为：

$$R_{ij}^0 = \sum_{(i,a,j)\in\delta} a$$

因此，所有的 0-路径都能够转化成正则表达式。

归纳假设：假设所有的 $k-1$-路径都能转化成正则表达式。

归纳步骤：一条从 i 到 j 的 k-路径，中间要么没有经过 k ，形如 R_{ij}^{k-1} ；要么经过了 k ，形如 $R_{ik}^{k-1}(R_{kk}^{k-1})^* R_{kj}^{k-1}$ ，其中 $(R_{kk}^{k-1})^*$ 考虑的是可能多次经过 k 的情况。如**图 14**，因此

$$R_{ij}^k = R_{ij}^{k-1} + R_{ik}^{k-1}(R_{kk}^{k-1})^* R_{kj}^{k-1}$$

于是，k-路径也能够转化成正则表达式 R_{ij}^k。（在编程实践当中，这个过程可以用动态规划来实现。）

到此为止，我们也证明了任意的 DFA D 总可以转化成正则表达式 E ，使得 $L(D) = L(E)$ 。因此，只要是正则语言，就一定能够被正则表达式定义。

综上所述，正则表达式的语言不多不少就是正则语言，它在语言的表达能力上和有穷自动机是等价的。

□

到此为止，本文的三个重要的定理——定理2.2、定理2.3、定理3.1——都已经证明完毕了。这三个定理告诉我们，DFA、NFA、ε-NFA 和正则表达式在语言的表达能力上是等价的，它们所能够定义的语言都是正则语言，每一个正则语言都可以用这四种方式来定义。后续我们在研究正则语言的性质的时候就不再区分到底使用哪一种表示方法了，换句话说，如果我们要证明正

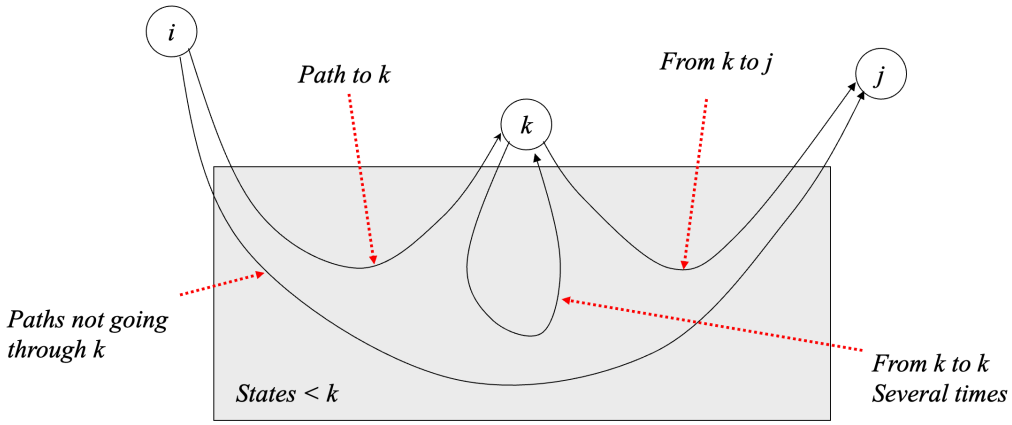


图 14: 从 $k-1$-路径拓展到 k 路径的方法

则语言具有某个性质，只需要基于这四种表示方法中的任意一种证明即可。

4 正则语言的性质

4.1 语言类的性质

语言类 (Language Class) 是一类语言的集合，如定义3.1所说的那样，正则语言类 RE 就是所有的正则语言的集合。对于一个语言类我们一般会讨论它的两种性质：判定性质和闭包性质。

4.1.1 判定性质

我们称一类语言的某个性质是**可判定的（Decidable）**，属于**判定性质（Decision Property）**，如果存在某种算法，这个算法以该种语言的某个形式化表示为输入，以输入语言是否满足这个性质为输出。

比如说，如果给定一个正则语言的 DFA，如果存在某种算法能够判断该正则语言是否为空，我们就称一个正则语言是否为空这个问题是可判定的，正则语言为空是一个判定性质。

因此，一般证明一种语言的某个性质可以判定，我们只需要给出一个算法即可，证明一种语言的某个性质不可判定，我们需要证明不存在任何的算法能做到这一点。这里还需要注意一下算法的定义，算法是一个必定可以终止的计算过程，如果一个计算过程不一定能够终止，那么它不是一个算法。

那么我们为什么需要研究一种语言有哪些判定性质呢？举个例子，加入我想要用 DFA 来表示一种网络通信协议，那么

- 这个协议是否能够正常终止其实就相当于这个 DFA 定义的语言是否是有限的，这就需要我们能够判定正则语言的有限性；
- 这个协议是否会失败其实就相当于这个 DFA 定义的语言是否为空，这就需要我们能够判定一个正则语言是否为空。

此外，我们还可能想要尽量简化一个语言的表示，比如说寻找一个和给定 DFA 等价的状态数最少的 DFA，这就需要我们能够判定正则语言的等价性，因为只有在等价性可判定的前提下，

我们才有可能作化简操作。

简而言之，一类语言具有怎样的判定性质决定了我们能够对这类语言做些什么，能够用这类语言做些什么；一类语言的哪些性质是不可判定的，决定了我们对这类语言，或者用这类语言，无论如何也做不到什么。

4.1.2 闭包性质

我们称一类语言具有对某种操作的**闭包性质（Closure Property）**，如果这类语言在该种操作下是**封闭的（Closed）**，即对于该语言类中的任意一个或者一些语言进行该种操作后得到的语言依旧在同一个语言类中。

