南開大學
Nankai University

计算机学院

编译原理实验报告

OT1: 实现词法分析器核心构造算法

姓名：谢畅

学号：2113665

专业：计算机科学与技术

2023 年 11 月 5 日

目录

1 程序链接与算法描述

程序链接在这里

实现词法分析器核心构造算法: 正则表达式 →NFA 的 Thompson 构造法、NFA→DFA 的子集构造法、DFA 的最小化算法

下面是对三个算法的描述:

- **正则表达式 →NFA 的 Thompson 构造法**:

  这里正则表达式的输入字符串的形式，当然还需要先给出字母表的所有字母。然后需要对正则表达式进行转换，构造对应的 NFA。这里使用的是 Thompson 构造法。值得一提的是，设计中正则表达式的运算限制于基本的 (), 连接，|，* 运算符。那么对于四种运算，只需要按照 Thompson 构造法的规则即可。

  但是这里有一些问题是输入的正则表达式对于连接运算而言，没有对应的符号。研究之后，我决定首先处理输入的正则表达式，补全 '.' 运算符作为连接符。

  最麻烦的是：中缀表达式中的运算符优先级和括号会增加转换的复杂度，因为我们需要考虑它们的顺序和嵌套关系。

  但是，如果将**中缀表达式转换为后缀表达式后**，所有的运算符都在操作数的后面，这样我们就可以直接从左到右处理后缀表达式，**不需要考虑优先级和括号等问题**，转换过程更加简单和直观。所以在这里采用中缀转后缀，再对后缀的正则表达式处理。

- **NFA→DFA 的子集构造法**:

  在得到 NFA 后，需要使用子集构造法进行 NFA→DFA 的转换。

  首先对 NFA 的初态计算 $\epsilon\_closure(s)$, 以此作为 DFA 的初态。在这个算法，**重要的是完成 $\epsilon\_closure(T)$，与 $\delta(T,a)$ 两个函数**，即状态的迁移函数、一个集合对应的 ϵ 闭包的计算，而且我们需要找到一个合适的数据结构表示集合等等结构。

- **DFA 的最小化算法**:

  这里 DFA 的最小化，首先需要检查 DFA 的所有状态是否对输入符号集 (字母表) 的所有输入符号都有转换。如果对某个输入符号，没有转换，那么**需要加入一个死状态**。在进行死状态完的处理后，才可以开始 DFA 最小化的构造。具体而言，就是对于一个组里面的状态 s、t 来说，必须对于任意输入符号 a，两者转换到同一组，才可以划分为一组，否则要继续进行划分。具体算法思路在算法实现一节介绍。

2 算法实现

2.1 正规表达式 →NFA

2.1.1 数据结构设计

```
string letters;//字母表
string expression, infix, suffix;//输入表达式、中缀、后缀字符串
int start = 0;//nfa 状态的名字，最后根据它可知道状态总数
char null_action = '~';//空字用'~'替代
struct edge {
    char weight;//letter，给定字母
    int next;  //下一个 nfa 状态
};
//在构造 nfa 用到，映射左端与右端
map<int, int>left_right;
map<int, vector<edge>>nfa;//nfa 起点，起点相连的边
#define nfaIT  map<int, vector<edge>>::iterator
//设置运算符优先级
void setPrecedence()
//添加连接符为'.'
void expr2infix()
//中缀转为后缀
void infix2suffix()
//后缀转为 nfa
int suffix2nfa()
```

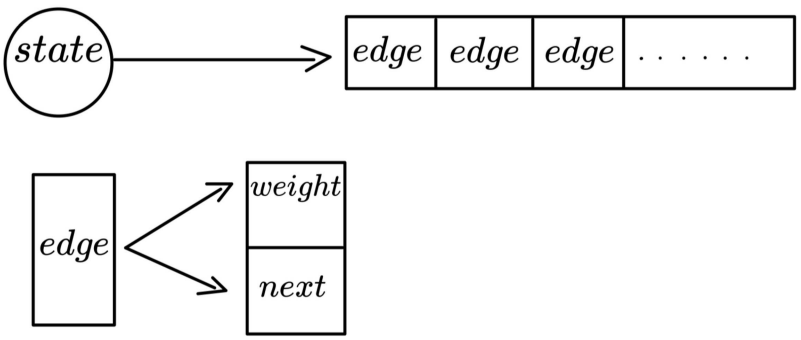


图 2.1: nfa 相关数据结构

下面对数据结构具体分析：

- nfa 的状态节点设置为 int 类型, 从 1 开始，累加
- nfa 的边用 *edge* 结构体表示，*weight* 表示边上的输入符号，*next* 表示某个 nfa 状态在此边上转换后的下一个状态。
- 可以看到这里用一个 map 映射记录点和边的映射情况。$map < int, vector < edge >>$ nfa 中的第一个 int 表示 nfa 的状态节点，而第二个 $vector < edge >$ 表示此状态对应的边。根据 nfa 我们可以清晰的观察到 nfa 图的边、节点的情况。

2.1.2 代码实现

具体逻辑如下图2.2所示。

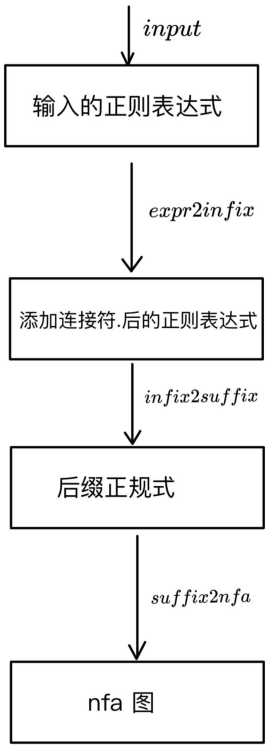


图 2.2: 正规式的处理流程

1.expr2infix: 给表达式增加连接符’．’

