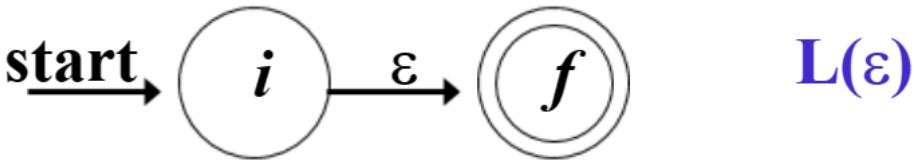
正则表达式 --> NFA

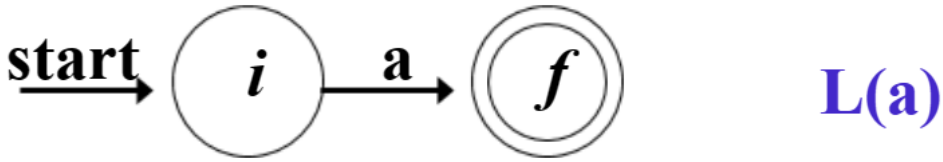
Thompson算法

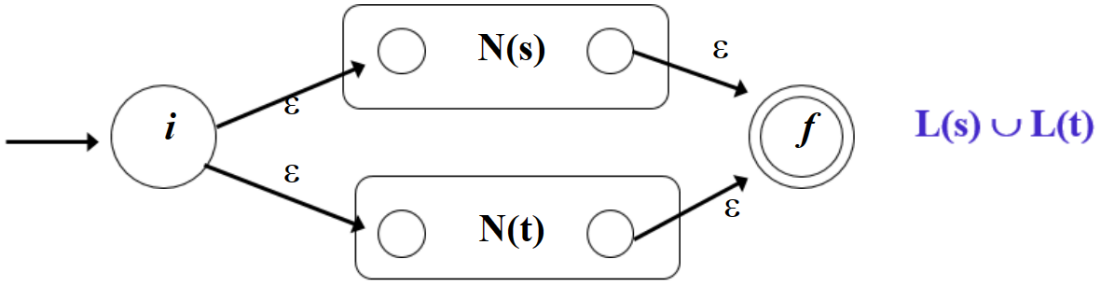
首先构造识别子表达式的ε-NFA，通过规则将ε-NFA合并，最终得到识别完整正则表达式的ε-NFA。汤普森构造法的优点是构造速度快，且构造的ε-NFA状态数较少。

