有限状态机

有限状态机（Finite State Machine，FSM）又称有限自动机（Finite Automation，FA），简称状态机，是表示有限个状态以及在这些状态之间的转移和动作等行为的模型。

有穷自动机看上去就像是一个有向图，其中状态是图的节点，而状态转换则是图的边。此外这些状态中还必须有一个初始状态和至少一个接受状态。

状态，存储了关于过去的信息，就是说：它反映从系统开始到现在时刻的输入变化。

转移指示状态变更，并且用必须满足确使转移发生的条件来描述它。

动作是在给定时刻要进行的活动的描述。有多种类型的动作：

进入动作（entry action）：在进入状态时进行

退出动作：在退出状态时进行

输入动作：依赖于当前状态和输入条件进行

转移动作：在进行特定转移时进行

FSM（有限状态机）可以使用状态图（或状态转移图）来表示。此外可以使用多种类型的状态转移表。下面展示最常见的表示：当前状态（B）和条件（Y）的组合指示出下一个状态（C）。完整的动作信息可以只使用脚注来增加。包括完整动作信息的FSM定义可以使用状态表。



有两个不同的群组：接受器／识别器和变换器。

接受器和识别器

接受器和识别器（也叫做序列检测器）产生一个二元输出，说要么“是”要么“否”来回答输入是否被机器接受。所有FSM的状态被称为要么接受要么不接受。在所有输入都被处理了的时候，如果当前状态是接受状态，输入被接受，否则被拒绝。作为规则，输入是符号（字符）；动作不使用。

现在我们来看看有穷自动机怎么处理输入的字符串：

一开始，自动机处于初始状态

输入字符串的第一个字符，这时自动机会查询当前状态上与输入字符相匹配的边，并沿这条边转换到下一个状态。

继续输入下一个字符，重复第二步，查询当前状态上的边并进行状态转换

当字符串全部输入后，如果自动机正好处于接受状态上，就说该自动机接受了这一字符串。

机器还可以被描述为定义了一个语言，它包含了这个机器所接受而非拒绝的所有字词；我们称这个语言被这个机器接受。通过定义，FSM接受的语言是正则语言 - 就是说，如果一个语言被某个FSM接受，那么它是正则的（cf. Kleene的定理）。

开始状态

开始状态通常用“没有起点的箭头”指向它来表示（Sipser (2006）p.34）

接受状态

接受状态是机器成功的进行了它的程序之后的状态，它通常表示为双重圆圈。

接受状态出现的下面确定有限状态自动机例子的状态图的左边，它确定二进制输入是否包含偶数个0: S1（它也是开始状态）指示已经输入了偶数个0的状态因此被定义为接受状态。

变换器

变换器使用动作基于给定输入和／或状态生成输出。它们用于控制应用。常分为两种类型：

Moore机(摩尔型有限状态机)

只使用进入动作的FSM，就是说输出只依赖于状态。Moore模型的好处是行为的简单性。图1的例子展示了一个电梯门的Moore FSM。这个状态机识别两个命令：“command\_open”和“command\_close”触发状态变更。在状态“Opening”中的进入动作 (E:)开启电机开门，在状态“Closing”中的进入动作以反方向开启电机关门。状态“Opened”和“Closed”不进行任何动作。它们信号通知外部世界（比如其他状态机）情况：“门开着”或“门关着”。

Mealy机(米利型有限状态机)

只使用输入动作的FSM，就是说输出依赖于输入和状态。Mealy FSM的使用经常导致状态数目的简约。在图4中的例子展示了实现同上面Moore机同样行为的Mealy FSM（行为依赖于实现的FSM执行模型，比如对虚拟FSM可工作但对事件驱动FSM不行）。有两个输入动作（I:）：“开启电机关门如果command\_close下达”和“反向开启电机开门如果command\_open下达”。