主要思路为: 对输入的正则表达式中，排除特殊操作符，剩余的即形如 ab 的形式，加入’．’即可。

```
1 //设置优先级
2 void setPrecedence() {
3     precedence['*'] = 3;
4     precedence['.'] = 2;
```

```
    precedence['|'] = 1;
}
//增加连接符
void expr2infix() {
    for (int i = 0; i < expression.length(); i++) {
        char cur = expression[i];
        char next = expression[i + 1];
        //默认连接符省略，这里增加'.'
        if ((cur != '|' && cur != '.' && cur != '(') && (next != ')' &&
        next != '|' && next != '*' && next != '.' && next != '\0')) {
            infix = infix + cur + '.';
        }
        else {
            infix += cur;
        }
    }
}
```

2.infix2suffix: 将中缀表达式转为后缀

主要思路为: 利用一个栈对输入的中缀表达式进行处理，对于左括号直接入栈即可，只有遇到右括号才会把栈里面的东西都出栈，然后弹出左括号。对于'|'、'*'、'.' 按照优先级判断是否入栈，如果现在的栈顶操作符优先级高，那么必须先让栈顶操作符出栈，才能让现在的操作符入栈；反之，直接让现有操作符入栈即可。对于字母，直接导入至后缀表达式即可；对于操作符，按照上述规则用栈处理。

```
//中缀转为后缀
//若栈非空，判断栈顶操作符
//当前操作符优先级低于栈顶操作符，那么要让栈顶操作符出栈。
//直到栈空或栈顶操作符优先级低于该操作符，该操作符再入栈。
void infix2suffix() {
    stack<char>cstack;
    for (int i = 0; i < infix.size(); i++) {
        char cur = infix[i];//当前字符
        if (cur == '(')
            cstack.push(cur);
        else if (cur == ')') {//需要把括号里面全部解决
            while (cstack.top() != '(') {
                char temp = cstack.top();
                suffix += temp;
```

```
                cstack.pop();
            }
            cstack.pop();//删掉左括号
        }
        else if (cur == '|' || cur == '*' || cur == '.') {
            while (!cstack.empty()) {
                char temp = cstack.top();
                //cur 优先级低，需要把过去的 temp 打印
                if (precedence[temp] >= precedence[cur]) {
                    suffix += temp;
                    cstack.pop();
                }
                else {//当前字符优先级高
                    cstack.push(cur);
                    break;
                }
            }
            if (cstack.empty())
                cstack.push(cur);
        }
        else {
            suffix += cur;
        }
    }
    //若栈还有操作符，把剩余操作符出栈
    while (!cstack.empty()) {
        char temp = cstack.top();
        suffix += temp;
        cstack.pop();
    }
}
```

3.suffix2nfa: 后缀正规式转 nfa

主要思路: 这里采用一个栈，储存操作数。具体而言，对于给定的 suffix，进行从左到右的扫描。如果当前字符为操作符，那么进行处理；如果当前字符为操作数，那么进行入栈，保留的是最左边的状态 (初态)。

如果当前字符为操作符，我们可以首先找到对应的操作数。如果是'|' 或'.'，那么栈顶以及栈顶的下一个元素就是我们的操作数；如果是'*'，栈顶即为操作数。由于我们转化为 nfa，经常会用到初态、终态，因此这里用 left_right 映射，储存初态与终态的映射，而我

们栈里面的元素是初态。

因此，对于字符为操作符的情况，可以直接找到对应的操作数进行处理，同时由于我们有 left_right 储存当前初态与当前终态的映射，所以在拼接的时候，可以很方便的进行处理，按照 Thompson 原则进行拼接即可。

```
int suffix2nfa() {
    stack<int>cstack;//储存状态
    for (int i = 0; i < suffix.size(); i++) {
        char cur = suffix[i];
        if (cur == '*') {
            int l_temp = cstack.top();
            int r_temp = left_right[l_temp];
            left_right.erase(l_temp);
            cstack.pop();
            start++;//作为开始状态
            edge e;
            e.weight = null_action;
            e.next = l_temp;
            nfa[start].push_back(e);
            e.next = l_temp;
            nfa[r_temp].push_back(e);
            e.next = start + 1;
            nfa[start].push_back(e);
            nfa[r_temp].push_back(e);
            start++;
            left_right[(start - 1)] = start;
            cstack.push(start - 1);//把最左边状态入栈

        }
        else if (cur == '.') {
            int l_temp2 = cstack.top();
            int r_temp2 = left_right[l_temp2];
            cstack.pop();
            int l_temp1 = cstack.top();
            int r_temp1 = left_right[l_temp1];
            left_right.erase(l_temp1);
            left_right.erase(l_temp2);
            edge e;
            e.weight = null_action;
```

```
35              e.next = l_temp2;
36              nfa[r_temp1].push_back(e);
37              left_right[l_temp1] = r_temp2;
38              //不需要再 pop
39          }
40          else if (cur == '|') {
41              int l_temp2 = cstack.top();
42              int r_temp2 = left_right[l_temp2];
43              cstack.pop();
44              int l_temp1 = cstack.top();
45              int r_temp1 = left_right[l_temp1];
46              cstack.pop();
47              left_right.erase(l_temp1);
48              left_right.erase(l_temp2);
49              start++;//初态
50              edge e;
51              e.weight = null_action;
52              e.next = l_temp1;
53              nfa[start].push_back(e);
54              e.next = l_temp2;
55              nfa[start].push_back(e);
56              start++;
57              e.next = start;
58              nfa[r_temp1].push_back(e);
59              nfa[r_temp2].push_back(e);
60              left_right[(start - 1)] = start;
61              cstack.push((start - 1));
62
63          }
64          else {
65              start++;//初态从 1 开始
66              edge e;
67              e.weight = cur;
68              e.next = start + 1;
69              nfa[start].push_back(e);
70              cstack.push(start);
71              left_right[start] = e.next;
72              start++;
73          }
```

```
74      }
75      return cstack.top();//初态
76  }
```

