# 关于正则表达式转NFA转DFA的课设代码分析

姓名：何炜

学号：2015014278

班级：计科1501

1. 编程环境

语言： c++ 11

IDE：qt creator、 CLion

界面库：qt、GraphViz

开发平台：mac，Windows下未做测试

编译该项目，必须本地已经下载安装GraphViz,并且需要修改lex.h中的dot常量为你的系统中dot命令的位置。如果是mac系统，你可以通过 which dot 查看具体位置。

const string dot = "/usr/local/bin/dot";

## 数据结构分析

* 正则表达式由用户输入，使用string类型简单存储，正则表达式分为两个部分：运算符部分和标识符部分
* NFA和DFA类的成员变量：
  + MyGraph：因为NFA,DFA每个节点有多个出度，多个入度，用**图**表示最合适的存储方式，定义了图的数据结构用来存储整个数据
  + StartStatus：存储开始状态序号，相当于头部指针
  + EndStatus：存储结束状态序号，相当于尾部指针
  + mVexs：对应的NFA或者DFA的节点集合，NFA集合是一维数组，DFA的节点集合是二维集合，每个节点的信息为对于的NFA转换的状态集合
* MyGraph：图的具体实现
  + mMatrix: 使用一个二维整形数组存储了节点的边之间的关系，值为一个char型的vector，这里是考虑到两个节点之间可能有多条同向边。
  + mVexNum:节点的数量，辅助变量，便于后面的分析。
  + mEdgeNum:边的数量，辅助变量，便于后面的分析。

## 正则表达式转NFA

在介绍我使用的算法之前，先介绍一下正则表达式的基本概念：

比如：(a|b)\*abb 就是一个很简单的表达式

其中分为两个部分：

* 标识符部分：在我们算法里就是普通的字母
* 运算符部分：运算符有三个
  + \*：重复符号，在正则表达式中运算级最高。
  + |：选择符号
  + &：连接符号（课本上的是·，但是着重符号不是英文符号，所以使用该符号代替）
  + ()：左右括号，右括号是运算符中优先级最高的

龙书上面给了一个McMaughton-Yamada-Thompson算法，但是给的很简略：

1. 对正则表达式的每个字符进行分析，如果是单个表示

这些步骤可以直接手写出NFA，但是如果编程的话，少了一个重点：**判断运算符的优先级，** 举个例子：a|b\*，我们人可以直接看出a 、 b、 b\*、a|b\* 的顺序，但是机器如何识别\*符号的优先级比较高，否则也会产生这样的顺序：a、b、a|b、(a|b)\*。

这个算法和四则运算的算法有点类似，类似之处在于判断运算符的优先级，区别是四则运算是普通的数字或者公式计算，但是该算法中的**运算符的结果是NFA图的连接变化（增加边或节点，或者边之间的连接关系改变），而且该算法的运算符多了一个一目运算符：\*符号。**

和四则运算类似，我们这里需要增加两个栈：

* **运算符栈：**该栈中存储的是上面介绍的基本运算符
* **NFA栈：最为重点的就是这栈，这个栈的作用类似于四则运算中的运算结果（运算数）的栈。**栈中存储的是什么数据呢？需要是图的结构吗？不需要，我们先来看下为这个栈里面会放什么内容呢？放的内容有两个：
  + 零散的NFA，比如a 的NFA就是很简单的两个节点，一个起始点一个终止状态点。
  + NFA的运算结果
    - * 形似b\*，因为一目运算符是需要计算出NFA，然后把计算结果压入栈中，而一目运算符不必压入运算符栈中
      * 形似(a|b)，这种情况也需要计算出括号中的NFA，然后把计算结果压入栈中，这个过程中，运算符栈中的一些符号和NFA栈中的一些元素还需要出栈

当NFA栈中元素会发生运算的时候，**最终的NFA的图结构**的边和点信息都会同步更新的（因为运算会涉及到边和节点的增加），**也就是说最终的NFA中是包含这些NFA中发生的运算的结果的**。栈中没有发生运算的零散的NFA，一共只有两个节点。**所有这里只需要存储栈中的NFA的起点和终点序号，而无需给栈中的NFA也建立图的结构。类比四则运算的结果，NFA栈对应着操作数栈。**