在实践中经常使用混合模型。

进一步可区分为确定型（DFA）和非确定型（NDFA、GNFA）自动机。在确定型自动机中，每个状态对每个可能输入只有精确的一个转移。在非确定型自动机中，给定状态对给定可能输入可以没有或有多于一个转移。这个区分在实践而非理论中更有用，因为存在算法把任何NDFA转换成等价的DFA，尽管这种转换一般会增加自动机的复杂性。

只有一个状态的FSM叫做组合FSM并只使用输入动作。这个概念在多个FSM要一起工作的情况下是有用的，这时把纯组合部分看作一种形式的FSM来适合设计工具可能是方便的。

FSM逻辑

FSM的下一个状态和输出是由输入和当前状态决定的。

数学模型

依据类型不同有多种定义。接受器有限状态机是五元组 (∑, S, s\_0, , F)，这里的：

∑是输入字母表（符号的非空有限集合）。

S是状态的非空有限集合。

s\_0是初始状态，它是S的元素。在非确定有限状态自动机中，s\_0是初始状态的集合。

是状态转移函数：: S × ∑ → S。

F是最终状态的集合，S的（可能为空）子集。

变换器有限状态自动机是六元组(∑, Γ, S, s\_0, , ω)，这里的：

∑a是输入字母表（符号的非空有限集合）。

Γ是输出字母表（符号的非空有限集合）。

S是状态的非空有限集合。

s\_0是初始状态，它是S的元素。在非确定有限状态自动机中，s\_0是初始状态的集合。

是状态转移函数：: S × ∑ → S。

ω是输出函数。

如果输出函数是状态和输入字母表的函数（ω: S × ∑ → Γ），则定义对应于Mealy模型，它可以建模为Mealy机。如果输出函数只依赖于状态 (ω: S →Γ），则定义对应于Moore模型，它可建模为Moore机。根本没有输出函数的有限状态机叫做半自动机或转移系统。

优化

优化一个FSM意味着找到带有极小数目个状态的进行同样功能的机器。一种可能是使用蕴涵表或Moore简约过程。另一种可能是无环FSA的自底向上算法。

确定有限状态自动机

确定有限状态自动机或确定有限自动机（Deterministic Finite Automation, DFA）是一个能实现状态转移的自动机。对于一个给定的属于该自动机的状态和一个属于该自动机字母表∑的字符，它都能根据事先给定的转移函数转移到下一个状态（这个状态可以是先前那个状态）。

确定有限状态自动机\mathcal{A}是由

一个非空有限的状态集合Q

一个输入字母表\Sigma（非空有限的字符集合）

一个转移函数\delta: Q \times \Sigma \rarr Q（例如：\delta \left( q,\sigma \right) = p, \left( p,q \in Q, \sigma \in \Sigma \right)）

一个开始状态s \in Q

一个接受状态的集合F \sube Q

所组成的5-元组。因此一个DFA可以写成这样的形式：\mathcal{A} = \left( Q,\Sigma,\delta,s,F \right)。

工作方式（非正式的语义）

确定有限状态自动机从起始状态开始，一个字符接一个字符地读入一个字符串w \in \Sigma^\*（这里的{}^\*指示Kleene星号算子。），并根据给定的转移函数一步一步地转移至下一个状态。在读完该字符串后，如果该自动机停在一个属于F的接受状态，那么它就接受该字符串，反之则拒绝该字符串。

扩展转移函数

为了在保证严谨的前提下，方便地叙述关于DFA的内容，我们定义如下扩展的转移函数：

\delta^\*: Q \times \Sigma^\* \rarr Q。

\delta^\* \left( q,w \right)是自动机从状态q顺序读入字符串w后达到的状态

扩展转移函数递归的定义为：

\delta^\* \left( q,\epsilon \right) = q

\delta^\* \left( q,u\sigma \right) = \delta(\delta^\*(q,u),\sigma), \forall u \in \Sigma^\*, \sigma \in \Sigma

工作方式（正式的语义）

对于一个确定有限状态自动机\mathcal{A} = \left( Q,\Sigma,\delta,s,F \right)，如果\delta^\* \left( s,w \right) \in F，我们就说该自动机接受字符串w，反之则表明该自动机拒绝字符串w。

