# NFA确定化实验报告

目录：



# 一、实验目的

1. 熟悉课本中关于NFA确定化的算法思路，能够通过纸笔演算推算出DFA的状态集合和转换路径。
2. 将课本中的思想通过代码实现，进一步掌握算法流程。
3. 实验将尽可能完善，包括考虑空弧 € ，并且以书上例 3.4 和 3.5 为验证。

# 二、算法分析

## 1. 算法思路

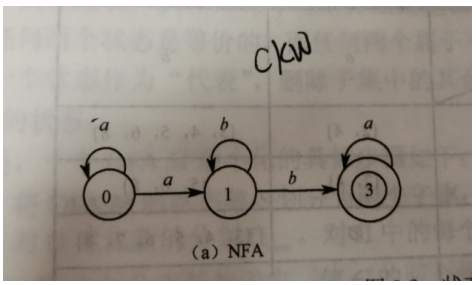
1.1 寻找初始状态集合 I，并且考虑空弧 € 的加入， 计 S0' 为 € - Closure(I)

1.2 将S0' 通过所有可能的状态进行转换， 生成新的状态， 查看新的状态是否已经存在，若存在进入下一步，否则嵌套循环1.2

1.3 将确定的新状态标记输出。

# 2. 代码详细解释

## 1. NFA状态文件




由于于程序需要将上图所示的状态图读入，因此需要固定格式的文本，所以我将文件格式定义为如下的样式，第一行为 q, 而 q ∈ € - Closure(I), 第二行至末尾，将遵守形如 " 0 a 0" 或者 " 1 ~ 2" ,表示 "0 通过输入 a , 进入状态 0" 或者" 1通过空弧 € 进入状态 2" ， 程序中用 " ~ " 代替空弧表示。

```
// 例 3.5 的输入文件格式 文件" states.in "
1
1~2
1~3
2a2
2~4
3b3
3~4
4b5
5~6
6b7
7a6
6~8
```

## 2. 程序中总体方法和变量

```
const int N = 1010;
string cfn[N]; //状态映射集合 同states.in
vector<char> inputs; // 输入集合
vector<string> newStates;

```

```
int Len=0; //状态映射集合长度
char E = '~';//空弧

queue<char> findInitState();
 queue<char> findNextState(char cur, char func);
 queue<char> nextStatesOneIpt(queue<char> curS, char func);
 string getIdentifyCode(queue<char> curS);
 void solve();
 bool isNewStateExist(string s);
 queue<char> findNextStateWithE(char cur, char func); // 包含空弧
```

程序中定义的 ***cfn*** 为当前状态通过输入走入下一个状态的字符串集合， 如文件中的 " ***6b7*** " ，而 ***Len*** 则表示NFA文件中有多少个映射转换； ***inputs*** 为 输入， 也就是 a, b 等的输入集合；***newStates*** 为新生成的状态集合， 通过字符串表示， 如 **{0, 1}** 表示为 **01；**

# 3. 方法具体介绍

## 3.1 "findNextState"

在文件读取中已经获得了状态和输入的映射关系，此方法中通过遍历循环，匹配满足条件的，因为一个状态和一个输入会映射至多个下一个状态，所以以队列的形式返回下一个状态；

```
queue<char> findNextState(char cur, char func) {
 queue<char> n;
 for(int i=0;i<Len;++i) {
 if(cfn[i][0]==cur && cfn[i][1]==func) {n.push(cfn[i][2]);}
 }
 return n;
}
```

## 3.1 ' "findNextStateWithE"

很快能够发现方法 "findNextState"不包含空弧的判定方法，因此在findNextStateWithE中加入了空弧的判定方法，即 通过先获得非空弧经过的状态置为初态集合，然后多次以空弧为判定方法调用findNextState，并以BFS的算法为基础去寻找所有空弧到达的状态。

```
queue<char> findNextStateWithE(char cur, char func){
 queue<char> q = findNextState(cur, func);
 queue<char> I;
 string s0 = "";
```

```
while(!q.empty()) {
char c = q.front();
q.pop();
if(s0.find(c)==-1) { //去重
I.push(c);
 s0 += c;
 }
 queue<char> ns = findNextState(c, E);
 while(!ns.empty()) {
 char c2 = ns.front();
 ns.pop();
 q.push(c2);
 }
 }
 return I;
}
```

### 3.2 "findInitState "

该方法是用来通过读取最开始的q集合，并且考虑空弧并且生成 € - Closure(I)的I；主要思路为，定义队列**Q0**，将最开始的 **q**集合放入至**Q0**中，并且以**q**集合中的元素 **qi** 为开始，通过方法 **findNextState** 寻找下一个状态，判断是否已经在**Q0**中以决定是否放入队列**Q0**， 并且重新放入**q**集合中再此循环空弧为条件找下一个状态，直至**q**集合， 则返回队列**Q0**；

```
queue<char> findInitState() {
 freopen(file,"r",stdin);
 queue<char> q;

 string init="";
 cin>>init;

 int i=0;
 //输入状态映射关系
 while(cin>>cfn[i++]){
 char ipt = cfn[i-1][1];
 bool flag = false;
 for(int i=0;i<inputs.size();++i) {
 if(ipt==inputs[i]) {flag = true;break;}
 }
 if(!flag && ipt!=E) {inputs.push_back(ipt);}
```

```
 }
 Len = i-1;
 for(int i=0;i<init.length();++i) {q.push(init[i]);}
 // 从q 经过任意条空弧到达的状态 q'
 queue<char> I;
 string s0 = "";
 while(!q.empty()) {
 char c = q.front();
 q.pop();
 if(s0.find(c)==-1) { //去重
 I.push(c);
 s0 += c;
// cout<<"i': " <<c<<endl;
 }
 queue<char> ns = findNextState(c, E);
 while(!ns.empty()) {
 char c2 = ns.front();
 ns.pop();
 q.push(c2);
 }
 }
 return I;
}

