# 实验报告书中有关每个系统功能的书写内容及格式范例

1.4 NFA 构造

一、存储结构

1.输入：正则表达式逆波兰表达式字符串 r

2.输出：二维数组保存邻接表 nfa_tab

3.具体存储结构，如表 1-7 所示

表 1-7

| 对象或变量名称 | 功能 | 存储结构 |
|---|---|---|
| NFA_TAB | 记录 NFA 类对象 | NFA* NFA_TAB=new NFA(); |
| NFA_state | 记录 NFA 结点编号栈 | QStack<int>NFA_state |
| s1,s2,s3,s4; | 记录 NFA 结点编号 | int s1,s2,s3,s4 |
| i，j | 计数变量 | int i，j |
| c | 记录当前访问字符 | char c; |
| NFA_STACK_NUM | NFA 的总结点数 | int NFA_STACK_NUM; |
| nfa_tab | 二维数组保存邻接表 | char nfa_tab[100][100] |

二、算法实现过程文字描述

1.清空 NFA_TAB，初始化 NFA_TAB 起始点

2.设置 i 初始值为 1，j 初始值为 0，从 r 获取当前位置 j 的符号 c

3.当 c 不为‘\0’时，反复执行以下步骤：

3.1 如果 c 为连接运算符,执行以下操作：

3.1.1s1、s2、s3、s4 依次保存 NFA_state 弹出的栈顶元素

3.1.2 向 NFA_TAB 中插入 s3 到 s2 权值为 ε 的边

3.1.3 依次将 s4 和 s1 压入 NFA_state 中，执行步骤 3.5

3.2 如果 c 为选择运算符，执行以下操作：

3.2.1s1、s2、s3、s4 依次保存 NFA_state 弹出的栈顶元素

3.2.2 向 NFA_TAB 插入两个新结点 i 和 i+1，

3.2.3 然后在结点编号 i 和 s4、i 和 s2、s3 和 i+1，s1 和 i+1 之间插入权值为 ε 的边

3.2.4 令 s4 等于 i，s1=i+1,将 s4 和 s1 依次压入 NFA_state 中，i 自增 2，执行步骤 3.5

3.3 如果 c 为闭包运算符，执行以下操作：

3.3.1s2、s1 依次保存 NFA_state 弹出的栈顶元素

3.3.2 向 NFA_TAB 插入两个新结点 i 和 i+1

3.3.3 在结点编号 i 和 i+1、s2 和 s1、i 和 s1、s2 和 i+1 之间建立权值为 ε 的边

3.3.4 令 s1=i，s2=i+1，将 s1 和 s2 依次压入 NFA_state 中，i 自增 2，执行步骤 3.5

3.4 否则，执行以下操作：

3.4.1 向 NFA_TAB 插入两个新结点 i 和 i+1

3.4.2 在结点编号 i 和 i+1 之间建立权值为 c 的边

3.4.3 令 s1=i，s2=i+1，将 s1 和 s2 依次压入 NFA_state 中，i 自增 2,执行步骤 3.5

3.5 j 自增，c=r[j]