被一个确定有限自动机接受的语言（或者叫“被识别的语言”）定义为：\mathcal{L} ( \mathcal{A} ) = \{ w \in \Sigma^\* | \mathcal{A}~接受字符串~w \}，也就是由所有被接受的字符串组成的集合。

DFA与有向图

除了数学上的严谨表述，通常为了讨论方便，也使用状态图直观地表示DFA。不难发现，对于一个给定的DFA，存在唯一一个对应的有向图（但是严格意义上一个有向图不能确定出唯一一个DFA）。有向图的每个结点对应一个状态，每条有向边对应一种转移。习惯上将结点画成两个圈表示接受状态，一个圈表示拒绝状态。用一条没有起点的边指向起始状态。

除了在表述上方便以外，在研究某些问题（如“给定的DFA的语言是否为无穷集合”）时，状态图也提供了有效的解法。

利弊

DFA是一种实际的计算模型，因为有平凡的线性时间、恒定空间的在线算法模拟在输入流上的DFA。给定两个DFA有有效算法找到识别它们所识别语言的并集、交集和补集的DFA。还有有效算法确定一个DFA是否接受任何给定字符串，一个DFA是否接受所有字符串，两个DFA是否识别同样的语言，和对特定正则语言找到状态数目最小的DFA（最小DFA）。

在另一方面，DFA在可识别的语言上有严格的限制—很多简单的语言，包括需要多于恒定空间来解决的任何问题，不能被DFA识别。经典的DFA不能识别的简单语言的例子是括号语言，就是由正确配对的括号组成的语言，比如 (()())。由形如anbn的字符串组成的语言，就是有限数目个a，随后是相等数目个b。可以证明没有DFA有足够状态来识别这种语言（通俗地说，因为需要至少2n个状态，而n是不恒定的）。

其它

能被确定有限状态自动机识别的语言是正则语言。

确定有限状态自动机是非确定有限状态自动机的一种极限形式。

确定有限状态自动机在计算能力上等价于非确定有限状态自动机。

没有接受状态列表并没有指定开始状态的确定有限状态机叫做转移系统或半自动机。

例子

下面是一个确定有限状态自动机的例子。

\mathcal{A}的状态图

确定有限状态自动机\mathcal{A} = \left( Q,\Sigma,\delta,s,F \right)

Q = \{ S\_1 , S\_2 \}

\Sigma = \{0,1\}

s=S\_1

F=\{S\_1\}

\delta由下面的状态转移表定义：

0

1

S1 S2 S1

S2 S1 S2

对应的转移函数为：

\delta(S\_1,0)=S\_2

\delta(S\_1,1)=S\_1

\delta(S\_2,0)=S\_1

\delta(S\_2,1)=S\_2

状态S\_1表示在输入的字符串中有偶数个0，而S\_2表示有奇数个0。在输入中1不改变自动机的状态。当读完输入的字符串的时候，状态将显示输入的字符串是否包含偶数个0。

\mathcal{A}能识别的的语言是\mathcal{L} ( \mathcal{A} ) = \{ w | \#\_0(w) \equiv 0~(mod~2) \}。用正则表达式表示为：(1^\*(01^\*0)^\*)^\*。

封闭性及一些运算

封闭性

确定有限状态自动机的交，并，差，补，连接，替换，同态，逆同态等运算是封闭的，也就是说确定有限状态自动机通过这些运算产生的新的自动机也是确定有限状态自动机。

补运算

\mathcal{A}=(Q,\Sigma,\delta,s,F)是一个DFA，那么由补运算产生的新DFA定义为：\bar{\mathcal{A}}=(Q,\Sigma,\delta,s,Q-F)。显然只要将\mathcal{A}中接受的状态设为不接受的状态，同时把不接受的状态设为接受的状态就得到\bar{\mathcal{A}}。补运算的复杂度是：O(\left| Q \right|)。

