**华北水利水电大学 形式语言与自动机 实验报告**

**2022-2023 学年 第**  一 **学期 2020 级 人工智能 专业**

**指导老师 张瑞霞 学号 202018526 姓名 高树林**

## 实验四 由NFA构造与之等价的DFA的程序实现

1. **实验目的**

（1）理解NFA与DFA的等价性；

（2）掌握NFA构造等价DFA的方法；

（3）进一步理解由问题形式化到解决问题自动化的过程。

1. **实验内容**

输入任意给定的NFA，转换为等价的DFA，并以五元组或状态转换图方式输出DFA。

1. **实验指导**

（1）用户输入NFA，可以自设以尽可能方便用户输入的形式限定NFA格式；

（2）实现子集法构造DFA；

（3）把DFA输出到屏幕或文件，以五元组或状态转换图方式输出自动机，要求状态结点大小位置布局合理，具有通用性，可以查找相关资料，使用相关包或插件，也可自己设计实现。

1. **实验设计与实现**

包括主要设计思想、主要数据结构、主要程序设计流程图或伪代码、主要程序代码。

**设计思想：**本实验难点在于多个状态之间的转换。当前状态只有一个的时候很好理解，只需要读取NFA的状态就好，但是当当前状态是多个状态时，需要对每一个元素进行遍历，读取最开始的映射表。因此需要设计一个函数用于接受每次的状态列表和即将读取的数据，函数在返回这些列表读取这些数据后得到的列表元素。这个列表在返回前需要进行去重处理，在实验3中我发现去重处理可以通过集合的数据结构进行实现。之后再将列表升序排序并返回。在遍历每个元素后，最终就能得到节点和节点对应的读取相应的值后得到的状态。用这两个结果的对应关系就能得到相应的状态转移函数和状态转移表格。

**数据结构：**本次实验采用了列表的数据结构，原因是如果采用双重字典嵌套的结构，操作性很，而且非常容易出错，因此我放弃了字典的数据结构转而采用列表的数据结构。每个列表代表一个状态转移函数，列表第一个元素是当前状态列表，第二个元素代表了读取的字符，第三个元素代表的是读取后的状态列表。其结构如下表1所示。

表1 数据结构表

|  |  |  |
| --- | --- | --- |
| [q0,q1,q2] | a | [q2] |

上表1表示为：当前状态为[q0.q1.q2]，读取字符a之后到达状态[q2]。

**流程图：**

输入NFA的状态转移函数

将状态函数转为本实验的数据结构并存储

构造DFA转移结果列表

获取每个节点

输出结果

图1 NFA转DFA流程图

**实验代码：**

1. S = input('输入开始符：')
2. E = input('输入结束符：')
3. status = input('输入所有状态(用空格分开）：').split(' ')
4. data = input('输入读入的符号：')
5. **print**('请输入NFA对应的转移函数,按#退出,如：(q0,a)=q0：')
6. NFA = []
7. **while** 1:
8. a = input()
9. **if** a=='#':
10. **break**
11. NFA.append([str(a.split('=')[0][1:3]),a.split('=')[0][-2],a.split('=')[-1]])
12. **print**(NFA)


16. **def** get\_item(stlis, word):  # 给当前状态集和读取的字符，返回可达状态
17. lis = []
18. **for** i **in** stlis:
19. **for** j **in** NFA:
20. **if** [i, word] == j[:2]:
21. lis.append(j[2])
22. **if** '$' **in** lis:
23. **del** lis[lis.index('$')]
24. **return** sorted(list(set(lis)))

