正则表达式

1 正则表达式的定义

正则表达式（Regular Expression）是一种强大的，便捷的，高效的文本处理工具，它可以表示比单字符、字符串集合等更加复杂的搜索模式。下面首先给出正则表达式和它所表达语言的形式化定义。

一个正则表达式 RE 是符号集合 $\Sigma\ \{\varepsilon, |, \bullet, *, (,)\}$上的一个字符串，它可以递归定义如下：

空字符 ε 是正则表达式。

任意字符 $\alpha \in \Sigma$ 是正则表达式。

如果 RE_1 和 RE_2 都是正则表达式，则（RE_1），（$RE_1 \cdot RE_2$），（$RE_1 \mid RE_2$）和（RE_1*）亦是正则表达式。

通常（$RE_1 \cdot RE_2$）可以简写为 RE_1RE_2。符号“·”，“*”，“|”称为操作符，可以通过为每个操作符赋予优先级来消除更多的括号。为了方便起见，这里使用了额外的后缀操作符“+”，它的含义是 $RE+=RE \cdot RE*$。其他所使用的操作符如”?”，字符组，”.”等实际上都可以用上面的方式来表达。下面定义正则表达式所表达的语言。

正则表达式 RE 所表达的语言是 Σ 上的一个字符串集合。根据 RE 的结构，可以将它递归的定义如下：

如果 RE 是 ε，则 $L(RE)=\{\varepsilon\}$，即空串。

如果 RE 是 $\alpha \in \Sigma$，则 $L(RE)=\{\alpha\}$，即包含一个字符的单串。

如果 RE 是（RE_1）这种形式，则 $L(RE) = L(RE_1)$。

如果 RE 是（RE_1RE_2）这种形式，则 $L(RE) = L(RE_1)\ L(RE_2)$，其中 W_1W_2 可以看成是字符串集 w 的集合，其中，$w = w_1w_2$ 并且 $w1 \in W_1$，$w_2 \in W_2$。操作符表示字符串的连接。

如果 RE 是（$RE_1|RE_2$）这种形式，则 $L(RE) = L(RE_1) \cup L(RE_2)$，是这两种语言的并集，“|”称为并操作符。

如果 RE 是（RE_1*）这种形式，则 $L(RE) = L(RE)^* = \cup_{i \geq 0} L(RE)^i$，其中 $L_0=\{\varepsilon\}$ 并且 $L_i = LL_{i-1}$，

它表示字符串集合是由 0 个或者多个 RE_1 表达的字符串连接而成。“*“称为星操作符。

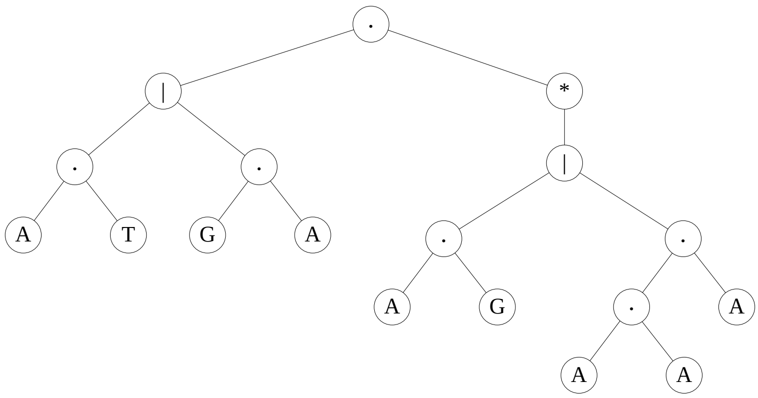
正则表达式 *RE* 的规模是指它所包含的属于字母表Σ 的字符的个数，在算法复杂性分析中，它是一个重要的度量。

在文本 T 中搜索正则表达式 *RE* 的问题就是找到文本中所有属于语言 *L*(*RE*)的字串。搜索的方法是首先将正则表达式解析成一颗表达式树，然后将表达式树转换成非确定性有限自动机（NFA）。直接使用 NFA 进行搜索是可行的，然而 NFA 算法处理速度通常比较慢，一般的，搜索过程最坏情况时间复杂度是 $O(mn)$，但是所需存储空间并不多。另外一种策略是将 NFA 转变成确定性有限自动机（DFA），它的搜索时间是 $O(n)$，但是构造这样的一个自动机所需的最坏情况时间和空间复杂度都是 $O(2^m)$。

2 构造解析树

通常来说，解析树并不是唯一的。在解析树中，每个叶节点都是使用 $\Sigma\cup\{\varepsilon\}$中的一个字符来标识的，而每个中间节点则使用操作符集合{|, •, *}中的一个进行标识。

一种可能的解析树使用二叉树来表示，二叉树的父节点是一个操作符，两个子节点表示这个操作符作用的两个子表达式。如正则表达式(AT|GA)((AG|AAA)*)的解析树可以表示如下：



。

程序中使用这个二叉树的后序遍历序列来存储这个解析树，那么上面那个正则表达式的存储序列如下：

A T • G A • | A G • A A • A • | * •。

函数 re2post 就是将输入的正则表达式字符串转换成解析树的后序遍历序列。解析过程中有两个重要的变量，natom 和 nalt，natom 表示解析到这个字符为止，已经有多少个原子结构，而 nalt 表示解析到这个字符为止，已经有多少个分支结构。正则表达式中的括号表示一个子表达式，这个子表达式对于括号外面的表达式来说是一个原子结构，它内部的 natom 和 nalt 的值和外部的表达式的这些值没有关系。为了正确的处理这种括号及其嵌套，程序中使用堆栈来辅助解析，每当碰见“(“，将当前的 natom 和 nalt 压入栈中，新的 natom 和 nalt 从零开始；而解析到”)“时，则根据当前的 natom 和 nalt 值进行后续处理，然后从栈中弹出上一层的 natom 和 nalt。具体的处理算法如下：

Parse(p = p1p2…pm, last)

v = 0;