比如说，正则语言在并、拼接和星闭包操作显然是封闭的，可以通过正则表达式很轻易地证明出来。

4.2 正则语言的判定性质

4.2.1 成员资格问题

语言类的**成员资格问题（Membership Problem）**问的是：给定任意字符串 w 以及该语言类中的任意一个语言 L，是否 $w \in L$？

对于正则语言 L，一定存在 DFA $A = (Q, \Sigma, \delta, q_0, F)$ 使得 $L = L(A)$。因为定理2.2、定理2.3和定理3.1已经告诉我们正则语言的四种表示形式都是等价的，并且在证明这些定理的过程中，我们已经给出了这些表示形式之间的转化方法。所以，在遇到不是 DFA 的其他形式的时候，我们只需要将其转化成 DFA 即可。

定理 4.1. *正则语言的成员资格问题是可判定的。*

证明. 考虑定义某正则语言 L 的 DFA A，我们只需要在 A 上模拟 w 输入情况下的状态转换，判断是否 $\delta(q_0, w) \in F$ 即可判定是否 $w \in L$。这是线性时间即可完成的模拟算法，因此正则语言的成员资格问题是可判定的。 □

例如在如**图 4**中，输入 01011 会让 DFA 从初始状态 A 走到状态 C，而 C 不是终止状态（接受状态），因此 01011 并不属于这个 DFA 所表示的正则语言。类似的，我们也可以判断出，0101 属于这个 DFA 所表示的正则语言。

4.2.2 空问题

语言类的**空问题（Emptiness Problem）**问的是：给定该语言类中的任意一个语言 L，是否 $L = \emptyset$？

定理 4.2. *正则语言的空问题是可判定的。*

证明. 给出判定算法如下：

将正则语言的表示转化成 DFA，从初始状态开始，计算所有可达状态的集合（也就是求初始状态在状态转换图上的传递闭包），如果至少有一个终止状态可达，那么这个语言就是非空的，否则该语言为空集。

因此，正则语言的空问题是可判定的。 □

4.2.3 无限性问题

语言类的**无限性问题（Infiniteness Problem）**问的是：给定该语言类中的任意一个语言 L，L 是无限集吗?

一个正则语言是无限集的直观感受就是 DFA 上面有环，从而存在无数个形如 xy^iz 的字符串都属于该正则语言，其中 y^i 就是在一个环上面不断循环的过程（$i \geq 0$ ）。于是，我们可以有如下两个引理。

引理 4.3. *对于任意正则语言 L，记其 DFA 的状态数为 n，若存在 $w \in L$ 且 $|w| \geq n$，则 L 是无限集；否则，若不存在这样的 w，则 L 是有限集。*

证明. 首先，如果不存在 $w \in L$ 且 $|w| \geq n$，那么意味着 L 中的字符串长度都小于 n，显然 L 是有限集。

下面我们来说明如果存在 $w \in L$ 且 $|w| \geq n$，为什么 L 就一定会是无限集。我们的证明方式是构造无数个在 L 当中的字符串，从而说明 L 的无限性。

如果一个 n 状态的 DFA 接受了一个长度至少为 n 的字符串 w，那么在标记为 w 的，从初始状态到终止状态的路径上面，至少有一个状态出现了两次（鸽笼原理，整个 DFA 一共 n 个状态，但是这条路径上面有至少 $n+1$ 个状态）。

记这个出现两次的状态为 q，记 $w = xyz$，其中在读完 x 和读完 xy 的时候，DFA 都处于状态 q，则 y 对应的路径是一个环，且 $y \neq \varepsilon$，如15所示。

于是，我们不难发现：$\forall i \geq 0, xy^iz \in L$，因为无论这个环被走了多少遍，所得到的路径都是从初始状态到终止状态的一条有效路径。于是，我们构造出了无数个在 L 中的字符串，L 是一个无限集。

□

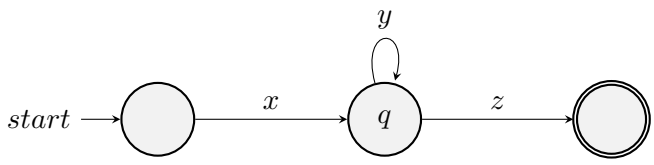


图 15: 带环的 DFA 状态转换图示意图（注：这里的边代表一条路径）

引理 4.4. *对于任意正则语言 L，记其 DFA 的状态数为 n，若存在 $w \in L$ 且 $|w| \geq n$，则存在 $w' \in L$ 且 $n \leq |w'| \leq 2n-1$。*

证明. 根据引理4.3的证明过程，结合**图 15**，不妨设 $w = xyz$ 且 y 是 w 中的第一个环，则 $|xy| \leq n$，特别地，$1 \leq |y| \leq n$。

- 如果 $|w| \leq 2n-1$，则直接取 $w' = w$ 即可；
- 如果 $|w| \geq 2n$，每个长度为 n 的子串中都可以消除一个 $|y|$，从而将 w 的长度缩减到 n 到 $2n-1$ 之间，将这个缩减后的 w 记为 w'。

上述过程简单理解就是，$|w| \geq n$ 说明有环（可能不止一个），但是我们可以控制第一个环只走 1 圈，剩余的环走 0 圈，从而将路径长度限制在 n 到 $2n-1$ 之间。

□

定理 4.5. 正则语言的无限性问题是可判定的。

证明. 给出判定正则语言 L 是否是无限集的算法如下：遍历所有 L 字母表中符号组成的，长度在 n 到 $2n-1$ 之间的有限个字符串 w，利用定理4.1中的判定算法判定是否 $w \in L$，如果存在这样的 $w \in L$，则 L 是无限集；否则 L 是有限集。