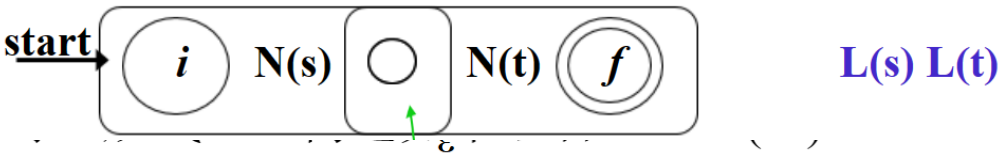
对于ε，构造NFA

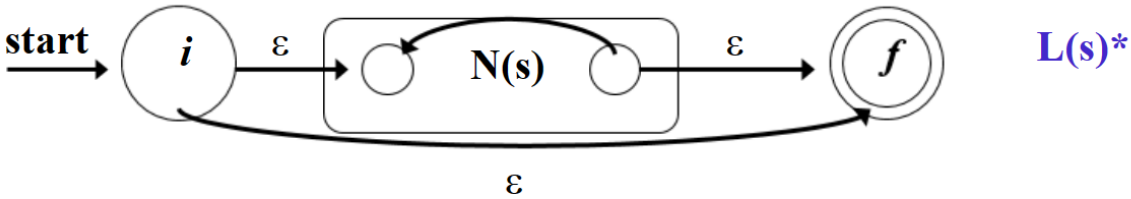


对于a ∈Σ，构造NFA









中缀表达式转后缀表达式

```
void preprocess(string& s)
{
    int i = 0;
    int length = s.size();

    while (i < length)
    {
        if ((s[i] >= 'a' && s[i] <= 'z') || (s[i] == '*') || (s[i] == ')'))
        {
            if ((s[i + 1] >= 'a' && s[i + 1] <= 'z') || s[i + 1] == '('
            {
                insert(s, i + 1, '&');
                length++;
            }
        }
        i++;
    }
}

//中缀转后缀
string infix_to_suffix(string infix_str)
{
    preprocess(infix_str);

    string suffix_str;      //输出后缀表达式
    stack<char> operator_stack; //运算符栈

    for (int i = 0; i < infix_str.size(); i++)
    {
        if (infix_str[i] >= 'a' && infix_str[i] <= 'z') //操作数直接输出
        {
            suffix_str += infix_str[i];
        }
        else                            //运算符
        {
            if (infix_str[i] == '('            //左括号入栈
            {
                operator_stack.push(infix_str[i]);
            }
            else if (infix_str[i] == ')')   //右括号
            {
                char temp = operator_stack.top();
                while (temp != '(')     //栈中元素出栈，直到栈顶为左括号
                {
                    suffix_str += temp;
                    operator_stack.pop();
                    temp = operator_stack.top();
                }
                operator_stack.pop();           //左括号出栈
            }
            else    //其他操作符
            {
```

```
                if (!operator_stack.empty())        //符号栈非空
                {
                    char temp = operator_stack.top();
                    while (priority(temp) >= priority(infix_str[i]))    //弹出栈中优先级大于等于当前运算符的
                    {
                        suffix_str += temp;
                        operator_stack.pop();
                        if (operator_stack.empty())     //运算符栈为空结束循环
                        {
                            break;
                        }
                        else temp = operator_stack.top();
                    }
                    operator_stack.push(infix_str[i]);      //当前运算符入栈
                }
                else        //符号栈为空，运算符入栈
                {
                    operator_stack.push(infix_str[i]);
                }
            }
        }
    }

    //最后如果运算符栈不为空，出栈，输出
    while (!operator_stack.empty())
    {
        char temp = operator_stack.top();
        operator_stack.pop();
        suffix_str += temp;
    }

    cout << "--------------------------------" << endl << endl;
    cout << "中缀表达式为：" << infix_str << endl << endl;
    cout << "后缀表达式为：" << suffix_str << endl << endl;

    return suffix_str;
}
```

- `preprocess`：对中缀表达式进行预处理，在操作数、闭包运算符*或右括号)之后，如果下一个字符是操作数或左括号(，则在它们之间插入连接符&，以确保两个相邻的操作数之间都有连接符，方便后续的处理。
- `infix_to_suffix`：先对中缀表达式进行预处理，定义了一个空字符串suffix_str用于保存后缀表达式，以及一个运算符栈operator_stack用于保存运算符。遍历中缀表达式的每个字符：
  - 如果当前字符是操作数（字母），直接将其添加到后缀表达式字符串suffix_str中。
  - 如果当前字符是左括号(，将其入栈。
  - 如果当前字符是右括号)，将运算符栈中的元素弹出，直到遇到左括号为止，并将这些弹出的运算符添加到后缀表达式字符串suffix_str中。
  - 如果当前字符是其他运算符，首先判断运算符栈是否为空：

        - 如果运算符栈不为空，取出栈顶的运算符temp，如果temp的优先级大于等于当前运算符的优先级，则将temp加入到后缀表达式字符串suffix_str中，并将其从运算符栈中弹出，重复此步骤直到栈顶的运算符优先级小于当前运算符的优先级，然后将当前运算符入栈。
        - 如果运算符栈为空，直接将当前运算符入栈。
    - 最后，如果运算符栈不为空，将剩余的运算符依次弹出并添加到后缀表达式字符串suffix_str中。