27. lower = ['a', 'b']
28. # print(get\_item(['q0', 'q1', 'q2', 'q3'], 'b'))
29. # print('\ \t a \t b')
30. point = [['q0']]
31. result = []
32. **for** i **in** point:
33. **for** j **in** lower:
34. # print(get\_item(i, j), end='\t')
35. result.append(get\_item(i,j))
36. **if** get\_item(i,j) **not** **in** point:
37. point.append(get\_item(i,j))
38. # print('')
39. # print(result)
40. # print(point)
41. DFA = {}
42. **print**(' '\*44+'NFA状态转移表'+' '\*44)
43. **print**('-'\*100)
44. **print**('%-40s | %-40s | %-40s'%('状态','a','b'))
45. **print**('-'\*100)
46. **for** i **in** range(0,len(result),2):
47. cur\_point=point[i//2]
48. nex\_point = [result[i],result[i+1]]
49. **print**('%-40s | %-40s | %-40s'%(point[i//2],result[i],result[i+1]))      # 2==len(lower)
50. **print**('-'\*100)
51. **print**('结束')
53. **print**('\n\n\n')
54. **print**(' '\*44+'NFA状态转移函数'+' '\*44)
55. **for** i **in** range(0,len(result),2):
56. cur\_point=point[i//2]
57. # print("$(%s,a) = %s"%(cur\_point,result[i]))
58. # print(str(cur\_point))
59. **print**("%50s = %s" % (str('$(' + str(cur\_point) + ', a)'), str(result[i]).replace('[', '{').replace(']', '}')))
60. **print**("%50s = %s" % (str('$('+str(cur\_point)+', b)'), str(result[i+1]).replace('[','{').replace(']','}')))
61. **print**('结束')

**代码流程解释：**

首先输入NFA五元组的各个部分——开始符、结束符、字母表、状态集、转移函数。在输入转移函数的时候，需要对输入的格式行转化，转化为前面提到的列表的格式。之后定义了一个得到DFA状态的函数，这个叫函数接受的是已经得到的DFA节点列表和他将要读取的字符，返回的是该DFA节点通过读取字符所到达的新的状态。之后定义一个point列表，这个列表用于保存所有的DFA节点，用作最后结果的输出和迭代的过程。最开始point只有一个元素，但是随着迭代读取，获得的DFA结果一旦符合新结点的要求，就立刻把该结果加到point列表中。以此来迭代，到最后得到的元素均在point中，循环停止。得到的point和result列表分别是节点和节点读取元素到达的新状态。最后通过控制输出就能输出状态转移函数和状态转移表了。