While $p_{last} \neq \$$ **Do**

If $p_{last} \in \Sigma$ **Then**

If (natom > 1) **Then**

```
            --natom;  v ←v+'.';
        EndIf
        v ←v + p_last; natom++;
    Else If = '|'
        v ←v + (natom-1)×'.'
        nalt++;
    Else If = '*' or '+' or '?'
    v ←v + p_last;
    Else If = '('
        If (natom > 1) Then
            --natom;  v ←v+'.';
        EndIf
        push(natom, nalt);
        nalt ← 0; natom ← 0;
    Else If = ')'
        v ←v + (natom-1)×'.'
    v ←v + nalt×'|'
        pop(natom, nalt);
        natom++;
    EndIf
Endwhile
v ←v + (natom-1)×'.'
v ←v + nalt×'|'
Return v;
```

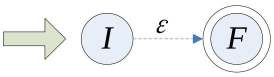
3　构造 NFA

有多种方式用来从正则表达式构造 NFA，最常见的两种，也是实践中经常使用的是 Thompson 构造法和 Glushkov 构造法。Thompson 方法简单，并且构造的 NFA 中状态数量（最多 2*m* 个）和转移数量（最多 4*m* 个）都是线性的。这种自动机存在 ε-转移，即空转移。

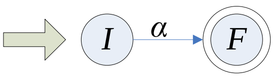
Thompson 自动机

Thompson 自动机构造的核心思想是先形成正则表达式 *RE* 对应的树表示 T_{re}，然后自底向上地对树的每个节点 v，构造一个自动机 $Th(v)$来识别以 v 为根的子树所表达的语言。根据不同类型的中间节点和叶节点，有不同的自动机构造方法，具体情况如下。

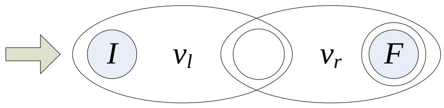
空字的构造方法。自动机由连接两个节点而组成



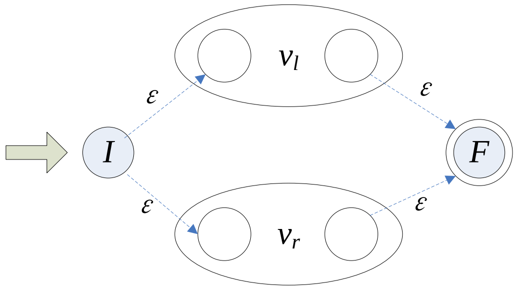
单字符 α 的构造方法，与空字类似，只不过转移是使用字符来标识，而不是使用空字符串。



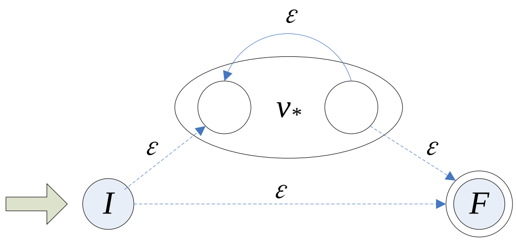
相连节点的构造法。将两个子节点 v_l 和 v_r 对应的 Thompson 自动机合并，即第一个自动机的终止状态成为第二个自动机的初始状态。



联合节点的构造法。对于联合节点，则必须通过子节点对应的自动机 $Th(v_l)$和 $Th(v_r)$中的一个。这时需要 ε-转移。构造过程中，必须添加两个新的状态：一个是初始状态 I，从它有两个 ε-转移分别到自动机 $Th(v_l)$和 $Th(v_r)$的初始状态；另一个是终止状态 F，从自动机 $Th(v_l)$和 $Th(v_r)$的终止状态分别由 ε-转移到达终止状态 F。它表达的语言是 $RE_{vl}|RE_{vr}$。

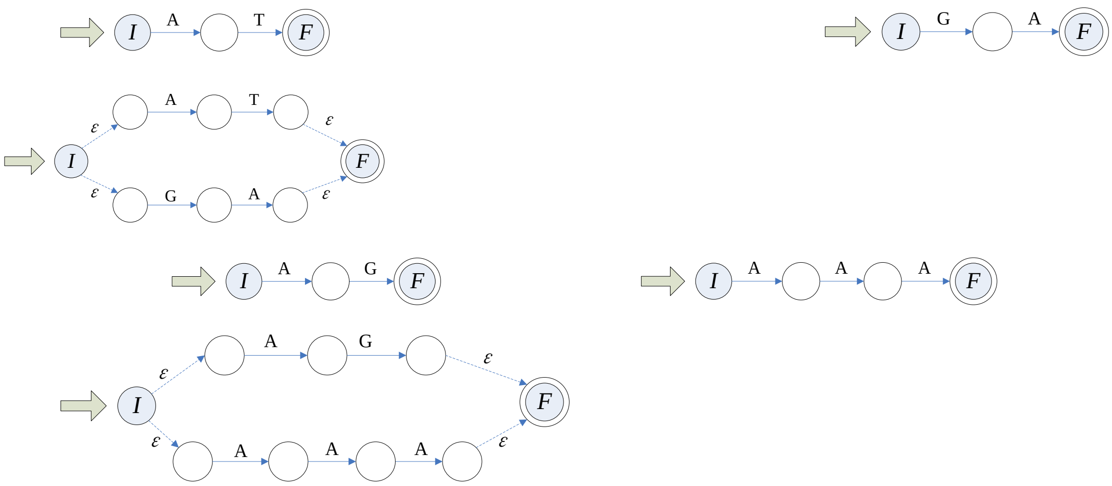


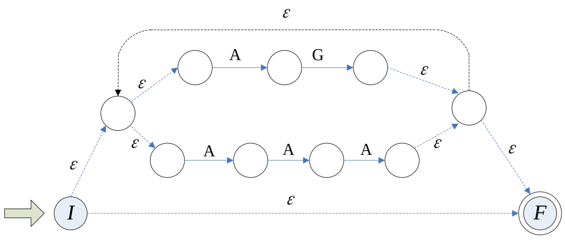
星节点的构造方法，它使用了同联合节点构造方法相同的思想。首先，因为对于语言 RE_{v*}，节点 v 的唯一子节点 $v*$可以被重复任意多次，所以需要创建一个从自动机 $Th(v_*)$的终止状态指向其初始状态的 ε-转移。但是星符号也意味着自动机 $Th(v_*)$可以被忽略。因此需要创建初始节点 I 和终止节点 F，并用一个 ε-转移把它们连接起来。另外，再创建两条 ε-转移分别用来从节点 I 指向 $Th(v_*)$的初始状态以及从 $Th(v_*)$的终止状态指向 F。最终，自动机识别的语言是$(RE_{v*})*$ 。