后缀表达式转NFA

- `NFA_State`：表示NFA状态，包含四个成员变量，分别为状态号、状态弧上的值、状态弧转移到的状态号和当前状态通过ε转移到的状态号集合。
- `NFA`：表示NFA，包含两个成员指针，分别为头指针和尾指针。

```
NFA str_to_NFA(string suffix_str)
{

    stack<NFA> NFA_stack;

    for (int i = 0; i < suffix_str.size(); i++)
    {
        //操作数
        if (suffix_str[i] >= 'a' && suffix_str[i] <= 'z')
        {
            NFA nfa = create_NFA(nfa_state_num);
            nfa_state_num += 2;     //状态总数加2

            add(nfa.head, nfa.tail, suffix_str[i]);     //NFA的头指向尾，弧上的值为
suffix_str[i]

            NFA_stack.push(nfa);        //NFA入栈
        }
        else if (suffix_str[i] == '*')      //闭包运算符
        {

            NFA nfa_1 = create_NFA(nfa_state_num);      //新建
            nfa_state_num += 2;                 //状态总数加2
            NFA nfa_2 = NFA_stack.top();            //从栈中弹出一个NFA
            NFA_stack.pop();

            add(nfa_2.tail, nfa_1.head);            //2的尾--ϵ->1的头
            add(nfa_2.tail, nfa_1.tail);            //2的尾--ϵ->1的尾
            add(nfa_1.head, nfa_2.head);            //1的头--ϵ->2的头
            add(nfa_1.head, nfa_1.tail);            //1的头--ϵ->1的尾

            NFA_stack.push(nfa_1);          //新生成的NFA入栈*/
        }
        else if (suffix_str[i] == '|')      //或运算符
        {
```

```
            NFA nfa_1, nfa_2;           //从栈中弹出两个NFA
            nfa_2 = NFA_stack.top();        //栈顶
            NFA_stack.pop();

            nfa_1 = NFA_stack.top();    //次栈顶
            NFA_stack.pop();

            NFA nfa = create_NFA(nfa_state_num);
            nfa_state_num += 2;         //状态总数加2

            add(nfa.head, nfa_1.head);      //nfa的头--ε->1的头
            add(nfa.head, nfa_2.head);      //nfa的头--ε->2的头
            add(nfa_1.tail, nfa.tail);      //1的尾--ε->nfa的尾
            add(nfa_2.tail, nfa.tail);      //2的尾--ε->nfa的尾

            NFA_stack.push(nfa);
        }
        else if (suffix_str[i] == '&')      //连接运算符
        {

            NFA nfa, nfa_1, nfa_2;
            nfa_2 = NFA_stack.top();
            NFA_stack.pop();
            nfa_1 = NFA_stack.top();
            NFA_stack.pop();

            add(nfa_1.tail, nfa_2.head);                //1的尾--ε->2的头

            nfa.head = nfa_1.head;                  //nfa的头为1的头
            nfa.tail = nfa_2.tail;                  //nfa的尾为2的尾

            NFA_stack.push(nfa);
        }
    }

    return NFA_stack.top(); //最后的栈顶元素即为生成好的NFA
}
```

- `str_to_NFA`：实现了根据后缀表达式生成NFA的算法。定义NFA_stack用于保存中间生成的NFA，并根据后缀表达式中不同的运算符进行不同的处理。
  - 如果当前字符是操作数，则新建一个NFA，并将该操作数添加到NFA的弧上，然后将该NFA入栈。
  - 如果当前字符是闭包运算符，则从栈中弹出一个NFA，新建一个NFA作为闭包运算的结果，并在新建的NFA中添加对原NFA的ε弧连接，最后将新建的NFA入栈。
  - 如果当前字符是或运算符，则从栈中弹出两个NFA，新建一个NFA作为或运算的结果，并在新建的NFA中分别添加对两个NFA的ε弧连接，最后将新建的NFA入栈。
  - 如果当前字符是连接运算符，则从栈中弹出两个NFA，将其中一个NFA的尾与另一个NFA的头进行ε弧连接，然后将这两个NFA合并为一个NFA，并将新建的NFA入栈。
  - 最终返回栈顶元素即为生成好的NFA。

