概念

记号

有字母表中的符号组成的有限长度的序列。记号 s 的**长度**记为|s|。长度为 0 的记号称为**空记号**，记为**ε** 。

有限自动机(Finite State Automaton)

为研究某种计算过程而抽象出的计算模型。拥有有限个状态，根据不同的输入每个状态可以迁移到其他的状态。

非确定有限自动机(Nondeterministic Finite Automaton)

简称 **NFA**，由以下元素组成：

1. 有限状态集合 *S*;
2. 有限输入符号的字母表**Σ** ;
3. 状态转移函数 *move*;
4. 开始状态 *sSUB{0}*;
5. 结束状态集合 *F*，*F* ∈ *S*。

自动机初始状态为 *sSUB{0}*，逐一读入输入字符串中的每一个字母，根据当前状态、读入的字母，由状态转移函数 *move* 控制进入下一个状态。如果输入字符串读入结束时自动机的状态属于结束状态集合 *F*，则说明该自动机接受该字符串，否则为不接受。

确定有限自动机(Deterministic Finite Automaton)

简称 **DFA**，是 NFA 的一种特例，有以下两条限制：

1. 对于空输入**ε** ，状态不发生迁移;
2. 某个状态对于每一种输入最多只有一种状态转移。

将正则表达式转换为 NFA(Thompson 构造法)

算法

算法 1 将正则表达式转换为 NFA(Thompson 构造法)

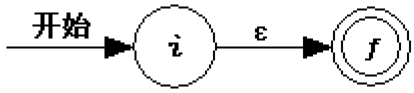
输入 字母表**Σ** 上的正则表达式 r

输出 能够接受 *L(r)*的 NFA *N*

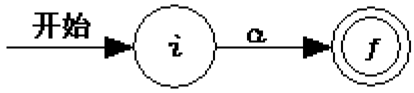
方法 首先将构成 *r* 的各个元素分解，对于每一个元素，按照下述**规则 1** 和**规则 2** 生成 NFA。 **注意**：如果 *r* 中记号 *a* 出现了多次，那么对于 *a* 的每次出现都需要生成一个单独的 NFA。