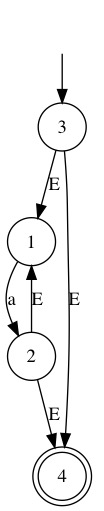
结合龙书上给个简略算法，和我们增加的两个栈（运算符栈和NFA栈）的数据结构，有以下的算法过程：

我们对输入的正则表达式如(a|b)\*abb 的每个字符进行判断：

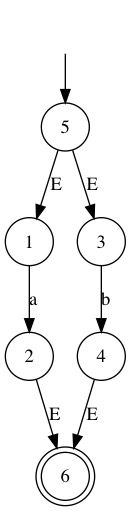
* **如果是非运算符**：也就是字母，我们以该字母为转换条件新建一个两节点一条边的基本NFA，并压入栈中。然后检查下一个符号，如果是左括号或者仍然是字母，并向运算符栈中压入一个连接符（&）
* **如果是运算符**：
  + **左括号**：直接向运算符栈中压入
  + **右括号**：此时需要从运算符栈顶开始出栈，一直到栈顶元素为左括号为止，没出栈一个符号，就同时把NFA栈中出两个操作数进行运算，然后把运算结果入NFA栈。最后也还需要检查下一个符号，如果是左括号或者是字母，就需要向运算符栈中压入一个连接符(&)
  + **连接符**：这个不需要任何操作和判断，因为程序中的正则表达式都是省略这个的
  + **选择符**：该符号是优先级第三，遇到该符号，首先需要看栈顶元素，如果栈顶元素是连接符，就把连接符出栈，把NFA栈出栈两个元素，计算后再把结果入栈，直到遇到了左括号或者选择符就会停止循环遍历。
  + **重复符**：重复符号是一目运算符，所有一旦遇到重复符号就是将NFA栈出栈一个元素进行计算，再讲计算结果入栈。最后也需要检查下一个字符，如果下一个字符是字母或者左括号就向运算符栈压入一个连接符。

经过这样的一轮扫描之后，NFA栈和运算符栈中可能并不为空，运算符中可能还有选择符和连接符，接下来我们需要对这些符号和NFA进行最后的运算：

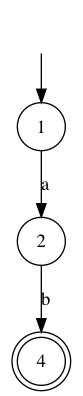
* 重复符运算：比如a\*,创建基本的NFA是两个节点一条边，1，2节点，边的值为a，然后再生生成两个节点，按照下图中的示例操作：



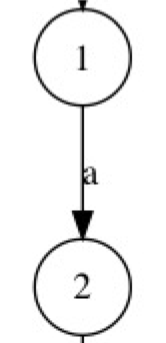
* 选择符运算：比如a|b



* 连接符运算：



* 创建基本的NFA操作：



## NFA转DFA

直接根据课本上面的**子集构造法**即可，重点是要在代码中完成以下两个函数：

* E\_closure(T)：T为NFA状态集合，找到T集合的每个E转换的集合。这里需要递归，我的做法是新建一个递归栈S，R，S初值是T集合的值，R栈为空。每个状态通过E转换获得的状态加入到递归栈S、R 中，并把该状态从S中出栈。这样直到递归栈为空结束循环。最后返回结果栈R。
* Move(T,a)：T为NFA状态集合，a为转换条件。初始化一个结果栈R，初始化一个栈S，S初始值和T集合值相同。每次进行一次a转换，就将该状态出栈，并把转换结果压入R栈中。直到S为空，结束循环。最后返回结果栈R。