上述过程便利的字符串个数是有限的，所以可以终止，因此是一个算法；

根据引理4.3，如果找到了长度大于 n 的 L 中的字符串 w，则 L 是无限集；

根据引理4.4，如果没有找到长度在 n 到 $2n-1$ 之间的 L 中的字符串 w，则不可能存在 $w \in L$ 且 $|w| \geq n$（否则就会找到了），因此 L 中的字符串长度都小于 n，从而 L 是一个有限集（因为有限的字母表中的符号组成的长度小于 n 的字符串是有限的）。

于是我们证明了给出的判定算法可以终止且结果正确，从而正则语言的无限性问题是可以判定的。

□

其实上面的穷举算法虽然能够终止且正确，但它其实不是一个很高效的算法，讲解这个算法是为了后面的泵引理作铺垫。一个更直接的想法是，在 DFA 的状态转换图中查找从初始状态到终止状态是否存在环，这样就变成一个图问题了，且复杂度也不高。一个可选的方案是：

1. 删除所有从初始状态不可达的结点；
2. 删除所有无法到达终止状态的结点；
3. 检查剩下的状态转换图中是否有环。

4.2.4 泵引理

在上面的无限性讨论中，我们其实已经快要证明一个很重要的命题了，这对于判断某个语言不是正则语言很有用。

定理 4.6. 正则语言的泵引理（*Pumping Lemma for Regular Languages*） 对于每一个正则语言 L 来说，存在一个整数 n，使得 L 中的每一个满足 $|w| \geq n$ 的字符串 w，我们可以写作 $w = xyz$，满足：

1. $|xy| \leq n$；
2. $|y| > 0$；
3. $\forall i \geq 0, xy^i z \in L$；

整数 n 也称为**泵引理常数（*Pumping Lemma Constant*）**。

证明. 取 n 为 L 的 DFA 的状态数，取 y 为 w 中的第一个环，结合引理4.3和引理4.4不难得出上述结论，这里不再赘述。 □

下面我们来看一个泵引理的简单应用：证明一个语言不是正则语言。定理2.1告诉我们 $L = \{0^n 1^n | n \geq 1\}$ 不是正则语言，之前我们已经给出了一个证明，这里再给出一个应用泵引理的证明。

证明. 反证法。假设 L 是正则语言，根据泵引理，存在泵引理常数 n 。考虑 $w = 0^n 1^n$ ，由于 $|w| \geq n$ ，根据泵引理，可以写作 $w = xyz$ ，其中 $|xy| \leq n$ 且 $|y| > 0$，于是 x 和 y 都是全 0 字符串，从而 $xyyz$ 中的 0 比 1 多，因此 $xyyz \notin L$ ，这与泵引理的第 3 条结论矛盾，因此 L 不是正则语言。 □

4.2.5 等价性问题

语言类的**等价性问题（Equivalence Problem）**问的是：给定该语言类中的任意两个语言 L_1, L_2 ，是否 $L_1 = L_2$?

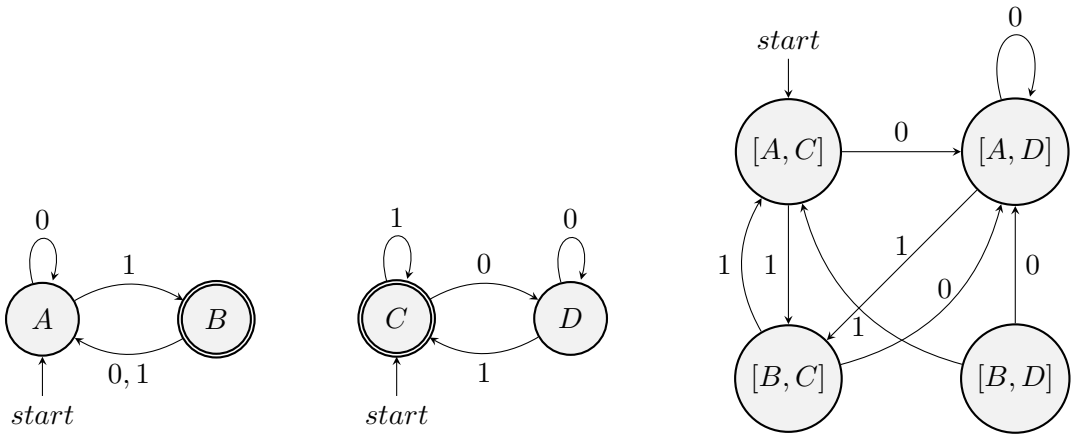
在讨论正则语言的等价性问题之前，我们先引入一个新的概念。

定义 4.1. 考虑 DFA $L = (Q, \Sigma_L, \delta_L, q_0, F_L)$ 和 $M = (R, \Sigma_M, \delta_M, r_0, F_M)$ ，定义其**乘积 DFA（Product DFA）**为一个 DFA

$$L \times M = (Q \times R, \Sigma_L \cup \Sigma_M, \delta, [q_0, r_0], F)$$

其中，$\forall [q, r] \in Q \times R, a \in \Sigma_L \cup \Sigma_M, \delta([q, r], a) = [\delta_L(q, a), \delta_M(r, a)]$ ，F 可以根据需求另行定义。

假设 DFA L 如**图 16a**所示，DFA M 如**图 16b**所示，则它们的乘积自动机如**图 16c**所示，其中终止状态还未标出，但乘积自动机的定义中并没有强制规定终止状态，可以根据自己的需求来构造符合自己目的的终止状态。