NFA --> DFA

主要流程

1. 初始化DFA结构，包括终结符集合和转移矩阵。
2. 将NFA的初始状态加入到一个集合中，并求出该集合的ε-closure，作为DFA的初态。
   将初态存入队列，并进行广度优先搜索（BFS）。
3. 对于队列中的每个状态，遍历终结符集合，计算每个终结符对应的ε-closure(move(ch))。
4. 如果新得到的状态集在之前的状态集中没有出现过，则创建一个新的DFA状态，并更新转移矩阵和弧的信息。
5. 如果新得到的状态集在之前的状态集中已经存在，则直接更新转移矩阵和弧的信息。
6. 终止条件是队列为空，即所有可能的状态都已经遍历完毕。
7. 最后计算终态集，将DFA中的终态加入到终态集合中。

核心代码

- `DFA_Edge`：DFA的转换弧，包含输入字符和转换到的状态。
- `DFA_State`：DFA的状态，包括是否为终态、状态号、ε-move()闭包、射出弧的数量和射出弧的数组。
- `DFA`：DFA的整体结构，包括初态、终态集合、终结符集合和转移矩阵。

```
//求ε-cloure
IntSet ep_cloure(IntSet s)
{
    stack<int> ep_stack;

    IntSet::iterator it;
    for (it = s.begin(); it != s.end(); it++)
    {
        ep_stack.push(*it);          //将状态集中的每一个元素都入栈
    }

    while (!ep_stack.empty())
    {
        int temp = ep_stack.top();       //弹出一个元素
        ep_stack.pop();

        IntSet::iterator iter;
        for (iter = NFA_States[temp].ep_trans.begin(); iter !=
NFA_States[temp].ep_trans.end(); iter++) //遍历通过ε能转换到的状态集
        {
            if (!s.count(*iter))                 //如果当前元素没有在集合中出现
            {
                s.insert(*iter);         //加入集合中
                ep_stack.push(*iter);        //入栈
            }
        }
    }
```

```
    return s;       //返回ε-cloure
}

//求ε-cloure(move(ch))
IntSet moveEpCloure(IntSet s, char ch)
{
    IntSet temp;

    IntSet::iterator it;
    for (it = s.begin(); it != s.end(); it++)
    {
        if (NFA_States[*it].input == ch)        //对应转换弧上的值为ch
        {
            temp.insert(NFA_States[*it].trans_state_index);     //把该弧通过ch转换到的
状态加入集合
        }
    }

    temp = ep_cloure(temp);     //求temp的ε闭包
    return temp;
}
```