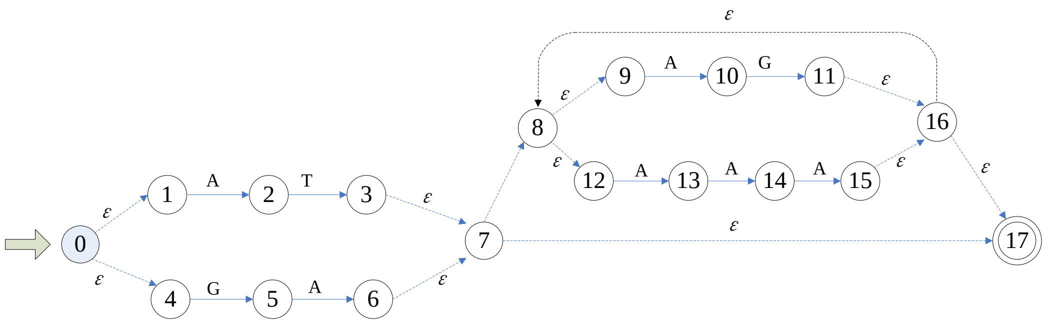


整个 Thompson 算法包含自底向上的树的遍历，同时保证根节点开始构造的自动机即为能够表示整个正则表达式的 Thompson 自动机。

在构造树表示中的每一个节点时，自动机中相应地最多增加 2 个状态和 4 个转移。因此，构造完成后，状态与转移的数量最多为 $2m$ 和 $4m$ 个。下面图表示了正则表达式(AT|GA)((AG|AAA)*)的构造过程。







Thompson 算法由函数 post2nfa 实现。post2nfa 函数输入一个数组表示的解析树，返回 Thompson 自动机，它使用了下面两种数据结构。

```
typedef struct _state
{
    int c;  表示状态的特性，小于 256 时，表示此状态输入 c 将转移到 out 指向的状态
    struct _state *out;  下一个状态
    struct _state *out1; c 是 Split 时，指向下一个分支状态
    int lastlist;
    int stateid;  状态编号
} State;

typedef union _ptrlist
{
    union _ptrlist *next;
    State *s;
} Ptrlist;
typedef struct _frag
{
    State *start;
    Ptrlist *out;
} Frag;
```

Frag 结构是一个部分 NFA，start 指向 NFA 的开始状态，out 指向一系列位置，这些位置需要被设置成这个部分 NFA 的下一个状态。函数中使用了一个 Frag 的栈来保存 NFA 的片段。

```
stackp = stack;
for(p=postfix; *p; p++){
    switch(*p){
    default:  单字符的构造
        /*  生成一个新的状态 s */
```

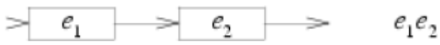
s = state(g, *p, NULL, NULL);

/* 生成一个新的 Frag，用 s 作为 start 状态，它的 out 作为终止状态，压入栈中 */

push(frag(s, list1(&s->out)));

break;

case '.'+256: 相连节点的构造



e2 = pop();

e1 = pop();

/* 连接两个相邻 Frag，第一个的 out 指向第二个的 start 状态 */

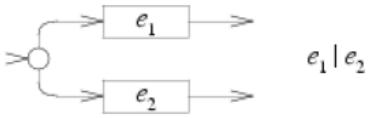
patch(e1.out, e2.start);

/* 生成一个新的 Frag，用 e1 的 start 状态作为 start 状态，e2 的 out 作为终止状态，压入栈中 */

push(frag(e1.start, e2.out));

break;

case '|'+256: 联合节点的构造



e2 = pop();

e1 = pop();

/* 生成一个新的 Split 状态 s，指向 e1 和 e2 的 start 状态*/

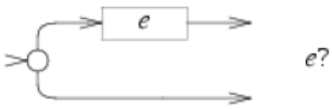
s = state(g, Split, e1.start, e2.start);

/* 生成一个新的 Frag，用 Split 的 start 状态作为 start 状态，终止状态为 e1 和 e2 的终止状态的连接，压入栈中 */

push(frag(s, append(e1.out, e2.out)));

break;

case '?'+256: 问号节点的构造



e = pop();

/* 生成一个新的 Split 状态 s，指向 e 的 start 状态和 ε-转移*/

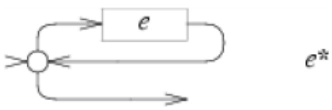
s = state(g, Split, e.start, NULL);

/* 生成一个新的 Frag，用 Split 的 start 状态作为 start 状态，终止状态为 e 和 s 的终止状态的连接，压入栈中 */

push(frag(s, append(e.out, list1(&s->out1))));

break;

case '*'+256: 星节点的构造



e = pop();

/* 生成一个新的 Split 状态 s，指向 e1 的 start 状态和 ε-转移*/

s = state(g, Split, e.start, NULL);

/* e 的下一个状态回指 s */

patch(e.out, s);

/* 生成一个新的 Frag，用 Split 的 start 状态作为 start 状态，终止状态为 s 的终止状态，压入栈中 */

push(frag(s, list1(&s->out1)));

break;

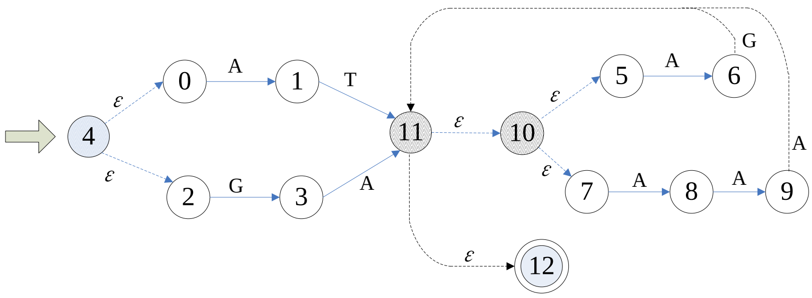
case '+'+256: 加号节点的构造

e = pop();

/* 生成一个新的 Split 状态 s，开始状态为 e1 的 start 状态和终止状态为 ε-转移*/

```
s = state(g, Split, e.start, NULL);
patch(e.out, s);
/* 生成一个新的 Frag，用 e 的 start 状态作为 start 状态，终止状态为 s 的终止状态，压入栈中 */
push(frag(e.start, list1(&s->out1)));
break;
}
}
```