(a) DFA L **(b)** DFA M **(c)** 乘积 DFA $L \times M$，未标终止状态

图 16: 乘积 DFA 示例

定理 4.7. 正则语言的等价性问题是可判定的。

证明. 考虑任意两个 DFA L 和 M，下面给出判断 $L(L) = L(M)$ 的算法。

构造 L 和 M 的乘积 DFA $L \times M$，其中终止状态定义为：$[q, r]$ 为终止状态，当且仅当 q 和 r 中有且仅有一个是原本 DFA 的终止状态，于是 $L(L) = L(M) \Leftrightarrow L(L \times M) = \emptyset$，根据定理4.2，$L(L \times M) = \emptyset$ 是可判定的，因为显然 $L(L \times M)$ 是正则语言，从而 $L(L) = L(M)$ 也是可判定的。

下面我们再解释一下为什么 $L(L) = L(M) \Leftrightarrow L(L \times M) = \emptyset$。$L \times M$ 接受 w 当且仅当

$$(w \in L \wedge w \notin M) \vee (w \notin L \wedge w \in M)$$

于是 $L(L \times M) = \emptyset$ 当且仅当

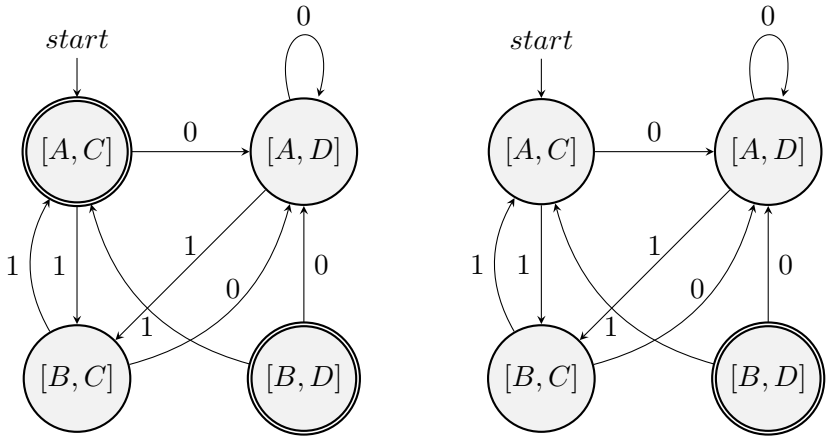
$$\neg\exists w, (w \in L(L) \wedge w \notin L(M)) \vee (w \notin L(L) \wedge w \in L(M))$$

即

$$\begin{aligned}
&\forall w, \neg((w \in L(L) \wedge w \notin L(M)) \vee (w \notin L(L) \wedge w \in L(M))) \\
\Leftrightarrow &\forall w, (w \notin L(L) \vee w \in L(M)) \wedge (w \in L(L) \vee w \notin L(M)) \\
\Leftrightarrow &\forall w, (w \notin L(L) \wedge w \notin L(M)) \vee (w \in L(L) \wedge w \in L(M)) \\
\Leftrightarrow &\forall w, w \in L(L) \leftrightarrow w \in L(M) \\
\Leftrightarrow &L(L) = L(M)
\end{aligned}$$

因此，$L(L) = L(M) \Leftrightarrow L(L \times M) = \emptyset$。综上，正则语言的等价性问题是可判定的。 □

在**图 16**的例子中，通过 $L \times M$ 空问题的判定来判定 L 和 M 的等价性所需要设置的终止状态如**图 17a**所示。



(a) 等价性问题 **(b)** 包含问题

图 17: 通过空问题判定等价性问题所需的乘积 DFA 的样例

4.2.6 包含问题

语言类的**包含问题（Equivalence Problem）**问的是：给定该语言类中的任意两个语言 L_1, L_2，是否 $L1 \subseteq L2$？

定理 4.8. *正则语言的包含问题是可判定的。*

证明. 考虑任意两个 DFA L 和 M，下面给出判断 $L(L) \subseteq L(M)$ 的算法。

构造 L 和 M 的乘积 DFA $L \times M$，其中终止状态定义为：$[q, r]$ 为终止状态，当且仅当 q 是 L 的终止状态且 r 不是 M 的终止状态。于是 $L(L) \subseteq L(M) \Leftrightarrow L(L \times M) = \emptyset$，证明和定理4.7类似，这里不再赘述。从而我们能够通过定理4.2中判定乘积 DFA 的空问题的算法来判定 $L(L)$ 和 $L(M)$ 的子集关系。因此，正则语言的包含问题是可判定的。 □

在**图 16**的例子中，通过 $L \times M$ 空问题的判定来判定 L 和 M 的包含关系所需要设置的终止状态如**图 17b**所示。

4.3 DFA 状态最小化

原则上，由于定理4.7中已经告诉我们检查 DFA 的等价性的算法，我们可以朴素地检查所有状态数比 DFA A 少的 DFA 与 A 的等价性，从中选出状态数最少的 A'；于是我们可以得到与 A 等价的且状态数最少的 A'，在没有改变所表达的语言的前提下使得表达的方式更加简单，后续如果想要做什么操作的话也会更简单，计算机实现的时候 DFA 的存储空间也会更小。

虽然上面的朴素的做法虽然理论上是可以做到的，但这实在是一个糟糕的算法。下面我们介绍一种**高效的状态最小化（Efficient State Minimization）**算法，其实这个算法的想法很直观很简单，就是“合并等价的状态”。

那么，对于 DFA 来说，怎样的状态是等价的呢？这里，我们定义一个“区分”的概念。

定义 4.2. 对于一个 DFA $A = (Q, \Sigma, \delta, q_0, F)$，考虑两个不同的状态 $\{q_1, q_2\} \subseteq Q$，称 q_1 与 q_2 是**可区分的（Distinguishable）**，如果

$$\exists w, (\delta(q_1, w) \in F \wedge \delta(q_2, w) \notin F) \vee (\delta(q_1, w) \notin F \wedge \delta(q_2, w) \in F)$$