1. **实验结果及分析**

**结果1：**获得的状态转移表如下图2所示。

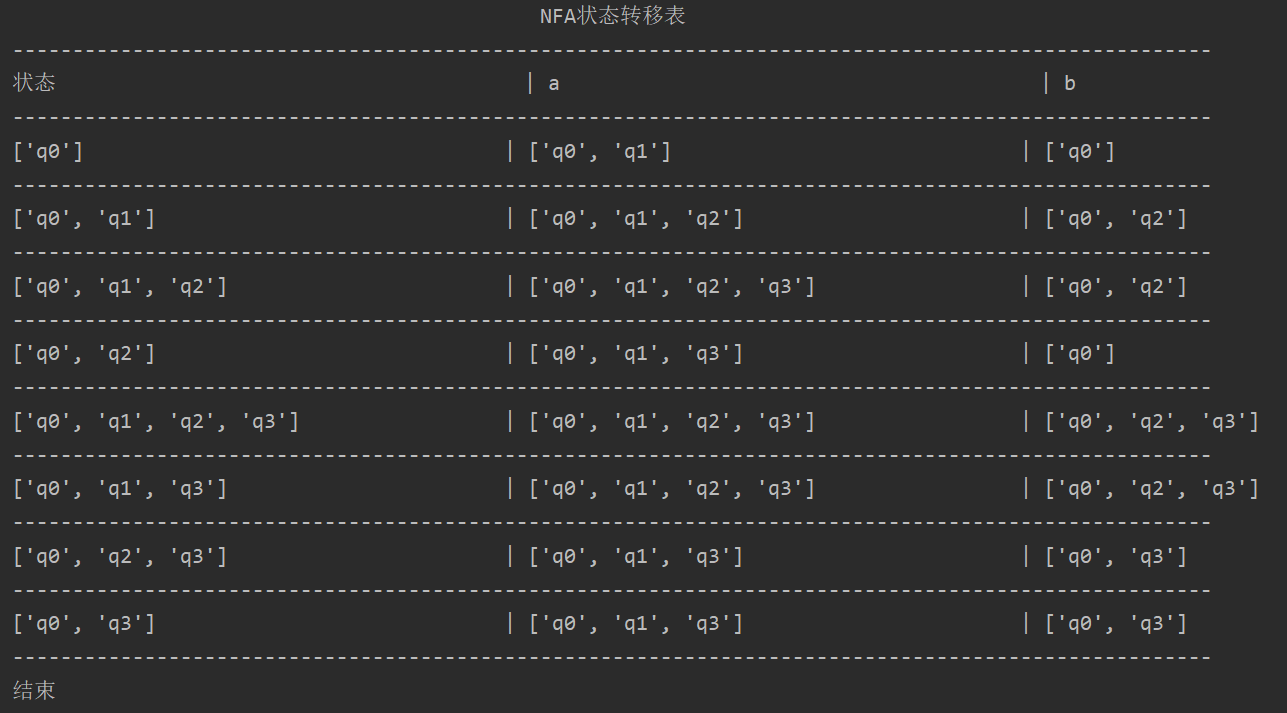


图2 NFA转DFA的状态转移表

**结果2：**获得的状态转移函数如下图3所示。

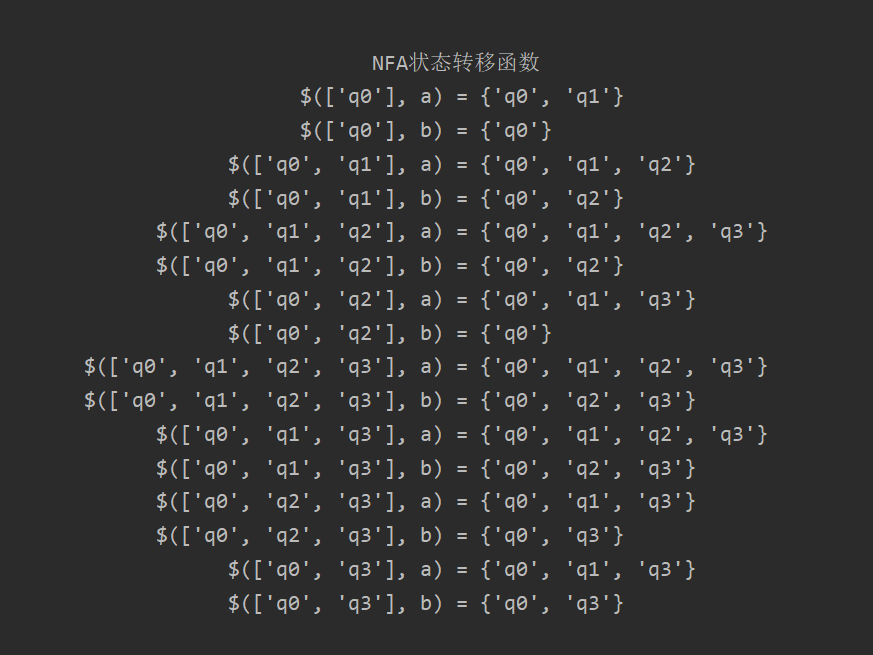


图3 图2 NFA转DFA的状态转移函数

**结果分析：**NFA转DFA的实质就是，把不确定的所有状态归为一个状态集，从而实现状态集之间的转换。在本题中，q0最先读取a时，能到达两个状态，分别是q0状态和q1状态，读取b的时候能到达q0状态，由于[q0,q1]不在此时的point中，因此这个状态集也是point中的一个元素，[q0]在point里面，因此这个状态集不应该被加到point里面，这和我们在纸上书写的时候忽略这种情况的实质是一样的。之后[q0,q1]最先读取a时，能到达三个状态，分别是q0状态、q1状态、q2状态，读取b的时候能到达q0状态、q2状态，由于[q0,q1,q2]和[q0,q2]不在此时的point中，因此这两个状态集也是point中的元素。之后[q0,q1,q2]最先读取a时，能到达四个状态，[q0,q1,q2,q3]，读取b的时候能到达q0状态、q2状态，由于[q0,q1,q2,q3]不在此时的point中，因此这它是状态集也是point中的元素。之后[q0,q2]最先读取a时，能到达三个状态，[q0,q1,q3]，读取b的时候能到达q0状态，由于[q0,q1,q3]不在此时的point中，因此这它是状态集也是point中的元素。之后[q0,q1,q2,q3]最先读取a时，能到达四个状态，[q0,q1,q2,q3]，读取b的时候能到达[q0,q2,q3]状态，由于[q0,q2,q3]不在此时的point中，因此这它是状态集也是point中的元素。之后[q0,q1,q3]最先读取a时，能到达四个状态，[q0,q1,q2,q3]，读取b的时候能到达[q0,q2,q3]状态，由于[q0,q2,q3]在此时的point中，因此这它步不是状态集也是point中的元素。