下面是正则表达式(AT|GA)((AG|AAA)*)的 NFA。图中阴影的状态是 Split 状态。



4 DFA 构造

DFA 算法的核心思想是，当使用 NFA 遍历文本时，会经过很多转移，因此会激活一个状态集合。然而，DFA 在一个时刻只有一个确定的活动状态，因此可以在 NFA 的状态集合上定义相对应的 DFA。该思想的关键在于，确定性自动机的当前唯一状态就是 NFA 的当前活动状态集合。

在 NFA 中，*E*(s)表示状态 s 对应的 ε 闭包，它是在 NFA 中状态 s 能够通过 ε-转移到达的所有状态的集合。

假设 NFA 为（Q, Σ，I，F，Δ），那么 DFA 定义为：

$$(\wp(Q), \Sigma, E(I), F', \delta)$$

其中，$F' = \{f \in \wp(Q), f \cap F \neq \emptyset\}$，并且$\delta(S,\sigma) = \bigcup_{s',\exists s\in S,(s,\sigma,s')\in\Delta} E(s')$。

上述定义中的 $\delta(S, \sigma)$表示：对于 S 中所有可能的活动状态 s，可以通过标记为字符 σ 的转移找到所有可能的状态 s'，然后再从跟随所有可能的 ε-转移寻找其他状态。

算法中 DFA 状态使用数据结构

```
typedef struct _dState
{
List l; 指向一个数组，数组中是这个 DFA 状态对应的 NFA 状态的集合
struct _dState *next[256];转移表
struct _dState *left;
struct _dState *right;
int id; 状态 id
int flags;
} DState;
```

Dstate 是用二叉树的形式组织起来的，以 *l* 作为关键字。算法的思想如下：

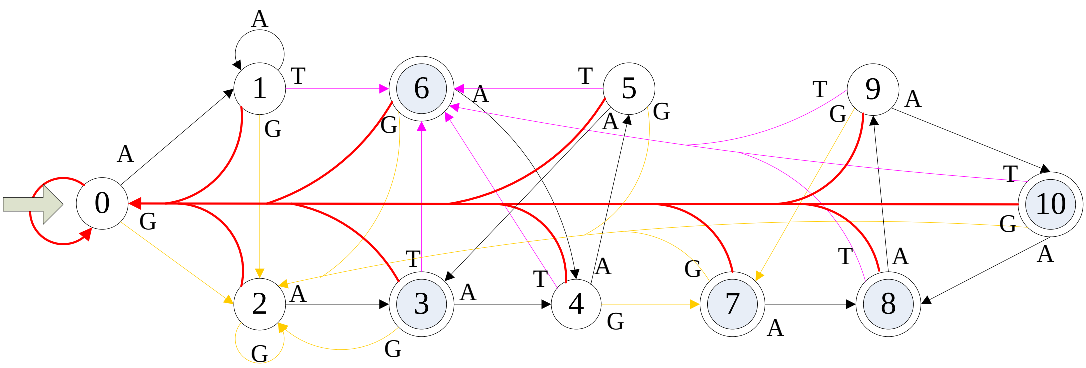
```
build_dstate（DState *d）
 For σ ∈ Σ Do
     If (d, σ, Null)  ∈  δ
       d′ = nextstate(d, σ)
       δ←δ ∪(d, σ, d′)
       build_dstate(d′)
     End If
 Endfor

 nextstate(DState *d, σ)
For s  ∈  d->l Do
  If s->c == σ
    addstate(l, s->out);
  End If
Endfor
d′=  dstate(l)  根据 l 生成一个 Dstate
Return d′
```

将 s 状态上能够通过 ε-转移到达的状态放入 *l* 中，实际上就是 *s* 的 ε 闭包

```
 addstate(List *l, State *s)
If(s->c == Split) s 状态具有 ε-转移
  addstate(l, s->out);
  addstate(l, s->out1);
  Return l;
EndIf
l ←l  ∪s
Return l;
```

下面是正则表达式(AT|GA)((AG|AAA)*)的 DFA，加入了初始自环，即.*(AT|GA)((AG|AAA)*).



上图中，黑色的箭头表示输入 A 后状态的转移，黄色的箭头表示输入 G，紫色的箭头表示输入 T 后状态的转移，红色的箭头表示其余的输入表示的状态转移，有两个圈的状态表示终止状态。

使用 DFA 进行搜索十分简单，从状态 0 开始，依次读入字符，转移到下一个状态，如

果这个状态是终止状态，则发现一个匹配，否则继续读入字符，直到读完为止。

5 Glushkov 自动机

Glushkov 自动机通过只对字符计数，可以标记出字符表Σ 中每个字符在正则表达式 RE 中的位置。例如，(AT|GA)((AG|AAA)*)标记为($A_1T_2|G_3A_4$)(($A_5G_6|A_7A_8A_9$)*)。我们使用$\overline{RE}$表示对正则表达式 RE 进行标记的标记表达式，使用 $L(\overline{RE})$表示它对应的语言，其中每个字符都包含它的位置的索引。于是，上面的例子正则表达式的语言为 {A_1T_2，G_3A_4，$A_1T_2A_5G_6$，…}。用 $Pos(\overline{RE})$ = {1…*m*}表示中位置的集合，用$\overline{\Sigma}$表示标记后的字母表。

首先，在标记表达式$\overline{RE}$上构造自动机，这个自动机识别语言 $L(\overline{RE})$。然后，通过消除所有字符的位置索引，可以从中抽取 Glushkov 自动机，从而识别语言 *L*(*RE*)。

字符位置的集合，加上一个初始状态 0，可以当做自动机状态集合的索引。创建 *m*+1 个状态，标记为 0 到 *m*。状态 *j* 表示已经从文本中读取了一个字符串，该字符串在 NFA 中的位置 *j* 结束。当再读入一个新的字符 α 时，需要知道从位置 j 通过 α 可以到达的位置。这个位置可以通过 Glushkov 自动机计算出来。

为了说明 Glushkov 算法，下面首先引入四个新的定义。其中，α_y表示$\overline{RE}$中 *y* 处的被索引字符。

$First(\overline{RE}) = \{x \in Pos(\overline{RE}), \exists u \in \overline{\Sigma}^*, \alpha_x u \in L(\overline{RE})\}$