2.2 NFA→DFA

2.2.1 数据结构设计

```
1   string d_state;//状态集合
2   //dfa 状态的名字从'A'开始, '@'+1='A'
3   char d_start = '@';
4   char null_action = '~';//空字用'~'替代
5   struct edge2 {
6       char weight;//letter, 给定字母
7       char next;//下一个 dfa 状态
8   };
9   map<char, vector<edge2>>dfa;//dfa
10  //初态与终态
11  char D_S;
12  string D_F;
13  //nfa 子集与 dfa 中状态的映射
14  map<set<int>, char>class2state;
```

下面对数据结构具体分析：

- dfa 的状态节点设置为 char 类型，从'A' 开始，累加。
- dfa 的边用 *edge*2 结构体表示，*weight* 表示边上的输入符号，*next* 表示某个 dfa 状态在此边上转换后的下一个状态。
- 可以看到这里用一个 map 映射记录点和边的映射情况。$map < char, vector < edge >> dfa$ 中的第一个 char 表示 dfa 的状态节点，而第二个 $vector < edge >$ 表示此状态对应的边。根据 dfa 我们可以清晰的观察到 dfa 图的边、节点的情况。
- 用 d_state 记录了 dfa 的状态集合，D_S 是初态，D_F 是终态集合。
- class2state，记录了 nfa 一个状态子集合，在 dfa 中对应的状态。

2.2.2 代码实现

1.$\epsilon\_closure(T)$: 计算子集的 ϵ 闭包

主要思路: 用 set<int> 记录 nfa 中的状态子集。这里首先将 subClass 中所有状态压栈，temp 储存了最终的状态集合。初始 temp 与 subClass 相等。当栈不空的情况，对栈顶元素进行如下处理：对于该栈顶状态，如果对某个 nfa 状态有一条 ϵ 边，且此状态不在 temp 集合中，那么将其加入 temp，将此状态压栈。

```
set<int> _closure(set<int>subClass) {
    stack<int>cstack;
    set<int>temp(subClass);
    for (auto i = subClass.begin(); i != subClass.end(); i++)
        cstack.push(*i);
    while (!cstack.empty()) {
        int cur = cstack.top();
        cstack.pop();
        for (auto i = 0; i < nfa[cur].size(); i++) {
            if (nfa[cur][i].weight == null_action) {
                //说明 temp 集合里面没有
                if (temp.find(nfa[cur][i].next) == temp.end()) {
                    cstack.push(nfa[cur][i].next);
                    temp.insert(nfa[cur][i].next);
                }
            }
        }
    }
    return temp;
}
```

2.$\delta(T,a)$: 计算 T 集合在输入符号 a 下的状态变换集合

主要思路为: 对于集合 subClass 的状态，如果此状态有一条边，输入为 a，且边的下一个状态不在 temp 集合里，那么将其加入 temp 集合。最后返回 temp 集合即可。

```
//a 是输入符号
set<int> transition(set<int>subClass, char a) {
    set<int>temp;
    //用 nfa 的状态转换
    for (auto i = subClass.begin(); i != subClass.end(); i++) {
        for (auto j = 0; j < nfa[*i].size(); j++) {
```

```
            if (nfa[*i][j].weight == a) {
                //说明 temp 集合里面没有
                if (temp.find(nfa[*i][j].next) == temp.end()) {
                    temp.insert(nfa[*i][j].next);
                }
            }
        }
    }
    return temp;
}
```

3.nfa→dfa: 完成 dfa 映射的设置

主要思路为：首先对于给定的 NFA 初态 S, 计算其 ϵ 闭包，得到初始状态集 set_S，作为初态，这里设置了 dstates 储存不同的 nfa 状态子集合 (与 dfa 状态一一对应)，而 dstates2 起一个查重的作用，利用 set 里面自带的 find，只让 dstates 加入之前没有的 nfa 状态子集。