交运算和并运算

有两个DFA，\mathcal{A}\_1=(Q\_1,\Sigma,\delta\_1,s\_1,F\_1)和\mathcal{A}\_2=(Q\_2,\Sigma,\delta\_2,s\_2,F\_2)，那么由这两个DFA创造出来的新的自动机定义为：\mathcal{B}= (Q\_1 \times Q\_2,\Sigma,\delta\_\mathcal{B},(s\_1,s\_2),M)。其中M \sube Q\_1 \times Q\_2，\left( s\_1 , s\_2 \right)为\mathcal{B}的开始状态，\delta\_\mathcal{B}为\mathcal{B}的转移函数，且作如下定义：\forall q\_1 \in Q\_1,~q\_2 \in Q\_2,~\sigma \in \Sigma : \delta\_\mathcal{B}((q\_1,q\_2),\sigma) = (\delta\_1 (q\_1,\sigma),\delta\_2 (q\_2,\sigma) )。

当M = F\_1 \times F\_2时，由上述方法得到的\mathcal{B}就是DFA \mathcal{A}\_1和\mathcal{A}\_2的交运算，记作：\mathcal{B} = \mathcal{A}\_1 \cap \mathcal{A}\_2。也就是说对于读入的字符串w，当且仅当\mathcal{A}\_1和\mathcal{A}\_2同时接受w的时候\mathcal{B}接受w。

当M = Q\_1 \times F\_2 \bigcup F\_1 \times Q\_2时，由上述方法得到的\mathcal{B}就是DFA \mathcal{A}\_1和\mathcal{A}\_2的并运算，记作：\mathcal{B} = \mathcal{A}\_1 \cup \mathcal{A}\_2。也就是说对于读入的字符串w，只要\mathcal{A}\_1或\mathcal{A}\_2中至少有一个接受w，\mathcal{B}就接受w。

交运算和并运算的复杂度都是O(\left| Q\_1 \right| \left| Q\_2 \right| \left| \Sigma \right|)。

同态和逆同态运算

一个同态函数h: \Sigma^\* \rarr \Gamma^\*可以递归的定义为：

~h(\epsilon)=\epsilon

~h(u\sigma)=h(u)h(\sigma)

于是则有~h(uv)=h(u)h(v)。（以上所述中~\epsilon为空字符，~u,v \in \Sigma^\* , \sigma \in \Sigma）

\mathcal{L} \sube \Sigma^\* : h(\mathcal{L}) := \{ h(w) ~| ~w \in \mathcal{L} \}：对于接受语言L的DFA，只要将其中代表~\delta(q,\sigma)的边替换成一个序列~h(\sigma)并在其中加入不属于原DFA状态的新状态，就产生了接受语言h(L)的DFA。

\mathcal{L} \sube \Gamma^\* : h^{-1}(\mathcal{L}) := \{ w ~| ~h(w) \in \mathcal{L} \}：定义一个~Q,\Sigma,s,F都不变的新DFA，并定义新的转移函数为~\delta'(q,\sigma) := \delta^\*(q,h(\sigma))，则~(Q,\Sigma,\delta',s,F)就是逆同态运算产生的新DFA。

此外替换运算和逆同态运算的方法近似。

最小自动机

等价类自动机

对于一个正则语言，接受该语言的等价类自动机是一个~(Q,\Sigma,\delta,s,F)的5-元组。其定义如下：

Q是等价关系~L的等价类的集合：[x], x \in \Sigma^\*的集合

~s=[\epsilon]

F = \{ [x] ~| ~x \in L \}

~\delta([x],\sigma) = [x\sigma]

~L被称为Nerode关系，是Myhill-Nerode定理的基础。简单的来说就是对于任意~x,y,z \in \Sigma^\*，如果 xz \in L \Leftrightarrow yz \in L，那么x~Ly。