集合 $First(\overline{RE})$表示 $L(\overline{RE})$的初始状态集合，也就是说，在这些位置可以开始读入字符。上面的例子中，*First*(($A_1T_2|G_3A_4$)(($A_5G_6|A_7A_8A_9$)*))={1，3}。

$Last(\overline{RE}) = \{x \in Pos(\overline{RE}), \exists u \in \overline{\Sigma}^*, u\alpha_x \in L(\overline{RE})\}$

集合 $Last(\overline{RE})$表示 的终止状态集合，也就是说，在这些位置可以识别出 *L*(*RE*)中的字符串。上面的例子中，*Last*(($A_1T_2|G_3A_4$)(($A_5G_6|A_7A_8A_9$)*))={2, 4, 6, 9}。

$Follow(\overline{RE}, x) = \{y \in Pos(\overline{RE}), \exists u, v \in \overline{\Sigma}^*, u\alpha_x\alpha_y v \in L(\overline{RE})\}$

集合 $Follow(\overline{RE})$表示在 $Pos(\overline{RE})$中从 *x* 可以到达的所有位置。上面的例子中，从位置 6 可以到达的位置集合为 *Follow*(($A_1T_2|G_3A_4$)(($A_5G_6|A_7A_8A_9$)*), 6)={7, 5}。

函数 $Empty_{RE}$ 可以递归的定义如下：

$$\begin{aligned} Empty_{\varepsilon} &= \{\varepsilon\} \\ Empty_{\alpha\in\Sigma} &= \emptyset \\ Empty_{RE_1|RE_2} &= Empty_{RE_1} \cup Empty_{RE_2} \\ Empty_{RE_1\cdot RE_2} &= Empty_{RE_1} \cap Empty_{RE_2} \\ Empty_{RE*} &= \{\varepsilon\} \end{aligned}$$

当 ε 属于 *L*(*RE*)时，它的值为{ε}；否则它的值为∅。

确定的 Glushkov 自动机$\overline{GL}$可以识别语言 $L(\overline{RE})$，它可以通过下面的方法进行构造：

$$\overline{GL} = (S, \Sigma, I, F, \bar{\delta})$$

其中，

（i）*S* = {0, 1, ⋯, *m*}是状态集合，它是位置 $Pos(\overline{RE})$的集合，初始状态为 *I* = 0。

（ii）*F* = $Last(\overline{RE})$∪($Empty_{RE}$ {0})是终止状态集合。通常，如果状态（位置）*i* 属于 $Pos(\overline{RE})$，则 *i* 是一个终止状态。当空字ε 属于 $L(\overline{RE})$时，初始状态 0 也是终止状态。这时，$Empty_{RE}$ = {ε}，则 $Empty_{RE}$ {0}= {0}；否则 $Empty_{RE}$ {0}= ∅

（iii）$\bar{\delta}$是自动机的转移函数，定义为：

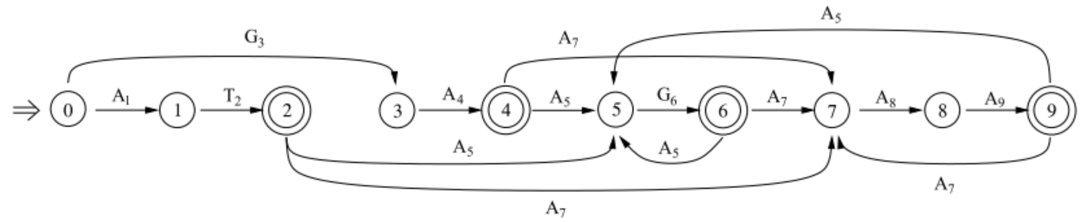
$$\exists x \in Pos(\overline{RE}), \exists y \in Follow(\overline{RE}, x), \bar{\delta}(x, \alpha_y) = y \qquad (5.1)$$

通常情况下，如果状态 *y* 在状态 *x* 之后，那么从 *x* 到 *y* 有一个转移 α_y。从初始状态开始

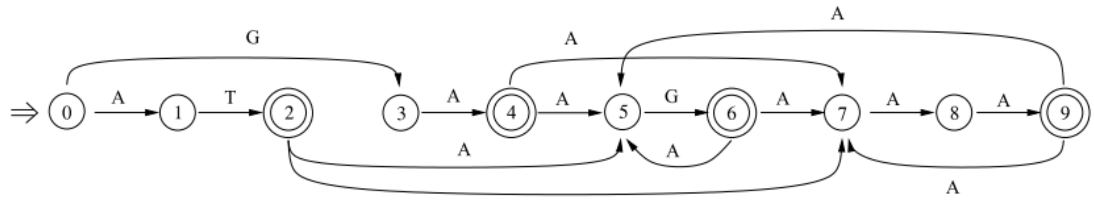
的转移定义为：

$$\exists y \in First(\overline{RE}), \bar{\delta}(0, \alpha_y) = y \tag{5.2}$$

图给出了标记正则表达式对应的 Glushkov 自动机。



为了获得原始的 RE 的 Glushkov，只要简单地去掉标记自动机的位置索引即可。在这个过程中，自动机通常会变成非确定的。新的自动机对应语言 L(RE)。例子 $((A_1T_2|G_3A_4)((A_5G_6|A_7A_8A_9)^*)$对应的 Glushkov 自动机如所示。



Glushkov 算法基于正则表达式 RE 对应的树表示 TRE。树中的每个节点 v 都代表 RE 的子表达式 REv。算法中使用了下列与 v 相关的变量：

First(v)：表示集合中所有位置的列表。

Last(v)：表示集合中所有位置的列表。

$Empty_v$：如果中包含空串 ε，它为 True，否则为 False。

对于每个节点，这些变量都是后序计算的，也就是说，先计算出 v 的所有子节点的变量值，然后再计算 v 的变量值。如果 v 是"|"或者"."，则把它的两个子节点分别记做 vl 和 vr；如果 v 表示"*"，则把它唯一的子节点记做 v*。

集合 Follow(x)是一个全局变量，对于每一个节点 v，Follow(x)的值要根据它在子表达式中位置的变化而不断更新。

Glushkov_variables(v_{RE}, *lpos*)