在整体的getDFA()函数中，（DFA状态有两个信息，一个是节点序号，一个是对于的NFA集合）。

1. 我们设置一个DFA状态栈S，当这个栈为空的时候，结束循环。
2. 首先求出NFA初始状态的e\_closure的E转换集合，记为DFA的初始状态A。并把该状态加入到DFA状态栈S中。并把该状态加入到DFA的节点集合中。
3. 从初始状态开始，循环字母表，字母表的每个元素都是转换条件。
   1. 对每个转换条件，执行e\_closure(move(<当前DFA状态>,<当前字母表>))，会得到一个新生成的NFA状态集合
   2. 检查该NFA状态集合在已有的DFA状态集合中是否已经存在。
   3. 如果不存在，就把该新生成的DFA状态加入到S栈中，把该状态加入到DFA的节点集合中，并把原来的DFA状态出栈。
   4. 并根据转换条件，给DFA的图中把对应的两个DFA节点连接起来。
4. 直至DFA状态栈S为空，停止循环。这样就构建成了DFA图了。

## 最小化DFA

根据课本上面的算法可以直接写出代码：

1. 定义一个划分集合dividedArrays，集合中初始有两个元素，第一个元素是非终止状态集合，第二个元素是终止状态集合。向划分集合中加入元素的时候，每个元素都会被标记为false，表示可划分。
2. 定义一个循环变量 flag。默认为true，当值为false时候结束循环。当且仅当划分集合的每个元素都不可划分，即每个集合都被标记为true，此时退出大循环。
3. 对该划分集合的每个元素遍历：
   1. 每个元素也是一个状态集合S，对该集合S的每个元素进行字母表的转换循环。
      1. 通过获取S集合中的每个元素经过某个字母转换的结果状态属于划分集合的序号，最后根据属于划分集合的序号将该状态集合进行划分。
   2. 如果某个状态集合S的所有元素经过所有字母转换的结果都属于划分集合的同一个元素，说明该集合不可划分，将该集合标记为true。
4. 最后根据划分集合的每个元素是否可继续划分，如果都已经被标记为true，退出循环
5. 最后，循环扫描划分集合的每个元素，如果某个元素的大小大于1，则将该元素内的状态都合并成同一个状态。具体做法就是将需要被合并的节点的所有边的状态都移到目标节点中。

## 图形化界面设计



* **主体界面的绘制使用Qt**

用户输入正确的表达式，点击确定按钮，再点击左侧工具栏的相应按钮，即可显示相应的节点图片。

* **节点图片的绘制使用GraphViz绘图工具**

执行的命令比如：

/usr/local/bin/dot -Tjpg ../../../../lexical/dots/dfa.dot -o ../../../../lexical/images/dfa.jpg

即可生成对应的dfa图片。

这里关键的文件是dfa.dot，是程序根据DFA图的数据结构生成的符合特定格式的文本文件，比如：

digraph G{

""[shape=none]

"1"[shape=circle]

"2"[shape=doublecircle]

"3"[shape=doublecircle]

"4"[shape=doublecircle]

"5"[shape=doublecircle]

"6"[shape=doublecircle]

"7"[shape=doublecircle]

""->"1"

"1" -> "2"[label="1"]

"1" -> "3"[label="2"]

"1" -> "4"[label="3"]