称 w **区分了（Distinguishes）** q_1 和 q_2。

也就是说，如果存在一个字符串，能够让一个状态前进到接受状态，另一个状态不能，那么这两个状态就是可区分的。反之，如果不存在任何字符串，能够让一个状态前进到接受状态，另一个状态不能，那么这两个状态就是不可区分的，从这两个状态开始能够识别的子串集合完全相同，我们就可以认为这两个状态是等价的，从而合并这两个状态。

引理 4.9. **（不可区分的传递性）***在 DFA 中，如果状态 p 和状态 q 是不可区分的，状态 q 和状态 r 是不可区分的，那么状态 p 和状态 r 也是不可区分的。*

证明. 证明：由于 p 和 q 不可区分且 q 和 r 不可区分，我们可以知道：$\forall w, \delta(p, w) \in F \leftrightarrow \delta(q, w) \in F \wedge \delta(q, w) \in F \leftrightarrow \delta(r, w) \in F$，从而 $\forall w, \delta(p, w) \in F \leftrightarrow \delta(r, w) \in F$，即 p 和 r 不可区分。 □

算法1是 DFA 的状态最小化算法，我们可以通过一个例子来感受一下这个算法，以**表 4**中的 DFA 为例，算法运行结果如**图 18**所示。其中，标记状态对的过程如**图 19**所示，简单叙述如下：

1. 开始时标记只有一个终止状态的对（**图 19**第 1 幅图）；
2. 由于对 $[B, D]$ 并没有被标记，而输入 r 的不同结果只可能是 $[B, D]$，因此输入 r 暂时并不能区分什么；

算法 1: DFA 状态最小化算法

输入: DFA $A = (Q, \Sigma, \delta, q_0, F)$

输出: A 的最小化 DFA $A' = (Q', \Sigma, \delta', q_0, F')$

建立一张包含所有状态对的表，标记所有可区分的状态对;

算法是一个基于最短的区分字符串长度的递归;

基础情况：标记所有的只有一个终止状态的对，因为这些对被 ε 区分;

递归情况：如果对于某个输入符号 $a \in \Sigma$ ，状态对 $[\delta(q,a), \delta(r,a)]$ 被标记了，那么将 $[q,r]$ 也标记；直到没有更多的对可以被标记为止;

最终剩下的还没有被标记的状态对所对应的两个状态是不可区分的，将所有的相互不可区分的状态合并即可得到最小化 DFA;

合并过程：

假设 $q_1, q2, \ldots, q_k$ 是不可区分的状态（因为不可区分具有传递性，这里 k 可以大于 2，即可以一下子合并多个状态），用一个**代表（Representative）**状态 q 来代替这些状态;

$\delta(q_1,a), \delta(q_2,a), \ldots, \delta(q_k,a)$ 也是不可区分的状态（否则 $q_1, q2, \ldots, q_k$ 就可区分了），记其代表状态为 r ;

则 $\delta'(q,a) = r$ ，按照这样的方式，我们可以将不可区分的各组状态都合并成一个。

3. 不过输入 b 可以将 $\{A,B,F\}$ 和 $\{C,D,E,G\}$ 区分开来，比如说 $[C,F] = [\delta(A,b), \delta(C,b)]$ 被标记了，从而 $[A,C]$ 也需要被标记，因为它们被 b 给区分了，其余的也是类似，不再一一说明（**图 19**第 2 幅图）;
4. 类似的，由于 $[F,G] = [\delta(C,b), \delta(D,b)] = [\delta(C,b), \delta(E,b)]$ 被标记了，$[C,D]$ 和 $[C,E]$ 也应该被标记（**图 19**第 3 幅图）;
5. 由于 $[B,D] = [\delta(A,r), \delta(B,r)]$ 被标记了，$[A,B]$ 也应该被标记（**图 19**第 4 幅图）;
6. $[D,E]$ 不可能被标记，因为无论输入 r 还是 b ，它们总是转换成同样的状态，至此，没有更多的可以标记的状态对了，标记算法终止，未被标记的 $[D,E]$ 就是唯一的不可区分的状态对（**图 19**第 5 幅图）。