当 dstates 有状态子集没有处理完时，进行如下操作：对于此状态子集 cur，对每个输入符号进行 $\epsilon\_closure(\delta(T\ a))$ 的计算，得到一个新的状态子集，如果此状态子集不在 dstates 中，那么加入 dstates。相应地，class2state 记录了状态子集与 dfa 状态的对应关系，那么 dfa[class2state[cur]].push_back(e) 即可完成相关 dfa 的边的赋值。

```
//nfa->dfa
//参数的 S 代表初态，F 代表终态 是 nfa 的
void nfa2dfa(int S, int F) {
    set<int>temp;
    temp.insert(S);
    set<int>set_S = _closure(temp);//作为初态
    set<set<int>>dstates2; //起一个查重的作用
    vector<set<int>>dstates;//dfa 状态集 int 是标记位
    int flag = 0;//代表标记状态
    dstates.push_back(set_S);
    dstates2.insert(set_S);
    d_start++;//设置开始状态
    D_S = d_start;
    d_state = d_state + (char)d_start;
    class2state[set_S] = d_start;
    while (flag < dstates.size()) {
        flag++;//标记的作用
        set<int>cur(dstates[flag - 1]);//当前状态集合
```

```
        //对于每个符号，letters 是字母表
        for (int i = 0; i < letters.size(); i++) {
            set<int>U = _closure(transition(cur, letters[i]));
            //不能为空，且必须之前没有
            if (dstates2.find(U) == dstates2.end() && !U.empty()) {
                dstates.push_back(U);
                dstates2.insert(U);
                d_start++;//新状态
                d_state = d_state + (char)d_start;
                class2state[U] = d_start;
            }
            if (U.empty())
                continue;
            edge2 e;
            e.weight = letters[i];
            e.next = class2state[U];
            dfa[class2state[cur]].push_back(e);//完成 dfa 的转换
        }
    }
    //最后完成终态设置
    for (int i = 0; i < dstates.size(); i++) {
        if (dstates[i].find(F) != dstates[i].end()) {
            D_F.push_back(class2state[dstates[i]]);//设置终态
        }
    }
}
```

2.3 最小化 DFA

2.3.1 数据结构设计

```
string min_dstate;//最小化 dfa 的状态集合
char dead_state = '@';
int dead_flag = 0;//是否有死状态标志
struct edge2 {
    char weight;//letter，给定字母
    char next;//下一个 dfa 状态
};
```

```
map<char, vector<edge2>>min_dfa;//最小化的 dfa
string MIN_DS;//最小化的初态
string MIN_DF;//最小化的终态
```

这里的数据结构与 dfa 相差不大。主要是增加对死状态判断的相关元素。如果当前 dfa 中的每个状态，不是对每个输入符号都有转换，那么需要引入一个死状态，这里我们把死状态定义为'@'。

2.3.2 代码实现

1.addDeadState: 若 dfa 不满足条件则添加死状态

这里的思路比较简单，检查 d_state 中的每个状态即可。如果对于一个状态，它在某输入字母上没有对应转换，那么加入一个指向死状态的边即可。当然最后死状态在每个输入字母上的转换，都是自身。

```
void addDeadState() {
    for (int i = 0; i < d_state.size(); i++) {
        //增加死状态
        if (dfa[d_state[i]].size() < letters.size()) {
            for (int j = 0; j < letters.size(); j++) {
                int cur_flag = 0;//是否存在该符号对应边
                for (int k = 0; k < dfa[d_state[i]].size(); k++) {
                    if (dfa[d_state[i]][k].weight == letters[j]) {
                        cur_flag = 1;
                        break;
                    }
                }
                //说明没有该符号对应输入，增加死状态对应转换
                if (cur_flag == 0) {
                    dead_flag = 1;
                    edge2 e;
                    e.next = dead_state;//死状态
                    e.weight = letters[j];
                    dfa[d_state[i]].push_back(e);
                }
                else {}//对于当前符号不用加边
            }
        }
        else {}//说明边的数量等于符号个数，没有死状态
```

```
25
26      }
27      if (dead_flag) {
28          for (int i = 0; i < letters.size(); i++) {
29              edge2 e;
30              e.next = dead_state;//死状态
31              e.weight = letters[i];
32              dfa[dead_state].push_back(e);
33          }
34
35      }
36  }
```

2.MIN_dfa: 最小化 dfa

这个算法是最复杂的一块。首先对应给定的 dfa 状态集合 d_state，我们根据终态与非终态分成两个集合 s1、s2。这里应用了一个关键的 count 结构，储存了每个状态与对应状态集合的映射关系，随着之后集合的划分不断更新。

M 储存了每个状态子集分组，而 Mnew 是每一轮更新的状态子集分组。其中 unordered_map<char, map<char, set<char»>transi_count，第一个 char 是该分组中对应的每个状态，而 map<char,set<char» 是储存了该状态，在该输入下的转换的状态集合，因此它们是一对多的关系，需要用 unordered_map 储存。

下面进入 while 循环，当 M 与 Mnew 不等时，说明上一轮更新了分组，需要继续进行；如果相等，说明上一轮分组未更新，可以结束循环。进行循环后，首先对于 M 里面的每个分组 i 进行如下操作: 先记录此分组里面的 transi_count 映射。在更新完映射后，进行分组划分的循环操作。对于当前状态 w，如果前面的状态 e，在每个输入下的状态转换子集相同，那么 w 可以加入 e 的分组；反之，新创建一个分组，包含 w。在这一轮的最后一个分组结束后，count 更新完毕，我们可以进行 Mnew 的更新操作，至此一轮 while 循环完成。

现在我们已经得到了 M 中的正确分组。需要完成 min_dfa 的赋值操作。首先我们这里运用一个 map<char, char>handle 映射，记录每个状态与对应分组中代表状态的映射。一个分组中的代表状态，取分组中的第一个状态即可。注意这里要把'@' 死状态给删除。

现在拿到 handle 后，可以完成 min_dfa 的边映射关系赋值。不过这里需要采用一个 set<string>NoCopy，具体作用是: 由于可能一个分组中，对于一个输入字母，有多个到此分组状态的边。那么这个时候我们需要去重，所以采用 set 里面的 find 去重。具体方式为记录 min_dfa 中的起始状态、输入字母、转换状态，如果三者都相同说明要去重。