之后[q0,q2,q3]最先读取a时，能到达三个状态，[q0,q1,q3]，读取b的时候能到达[q0，q3]状态，由于[q0,q3]不在此时的point中，因此这它是状态集也是point中的元素。之后[q0，q3]最先读取a时，能到达四个状态[q0,q1,q3]，读取b的时候能到达[q0，q3]状态，由于[q0,q1,q3]和[q0,q3]均不在此时的point中，因此这它俩均不是状态集也是point中的元素，至此循环终止。得到的point就是最终的节点列表。

1. **实验总结**

主要总结所遇到的问题及其解决方法（可以是成功解决的方法或部分解决方法或无法解决的原因）。

**问题1：** 最开始我没有选择使用列表作为存储的数据结构，而是仍然使用前面提到的字典存储数据，结果出现了很严重的错误，就是无法解析对应的状态。甚至还报出了错误。报错截图如下图4所示。

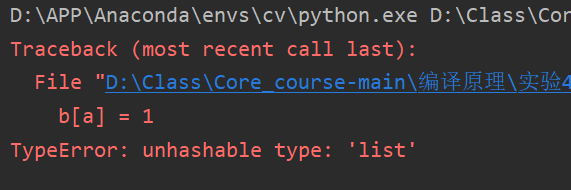


图4 问题1报错截图

出现这个问题就是层层的嵌套出现了用列表作为字典的键，由于列表是不可哈希的数据结构，因此在这里会报错。最终我放弃了字典的存储结构，使用了列表进行存储，解决了问题。

**问题2：**在迭代的全过程中，我没有注意到得到的状态结果已经存在的情况，导致程序出现了死循环的状态。

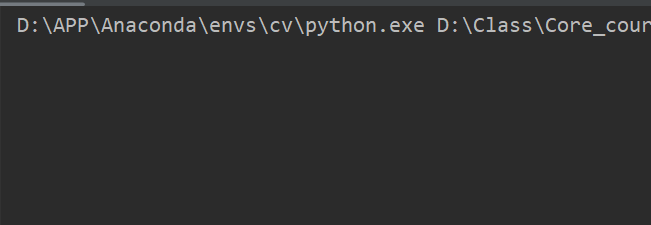


图5 问题2运行情况

出现这种问题的原因是，point一直在迭代，一直在加，因为不考虑重复的，所以他一直处于自己迭代自己的情况。我查看了一下代码，结果真的如我料想的一样。代码如下图6所示。在这个程序中，最开始是没有if判断条件的，这就导致生成的状态列表毫无顾忌的加到point里面。最终导致程序一直在for循环里面。之后我修改了代码，添加了判断语句，解决了这个问题，并最终得出了正确答案。

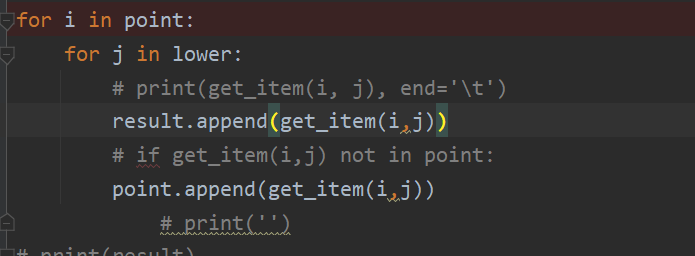


图6 导致问题2的代码段