4.s2、s1 保存 NFA_state 栈顶元素

5.向结点编号 0 和 s1 之间建立权值为 ε 的边

6.NFA 结点总数为 s2+1

7.将 NFA 邻接表存入 nfa_tab 中

三、算法实现流程图，如图 1-4 所示

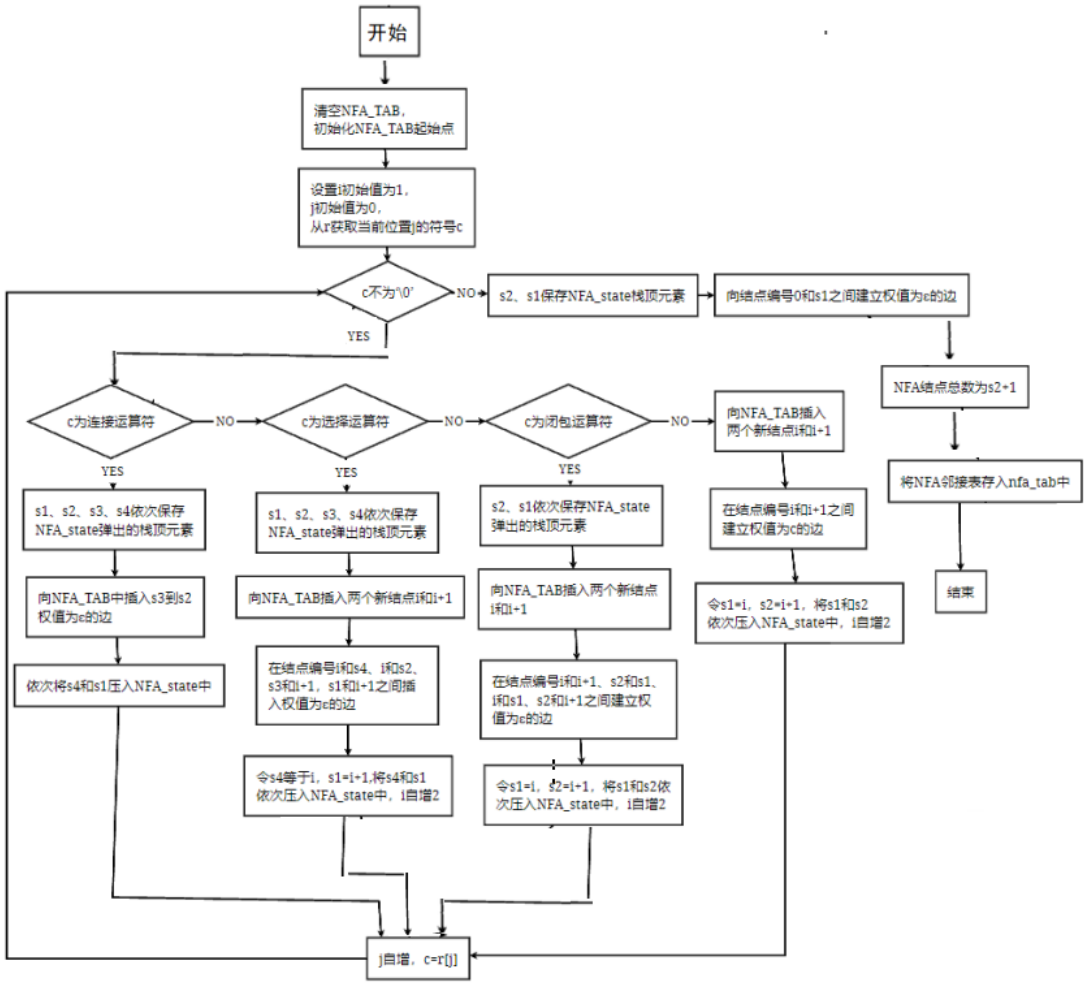


图 1-4

四、单元测试

1. 输入：a(b|c)*
2. 预测结果：如图 1-5 所示

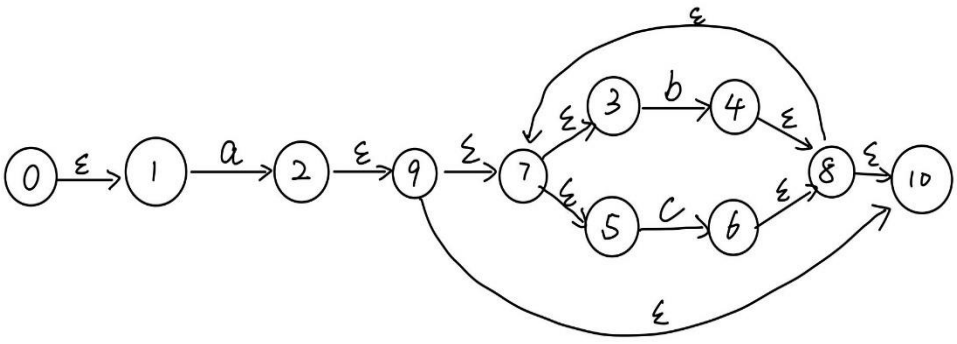


图 1-5

3.测试结果：如图 1-6 所示，经检验得符合预测结果


图 1-6

五、关键源程序代码段

1.处理连接符的代码段

```
if(c=='.')
    {
        s1=NFA_state.pop();
        s2=NFA_state.pop();
        s3=NFA_state.pop();
        s4=NFA_state.pop();
        NFA_TAB->value_insert_edge(s3,s2,epsilon);
        NFA_state.push(s4);
        NFA_state.push(s1);
    }
```

2.处理选择运算符的代码段：

```
if(c=='|')
    {
        s1=NFA_state.pop();
        s2=NFA_state.pop();
```

```
            s3=NFA_state.pop();
            s4=NFA_state.pop();
            NFA_TAB->insertNode(i);
            NFA_TAB->insertNode(i+1);
            NFA_TAB->value_insert_edge(i,s4,epsilon);
            NFA_TAB->value_insert_edge(i,s2,epsilon);
            NFA_TAB->value_insert_edge(s3,i+1,epsilon);
            NFA_TAB->value_insert_edge(s1,i+1,epsilon);
            s4=i;
            s1=i+1;
            NFA_state.push(s4);
            NFA_state.push(s1);
            i=i+2;
        }
```

3.处理闭包运算符

```
    if(c=='*')
        {
            s2=NFA_state.pop();
            s1=NFA_state.pop();
            NFA_TAB->insertNode(i);
            NFA_TAB->insertNode(i+1);
            NFA_TAB->value_insert_edge(i,i+1,epsilon);
            NFA_TAB->value_insert_edge(s2,s1,epsilon);
            NFA_TAB->value_insert_edge(i,s1,epsilon);
            NFA_TAB->value_insert_edge(s2,i+1,epsilon);
            s1=i;
            s2=i+1;
            NFA_state.push(s1);
            NFA_state.push(s2);
            i=i+2;
        }
```

### 1.5 DFA 构造

一、存储结构

1.输入：二维数组保存邻接表 nfa_tab

2.输出：Dtran 状态表

3.具体存储结构，如表 1-8 所示

表 1-8

| 对象或变量名称 | 功能 | 存储结构 |
|---|---|---|
| nfa_tab | 二维数组保存邻接表 | char nfa_tab[100][100] |
| NFA_node_set | NFA 结点得到的 DFA 状态集合 | QVector<QVector<int> >NFA_node_set |
| num_nfa_node_set | NFA 状态集合数 | int num_nfa_node_set |
| i,j | 遍历计数变量 | int i,j |

| | | |
|---|---|---|
| Dtran | Dtran 状态表 | int Dtran[100][100] |
| num_nfa_node | NFA 结点数 | int num_nfa_node |
| num_Nonterminal | 非终结符的数量 | int num_Nonterminal |
| T | 暂存 NFA 结点集合 | QVector<int>T |
| t | 暂存 NFA 结点集合 | QVector<int>t |
| temp | 暂存 NFA 结点集合 | QVector<int>temp |
| k | 计数变量 | int k |
| ch | 存放非终结符 | char ch[100]={'\0'} |