然后我们只需要对 handle 遍历即可，再借助 dfa 中的边，即可完成 min_dfa 的赋值。

```
1  void MIN_dfa() {
```

```
    set<char>s1;
    set<char>s2;
    //记录了 dfa 每个状态与集合的对应关系，不断更新
    map<char, set<char>>count;
    //map<char, set<char>>cur_count;
    if (dead_flag)
        d_state += '@';
    for (int i = 0; i < D_F.size(); i++)
        s2.insert(D_F[i]);
    for (int i = 0; i < d_state.size(); i++) {
        if (s2.find(d_state[i]) == s2.end()) {
            s1.insert(d_state[i]);
        }
    }
    for (auto i = s1.begin(); i != s1.end(); i++) {
        count[*i] = s1;
    }
    for (auto i = s2.begin(); i != s2.end(); i++) {
        count[*i] = s2;
    }
    set<set<char>>M;
    M.insert(s1); M.insert(s2);
    //第一个 char 对应状态 第二个对应动作与组别的映射
    unordered_map<char, map<char, set<char>>>transi_count;
    //s1 是不可接受,s2 可接受
    set<set<char>>Mnew;
    int size_flag;
    while (M != Mnew) {
        //count 的映射会不断被覆盖
        if (Mnew.size() != 0)
            M = Mnew;
        Mnew.clear();//新的重置
        for (auto i = M.begin(); i != M.end(); i++) {
            //对每个组执行如下操作
            transi_count.clear();
            for (int k = 0; k < letters.size(); k++) {
                for (auto s = (*i).begin(); s != (*i).end(); s++) {
                    transi_count[*s][dfa[*s][k].weight] = count[dfa[*s][k].next];

```

```
41                  }
42              }
43              /*for (auto xy = transi_count.begin(); xy != transi_count.end(); xy++) {
44                  cout << xy->first << " ";
45                  for (auto fk = xy->second.begin(); fk != xy->second.end(); fk++)
46                      cout << fk->first << " ";
47                  cout << endl;
48              }*/
49              for (auto w = transi_count.begin(); w != transi_count.end(); w++) {
50                  char flag = '0';//判断是否可以找到一组的状态标志
51                  for (auto e = transi_count.begin(); e != w; e++) {
52                      //遍历所有输入
53                      for (int k = 0; k < letters.size(); k++) {
54                          if (w->second[letters[k]] != e->second[letters[k]]) {
55                              break;
56                          }
57                          if (k == letters.size() - 1)
58                              flag = e->first;
59                      }
60                      //如果当前找到了一个，就退出
61                      if (flag != '0')
62                          break;
63                  }
64                  if (flag != '0') {
65                      count[flag].insert(w->first);
66                      count[w->first] = count[flag];//count 映射一个状态，与一个集合
67                      //更新 count 这里必须要更新，避免没有同步增加 w->first
68                      //因为 stl 的 = operator 是一个深拷贝，所以实际上映射的不是同一个集合
69                      //直接把所有都拷贝一遍简单
70                      for (auto member = count[flag].begin(); member != count[flag].end()
71                      ; member++)
72                          count[*member] = count[flag];
73                  }
74                  else {
75                      set<char>temp;
76                      temp.insert(w->first);
77                      count[w->first] = temp;//实际上创建了一个新集合与 w->first 对应。
78
79                  }
```

```
80              }
81              if (&(*i) != &(*M.rbegin()))continue;
82              for (auto iter = count.begin(); iter != count.end(); iter++) {
83                  if (Mnew.find(iter->second) == Mnew.end()) {
84                      Mnew.insert(iter->second);//新的组
85                      /*for (auto sk = iter->second.begin(); sk != iter->second.end(); sk++)
86                          cout << *sk << " ";
87                      cout << endl;*/
88                  }
89
90              }
91          }
92      }
93      //现在 M 是所有组的集合，需要清除死状态'@'
94      map<char, char>handle;//第一个为 dfa 中所有状态，第二个为最小化后的每组代表元素
95      for (auto i = M.begin(); i != M.end(); i++) {
96          //对于第 i 组
97          for (auto j = i->begin(); j != i->end(); j++) {
98              if (*j != '@') {
99                  handle[*j] = *(i->begin());//选用 *(i  ->begin()) 作为代表
100             }
101         }
102         if (*(i->begin()) != '@')
103             min_dstate += *(i->begin());
104     }
105
106     //处理边 min_dfa
107     set<string>NoCopy;//处理重复映射的问题
108     for (auto i = handle.begin(); i != handle.end(); i++) {
109         if (i->first == D_S)
110             MIN_DS += i->second;
111         if (D_F.find(i->first) != D_F.npos && MIN_DF.find(i->second) == MIN_DF.npos)
112             MIN_DF += i->second;
113
114         for (auto j = dfa[i->first].begin(); j != dfa[i->first].end(); j++) {
115             if (j->next == '@')
116                 continue;//死状态去除
117             edge2 e;
118             e.next = handle[j->next];
```

```
119             e.weight = j->weight;
120             string ttemp;
121             ttemp.push_back(i->second);
122             ttemp.push_back(e.weight);
123             ttemp.push_back(e.next);
124             if (NoCopy.find(ttemp) == NoCopy.end()) {
125                 min_dfa[handle[i->second]].push_back(e);
126                 NoCopy.insert(ttemp);
127 
128             }
129         }
130     }
131 }
```