If v = [|] (v_l,v_r) **OR** $v = \boxed{\cdot}$ (v_l,v_r) **Then**

lpos ← **Glushkov_variables** (v_l,*lpos*)

lpos ← **Glushkov_variables** (v_r,*lpos*)

Else If $v = \boxed{*}(v_*)$ **Then** *lpos* ← Glushkov_variables (v_*,*lpos*)

End of If

If $v = (\varepsilon)$ **Then**

$First(v) \leftarrow \emptyset$，$Last(v) \leftarrow \emptyset$, $Empty_v \leftarrow \{\varepsilon\}$

Then If $v = (\alpha)$，$\alpha \in \Sigma$ **Then**

$lpos \leftarrow lpos + 1$

$First(v) \leftarrow \{lpos\}$, $Last(v) \leftarrow \{lpos\}$, $Empty_v \leftarrow \emptyset$, $Follow(lpos) \leftarrow \emptyset$

Then If v = [|] (v_l,v_r) **Then**

$First(v) \leftarrow First(v_l) \cup First(v_r)$

$Last(v) \leftarrow Last(v_l) \cup Last(v_r)$

$Empty_v \leftarrow Empty_{vl} \cup Empty_{vr}$

Then If $v = \boxed{\cdot}$ (v_l,v_r) **Then**

$First(v) \leftarrow First(v_l) \cup (Empty_{vl} \cdot First(v_r))$

Last (v) ←(*Empty*$_{vr}$ ·*Last* (v_l)) ∪*Last* (v_r)

Empty$_v$ ← *Empty*$_{vl}$ ∩ *Empty*$_{vr}$

  For x∈ *Last*(v_l) **Do** *Follow*(x) ← *Follow* (x) ∪ *First*(v_r)

Then If v = $\boxed{*}$(v_*) **Then**

  First(v) ← *First*(v_*), *Last*(v) ← *Last*(v_*), *Empty*$_v$ ← {ε}

For x∈ *Last*(v_*) **Do** *Follow*(x) ← *Follow* (x) ∪ *First*(v_*)

End of If

Return *lpos*

Glushkov(*RE*)

v_{RE} ← **Parse**(*RE*,1)

m ← **Glushkov_variables**(v_{RE}, 0)

Δ = ∅

For i ∈0…m **Do** create state i

For x ∈*First*(vRE) **Do**Δ ←Δ ∪{(0, α_x, x)}

For i ∈0…m **Do**

  For x ∈*Follow*(i) **Do**Δ ←Δ ∪{(i, α_x, x)}

End of for

For x ∈*Last*(v_{RE}) ∪(*Empty* $_{vRE}$ ·{0}) **Do** mark x as terminal

上面给出了 Glushkov_variables(vRE,lpos)的递归算法。对于正则表达式 RE 的树表示中的每个节点 v，该算法提供计算变量 First(v)，Last(v)，Follow(x)和 Emptyv 的方法。为了计算每个节点 vRE 的值，该算法使用后续遍历树中所有的节点。

整个 Glushkov 算法包括：将 RE 转化成树 vRE，并且使用 Glushkov_variables(vRE,0)计算它对应的变量，然后从树 vRE 的根对应的变量开始构造 Glushkov 自动机。

6 Glushkov 自动机的位并行算法

存储 DFA 状态（也就是 NFA 的多个状态集合）的一种可行方法就是使用 $O(m)$位的位掩码。其中，如果 NFA 的第 i 个状态属于 DFA，那么位掩码的第 i 位就为 1。

NFA（$Q=\{s_0 ... s_{|Q|-1}\}, \Sigma, I=s_0, F, \Delta$）可以表示如下：

$Q_n=\{0...|Q|-1\}$，$I_n=0^{|Q|-1}$，$F_n=|_{s_j\in F}0^{|Q|-1-j}10^j$（即对终止状态位置的位或运算）。

使用位掩码的 Glushkov_variables 算法如下所示：

```
Glushkov_variables(v_RE, lpos)
    If v = [ | ] (v_l, v_r) OR v = [·] (v_l, v_r) Then
        lpos ← Glushkov_variables(v_l, lpos)
        lpos ← Glushkov_variables(v_r, lpos)
    Else If v = * Then lpos ← Glushkov_variables(v_*, lpos)
    End of if
    If v = (ε) Then
        First(v) ← 0^{m+1}, Last(v) ← 0^{m+1}, Empty_v ← TRUE
    Else If v = (C) , C ⊆ Σ Then
        lpos ← lpos + 1
        For σ ∈ C Do B[σ] ← B[σ] | 0^{m−lpos}10^{lpos}
        First(v) ← 0^{m−lpos}10^{lpos}, Last(v) ← 0^{m−lpos}10^{lpos}
        Empty_v ← FALSE , Follow(lpos) ← 0^{m+1}
    Else If v = [ | ] (v_l, v_r) Then
        First(v) ← First(v_l) | First(v_r), Last(v) ← Last(v_l) | Last(v_r)
        Empty_v ← Empty_{v_l} OR Empty_{v_r}
    Else If v = [·] (v_l, v_r) Then
        First(v) ← First(v_l), Last(v) ← Last(v_r)
        If Empty_{v_l} = TRUE  Then First(v) ← First(v) | First(v_r)
        If Empty_{v_r} = TRUE  Then Last(v) ← Last(v_l) | Last(v)
        Empty_v ← Empty_{v_l} AND Empty_{v_r}
        For x ∈ Last(v_l) Do Follow(x) ← Follow(x) | First(v_r)
    Else If v = [*] (v_*) Then
        First(v) ← First(v_*), Last(v) ← Last(v_*), Empty_v ← TRUE
        For x ∈ Last(v_*) Do Follow(x) ← Follow(x) | First(v_*)
    End of if
    Return lpos
```

为了说明使用位掩码表示 Glushkov 算法，我们先介绍 Glushkov 自动机的几个性质。

（i） Glushkov 自动机的 NFA 是 ε-free 的

证明：由公式(5.1)，标记正则表达式的转移函数为$\bar{\delta}(x, \alpha_y) = y$，即当前为 x 状态，输入 α_y，转移到状态 y。注意 α_y 是正则表达式第 y 个字符和 y 的组合，显然不为空。转换后的正则表达式只是去掉位置索引，因此转移函数中 α_y 为正则表达式第 y 个字符，也是不为空的。故自动机的 NFA 是 ε-free 的。

