南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

编译原理编程实验报告

OT1 实现词法分析器构造算法

马可

年级：2021 级

专业：计算机科学与技术

指导教师：王刚

摘要

关键字：Parallel

目录

一、 算法概述

本次的任务是手动实现词法分析器构造算法。我们使用的编程语言是 C++，编程环境是 VS2022。由于本次的工作量比较大，所以在本次作业中的数据结构将在具体的过程中展示，我们先详细讲述如何将一个正则表达式转换为一个可识别输入的 DFA。

首先我们先不从编程的逻辑来考虑这件事，我们从逻辑上来考虑这件事。以下面这个正则表达式为例，$(a|b)^*a|bcd$。从人的角度来说，我们看到它后其实已经根据我们默认的运算符优先级指定了运算顺序。那么很自然的我们可以根据 Thompthon 构造法，来构造 NFA。具体流程我们不在这里赘述。构造完 NFA 后，我们根据 NFA 转 DFA 的算法，用状态集代替状态，然后就能自然构造出 DFA，然后再进行 DFA 的化简，就得到了我们最终的图，就如同我们上次的练习一样。

但是如果从计算机的角度来看，我们要实现这个事好像要考虑的细节就变得特别的多。

首先从第一步来看，我们怎么让计算机知道有优先级，或者怎么让计算机像人一样去根据优先级来识别正则表达式呢。然后它又怎么在识别正则表达式这个过程中，构建 NFA。现在我们甚至不知道这是一步做的还是分成两步做的，然后我们需要的数据结构有哪些，我们应该从哪个角度设计。我们将在专题详细阐述。我们采用循序渐进式的方法来写这次的实验报告。将在把这个问题解决之后再考虑接下来的问题。

(一) 正则表达式的识别

回溯正则表达式的识别的这个过程，这其实是一个语法分析树的构造过程。为什么？其实识别正则表达式的过程，是不是一直在从左往右，或者说两边同时（这是正则表达式有个 | 的情况）构造子字符串, 直到我们遍历完这个字符串。我们再来回想一下，首先我们先识别出来这是一个合格的正则表达式。然后根据优先级来构造语法分析树，然后顺着这个优先级语义分析。

那第一个困难的事情就来了，我们怎么构造这个语法分析树呢，或者说怎么根据它构造出 NFA 呢。其实利用 lex 和 yacc 这两个工具这个事就变得尤其简单，先利用 lex 识别我们的原始输入，转化为我们规定的 token 后，指定相应的语义规则，然后 yacc 会在构造这颗树的同时来构造出 NFA。那其实这个问题就转变为了：用 lex 识别 token, 编写语义规则，NFA 的数据结构应该怎么设计。

本次实验我们决定手搓这个部分的算法。首先我们要做的是直接根据输入构造出语法分析树。输入一个正则表达式，展现出语法分析树，输入和运行结果如下图所示：

图 1: 语法分析树构造

我们为什么要构造语法分析树？看看后序遍历这颗语法分析树的顺序，是不是相当于我们人

的默认的优先级！那么每个结点代表什么呢？代表我们遍历到这后我们已经构造的子字符串。这就是我们告诉计算机怎么去看一个正则表达式的方法，构造一个语法分析树，然后让它后序遍历。那应该怎么构造呢？

本次实验输入的正则表达式规定只能有 $(,), *, |$ 这四种运算符，以及 26 个小写字母。我们先给出我们设计的数据结构，然后再以具体的例子说明我们为什么要这么设计这个数据结构。