3 代码测试与结果分析

3.1 完整测试 1

测试题目来自第三章书面作业

5. 使用Thompson构造法为下面正规式构造NFA，写出NFA处理符号串bbabb过程中的状态转换序列（1分）
b*a((b | ε)(a | b | ε))

图 3.3: 测试题目 1

答案如下:

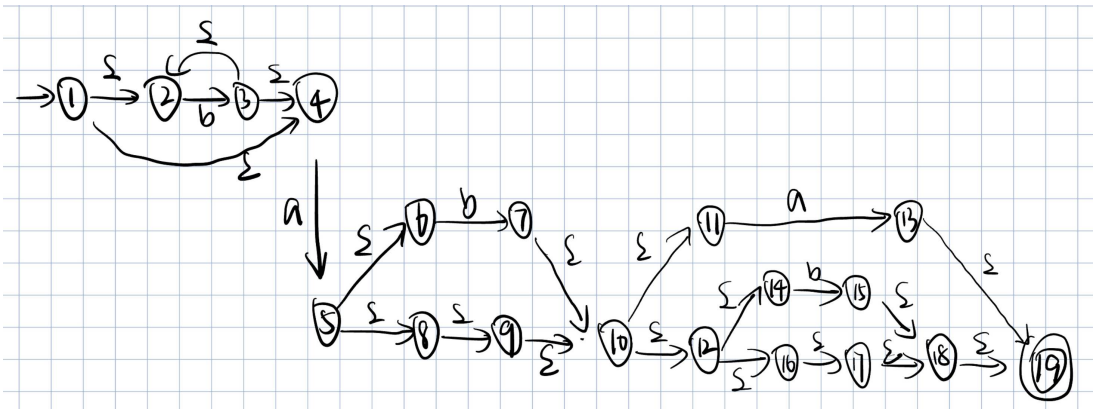


图 3.4: 测试题目 1:nfa

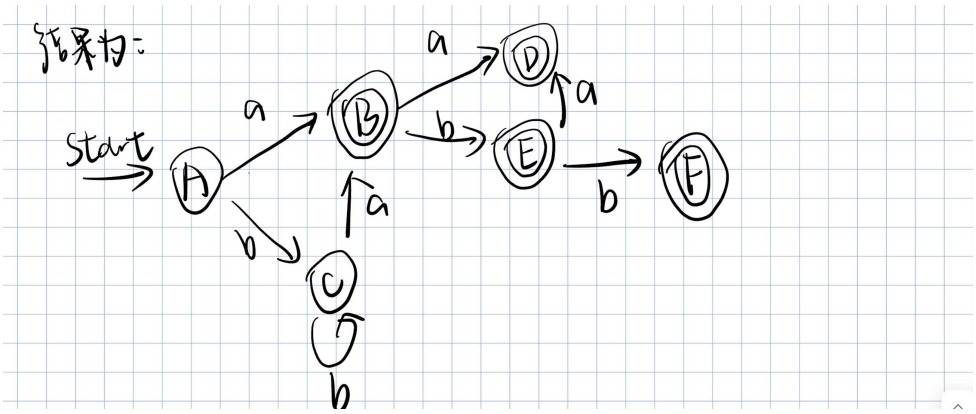


图 3.5: 测试题目 1:dfa

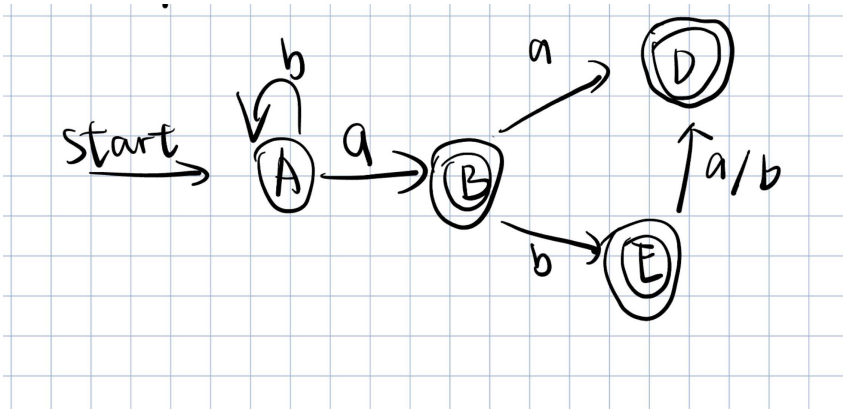


图 3.6: 测试题目 1:MIN_dfa

其中‘~’代表 ϵ 字符。由于为了简便，关于连接运算符转化成 nfa，我直接在前一个操作数的终态与后一个操作数的初态间增加一条 ϵ 边。

```
请输入字母表的所有字母:
ab
请输入正则表达式:
b*a((b|~)(a|b|~))
加入.连接符后:
b*.a.((b|~).(a|b|~))
将中缀表达式转为后缀表达式后:
b*a.b~|ab|~|..
NFA初态:3
NFA终态:22
状态集:1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22
[1—b—>2]
[2—~—>1]
[2—~—>4]
```

```
[3—~—>1]
[3—~—>4]
[4—~—>5]
[5—a—>6]
[6—~—>11]
[7—b—>8]
[8—~—>12]
[9—~—>10]
[10—~—>12]
[11—~—>7]
[11—~—>9]
[12—~—>21]
[13—a—>14]
[14—~—>18]
[15—b—>16]
[16—~—>18]
[17—~—>13]
[17—~—>15]
[18—~—>22]
[19—~—>20]
[20—~—>22]
[21—~—>17]
[21—~—>19]
DFA初态:A
DFA终态集合:BDEF
状态集合:ABCDEF
[A—a—>B]
[A—b—>C]
[B—a—>D]
[B—b—>E]
[C—a—>B]
[C—b—>C]
[E—a—>D]
[E—b—>F]
对DFA增加死状态后
[@—a—>@]
[@—b—>@]
[A—a—>B]
[A—b—>C]
```

```
[B—a—>D]
[B—b—>E]
[C—a—>B]
[C—b—>C]
[D—a—>@]
[D—b—>@]
[E—a—>D]
[E—b—>F]
[F—a—>@]
[F—b—>@]
最小化DFA
最小化DFA的初态:A
最小化DFA的终态:BDE
MIN_DFA状态集合:ABDE
[A—a—>B]
[A—b—>A]
[B—a—>D]
[B—b—>E]
[E—a—>D]
[E—b—>D]
```