```

### 3.3 "nextStatesOneIpt"

该方法是给定一个过程性的集合并且根据输入产生新的状态集合， 如 例 3.4 中 **{0 , 1}** 通过输入**1**, 产生新的状态 **{1, 3}** ， 而方法需要考虑空弧的因素，如 例 3.5中 **{4}** 通过输入 **b** 到达的状态集合为 **{ 5, 6, 8}**， 如果不考虑空弧，则只到达状态 **{5}** ；

```
queue<char> nextStatesOneIpt(queue<char> curS, char func) {
 queue<char> res;
 string s0 = "";
 while(!curS.empty()){
 char c = curS.front();
 curS.pop();
 queue<char> ns = findNextState(c, func);
 while(!ns.empty()) {
 char c2 = ns.front();
 ns.pop();
```

```
  if(s0.find(c2)==-1) {s0 += c2;res.push(c2);}
  }
  }
  return res;
}
```

### 3.4 " getIdentifyCode"

由于表中如果多次出现 状态 **{0, 1}**， 只会进行一次将 **{0，1}**作为新的状态， 而 **{0, 1}** 和 **{1,0}**本质上是同一个状态，因此需要通过为每个集合生成一个码值，作为唯一的标记，而此处的方法是将放入数组中非递减排序，然后生成一个字符串如 **{2,3}** 生成 **"23"**，并为后序放入***newStates*** 并做匹配提供依据。

```
string getIdentifyCode(queue<char> curS) {
 int all[N];
 int i=0;
 while(!curS.empty()) {
 char c = curS.front();
 curS.pop();
 all[i++] = c-'0';
 }
 sort(all, all + i);
  string s0 = "";
  for(int j=0;j<i;++j) {
  char c1 = '0' + all[j];
  s0 += c1;
  }
  return s0;
}
```

### 3.5 "isNewStateExist"

此方法就是用来检查给定字符串 s， 是否已经出现在 ***newStates*** 中。

```
bool isNewStateExist(string s) {
 for(int i=0;i<newStates.size();++i) {
 if(s==newStates[i]) {return true;}
 }
 return false;
}
```

### 3.6 "solve"

此方法为总体的方法，通过多次BFS算法，不断获取新状态并且放入至队列中进行迭代，直至队列为空。

```
void solve() {
 queue<char> I = findInitState();
 queue<queue<char> > allStates;
 allStates.push(I);
 int sizIpt = inputs.size();
 while(!allStates.empty()) {
 queue<char> q = allStates.front();
 allStates.pop();
 string id0 = getIdentifyCode(q);
  for(int i=0;i<sizIpt;++i) {
  queue<char> ns = nextStatesOneIpt(q, inputs[i]);
  if(ns.empty()) {
  continue;
  }
  string id = getIdentifyCode(ns);
  if(!isNewStateExist(id)) {
  newStates.push_back(id);
  allStates.push(ns);
  }
  cout<<"state: " << id0 << " -> " << id
  << " by " << inputs[i] << endl;
  }
  }
}
```

## 4. 实验结果

由于算法中对于空的队列直接跳过，因此对于表格中的空集不输出。

### 4.1 例3.4

输入文件：

```
0
0a0
0a1
1b1
1b3
3a3
```

结果：

```
state: 0 -> 01 by a
state: 01 -> 01 by a
state: 01 -> 13 by b
state: 13 -> 3 by a
state: 13 -> 13 by b
state: 3 -> 3 by a
```

```
新的所有状态为:
0
01
13
3
```

### 4.2 例3.5

输入文件：

```
1
1~2
1~3
2a2
2~4
3b3
3~4
4b5
5~6
 6b7
 7a6
 6~8
```

结果：

```
state: 1234 -> 24 by a
state: 1234 -> 34568 by b
state: 24 -> 24 by a
state: 24 -> 568 by b
state: 34568 -> 345678 by b
state: 568 -> 7 by b
state: 345678 -> 68 by a
state: 345678 -> 345678 by b
state: 7 -> 68 by a
```

```
state: 68 -> 7 by b
```

```
新的所有状态为:
1234
24
34568
568
345678
7
68
```

## 三、实验感想和改进

1.感想

通过此次实验，加深了对NFA确定化为DFA的过程，并且更加掌握了对于空弧存在的重要性的了解，并且在方法 "findNextStateWithE" 和 "findNextState"中利用BFS算法进行状态寻找。

2.改进

该算法在匹配状态和输入映射中大量使用了遍历，算法复杂度偏高，而且在输入格式上略微偏向于机械化的算法竞赛输入格式，需要之后进一步在算法复杂度上进行优化。