

NFA确定化实验报告

名字：蔡铠蔚

学号：19180435

目录：

1. 实验目的
2. 算法分析
3. 实验感想和改进

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 ϵ ，并且以书上例 3.4 和 3.5 为验证。

二、算法分析

1. 算法思路

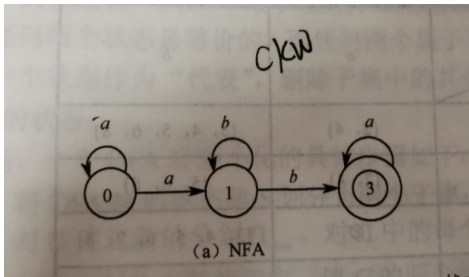
1.1 寻找初始状态集合 I , 并且考虑空弧 ϵ 的加入, 计 S_0' 为 ϵ -Closure(I)

1.2 将 S_0' 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 $q \in \epsilon$ -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 ϵ 进入状态 2", 程序中用 "~" 代替空弧表示。

```
1 // 例 3.5 的输入文件格式 文件 " states.in "
2 1
3 1~2
4 1~3
5 2a2
6 2~4
7 3b3
8 3~4
9 4b5
10 5~6
11 6b7
12 7a6
13 6~8
```

2. 程序中总体方法和变量

```
1  const int N = 1010;
2  string cfn[N]; //状态映射集合 同states.in
3  vector<char> inputs; // 输入集合
4  vector<string> newStates;
5
6  int Len=0; //状态映射集合长度
7  char E = '~';//空弧
8
9  queue<char> findInitState();
10 queue<char> findNextState(char cur, char func);
11 queue<char> nextStatesOneIpt(queue<char> curS, char func);
12 string getIdentifyCode(queue<char> curS);
13 void solve();
14 bool isNewStateExist(string s);
15 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；

```
1  queue<char> findNextState(char cur, char func) {
2      queue<char> n;
3      for(int i=0;i<Len;++i) {
4          if(cfn[i][0]==cur && cfn[i][1]==func) {n.push(cfn[i][2]);}
5      }
6      return n;
7  }
```

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。

```
1 queue<char> findNextStateWithE(char cur, char func){
2   queue<char> q = findNextState(cur, func);
3   queue<char> I;
4   string s0 = "";
5   while(!q.empty()) {
6     char c = q.front();
7     q.pop();
8     if(s0.find(c)==-1) { //去重
9       I.push(c);
10      s0 += c;
11    }
12    queue<char> ns = findNextState(c, E);
13    while(!ns.empty()) {
14      char c2 = ns.front();
15      ns.pop();
16      q.push(c2);
17    }
18  }
19  return I;
20 }
```

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 ϵ - 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；

```
1 queue<char> findInitState() {
2   freopen(file,"r",stdin);
3   queue<char> q;
4
5   string init="";
6   cin>>init;
7
8   int i=0;
```

```

9 //输入状态映射关系
10 while(cin>>cfn[i++] ){
11     char ipt = cfn[i-1][1];
12     bool flag = false;
13     for(int i=0;i<inputs.size();++i) {
14         if(ipt==inputs[i]) {flag = true;break;}
15     }
16     if(!flag && ipt!=E) {inputs.push_back(ipt);}
17 }
18 Len = i-1;
19 for(int i=0;i<init.length();++i) {q.push(init[i]);}
20 // 从q 经过任意条空弧到达的状态 q'
21 queue<char> I;
22 string s0 = "";
23 while(!q.empty()) {
24     char c = q.front();
25     q.pop();
26     if(s0.find(c)==-1) { //去重
27         I.push(c);
28         s0 += c;
29     }
30 }
31 queue<char> ns = findNextState(c, E);
32 while(!ns.empty()) {
33     char c2 = ns.front();
34     ns.pop();
35     q.push(c2);
36 }
37 }
38 return I;
39 }
40

```

3.3 "nextStatesOneIpt"

该方法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 $\{5\}$ ；

```

1 queue<char> nextStatesOneIpt(queue<char> curS, char func) {
2     queue<char> res;

```

```

3  string s0 = "";
4  while(!curS.empty()){
5      char c = curS.front();
6      curS.pop();
7      queue<char> ns = findNextState(c, func);
8      while(!ns.empty()) {
9          char c2 = ns.front();
10         ns.pop();
11         if(s0.find(c2)==-1) {s0 += c2;res.push(c2);}
12     }
13 }
14 return res;
15 }

```

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。

```

1  string getIdentifyCode(queue<char> curS) {
2      int all[N];
3      int i=0;
4      while(!curS.empty()) {
5          char c = curS.front();
6          curS.pop();
7          all[i++] = c-'0';
8      }
9      sort(all, all + i);
10     string s0 = "";
11     for(int j=0;j<i;++j) {
12         char c1 = '0' + all[j];
13         s0 += c1;
14     }
15     return s0;
16 }

```

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。

```

1  bool isNewStateExist(string s) {

```

```

2  for(int i=0;i<newStates.size();++i) {
3  if(s==newStates[i]) {return true;}
4  }
5  return false;
6  }

```

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。

```

1  void solve() {
2  queue<char> I = findInitState();
3  queue<queue<char> > allStates;
4  allStates.push(I);
5  int sizIpt = inputs.size();
6  while(!allStates.empty()) {
7  queue<char> q = allStates.front();
8  allStates.pop();
9  string id0 = getIdentifyCode(q);
10 for(int i=0;i<sizIpt;++i) {
11 queue<char> ns = nextStatesOneIpt(q, inputs[i]);
12 if(ns.empty()) {
13 continue;
14 }
15 string id = getIdentifyCode(ns);
16 if(!isNewStateExist(id)) {
17 newStates.push_back(id);
18 allStates.push(ns);
19 }
20 cout<<"state: " << id0 << " -> " << id
21 << " by " << inputs[i] << endl;
22 }
23 }
24 }

```

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：

```
1 0
2 0a0
3 0a1
4 1b1
5 1b3
6 3a3
```

结果:

```
1 state: 0 -> 01 by a
2 state: 01 -> 01 by a
3 state: 01 -> 13 by b
4 state: 13 -> 3 by a
5 state: 13 -> 13 by b
6 state: 3 -> 3 by a
```

```
1 新的所有状态为:
2 0
3 01
4 13
5 3
```

4.2 例3.4

输入文件:

```
1 1
2 1~2
3 1~3
4 2a2
5 2~4
6 3b3
7 3~4
8 4b5
9 5~6
10 6b7
11 7a6
12 6~8
```

结果:

```
1 state: 1234 -> 24 by a
```



```
2 state: 1234 -> 34568 by b
3 state: 24 -> 24 by a
4 state: 24 -> 568 by b
5 state: 34568 -> 345678 by b
6 state: 568 -> 7 by b
7 state: 345678 -> 68 by a
8 state: 345678 -> 345678 by b
9 state: 7 -> 68 by a
10 state: 68 -> 7 by b
```

1 新的所有状态为：

```
2 1234
3 24
4 34568
5 568
6 345678
7 7
8 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。