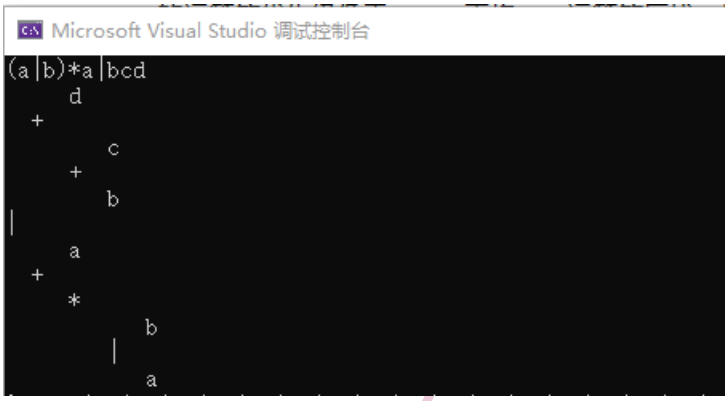

图 2: 语法分析树构造所需要的数据结构

以 $(a|b)^*a|bcd$ 为例，我们来说明计算机怎么构造出语法分析树。首先我们准备两个字符栈，一个用来存放运算符，一个用来存放字母。其实这个过程非常像后缀表达式的计算。我们站在计算机的角度和人的角度来说明这整个过程。这次的程序没设计排错功能，所以默认输入正确。具体的构造方法可以参见代码，这里仅举左半边构造的例子。

如果读入 (，说明在读入) 之前，下面的我们读到的所有东西，都应该优先处理，看成一个子字符串。那么我们就接着读取字符，首先读入字符 a，然后构造出一个结点，里面内容是 a，将其放入字符栈中。注意，字符栈对应的是我们构造的子字符串的结点。然后我们读入字符 |，然后将其放入运算符栈中。注意，真实代码相比于这个情况更复杂。比如你是不是要考虑把 | 放到运算符的栈里面的时候，里面的运算符有些啥？或者说你的 | 是不是要马上把字符栈里面的两个子字符串做运算，然后把运算对象弹出，把运算的结果放到栈中。或者我先不计算，先把运算符放到栈中，继续读入。这其实得根据符号栈与运算符栈中的具体内容而定，在此不再赘述，代码中均有声明。然后继续读入字符 b，再读入 (后将 a 与 b 做或运算后扔进字符栈。再读入 $*$ 运算后，取出符号栈中的子字符串，然后做闭包运算后扔回去。

仅以左半边为例可以看到我们生成的语法分析树成功的印证了我们的例子。这一段的数据结构相对而言比较简单，我们只设计了个可以存放一个字符的结点。如果这个结点存放的运算符，那么就代表经过这个运算符对其相应的运算对象运算后得到的子字符串。如果这个结点存放的是单个字符，实际上也代表着一个字符的子字符串。通过后序遍历这颗语法分析树就是这个正则表达式的运算顺序。

(二) NFA 的构造

上一步主要阐述怎么告诉计算机我们构造字符串的方法和顺序。这一步当然要进入我们的 NFA 的构造了！那就得引入我们的 Thompthon 构造法了。直到这一步我们更看到了上一步的好处，因为上一步不仅告诉了我们运算顺序，而且它表示运算顺序的方法，恰恰是让正则表达式里面所有的有用的字符和运算符分层而达到！Thompthon 运算法就是需要这两样东西：顺序，字符（现在的字符涵盖了运算符）。对于每一个字符，都构造图的新的两个结点和一些对应的边，然后与我们之前的图（也就是已经构造的子字符串的图）做一些个连接操作，这里具体不再赘述。

那又到了考虑数据结构的时候了。图的数据结构，非常简单，我们在这里选择邻接矩阵就行了。形式上非常常规，即是 vector<vector<vector<int»>。注意，由于 NFA 构造的原理，一个字符边或者 epsilon 边可以去往两个结点（至多也就两个了），所以有一个三层的嵌套。

```
vector<vector<vector<int>>> NFA;
```

图 3: 图的数据结构

OK！那开始构造我们的 NFA 图吧！仍然以 $(a|b)^*a|bcd$ 为例，后序遍历，先从最左下角开始。一个 a 字符，两个结点一个边添加到 NFA 中。然后 b 字符，两个结点一个边添加到 NFA 中。诶，回到 | 这个结点的时候，我们发现了问题了。好像我们用运算符做运算的时候，需要知道两个运算对象“在哪”。用人的话来说，我得知道两个子字符串是什么啊。用计算机的话来说，我得知道左边这个子图的开始结点和结束结点，右边这个子图的开始结点和结束结点啊！怎么办？设计新的遍历函数和数据结构！