之后依照正规表达式 r 的文法规则，将生成的 NFA 按照下述**规则 3** 组合在一起。

规则 1 对于空记号ε ，生成下面的 NFA。

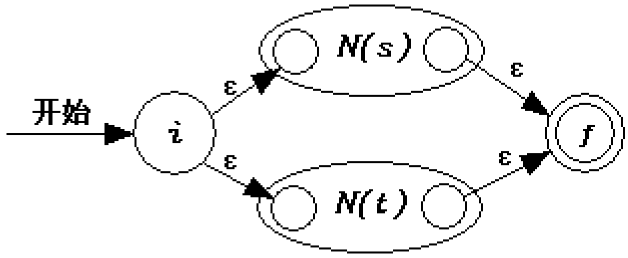


规则 2 对于Σ 的字母表中的元素 a，生成下面的 NFA。

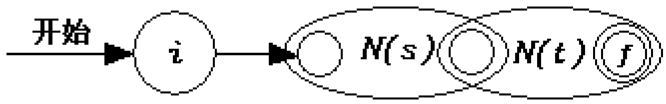


规则 3 令正规表达式 s 和 t 的 NFA 分别为 $N(s)$和 $N(t)$。

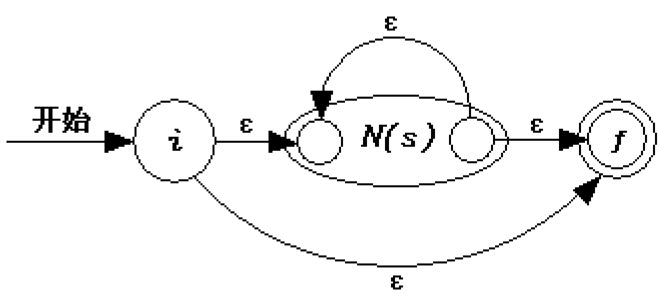
a) 对于 s/t，按照以下的方式生成 NFA $N(s/t)$。



b) 对于 st，按照以下的方式生成 NFA $N(st)$。



c) 对于 $s*$，按照以下的方式生成 NFA $N(s*)$。



d) 对于(s)，使用 s 本身的 NFA $N(s)$。

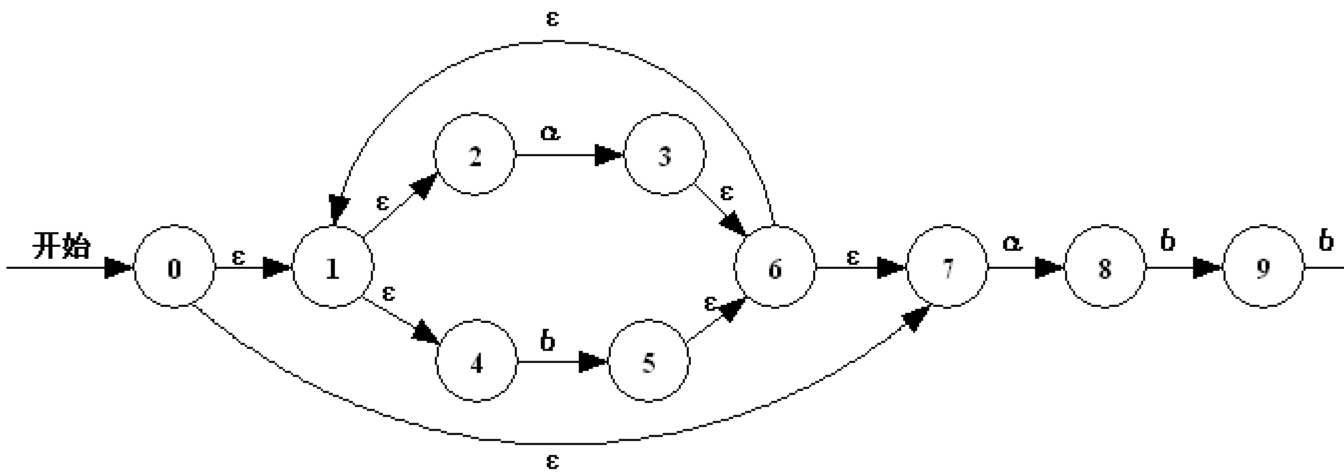
性质

算法 1 生成的 NFA 能够正确地识别正则表达式，并且具有如下的性质：

1. *N(r)*的状态数最多为 *r* 中出现的记号和运算符的个数的 2 倍。
2. *N(r)*的开始状态和结束状态有且只有一个。
3. *N(r)*的各个状态对于Σ 中的一个符号，或者拥有一个状态迁移，或者拥有最多两个ε 迁移。

示例

利用算法 1，根据正则表达式 *r=(a|b)*abb* 可以生成以下的 NFA。



将 NFA 转化为 DFA

算法

使用以下的算法可以将 NFA 转换成等价的 DFA。

算法 2 将 NFA 转化为 DFA

输入 NFA *N*

输出 能够接受与 N 相同语言的 DFA *D*

方法 本算法生成 D 对应的状态迁移表 *Dtran*。DFA 的各个状态为 NFA 的状态集合，对于每一个输入符号，*D* 模拟 *N* 中可能的状态迁移。

定义以下的操作。

操作	说明
ε *-closure(s)*	从 NFA 的状态 *s* 出发，仅通过ε 迁移能够到达的 NFA 的状态集合

操作	说明
ε -*closure(T)*	从 *T* 中包含的某个 NFA 的状态 *s* 出发，仅通过 ε 迁移能够到达的 NFA 的状态集合
move(T, a)	从 *T* 中包含的某个 NFA 的状态 *s* 出发，通过输入符号 *a* 迁移能够到达的 NFA 的状态集合

```
令 Dstates 中仅包含ε -closure(s), 并设置状态为未标记;
while Dstates 中包含未标记的状态 T do
begin
  标记 T;
  for 各输入记号 a do
  begin
    U := ε -closure(move(T, a));
    if U 不在 Dstates 中 then
      将 U 追加到 Dstates 中，设置状态为未标记;
    Dtrans[T, a] := U;
  end
end
```