		r	b
→	A	B	C
	B	D	E
	C	D	F
	D	D	G
	E	D	G
*	F	D	C
*	G	D	G

(a) 表 4中 DFA

		r	b
→	A	B	C
	B	H	H
	C	H	F
	H	H	G
*	F	H	C
*	G	H	G

(b) 表 18a中 DFA 的状态最小化结果

图 18: DFA 状态最小化的结果样例

合并不可区分的状态可能会在最小化状态的 DFA 中留下不可达的状态，因此，在最小化之

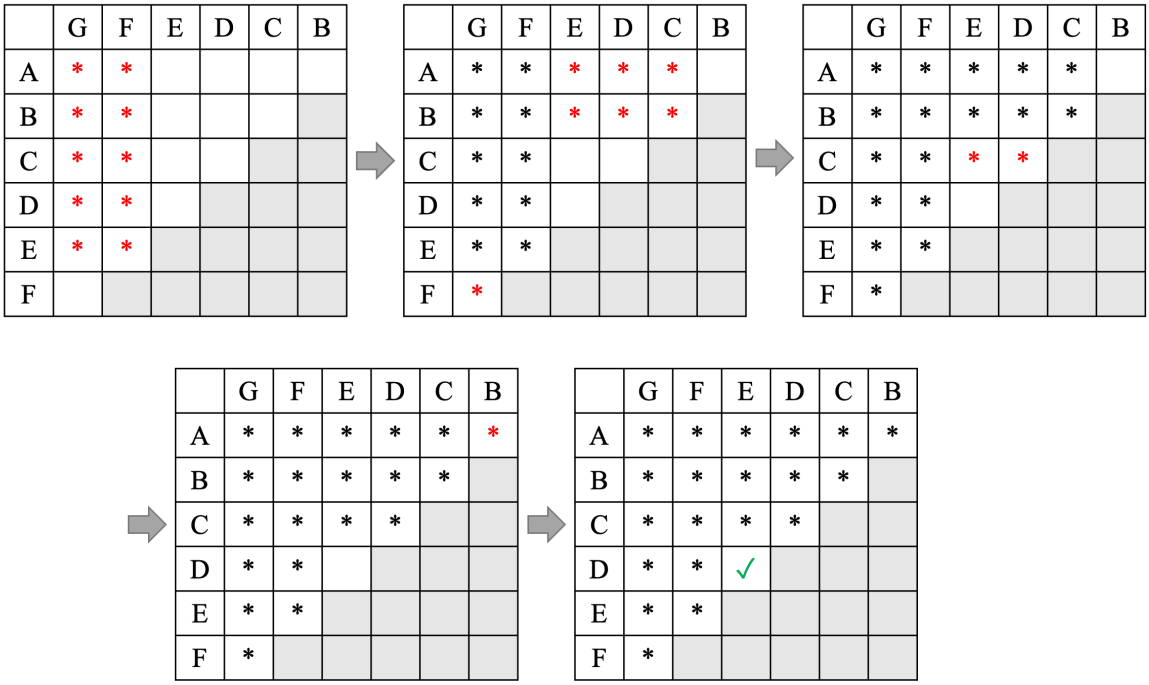


图 19: 图 18示例中 DFA 状态最小化算法的标记过程

前和之后，我们应该将不可达的状态从 DFA 中去除，达到更好的化简效果。

定理 4.10. *DFA 状态最小化算法（算法1）是正确的。*

证明. 要证明 DFA 状态最小化算法的正确性，需要证明两件事情：

1. 该算法得到的 DFA 和原来等价；
2. 该算法得到的 DFA 的状态数是最优的（最少的）。

第一件事情是很容易证明的，合并不可区分的状态并不会改变 DFA 的语言，这里不作赘述。

下面证明第二件事情。反证法。考虑算法1给出的最小化 DFA $A = (Q, \Sigma, \delta_A, q_0, F_A)$，假设存在一个状态数更少的 DFA $B = (P, \Sigma, \delta_B, p_0, F_B)$ 满足 $L(A) = L(B)$。

将 DFA A 和 B 拼在一起形成一个新的 DFA $A + B = (Q \cup P, \Sigma, \delta_A \cup \delta_B, r_0, F_A \cup F_B)$，其中 r_0 随便，可以是任意状态。

由于 $L(A) = L(B)$，在 $A + B$ 中，q_0 和 p_0 显然是不可区分的，从而它们的后继状态也是不可区分的。

下面我们证明 A 中的任意状态，在 B 中都存在某个状态和它不可区分。考虑 A 中任意状态 q，记 w 是使得 A 从状态 q_0 转换成 q 所需要消耗的最短字符串，对 $|w|$ 进行归纳。

基础情况：$|w| = 0$ 时，q_0 与 B 中的 p_0 不可区分。

归纳情况：记 $w = xa$ 是最短的 A 从 q_0 转换成 q 所需要消耗的字符串，记 $q' = \delta_A(q_0, w)$，根据归纳假设，存在 $p \in P$ 使得 q' 与 p 不可区分，于是 $\delta(q', a) = q$ 和 $\delta(p, a) \in P$ 是不可区分的。

综上，对于 A 中的任意状态，在 B 中都存在某个状态与它不可区分。

因为 B 的状态数比 A 少，根据鸽笼原理，B 中存在状态 q，至少与 A 中的状态 q_1 和 q_2 都

不可区分。根据引理4.9（不可区分的传递性），q_1 和 q_2 不可区分，矛盾（因为算法1终止就说明 A 中不存在不可区分的状态了）。

因此，该算法得到的 DFA 的状态数是最优的。

□

在了解完正则语言的判定性质，特别是等价性判定之后，我们花了一点时间来研究 DFA 的最小化，下面我们继续回到正则语言的性质，去介绍它的另一类性质——闭包性质。

4.4 正则语言的闭包性质

4.4.1 正则语言在正则表达式操作下的封闭性

正则语言天然是在正则表达式提供的三种基本操作下封闭的。

定理 4.11. *正则语言在并集、拼接操作以及星闭包操作下是封闭的。*

证明. 下面对并集操作证明：如果 L 和 M 是正则语言，那么 $L \cup M$ 也是正则语言。记 L 和 M 所对应的正则表达式分别为 E_L 和 E_M，则

$$L \cup M = L(E_L) \cup L(E_M) = L(E_L + E_M)$$

因为 $L \cup M$ 可以用正则表达式 $E_L + E_M$ 来定义，因此 $L \cup M$ 也是正则语言，从而我们证明了正则语言在并操作下面封闭的。