唯一性

对于任意给定的确定有限状态自动机都能找到一个与之计算能力等价的最小确定有限状态自动机，简称最小自动机。该最小自动机中状态的数量等于能识别相同语言的等价类自动机中等价关系的数量，我们可以称最小自动机和等价类自动机“实际上”是相等的，也就是同构。非正式的说法是：对于最小自动机上的任意状态都可以通过一个同构函数变换成等价类自动机上的一个状态。

能识别一个正则语言的等价类自动机是唯一的，因此能识别该语言的最小自动机也是唯一的。

算法

定义一个非等价关系：N := \{(p,q) ~| ~p,q \in Q, \exists w \in \Sigma^\* : \delta^\*(p,w) \in F \leftrightarrow \delta^\*(q,w) \notin F \}，如下步骤可以得到这个集合N：

如果p \in F, ~q \notin F，就给所有的状态对(p,q)和(q,p)打上标记

重复步骤3，直到所标记的状态对没有变化为止

对于未标记的状态对(p,q)和σ，如果~(\delta(p,\sigma),\delta(q,\sigma))被标记过了就把(p,q)也标记上

以上所有标记了的状态对的集合就是非等价关系N

以下是由一个任意DFA转换到一个最小DFA的步骤：

把所有不能从开始状态达到的状态删除

通过上述标记算法计算非等价关系N

一步一步将不属于N的状态对中的两个状态合成一个状态

这样就得到了接受相同语言的最小自动机。复杂度为O(\left| Q \right| ^2 \left| \Sigma \right|)。

非确定有限状态自动机

非确定有限状态自动机或非确定有限自动机（NFA）是对每个状态和输入符号对可以有多个可能的下一个状态的有限状态自动机。这区别于确定有限状态自动机（DFA），它的下一个可能状态是唯一确定的。尽管DFA和NFA有不同的定义，在形式理论中可以证明它们是等价的；就是说，对于任何给定NFA，都可以构造一个等价的DFA，反之亦然：通过使用幂集构造。两种类型的自动机只识别正则语言。非确定有限自动机有时被称为有限类型的子移位（subshift）。非确定有限状态自动机可推广为概率自动机，它为每个状态转移指派概率。

非确定有限自动机是Michael O. Rabin和Dana Scott在1959年介入的[1]，他们证明了它与确定自动机的等价性。

目录 [隐藏]

1 直观介绍

2 形式定义

3 性质

4 NFA-ε的性质

5 实现

6 例子

7 NFA与DFA

8 NFA-ε的应用

9 引用

9.1 注释

9.2 参考文献

直观介绍

NFA同DFA一样，消耗输入符号的字符串。对每个输入符号它变换到一个新状态直到所有输入符号到被耗尽。

不像DFA，非确定意味着对于任何输入符号，它的下一个状态不是唯一确定的，可以是多个可能状态中的任何一个。因此在形式定义中，一般都谈论状态幂集的子集：转移函数不提供一个单一状态，而是提供所有可能状态的某个子集。

一种扩展的NFA是NFA-λ（也叫做NFA-ε或有ε移动的NFA），它允许到新状态的变换不消耗任何输入符号。例如，如果它处于状态1，下一个输入符号是a，它可以移动到状态2而不消耗任何输入符号，因此就有了歧义：在消耗字母a之前系统是处于状态1还是状态2呢?由于这种歧义性，可以更加方便的谈论系统可以处在的可能状态的集合。因此在消耗字母a之前，NFA-ε可以处于集合{1,2}内的状态中的任何一个。等价的说，你可以想象这个NFA同时处于状态1和状态2:这给出了对幂集构造的非正式提示：等价于这个NFA的DFA被定义为此时处于状态q={1,2}中。不消耗输入符号的到新状态的变换叫做λ转移或ε转移。它们通常标记着希腊字母λ或ε。

接受输入的概念类似于DFA。当最后的输入字符被消耗的时候，NFA接受这个字符串，当且仅当有某个转移集合把它带到一个接受状态。等价的说，它拒绝这个字符串，如果不管应用什么转移，它都不能结束于接受状态。