（ii） 所有指向状态 y 的箭头都标记着同样的字符

证明：同样，根据转移函数$\bar{\delta}(x, \alpha_y) = y$，到达状态 y 的输入是 α_y，显然是同一个字符。

（iii） 假设 $B(\sigma)$为包含字符 σ 的位置的集合，则有：

$$\delta(x, \sigma) = Follow(x) \cap B(\sigma) \tag{6.1}$$

证明：设 $y \in \delta(x, \sigma)$。这意味着 y 能够从 x 位置输入 σ 到达，因此$y \in Follow(x) \cap B(\sigma)$。相反，假设$y \in Follow(x) \cap B(\sigma)$，那么表明可以从 x 位置转移到 y，并且输入 σ 也能到达 y。根据性质（ii），到达 y 的所有箭头都标记为 σ，显然也包括离开 x 的，因此 $y \in \delta(x, \sigma)$。

（iv） 不会有任何箭头到达 Glushkov 自动机的 NFA 中的 0 状态

证明：所有的箭头都是根据公式(5.1)生成的，并且根据 Follow 函数的定义，0 状态不在任何其他状态的 Follow 状态集合中。

下面我们使用性质（iii）来获得 DFA 的表示。计算 DFA 的转移表可以通过两个表来进行，一个是 $B[\sigma]$，给出了每个字符可达到状态的位掩码。第二个是 Follow 的确定性版本，一个从状态的集合到状态的集合的表 T（位掩码形式），满足：

$$T[D] = \bigcup_{i \in D} Follow(i) \qquad (6.2)$$

这个表给出了从一个 D 中的激活状态，不管通过哪个字符，可到达的状态集合。由性质（iii），下式成立：

$$\delta(D, \sigma) = T[D] \& B[\sigma] \qquad (6.3)$$

注意，存储 δ：$2^{m+1} \times \Sigma \rightarrow 2^{m+1}$，需要$(m+1)(2^{m+1} \times \Sigma)$位，而存储 T：$2^{m+1} \rightarrow 2^{m+1}$ 和 B：$\Sigma \rightarrow 2^{m+1}$ 需要$(m+1)(2^{m+1}+\Sigma)$位，节省了很多的空间。

下面给出从 Follow 计算 T 的算法：

BuildT $(Follow, m)$
    $T[0] \leftarrow 0^{m+1}$
    For $i \in 0 \ldots m$ **Do**
        For $j \in 0 \ldots 2^i - 1$ **Do**
            $T[2^i + j] \leftarrow Follow(i) \mid T[j]$
        End of for
    End of for
    Return T

我们现在给出基于位掩码以及上面构建的搜索算法。首先，用 First 和 Last 表示整个正则表达式的对应的变量。从技术的便利上看，可以设置 Follow(0)=First。其次，我们增加一个自循环在表达式的开始，以便可以搜索到从任意位置开始的正则表达式的语言。下面给出搜索算法：

Search$(RE, T = t_1 t_2 \ldots t_n)$
    Preprocessing
        $(v_{RE}, m) \leftarrow$ **Parse**(RE) /* parse the regular expression */
        Glushkov_variables$(v_{RE}, 0)$ /* build the variables on the tree */
        $Follow(0) \leftarrow 0^m 1 \mid First$ /* add initial self-loop */
        For $\sigma \in \Sigma$ **Do** $B[\sigma] \leftarrow B[\sigma] \mid 0^m 1$
        $T \leftarrow$ **BuildT**$(Follow, m)$ /* build T table */
    Searching
        $D \leftarrow 0^m 1$ /* the initial state */
        For $j \in 1 \ldots n$ **Do**
            If $D \,\&\, Last \neq 0^{m+1}$ **Then** report an occurrence ending at $j-1$
            $D \leftarrow T[D] \,\&\, B[t_j]$ /* simulate transition */
        End of for

上面提到，存储 T 和 B 需要（最坏的情况）$(m+1)(2^{m+1}+\Sigma)$位。但是在经典的 DFA 算法中，只需要存储能够达到的状态，这个状态远小于 2^{m+1} 所有可能的状态。这里也可以使用类似的算法，只计算可以到达的 *T*。下面的算法给出了递归构建 *T* 的过程，从 D = $0^m 1$ 开始，假定 T 已经初始化为 0，B，Follow 和 m 已经计算出来了。

BuildTrec (D)
    For $i \in 0 \ldots m$ **Do** /* first build $T[D]$ */
        If $D \,\&\, 0^{m-i} 1 0^i \neq 0^{m+1}$ **Then** $T[D] \leftarrow T[D] \mid Follow(i)$
    End of for
    For $\sigma \in \Sigma$ **Do**
        If $T[N \,\&\, B[\sigma]] = 0^{m+1}$ **Then** /* not built yet */
            BuildTrec $(N \,\&\, B[\sigma])$
    End of for

7 位并行 Glushkov 算法的实现

7.1 位掩码

我们使用位掩码来表示集合。位掩码实际上是内存中一块连续的内存，内存地址低的是位掩码的低位。程序中提供了一系列对位掩码的操作函数和宏。如下所示：

static BITMAP* makebitmap(unsigned numbytes,unsigned num)；

生成 num 个位掩码，每个占用 numbytes 个字节，返回指向位掩码的指针；

void free_bitmap(BITMAP* map)

释放位掩码的空间；

SETBIT(c,map,val)

宏，设置位掩码 map 的第 c 位的值为 val（0 或 1）;

TESTBIT(c,map)

宏，位掩码 map 的第 c 位是否为 1？；

BITCOPY(size,D,S)

宏，拷贝位掩码，D←S，size 个字节；

TRAVEL_BITMAP_BEGIN(size,map,i)
TRAVEL_BITMAP_END

遍历位掩码 map 中所有为 1 的位；

int compare_bitmap(int numbytes, BITMAP *A, BITMAP* B)

比较位掩码 A 和 B 的大小，从最高字节开始，逐个字节比较，每个字节看成无符号数；

void bitmap_AND(int numbytes, BITMAP *C, BITMAP* A, BITMAP* B)