其他两种操作类似可证封闭性，不再赘述。 □

4.4.2 正则语言在其他集合操作下的封闭性

正则语言在其他一些集合操作上也是封闭的。

定理 4.12. *正则语言对交集操作封闭。如果 L 和 M 是正则语言，那么 $L \cap M$ 也是正则语言。*

证明. 为了表示方便，我们直接用 L 和 M 同时表示 DFA 以及 DFA 的语言。构造 L 和 M 的乘积 DFA $L \times M$，其中终止状态定义为：$[q, r]$ 为终止状态，当且仅当 q 是 L 的终止状态且 r 是 M 的终止状态。不难证明，DFA $L \times M$ 的语言就是 $L \cap M$，从而 $L \cap M$ 可以用 DFA 来定义，所以 $L \cap M$ 是正则语言。 □

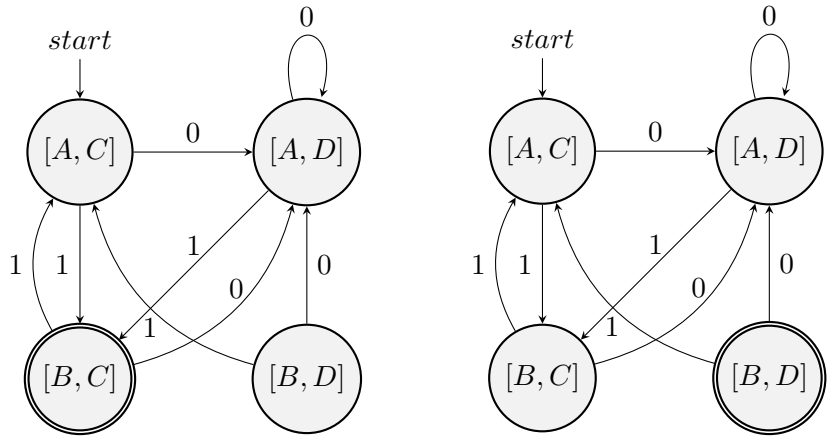
正则语言的闭包性质其实也可以用来判断一些语言是否是正则语言。比如说 $L_1 =$ { 所有由相等个数的 0 和 1 组成的字符串 } 是否是正则语言？答案是否。一种证明方式如下。

证明. 反证法。假设 L_1 是正则语言，记 $L_2 = L(0^*1^*)$，L_2 显然也是正则语言，那么根据定理4.12，$L_1 \cap L_2 = \{0^n 1^n | n \geq 0\}$ 也是正则语言，和定理2.1矛盾。因此，L_1 不是正则语言。 □

定理 4.13. *正则语言对差集操作封闭。如果 L 和 M 是正则语言，那么 $L - M$ 也是正则语言。*

证明. 为了表示方便，我们直接用 L 和 M 同时表示 DFA 以及 DFA 的语言。构造 L 和 M 的乘积 DFA $L \times M$，其中终止状态定义为：$[q, r]$ 为终止状态，当且仅当 q 是 L 的终止状态且 r 不是 M 的终止状态。不难证明，DFA $L \times M$ 的语言就是 $L - M$，从而 $L - M$ 可以用 DFA 来定义，所以 $L - M$ 是正则语言。 □

以图 16中的 L 和 M 为例，所构造的 $L \cap M$ 和 $L - M$ 的 DFA 如图 20所示。



(a) $L \cap M$ 的 DFA　　**(b)** $L - M$ 的 DFA

图 20: 交集和差集操作对应的乘积 DFA

定理 4.14. *正则语言对补集操作封闭。如果 L 是正则语言，那么 $\Sigma^* - L$ 也是正则语言。*

证明. 因为 Σ^* 是正则语言（可以用正则表达式表示），且 L 是正则语言（已知条件），所以 $\Sigma^* - L$ 也是正则语言（正则语言对差集操作封闭，定理4.13）。 □

4.4.3 正则语言在逆运算下的封闭性

定理 4.15. *正则语言对逆运算封闭。如果 L 是正则语言，那么 L^R 也是正则语言。*

证明. 另 L 的正则表达式为 E，下面构造 L^R 的正则表达式 E^R，从而说明 L^R 是正则语言。

基础情况：如果 E 是单个符号 a 或者 ε 或者 $\emptyset$，那么 $E^R = E$。

归纳情况：如果 E 形如：

- $F + G$，那么 $E^R = F^R + G^R$；
- FG，那么 $E^R = G^R F^R$；
- F^*，那么 $E^R = (F^R)^*$；

不难发现 $L(E^R) = L^R$，所以 L^R 也是正则语言。 □

比如说 $E = 01^* + 10^*$，根据上面证明中的构造方法，我们不难得到 $E^R = 1^*0 + 0^*1$，只需递归地应用上面的规则即可。

4.4.4 正则语言在同态运算下的封闭性

定义 4.3. 定义字母表 Σ 上的**同态（Homomorphism）**运算是一个从 Σ 到 Σ'^* 的函数，$h \subseteq \Sigma \times \Sigma'^*$，其中 Σ' 是另一个字母表，和 Σ 没有必然联系。

同态函数简单理解就是将一个语言中的每个符号映射成一个字符串。

定义 4.4. 定义**扩展同态函数（Extended Homomorphism Function）**如下：

$$\hat{h}(a_1a_2\dots a_n) = h(a_1)h(a_2)\dots h(a_n)$$

其中，$a_i \in \Sigma$ 。

习惯上，不区分书写 $\hat{h}$ 和 h ，统一都用 h 表示。至于其含义到底是定义4.3中的同态函数，还是定义4.4中的扩展同态函数，由具体语境下的参数类型决定。

比如说，对于 $\Sigma = \{0,1\}$ ，定义 $h(0) = ab, h(1) = \varepsilon$ ，则 $h(01010) = ababab$ 。