可以看到初态、终态、状态转移一致，说明正确。

3.2 完整测试 2

测试题目来自龙书。

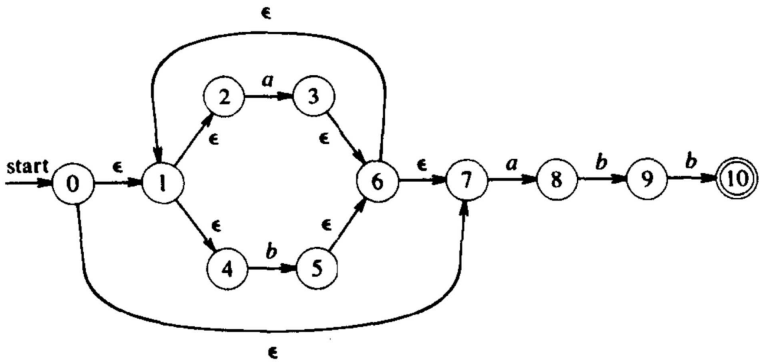


图3-27 $(a|b)^*\ abb$ 的 NFA N

图 3.7: 测试题目 2:nfa

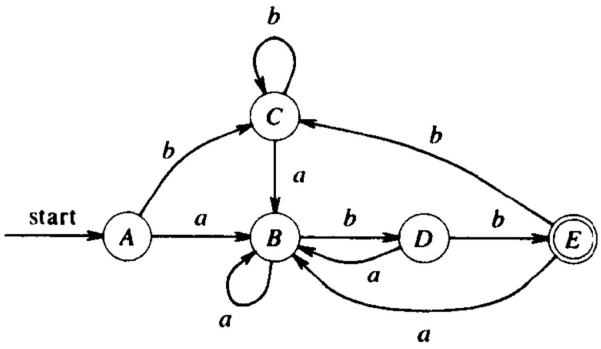


图3-29 对图3-27应用子集构造法得到的结果

图 3.8: 测试题目 2:dfa

状 态	输入符号	
	a	*b*
A	*B*	*A*
B	*B*	*D*
D	*B*	*E*
E	*B*	*A*

图3-46 简化的DFA的转换表

图 3.9: 测试题目 2:MIN_dfa

```
请输入字母表的所有字母:
ab
请输入正则表达式:
(a|b)*abb
加入.连接符后:
(a|b)*.a.b.b
将中缀表达式转为后缀表达式后:
ab|*a.b.b.
NFA初态:7
NFA终态:14
状态集:1,2,3,4,5,6,7,8,9,10,11,12,13,14
[1—a—>2]
[2—~—>6]
[3—b—>4]
[4—~—>6]
[5—~—>1]
```

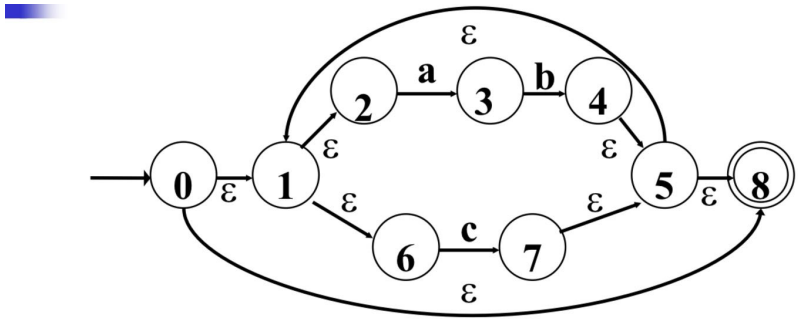
```
[5—~—>3]
[6—~—>5]
[6—~—>8]
[7—~—>5]
[7—~—>8]
[8—~—>9]
[9—a—>10]
[10—~—>11]
[11—b—>12]
[12—~—>13]
[13—b—>14]
DFA初态:A
DFA终态集合:E
状态集合:ABCDE
[A—a—>B]
[A—b—>C]
[B—a—>B]
[B—b—>D]
[C—a—>B]
[C—b—>C]
[D—a—>B]
[D—b—>E]
[E—a—>B]
[E—b—>C]
对DFA增加死状态后
[A—a—>B]
[A—b—>C]
[B—a—>B]
[B—b—>D]
[C—a—>B]
[C—b—>C]
[D—a—>B]
[D—b—>E]
[E—a—>B]
[E—b—>C]
最小化DFA
最小化DFA的初态:A
最小化DFA的终态:E
MIN_DFA状态集合:ABDE
```

```
[A—a—>B]
[A—b—>A]
[B—a—>B]
[B—b—>D]
[D—a—>B]
[D—b—>E]
[E—a—>B]
[E—b—>A]
```