- `bool IsEnd(NFA n, IntSet s)`:
  - 用于判断给定的NFA状态集合s中是否包含终态。
  - 遍历状态集合s中的每个状态，如果某个状态与NFA的终态状态匹配，则返回true表示该状态为DFA的终态。
  - 如果循环结束后没有找到与终态匹配的状态，则返回false表示该状态不是终态。
- `IntSet ep_cloure(IntSet s)`:
  - 计算给定状态集合s的ε-closure（ε闭包）。
  - 创建一个栈ep_stack，并将状态集合s中的每个状态入栈。
  - 当栈不为空时，从栈顶弹出一个状态temp。
  - 遍历状态temp通过ε能够转换到的状态集合，对于每个状态：
    - 如果该状态不在集合s中，则将其加入集合s，并将其入栈。
  - 循环结束后，返回集合s作为ε-closure。
- `IntSet moveEpCloure(IntSet s, char ch)`:
  - 计算给定状态集合s在输入字符ch下的ε-closure(move(ch))。
  - 创建一个空的集合temp。
  - 遍历状态集合s中的每个状态，对于每个状态：
    - 如果该状态能够通过输入字符ch转换到另一个状态，将该状态通过输入字符ch转换到的状态加入到集合temp中。
  - 调用函数ep_cloure对集合temp求ε-closure。
  - 返回求得的ε-closure作为move(ch)的结果。

```
DFA NFA_To_DFA(NFA n, string str)
{
    cout << "----------     DFA     ----------" << endl << endl;

    int i;
    DFA d;
    set<IntSet> states;     //判断求出一个状态集s的ε-cloure(move(ch))后是否出现新状态
    memset(d.trans, -1, sizeof(d.trans));   //初始化转移矩阵

    for (i = 0; i < str.size(); i++)            //遍历后缀表达式
    {
        if (str[i] >= 'a' && str[i] <= 'z')     //操作数，加入到终结符集中
        {
            d.terminator.insert(str[i]);
        }
    }

    d.startState = 0;
    IntSet tempSet;
    tempSet.insert(n.head->state_index);        //初态加入集合

    DFA_states[0].closure = ep_cloure(tempSet);     //求dfa的初态
    DFA_states[0].is_end = IsEnd(n, DFA_states[0].closure);     //判断初态是否为终态

    DFA_state_num++;

    queue<int> q;
    q.push(d.startState);       //dfa的初态存入队列

    while (!q.empty())
    {
        int num = q.front();                //出队首元素
        q.pop();

        CharSet::iterator it;
        for (it = d.terminator.begin(); it != d.terminator.end(); it++)     //遍历终
结符集
        {
            IntSet temp = moveEpCloure(DFA_states[num].closure, *it);       //计算每
个终结符的ε-cloure(move(ch))
            if (!states.count(temp) && !temp.empty())   //状态集不为空且与之前求出来的状
态集不同，新建DFA状态
            {
                states.insert(temp);
                DFA_states[DFA_state_num].closure = temp;
                DFA_states[num].Edges[DFA_states[num].edgeNum].input = *it;
//状态弧的输入即为当前终结符
                DFA_states[num].Edges[DFA_states[num].edgeNum].trans =
DFA_state_num;       //弧转移到的状态为最后一个DFA状态
                DFA_states[num].edgeNum++;
                d.trans[num][*it - 'a'] = DFA_state_num;        //更新转移矩阵
```

```
                DFA_states[DFA_state_num].is_end = IsEnd(n,
DFA_states[DFA_state_num].closure); //是否为终态

                q.push(DFA_state_num);      //新状态号加入队列
                DFA_state_num++;
            }
            else            //求出状态集在之前求出的某个状态集相同
            {
                for (i = 0; i < DFA_state_num; i++)
                {
                    if (temp == DFA_states[i].closure)      //找到与该集合相同的DFA状态
                    {
                        DFA_states[num].Edges[DFA_states[num].edgeNum].input = *it;
//状态弧的输入即为当前终结符
                        DFA_states[num].Edges[DFA_states[num].edgeNum].trans = i;
//转移到的状态即为i
                        DFA_states[num].edgeNum++;
                        d.trans[num][*it - 'a'] = i;        //更新转移矩阵
                        break;
                    }
                }
            }
        }
    }

    //计算终态集
    for (i = 0; i < DFA_state_num; i++) //遍历dfa的所有状态
    {
        if (DFA_states[i].is_end == true)       //是终态
        {
            d.endStates.insert(i);      //将该状态号加入终态集
        }
    }

    return d;
}
```

- `NFA_To_DFA(NFA n, string str)`
  - 定义DFA类型的变量d和保存状态集的set类型变量states，用于判断是否出现新状态。函数的核心是使用队列实现广度优先搜索，依次遍历每个状态，并对其进行操作。
  - 在遍历后缀表达式的过程中，如果当前字符是操作数，则将其加入到终结符集中。接着，将NFA的初态添加到一个集合tempSet中，并应用epsilon闭包函数ep_closure()求出DFA的初态，并判断该初态是否为终态。
  - 队列不为空时，从队列中取出当前状态num，然后遍历终结符集，计算每个终结符对应的epsilon闭包的移动，并判断是否出现了新状态集。
    - 如果存在新状态集，则新建一个DFA状态，并更新状态弧以及转移矩阵。同时，如果该新状态是终态，则将其加入到终态集中，并将其状态号加入队列。
    - 如果没有出现新状态集，则找到与之前求出DFA状态集相同的DFA状态，并更新状态弧和转移矩阵即可。