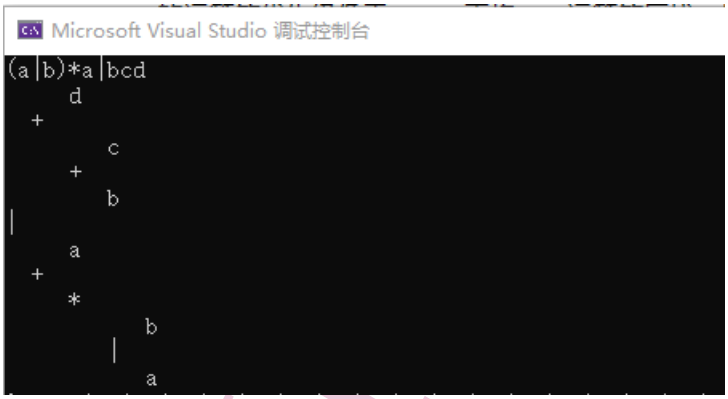

图 4: 语法分析树

我们设计新的遍历函数，在从这个结点（子字符串返回的时候），可以返回这个子字符串（子图）的开始结点和结束结点。我们设计的数据结构如下图所示

```
class NFANode {
public:
    int start;
    int end;
    NFANode* Left;
    NFANode* Right;
    NFANode() {
        Left = Right = nullptr;
        start = -1;
        end = -1;
    }
};
```

图 5: NFANode

在遍历语法树的时候，我们构造了一颗全新的树。实际上这树只有点辅助的用处，就是记录运算符的运算对象，以及子字符串的开始结点与结束结点。设计新的遍历函数，这个函数在经过原来的语法分析树结点时，会根据 Thompthon 构造法构造新的结点和边，并且和它的子字符串

做一些个运算（如果需要的话），然后返回新的子字符串的开始结点和结束节点。这样递归过后就能得到我们的 NFA 和 NFA 的开始结点和结束结点（开始结点和结束结点），NFA 的构造就完成了！

```
class NFANode {
public:
    int start;
    int end;
    NFANode* Left;
    NFANode* Right;
    NFANode() {
        Left = Right = nullptr;
        start = -1;
        end = -1;
    }
};
```