可以看到最后 dfa、min_dfa 完全一致。nfa 不一致，是因为我的连接运算为了方便，直接在前一个操作数的终态与后一个操作数的初态间增加一条 ϵ 边。

3.3 完整测试 3

题目来自讲义



ε -closure({0}) = { 0, 1, 2, 6, 8}
ε -closure(δ({ 0, 1, 2, 6, 8}, a) = ε -closure({3}) = {3}
ε -closure(δ({ 0, 1, 2, 6, 8}, b) = ε -closure({}) = {}
ε -closure(δ({ 0, 1, 2, 6, 8}, c) = ε -closure({7}) = {1, 2, 5, 6, 7, 8}

图 3.10: 测试题目 3:nfa

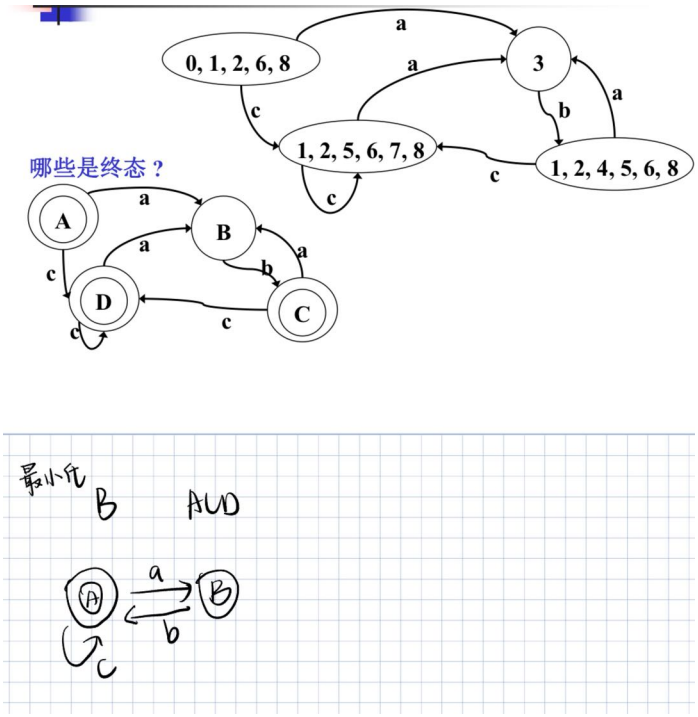


图 3.11: 测试题目 3:dfa 与 min_dfa

```
请输入字母表的所有字母:
abc
请输入正则表达式:
((ab)|c)*
加入.连接符后:
((a.b)|c)*
将中缀表达式转为后缀表达式后:
ab.c|*
NFA初态:9
NFA终态:10
状态集:1,2,3,4,5,6,7,8,9,10
[1—a—>2]
[2—~—>3]
[3—b—>4]
[4—~—>8]
[5—c—>6]
[6—~—>8]
[7—~—>1]
[7—~—>5]
[8—~—>7]
[8—~—>10]
```

```
[9—~—>7]
[9—~—>10]
DFA初态:A
DFA终态集合:ACD
状态集合:ABCD
[A—a—>B]
[A—c—>C]
[B—b—>D]
[C—a—>B]
[C—c—>C]
[D—a—>B]
[D—c—>C]
对DFA增加死状态后
[@—a—>@]
[@—b—>@]
[@—c—>@]
[A—a—>B]
[A—c—>C]
[A—b—>@]
[B—b—>D]
[B—a—>@]
[B—c—>@]
[C—a—>B]
[C—c—>C]
[C—b—>@]
[D—a—>B]
[D—c—>C]
[D—b—>@]
最小化DFA
最小化DFA的初态:A
最小化DFA的终态:A
MIN_DFA状态集合:AB
[A—a—>B]
[A—c—>A]
[B—b—>A]
```

可以发现初态、终态、状态转移完全一致，说明正确

3.4 单独测试

由于最小化 DFA 难度最大，这里仅测试最小化 DFA 的正确性。

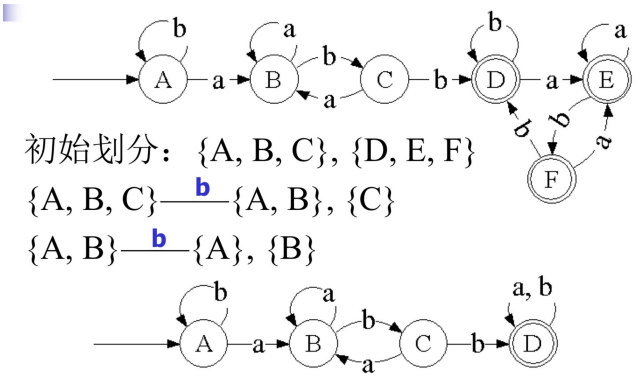


图 3.12: 最小化 dfa 测试

```
对DFA增加死状态后
[A—a—>B]
[A—b—>A]
[B—a—>B]
[B—b—>C]
[C—b—>D]
[C—a—>B]
[D—b—>D]
[D—a—>E]
[E—b—>F]
[E—a—>E]
[F—b—>D]
[F—a—>E]
最小化DFA
最小化DFA的初态:A
最小化DFA的终态:D
MIN_DFA状态集合:ABCD
[A—a—>B]
[A—b—>A]
[B—a—>B]
[B—b—>C]
[C—b—>D]
[C—a—>B]
[D—b—>D]
[D—a—>D]
```

可以发现程序输出正确。