ε -closure(T)的计算方法如下：

```
将 T 中的所有状态入栈;
设置ε -closure(T)的初始值为 T;
while 栈非空 do
begin
  从栈顶取出元素 t;
  for 从 t 出发以ε 为边能够到达的各个状态 u do
    if u 不在ε -closure(T)中 then
    begin
      将 u 追加到ε -closure(T)中;
      将 u 入栈;
    end
end
```

示例

将上面生成的 NFA 转化为 DFA。

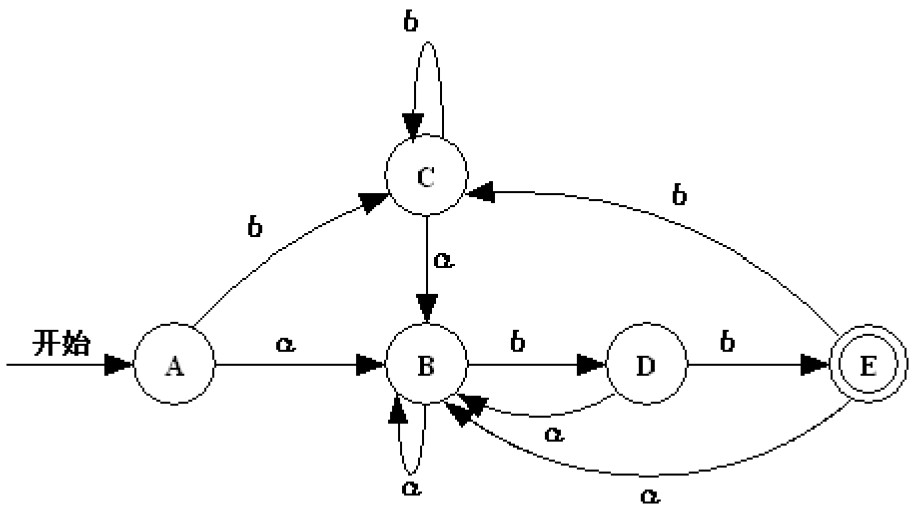
最初，*Dstates* 内仅有ε -closure(0) = A = {0, 1, 2, 4, 7}。然后对于状态 *A*，对于输入记号 *a*，计算 ε *-closure(move(A, a))* = ε *-closure*(*move*({0, 1, 2, 4, 7}, a)) = ε *-closure*({3, 8}) = {1, 2, 3, 4, 6, 7, 8}，即 B={1, 2, 3, 4, 6, 7, 8}，*Dtran[A, a]=B*。对于状态 *A*，由输入记号 *b* 能够到达的仅有 4->5，因此 C = ε *-closure*({5}) = {1, 2, 4, 5, 6, 7}，即 *Dtran[A, b] = C*。

以此类推，可得到以下的状态和 *Dtran*。

A = {0, 1, 2, 4, 7}
B = {1, 2, 3, 4, 6, 7, 8}
C = {1, 2, 4, 5, 6, 7}
D = {1, 2, 4, 5, 6, 7, 9}
E = {1, 2, 4, 5, 6, 7, 10}

状态	输入符号	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

由此得出 DFA 如下图所示。



NFA 和 DFA 的效率

给定正则表达式 *r* 和输入记号序列 *x*，判断 *r* 是否能够接受 *x*。

使用 NFA 的情况下，由正则表达式生成 NFA 的时间复杂度为 *O(|r|)*，另外由于 NFA 的状态数最多为 r 的 2 倍，因此空间复杂度为 *O(|r|)*。由 NFA 判断是否接受 *x* 时，时间复杂度为 *O(|r|×|x|)*。因此，总体上处理时间与 r、x 的长度之积成比例。这种处理方法在 x 不是很长时十分有效。

如果使用 DFA，由于利用 DFA 判断是否接受 *x* 与状态数无关，因此时间复杂度为 *O(|x|)*。但是 DFA 的状态数与正则表达式的长度呈指数关系。例如，正规表达式 *(a|b)*a(a|b)(a|b)...(a|b)*，尾部有 n-1 个 *(a–b)*的话， DFA 最小状态数也会超过 2SUP{n}。