图 6: VisitParseTree

这里给一下 NFA 的构造结果，以邻接矩阵的方式存储。

```
0:   -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
1:   -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 3,15
2:   -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 0
3:   -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 5
4:   -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 13
5:   -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 7,4
6:   -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 7,4
7:   -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 9,11
8:   -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 6
9:   8 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
10:   -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 6
11:   -1 10 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
12:   -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 2
13:   12 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
14:   -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 0
15:   -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 17
16:   -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 23
17:   -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 19
18:   -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 21
19:   -1 18 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
20:   -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 16
21:   -1 -1 20 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
22:   -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 14
23:   -1 -1 -1 22 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

图 7: NFA（邻接矩阵表示形式）

经过我们手工验证，这个结果没有问题。其它的例子也经过验证。

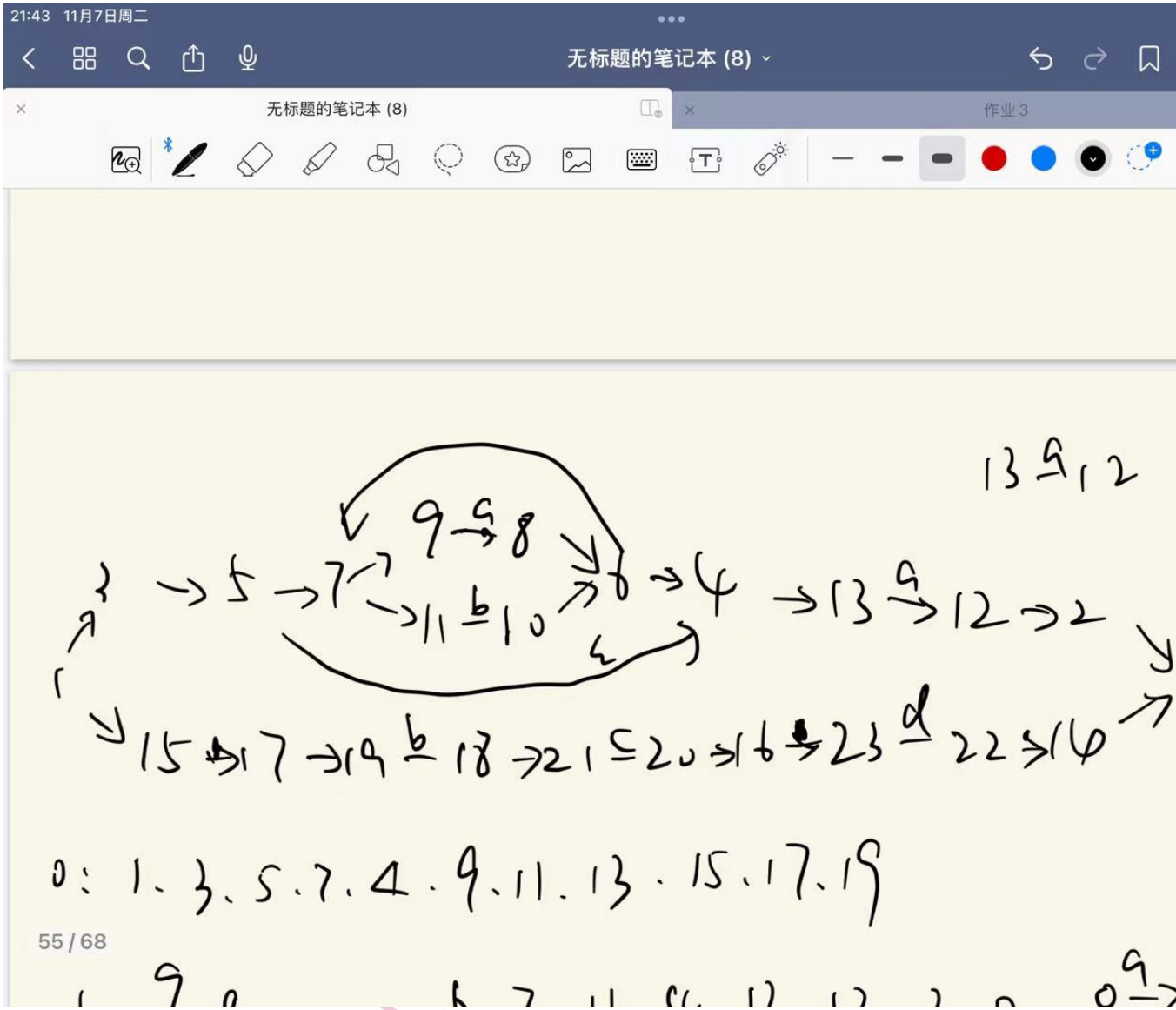

图 8: NFA（手工表示形式）

(三) NFA 转 DFA

好了我们走到了 NFA 转 DFA 这步。其实前面并没有涉及一些比较个复杂的数据结构。基本上都是一些需要思考的小点，和一些个简单的数据结构。真正需要考虑的是 NFA 转 DFA 的数据结构。NFA 转 DFA 的算法也不在这赘述了，我们只阐述用了哪些数据结构。

首先 DFA 的存储只用 vector<vector<int» 就好了，因为一条边只会有一个去向。好，那我们想想这个过程，基本也就设计出我们想要的数据结构了。

首先取开始态的 epislon 闭包，就得到了一堆状态，我们用 set<int> 存储就好了。

取到闭包后得到状态集,我得给它编个号吧,也就是要有个映射关系,那就用 map<set<int>,int> 就好了。注意一点编程上的问题，map 模板要求两个运算类型都是可比较的，所以你得自己重载一下 set<int> 的运算符。

编号完了，我是不是要记录这个 set 已经被编过号了，将来出现同样的 set 的时候我可以查找到这个编号记录。这里有很多方法，我们简单一点，就是 set<set<int» 就好了。

好，那是不是要根据可以输入的字符，去尝试所有输入，然后得到新的状态集了？那我们得

排队吧，那就用个 queue<set<int>> 就好。