- 最后遍历所有DFA状态并判断其是否是终态，如果是，则加入到终态集中并返回DFA。

最小化DFA

```
DFA min_DFA(DFA d)
{

    int i, j;
    cout << endl << "----------     minDFA     ----------" << endl << endl;

    DFA minDfa;
    minDfa.terminator = d.terminator;       //终结符集赋给minDfa
    memset(minDfa.trans, -1, sizeof(minDfa.trans));     //初始化minDfa转移矩阵

    //第一次划分，终态与非终态分开
    bool endFlag = true;                    //判断所有状态是否全为终态
    bool cutFlag = true;        //上一次是否产生新的划分的标志
    for (i = 0; i < DFA_state_num; i++)
    {
        if (DFA_states[i].is_end == false)      //不是终态
        {
            endFlag = false;
            min_DFA_state_num = 2;
            s[1].insert(DFA_states[i].index);       //该状态的状态号加入s[1]
        }
        else                        //是终态
        {
            s[0].insert(DFA_states[i].index);       //该状态的状态号加入s[0]
        }
    }

    if (endFlag)            //所有dfa状态都是终态
    {
        min_DFA_state_num = 1;          //只有一个集合
    }

    while (cutFlag)     //上一次产生新划分
    {
        int cutCount = 0;           //需要产生新的划分的数量
        for (i = 0; i < min_DFA_state_num; i++)         //遍历每个划分集合
        {
            CharSet::iterator it;
            for (it = d.terminator.begin(); it != d.terminator.end(); it++)     //遍
历dfa的终结符集
            {
                int setNum = 0;             //当前缓冲区中的状态集个数
                stateSet temp[20];          //划分状态集“缓冲区”

                IntSet::iterator iter;
                for (iter = s[i].begin(); iter != s[i].end(); iter++)       //遍历每
个状态号
```

```
                {
                    bool epFlag = true;         //该集合中是否存在没有该终结符对应的转换弧
的状态
                    for (j = 0; j < DFA_states[*iter].edgeNum; j++)     //遍历该状态的
所有边
                    {
                        if (DFA_states[*iter].Edges[j].input == *it)        //输入为
该终结符
                        {
                            epFlag = false;

                            //计算该状态转换到的状态集的标号
                            int transNum = find_set_num(min_DFA_state_num,
DFA_states[*iter].Edges[j].trans);

                            int curSetNum = 0;          //遍历缓冲区，寻找是否存在到达这
个标号的状态集
                            while ((temp[curSetNum].index != transNum) && (curSetNum
< setNum))
                            {
                                curSetNum++;
                            }

                            if (curSetNum == setNum)        //缓冲区中不存在到达这个标号
的状态集
                            {
                                //缓冲区中新建状态集
                                temp[setNum].index = transNum;      //能转换到的状态集
                                temp[setNum].s.insert(*iter);

                                setNum++;       //缓冲区中的状态集个数加一
                            }
                            else            //缓冲区中存在到达这个标号的状态集
                            {
                                temp[curSetNum].s.insert(*iter);    //加入到该状态集中
                            }
                        }
                    }

                    if (epFlag)     //该状态不存在与该终结符对应的转换弧
                    {
                        int curSetNum = 0;
                        while ((temp[curSetNum].index != -1) && (curSetNum <
setNum))
                        {
                            curSetNum++;
                        }

                        if (curSetNum == setNum)    //不存在这样的状态集
                        {
                            //缓冲区中新建状态集
```

```
                            temp[setNum].index = -1;            //转移到的状态集标号为-1