定义 4.5. 考虑语言 L 和同态函数 h ，记 L 在 h 下的同态变换结果为 $h(L)$ ，定义如下：

$$h(L) = \{h(w)|w \in L\}$$

定理 4.16. *正则语言对同态运算封闭。如果 L 是正则语言，h 是同态运算，那么 $h(L)$ 也是正则语言。*

证明. 令 E 是定义 L 的正则表达式，将 h 应用在 E 的每个符号上面得到的新正则表达式为 E' ，不难发现 $h(L) = L(E')$ ，所以 $h(L)$ 也是正则语言。 □

举个例子，依旧定义同态函数 $h(0) = ab, h(1) = \varepsilon$ ，考虑正则语言 $L = L(01^* + 10^*)$ ，则 $h(L) = L((ab)\varepsilon^* + \varepsilon(ab)^*)$ 。将正则表达式中的单个字符替换成字符串的时候，为了保持运算顺序不变，可能需要添加括号。

这里的正则表达式 $(ab)\varepsilon^* + \varepsilon(ab)^*$ 是可以化简的，由 $\varepsilon^* = \varepsilon, \varepsilon E = E\varepsilon = E$ ，原式可化为 $ab + (ab)^*$ 。而 $L(ab) \subseteq L((ab)^*)$ ，所以 $L(ab + (ab)^*) = L(ab) \cup L((ab)^*) = L((ab)^*)$ ，即 $ab + (ab)^* = (ab)^*$ 。这里正则表达式之间的等于号表示等号两边的正则表达式定义的是同一个语言。

4.4.5 正则语言在逆同态运算下的封闭性

定义 4.6. 考虑语言 L 和同态函数 h ，定义该语言 L 的**逆同态（Inverse Homomorphism）**运算 $h^{-1}(L)$ 为：

$$h^{-1}(L) = \{w|h(w) \in L\}$$

举一个简单的例子，$h(0) = ab, h(1) = \varepsilon, L = \{abab, baba\}$ ，则 $h^{-1}(L) = L(1^*01^*01^*)$ 。

定理 4.17. *正则语言对逆同态运算封闭。如果 L 是正则语言，h 是同态运算，那么 $h^{-1}(L)$ 也是正则语言。*

证明. 考虑 L 的 DFA 为 $A = (Q, \Sigma, \delta_A, q_0, F)$ ，下面为 $h^{-1}(L)$ 构造 DFA B 来证明 $h^{-1}(L)$ 是正则语言。

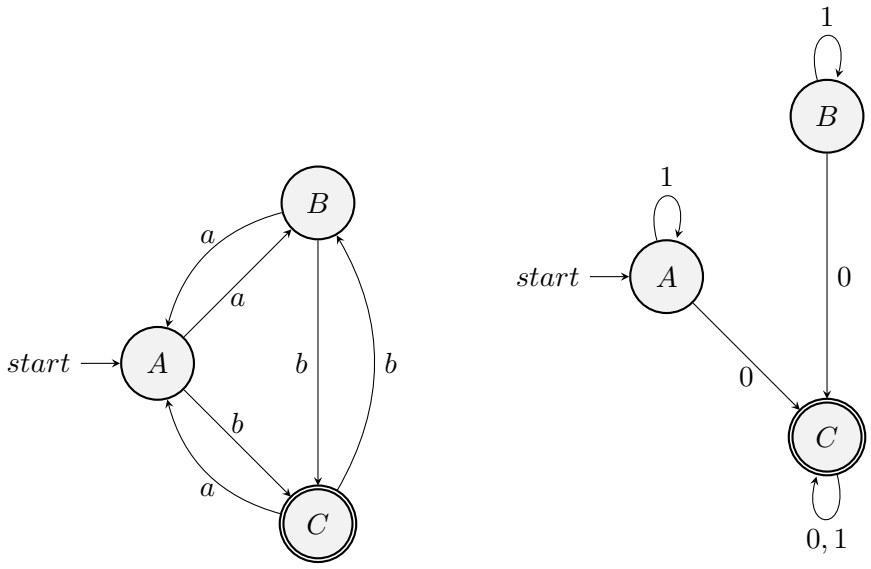
$$B = (Q, dom(h), \delta_B, q_0, F)$$

其中，$dom(h)$ 表示 h 的**定义域（Domain）**，δ_B 满足

$$\forall q \in Q, a \in dom(h), \delta_B(q,a) = \delta_A(q, h(a))$$

下面只需要证明 $h^{-1}(L) = L(B)$ 即可。这可以通过对 $|w|$ 进行归纳来证明 $\delta_B(q_0, w) = \delta_A(q_0, h(w))$ 恒成立，从而 B 接受 w 当且仅当 A 接受 $h(w)$ ，也就是 $L(B) = \{w|h(w) \in L\} = h^{-1}(L)$ 。归纳过程并不困难，这里不再赘述。 □

针对之前举的例子：“$h(0) = ab, h(1) = \varepsilon, L = \{abab, baba\}$ ，则 $h^{-1}(L) = L(1^*01^*01^*)$ ”，L 和 $h^{-1}(L)$ 的 DFA 如图 21所示。



(a) L 的 DFA　　**(b)** $h^{-1}(L)$ 的 DFA

图 21: 正则语言的逆同态运算的 DFA 样例

5 总结

到此为止，读者应当能够

- 对形式语言的基本定义有所了解；
- 理解正则语言的四种定义即这四种定义之间的等价环以及相互转化的算法；
- 对正则语言的表达能力有一些直觉上的认知，能大致判断一些语言是不是正则语言；
- 了解正则语言的各种判定性质和这些判定性质的判定算法，以及 DFA 的状态最小化算法。
- 了解正则语言的闭包性质和泵引理，并能够利用这些性质来证明一个语言是或者不是正则语言。