```
set<set<int>> Encoded_DFANode;//记录已经编过号的状态集合
queue<set<int>> UnMatchedNode;//记录还没有用字符寻找过的状态集合
map<set<int>, int, SetComparer> NodeMap;//记录状态集合的编号
set<int> StartNodeSet = Episonal_closure(start);//开始状态集合
int DFAcount = 0;//已经有的DFA结点个数
Encoded_DFANode.insert(StartNodeSet);
NodeMap[StartNodeSet] = DFAcount++;//给开始结点赋值为0
UnMatchedNode.push(StartNodeSet);
```

图 9: NFA 转 DFA 所需的数据结构

总结下来也就这么多数据结构，编程的细节就不赘述了，代码里面注释给的比较详细，这儿给一个运行的结果。

```
1 3 4 5 7 9 11 13 15 17 19
0通过a走1:0 2 4 6 7 8 9 11 12 13
0通过b走2:4 6 7 9 10 11 13 18 21
0 2 4 6 7 8 9 11 12 13
1通过a走1:0 2 4 6 7 8 9 11 12 13
1通过b走3:4 6 7 9 10 11 13
4 6 7 9 10 11 13 18 21
2通过a走1:0 2 4 6 7 8 9 11 12 13
2通过b走3:4 6 7 9 10 11 13
2通过c走4:16 20 23
4 6 7 9 10 11 13
3通过a走1:0 2 4 6 7 8 9 11 12 13
3通过b走3:4 6 7 9 10 11 13
16 20 23
4通过d走5:0 14 22
0 14 22


1 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
1 3 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
1 3 4 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
1 3 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 5 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

图 10: DFA 运行结果

(四) DFA 的最小化

实际上走到这一步了，还是一个数据结构的事儿。算法上很简单就实现了。仍然不赘述算法。我们谈一谈数据结构和一些需要具体考虑的细节。

首先也是记录状态集的集合，然后把里面的状态集的结点拿出来遍历。如果有一个结点去的地方和其它结点不在一个状态集，那就把它踢出去，然后去找它应该归属的状态集（如果找不到就给它新创一个）。

最后我们得到一堆状态集，然后根据原来的 DFA 创建一个新的 DFA 就行了，没有比较新奇的数据结构。在这儿给个运行结果。运行的是课上的例子。

```
1 3 4 5 7 9 11 13 15 17 19
0通过a走1:0 2 4 6 7 8 9 11 12 13
0通过b走2:4 6 7 9 10 11 13 18 21
0 2 4 6 7 8 9 11 12 13
1通过a走1:0 2 4 6 7 8 9 11 12 13
1通过b走3:4 6 7 9 10 11 13
4 6 7 9 10 11 13 18 21
2通过a走1:0 2 4 6 7 8 9 11 12 13
2通过b走3:4 6 7 9 10 11 13
2通过c走4:16 20 23
4 6 7 9 10 11 13
3通过a走1:0 2 4 6 7 8 9 11 12 13
3通过b走3:4 6 7 9 10 11 13
16 20 23
4通过d走5:0 14 22
0 14 22


1 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
1 3 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
1 3 4 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
1 3 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 5 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

图 11: DFA 最小化运行结果

二、 总结

这次的正则表达式的识别，手动构造词法分析器，收获还是挺大的吧。自己也没想到，课上看似轻描淡写的东西，实际实现的过程中需要这么多的细节和转化。总的来说收获很多，代码能

力有提升，这次作业完成的非常有意义，谢谢王老师给出这么好的作业起到锻炼机会。希望以后也有这种作业。

参考文献