# 编译原理-自动机

## 一、实验内容

利用状态表和有限自动机的运行原理编写和设计程序，判断输入的自动机是DFA还是NFA，如果是NFA，利用子集法将其确定化，然后利用求同法或求异法将所得的DFA最小化。

## 二、实验目的

1. 理解有限自动机的作用，进一步理解有限自动机理论

2. 设计有限自动机的表示方式，采用合理的数据结构表示自动机的五个组成部分

3.掌握ε闭包的求法和子集构造算法，以程序实现NFA到DFA的转换，并且最小化DFA，提高算法的理解和实现能力

## 三、具体实现

### 1.数据结构

首先我们需要考虑存储DFA与NFA的数据结构，两种状态机都需要存储转移规则（使用键值对），所有接受的输入字符(使用集合),开始状态与接收状态，因此NFA数据结构如下:

class NFA:

def \_\_init\_\_(self, state\_automaton: dict, alphabet: set, start\_status: set, terminal\_status: set):

self.state\_automaton = state\_automaton

self.start\_status = start\_status

self.terminal\_status = terminal\_status

self.alphabet = alphabet

DFA的数据结构与此类似。

### 2.求解epsilon\_closure(#闭包)

#闭包也就是在不输入任何符号的情况下，自动机所能到达的所有状态。

**算法如下:**

将开始状态压入队列

当队列非空时:

弹出队头元素

如果队头元素不在闭包中:

将之加入

如果这个状态在NFA中 且 其接受#：

将这个状态输入#转移到的状态全部压入队列

这是一个广度优先的算法，用递归的形式可以改写为深度优先的算法。

通过这个算法我们可以计算出NFA任意一个状态不输入任何东西可以到达的所有状态，这个函数被用于NFA的确定化中。

### 3.NFA的确定话

使用子集构造算法，伪码来自中科大的课件

将NFA开始状态的闭包压入队列并加入获得的DFA状态Q中

队列非空的时候：

取出队头元素

对于当前状态每一个可能的输入:

分别计算所有可能到达的状态的#闭包，并将之存放在t中

将其存放在二级字典中，即以当前的状态(一个字典)为键，再以输入c为键，以获得的t为值构造状态机

如果这是一个新的状态:

将这个状态t加入DFA的状态集合Q

我当时在实现的时候使用了以字典的字典来存储新的DFA，因为字典是不可哈希的(因为他是可变的)，所以我使用了frozenset，使之变为不可变的对象；同时，如果不想两次访问键值对

(如dfa\_transitions[frozenset(curr\_status)][one\_input] = frozentmpset)可以使用元组来达到相同的功能。

q0 <- eps\_closure(n0)

Q <- {q0}

workList <- q0

while(workList != [])

remove q from workList

foreach(character c)

t <- e-closure(delta(q,c))

D[q,c] <- t

if(t not in Q)

add t to Q and workList

到了这一步我们完成了最后还需要转移规则的获得，至于剩下的部分:

DFA的开始状态=NFA开始状态的闭包

dfa\_alphabet = nfa.alphabet

DFA的终结状态={DFA的状态 for DFA的状态 in ALL\_DFA

if NFA接收状态 in DFA的状态}

### 4.DFA的最小化

DFA的最小化我打算使用Moore算法，伪码如下

Moore DFA最小化

将DFA划分为接收状态与非接受状态

while set is still changing :

for s in set:

split(s)

def split():

for i in alphabet:

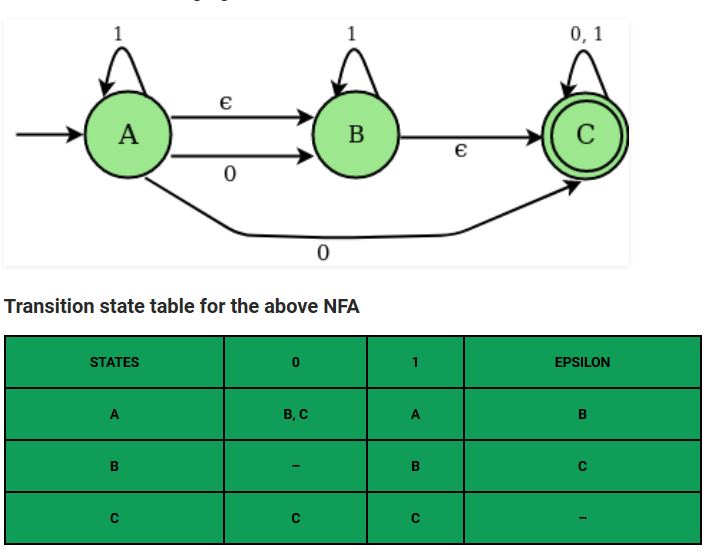
if i can split s:

split s => T1,T2,...Tk

但是最后没有想出合适的数据结构来实现这个算法。

## 测试

测试的NFA是我在<https://www.geeksforgeeks.org/>上找到的一个比较简单的例子，状态机如下图所示:



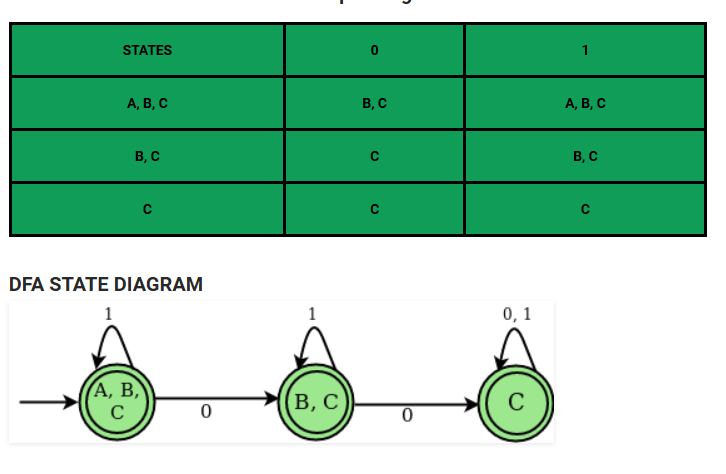
对应的闭包如下

**∈ closure(A) :** {A, B, C}

**∈ closure(B) :** {B, C}

**∈ closure(C) :** {C}

转换为DFA的结果如下



我们的测试代码如下图所示:

if \_\_name\_\_ == '\_\_main\_\_':

delta = {'A': {'0': {'C', 'B'}, '1': {'A'}, 'ε': {'B'}}, 'B': {'1': {'B'}, 'ε': {'C'}},

'C': {'0': 'C', '1': 'C'}} # type:dict

nfa = NFA(state\_automaton=delta, start\_status={'A'}, terminal\_status={'C'}, alphabet={'0', '1'}, )

work\_list(nfa=nfa, start\_status='A')

输出结果:

所有的状态

{frozenset({'C', 'B'}), frozenset({'C'}), frozenset({'C', 'B', 'A'})}

DFA状态转移字典

{frozenset({'C', 'B', 'A'}): {'1': frozenset({'C', 'B', 'A'}), '0': frozenset({'C', 'B'})}, frozenset({'C', 'B'}): {'1': frozenset({'C', 'B'}), '0': frozenset({'C'})}, frozenset({'C'}): {'1': frozenset({'C'}), '0': frozenset({'C'})}}

终结状态

{frozenset({'C'}), frozenset({'B', 'C', 'A'}), frozenset({'B', 'C'})}

输入(alphabet)

{'0', '1'}

与上图参照可以看出结果是正确的。

## 总结

这个实验的主要难点一是自动机的最小化、确定话算法的理解；第二是合适的数据结构的设计。

对于一些不足，一是没能实现最小化，想了不少的写法，落实下来都有问题；另一方面，后期可以再用一个字典映射将frozenset({'C', 'B', 'A'})这样的形式改为更简单的如’A’,现在的这种形式有利于我验证自己的代码是否正确，二将状态名简化有助于降低使用者的困惑与记忆负担。