                            temp[setNum].s.insert(*iter);
                            setNum++;
                        }
                        else            //缓冲区中存在到达这个标号的状态集
                        {
                            temp[curSetNum].s.insert(*iter);
                        }
                    }
                }
                if (setNum > 1) //缓冲区中的状态集个数大于1，表示同一个状态集中的元素能转换到不同的状态集，需要划分
                {
                    cutCount++;
                    //每组划分创建新的dfa
                    for (j = 1; j < setNum; j++)
                    {
                        IntSet::iterator t;
                        for (t = temp[j].s.begin(); t != temp[j].s.end(); t++)
                        {
                            s[i].erase(*t);             //原状态集删除该状态
                            s[min_DFA_state_num].insert(*t);        //新的状态集加入该状态
                        }
                        min_DFA_state_num++;
                    }
                }
            }
        }
        if (cutCount == 0)      //本次不需要进行划分
        {
            cutFlag = false;
        }
    }

    //遍历每个划分好的状态集
    for (i = 0; i < min_DFA_state_num; i++)
    {
        IntSet::iterator y;
        for (y = s[i].begin(); y != s[i].end(); y++)
        {
            if (*y == d.startState)     //最小化DFA状态为初态
            {
                minDfa.startState = i;
            }
            if (d.endStates.count(*y))      //最小化DFA状态也为终态
            {
                min_DFA_states[i].is_end = true;
                minDfa.endStates.insert(i);     //最小化DFA状态加入终态集
            }
```

```
            for (j = 0; j < DFA_states[*y].edgeNum; j++)        //遍历该DFA状态的每条
弧，为最小化DFA创建弧
            {
                for (int t = 0; t < min_DFA_state_num; t++)
                {
                    if (s[t].count(DFA_states[*y].Edges[j].trans))
                    {
                        bool haveEdge = false;      //判断该弧是否已经创建的标志
                        for (int l = 0; l < min_DFA_states[i].edgeNum; l++)
                        {
                            if ((min_DFA_states[i].Edges[l].input ==
DFA_states[*y].Edges[j].input) && (min_DFA_states[i].Edges[l].trans == t))
                            {
                                haveEdge = true;
                            }
                        }
                        if (!haveEdge)      //创建一条新的弧
                        {
                            min_DFA_states[i].Edges[min_DFA_states[i].edgeNum].input
= DFA_states[*y].Edges[j].input;
                            min_DFA_states[i].Edges[min_DFA_states[i].edgeNum].trans
= t;
                            minDfa.trans[i][DFA_states[*y].Edges[j].input - 'a'] =
t;   //更新转移矩阵
                            min_DFA_states[i].edgeNum++;
                        }
                        break;
                    }
                }
            }
        }
    }
    return minDfa;
}
```

- `min_DFA(DFA d)`：
  - 定义DFA对象minDfa，并将原始DFA的终结符集赋值给它。
  - 进行第一次划分。判断原始DFA状态中哪些是终态，哪些不是，根据不同的结果将DFA状态划分为两个集合。如果原始DFA所有状态都是终态，则只有一个集合。如果不是，则会有两个集合。
  - 进入while，对于每个划分好的状态集合，遍历DFA的终结符集。对于每个终结符，遍历该状态集合中的每个状态，统计它们能转移到的状态集合，并将它们划分到一个新的状态集合中。当一个状态集合中的元素能够转移到的状态集合有多个时，需要进行划分，将它们分别放到不同的新的状态集合中。循环直到没有状态集合需要划分为止。
  - 遍历每个划分好的状态集合，将它们转换为最小化DFA对象。对于每个最小化DFA状态，将其对应到原始DFA状态集合中的某个状态，根据该状态是否为原始DFA的起始状态或终结状态，更新最小化DFA对象的起始状态和终结状态。同时，遍历该状态在原始DFA中能够转移到的所有状态，为最小化DFA创建对应的弧，建立转移矩阵。
  - 最后返回最小化DFA对象minDfa。