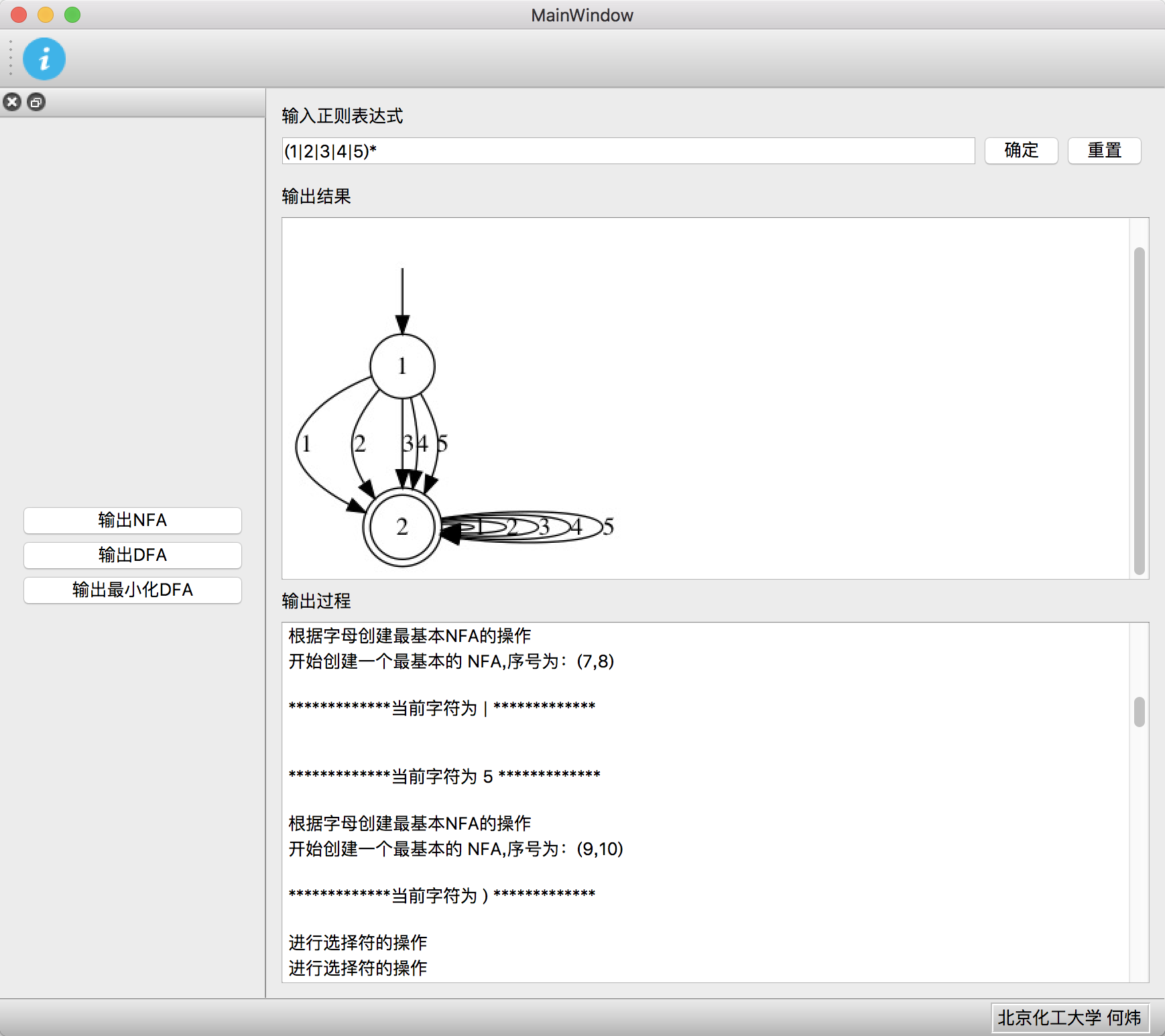
"1" -> "5"[label="4"]

"1" -> "6"[label="5"]

"1" -> "7"[label="6"]

}

**测试数据：(1|2|3|4|5)\***

****

## 七、 可以改进的地方

由于时间问题，以下内容作为待改进部分：

* **对输入的正则表达式有一个判断，可以判断出错误的正则表达式**：这一步主要在运算符和NFA压栈的过程中，发现符号不匹配或者异常字符，程序会提示错误，反正异常退出程序
* **动态化显示NFA 、DFA生成过程**：由于使用的是GraphViz绘图工具，生成的是图片，目前想到的办法是，程序中每生成一个节点或者每生成一条边，都会及时的重写相应的dot文件，与此同时执行绘图命令，生成对于的图片显示在界面中。在显示的过程中，使用sleep(2000)以便能够观察到生成的过程。
* **对相对路径的处理上优化**：现在程序中的相对路径是相对于编译目录的，如果打包成app或者exe应该是无法调用路径的文件的。因为对qt的资源文件不熟悉，所以暂时没有优化。