形式定义

通常定义两种类似类型的NFA: NFA和“有ε-移动的NFA”。普通的NFA被定义为5-元组(Q, Σ, T, q0, F)，它构成自

状态的有限集合Q

输入符号的有限集合Σ

转移函数T : Q×Σ → P(Q)

“初始”（或“开始”）状态q0，有着q0 ∈ Q

“接受”（或“最终”）状态的集合F，有着F ⊆ Q

这里的P(Q)指示Q的幂集。“有ε-移动的NFA”（有时也叫做“NFA-ε”或“NFA-λ”）修订转移函数为允许空串ε作为可能的输入，结果为

T : Q ×(Σ ∪{ε})→ P(Q)。

可以证明普通NFA和有ε移动的NFA是等价的，给定任何一个都可以构造出识别同样语言的另一个。

性质

机器开始于任意初始状态并读取来自它的符号表的符号的字符串。自动机使用状态转移函数T来使用当前状态，和刚读入的符号或空串来确定下一个状态。但是，“NFA的下一个状态不只依赖于当前输入事件，还依赖于任意数目的后续输入事件。直到这些后续事件出现才有可能确定这个机器所处的状态” [2]。如果在自动机完成读取的时候，它处于接受状态，则称NFA接受了这个字符串，否则称为它拒绝了这个字符串。

NFA接受的所有字符串的集合是NFA接受的语言。这个语言是正则语言。

对于所有NFA都可以找到接受同样语言的一个确定有限状态自动机（DFA）。所以出于实现（可能）更简单的机器的目的把现存的NFA转换成DFA是可行的。这是使用幂集构造进行的，这可能导致在必需状态的数目上的指数增长。幂集构造的形式证明在这里给出。

对于有状态的可数无限集合的非确定有限自动机，幂集构造给出有状态的连续统的确定自动机因为可数无限集合的幂集是连续统：2^{\aleph \_{0}}=\aleph \_{1})。在这种情况下，为了使状态转移有意义，状态的集合必须被赋予一个拓扑。这种系统叫做拓扑自动机。

NFA-ε的性质

对于所有p,q\in Q，写p{\stackrel {\epsilon }{\rightarrow }}q当且仅当从p沿着零或更多个\epsilon 箭头前进可到达q。换句话说，p{\stackrel {\epsilon }{\rightarrow }}q当且仅当存在q\_{1},q\_{2},\cdots q\_{k}\in Q这里的k\geq 0使得

q\_{1}\in T(p,\epsilon ),q\_{2}\in T(q\_{1},\epsilon )\cdots ,q\_{k}\in T(q\_{k-1},\epsilon ),q\in T(q\_{k},\epsilon )。

对于任何p\in Q，从p可到达的状态的集合叫做p的ε-闭包，并写为

\,E(\{p\})=\{q\in Q:p{\stackrel {\epsilon }{\rightarrow }}q\}。

对于P\subset Q的任何子集，定义P的ε-闭包为

E(P)=\bigcup \limits \_{p\in P}E(\{p\})。

ε-转移是传递的，这可以证明，对于所有q\_{0},q\_{1},q\_{2}\in Q和P\subset Q，如果q\_{1}\in E(\{q\_{0}\})且q\_{2}\in E(\{q\_{1}\})，则q\_{2}\in E(\{q\_{0}\})。

类似的，如果q\_{1}\in E(P)且q\_{2}\in E(\{q\_{1}\})则q\_{2}\in E(P)

设x是字母表Σ上的字符串。一个NFA-ε M接受字符串x，如果存在x的形如x1x2 ... xn的表示，这里的xi ∈(Σ ∪{ε})，和状态序列 p0,p1, ..., pn二者，这里的pi ∈ Q，满足下列条件：

p0 \in E({q0})

pi \in E(T(pi-1, xi ))对于i = 1, ..., n

pn \in F。

特别注意某些字母xi可以是{ε}；它们不是选择自单独的Σ，而是来自Σ ∪{ε}。

实现

有多种方式实现NFA:

转换成等价的DFA。在某些情况下这导致在自动机大小上的指数爆炸，从而辅助空间正比于在NFA中状态的数目（因为状态值存储要求给在NFA中的每个状态最多一位）。

保持机器当前可能处在的所有状态的集合数据结构。在消耗最后一个输入符号的时候，如果这些状态之一是最终状态，则机器接受这个字符串。在最坏的情况下，这要求辅助空间正比于在NFA中状态的数目；如果集合结构为每个NFA状态使用一位，则这个方式等价于前者。

建立多个复件。对于每个n路决定，NFA建立这个机器的直到n-1个复件。每个都进入单独的状态。如果在耗尽最后的输入符号的时候，至少一个NFA复件处在接受状态，则NFA就接受它。（这也要求关于NFA状态数目的线性存储，因为对所有NFA状态都可能有一个机器）。

通过NFA的转移结构明确的传播（propagate）记号（token）并在一个记号到达最终状态的时候匹配。这在NFA应当编码触发转移的事件的额外上下文的时候是有用的。（对使用这种技术来跟踪对象引用的实现可查看Tracematches）。

例子

下面的例子展示一个NFA M，带有二进制字母表，它确定输入是否包含偶数个0或偶数个1。设M =(Q, Σ, T, s0, F)这里的

Σ = {0, 1},

Q = {s0, s1, s2, s3, s4},

E({s0}) = { s0, s1, s3 }

F = {s1, s3}，而

转移函数T定义自下列状态转移表：

0

1

ε

S0

{}

{}

{S1, S3}

S1

{S2}

{S1}

{}

S2

{S1}

{S2}

{}

S3

{S3}

{S4}

{}

S4

{S4}

{S3}

{}

M的状态图是：

NFAexample.svg

M可以被看作两个DFA的并集：一个有状态{S2, S1}而另一个有状态{S3, S4}。

M的语言可以描述为如下正则表达式给出的正则语言：

(1^{\*}(01^{\*}01^{\*})^{\*})\cup (0^{\*}(10^{\*}10^{\*})^{\*})

NFA与DFA

NFA与确定有限自机动（DFA，或简称FA）的辨识能力是一样的，因而两者是等价的。每个FA也可以写成NFA的形式，只要把转换函式由\delta (q\_{n-1},x)=q\_{n}改写成\delta (q\_{n-1},x)=\{q\_{n}\}就可以，即是与之等价的NFA的转换函数的输出结果即是FA的转换函数的输出结果的单元集。反之，对任何NFA M=(Q, Σ, δ, q0, F)来说，如果它可以接受语言L，则必定存在某个FA M1=(Q1, Σ, δ1, q1, F1)，也可以接受L。可以从“状态”的定义下手“消除”NFA的不确定性。NFA的转换函数的输出结果本为状态集合Q的子集合，现在把这一个子集合当成一个状态看待，即是FA M1中的状态是NFA中状态集合的子集合。这技巧叫做子集合的建构（subset construction）。即是

Q\_{1}=2^{Q}{\text{ and }}q\_{1}=\{q\_{0}\}{\text{ for }}q\in Q\_{1}{\text{ and }}a\in \Sigma ,\delta \_{1}(q,a)=\bigcup \_{p\in q}\delta (p,a)

F\_{1}=\{q\in Q\_{1}|q\cap F\neq 0\}

NFA-ε的应用

NFA和DFA是等价的，如果一个语言可被一个NFA识别，则它也可以被一个DFA识别，反之亦然。这种等价性的建立是重要和有用的。有用是因为构造识别给定语言的NFA有时比构造这个语言的DFA要容易很多。重要是因为NFA可以用来减少建立计算理论中很多重要性质的数学工作的复杂性。例如，使用NFA比使用DFA证明下列性质要更加容易：

（i）两个正则语言的并集是正则的。

（ii）两个正则语言的串接是正则的。

（iii）一个正则语言的Kleene闭包是正则的。