位掩码与运算，C=A&B；

void bitmap_OR(int numbytes, BITMAP *C, BITMAP* A, BITMAP* B)

位掩码或运算，C=A | B；

int is_bitmap_empty(unsigned char* map, unsigned numbytes)

位掩码 map 是否为 0，是，返回 1，否则返回 0；

7.2 Hash 表

程序中 T 表是用位掩码来索引的，如果位数较多的话，直接通过位掩码找到对应的 T 的表项很麻烦。这里使用 Hash 函数将位掩码换算成一个无符号整数。对于 T 表，使用一个 hash 表来保存。每个位掩码换算成整数后，再 hash 成 Key 值，在表中查询。对于两个一样的 Key，使用链表将它们穿在一个 Key 下面。Hash 表的结构如下：

typedef struct hash_key_node{

```
  struct hash_key_node* next;
  unsigned int value;
  unsigned int key;
  BITMAP* D;
  BITMAP* T;
}hash_key_node;

typedef struct hash_node{
  struct hash_key_node* head;
  unsigned int value; //这个 key 下面有几个节点
}hash_node;

typedef struct hashtable{
  unsigned int bucket_num;
  hash_node* bucket;
}hashtable;
```

每个 hash_key_node 节点中都存储了 D 和 T，保存 D 目的是在插入一个相同 key 时，比较是否 D 相同。由于在生成 T 表前不知道有效的 T 的个数，故 hash 表的大小是动态变化的。为了不会在一个 Key 下面挂多个节点，在当前 T 的表项个数大于 2/3 的 hash 表的 bucket 数目时，就重新生成一个更大数目的表。

7.3 Glushkov 算法的实现

程序中实现的 Glushkov 算法和上面的伪代码算法很相似，这里就不在详述了。

8　一个具体的例子

正则表达式为(AT|GA)((AG|AAA)*)，字符串为 AAAGATAAGATAGAAAA。

B 表如下：（只显示所用到的）

```
B[A(0x41)]:......11 10110011
B[G(0x47)]:......00 01001001
B[T(0x54)]:......00 00000101
```

T 表如下：

```
Hash[0] :T[......00 00000000 ]=......00 00000000
Hash[6] :T[......00 00010011 ]=......00 10101111
Hash[14]:T[......00 00001001 ]=......00 00011011
Hash[18]:T[......00 10100011 ]=......01 01001111
Hash[19]:T[......01 10100011 ]=......11 01001111
Hash[20]:T[......10 10100011 ]=......01 11101111
Hash[31]:T[......00 00000001 ]=......00 00001011
Hash[37]:T[......00 10110011 ]=......01 11101111
         T[......00 01001001 ]=......00 10111011
Hash[40]:T[......00 00000011 ]=......00 00001111
```

```
Hash[41]:T[......01 00000011 ]=......10 00001111
Hash[42]:T[......10 00000011 ]=......00 10101111
Hash[43]:T[......11 00000011 ]=......10 10101111
Hash[49]:T[......00 00000101 ]=......00 10101011
Final: ......10 01010100
```

从 D 等于......00 00000001 开始：

1. 读入 A

```
T[......00 00000001 ] ......00 00001011
                 & B[A]:......11 10110011
                       D:......00 00000011
```

2. 读入 A

```
T[......00 00000011 ] ......00 00001111
                 & B[A]:......11 10110011
                       D:......00 00000011
```

3. 读入 A

```
T[......00 00000011 ] ......00 00001111
                 & B[A]:......11 10110011
                       D:......00 00000011
```

4. 读入 G

```
T[......00 00000011 ] ......00 00001111
                 & B[G]:......00 01001001
                       D:......00 00001001
```

5. 读入 A

```
T[......00 00001001 ] ......00 00011011
                 & B[A]:......11 10110011
                       D:......00 00010011
                  Final:......10 01010100
```

因为 $D\&F \neq 0^{m+1}$，所以标记一个成功匹配

6. 读入 T

```
T[......00 00010011 ] ......00 10101111
                 & B[T]:......00 00000101
                       D:......00 00000101
                  Final:......10 01010100
```

因为 $D\&F \neq 0^{m+1}$，所以标记一个成功匹配

7. 读入 A

```
T[......00 00000101 ] ......00 10101011
                 & B[A]:......11 10110011
                       D:......00 10100011
```

8. 读入 A

```
T[......00 10100011 ] ......01 01001111
                 & B[A]:......11 10110011
                       D:......01 00000011
```

9. 读入 G

```
T[......01 00000011 ] ......10 00001111
```

```
             & B[G]:......00 01001001
                D:......00 00001001
```

10. 读入 A

```
T[......00 00001001 ] ......00 00011011
             & B[A]:......11 10110011
                D:......00 00010011
            Final:......10 01010100
```

因为 $D\&F \neq 0^{m+1}$，所以标记一个成功匹配

11. 读入 T

```
T[......00 00010011 ] ......00 10101111
             & B[T]:......00 00000101
                D:......00 00000101
            Final:......10 01010100
```

因为 $D\&F \neq 0^{m+1}$，所以标记一个成功匹配

12. 读入 A

```
T[......00 00000101 ] ......00 10101011
             & B[A]:......11 10110011
                D:......00 10100011
```

13. 读入 G

```
T[......00 10100011 ] ......01 01001111
             & B[G]:......00 01001001
                D:......00 01001001
            Final:......10 01010100
```

因为 $D\&F \neq 0^{m+1}$，所以标记一个成功匹配

14. 读入 A

```
T[......00 01001001 ] ......00 10111011
             & B[A]:......11 10110011
                D:......00 10110011
            Final:......10 01010100
```

因为 $D\&F \neq 0^{m+1}$，所以标记一个成功匹配

15. 读入 A

```
T[......00 10110011 ] ......01 11101111
             & B[A]:......11 10110011
                D:......01 10100011
```

16. 读入 A

```
T[......01 10100011 ] ......11 01001111
             & B[A]:......11 10110011
                D:......11 00000011
            Final:......10 01010100
```

因为 $D\&F \neq 0^{m+1}$，所以标记一个成功匹配

17. 读入 A

```
T[......11 00000011 ] ......10 10101111
             & B[A]:......11 10110011
                D:......10 10100011
```

```
Final:......10 01010100
```

因为 $D\&F \neq 0^{m+1}$，所以标记一个成功匹配