北京郵電大學

计算机学院（国家示范性软件学院）

NFA 转换为 DFA 实验报告

课程名称：形式语言与自动机

项目名称：NFA 到 DFA 的转换

项目成员：艾宇婧、陈韵涵、毛杨帆

时间：2024 年 6 月 2 日

目录

一、项目主要内容

1.1 实现的功能描述

1. NFA 输入

输入转换数量和每个转换的源状态、转换条件（包括空转换条件’&’）和目标状态。这些信息构成了原始的 NFA。

2. NFA 可视化

使用 Graphviz 库创建了一个 NFA 的可视化图形，其中节点表示状态，边表示状态之间的转换，边上的标签表示转换条件。

3. DFA 转换

计算 ε -closure：对于给定的状态集合，计算其 ε -closure，即可以通过空转换到达的状态集合。

计算状态的转移：给定状态和输入字符，根据已知的 NFA 计算状态的转移结果。

构建 DFA：使用 NFA 中的转换规则，逐步构建 DFA，其中每个状态是 NFA 状态的子集。

4. DFA 可视化

使用 Graphviz 库创建了一个 DFA 的可视化图形，其中节点表示 DFA 的状态，边表示状态之间的转换，节点标签表示状态集合。

5. 打印终止状态

打印出 DFA 中的终止状态，这些状态由输入的终止状态集合确定。

1.2 实验环境描述

1. IDE：vscode
2. 编程语言：python
3. 可视化工具：graphviz

二、程序的设计思路及核心算法

2.1 设计思路

1. 理解自动机的概念和特点

NFA 到 DFA 的转化原理基于子集构造法，核心思想是将 NFA 的状态集合转换为 DFA 的状态集合，并确保在转换过程中考虑到所有可能的状态组合。要包括 ε 闭包（ε-closure）的计算、状态转换的确定化、构建 DFA 的状态集合和转换函数、确定 DFA 的终态、重复“状态转换的确定化”至“确定 DFA 的终态”，直到没有新的状态集合可以被加入到 DFA 为止，最后简化 DFA。

算法参考：NFA 到 DFA 的转化（保证能讲明白）-CSDN 博客

2. 确定化算法

关键是将 NFA 的状态集合转换为 DFA 的状态集合，并根据 NFA 的转换函数得到 DFA 的转换函数。其中，ε-closure 函数和状态转移函数是核心概念。其中计算 ε-closure 函数 closure(setp, NFA)用于计算状态 setp 的 ε 闭包，内部定义了一个递归函数 recursive_closure(state, current_closure)，通过递归查找状态 state 经过 ε 边可达的所有状态，并将其添加到当前闭包中，外部循环遍历输入的状态集合 setp，对每个状态调用递归函数，并返回闭包的排序后的字符串表示；状态转移函数 move(t, a, NFA) 用于给定状态 t 和输入字符 a，根据已知的 NFA 计算状态的转移，遍历当前状态 t 中的每个状态，并在 NFA 中寻找与状态 t 和输入字符 a 匹配的转移，将目标状态添加到临时变量 temp 中，返回临时变量 temp，其中包含了状态 t 经过输入字符 a 后的目标状态集合。

3. 状态合并

在确定化后可能会出现一些等价状态，需要进行状态合并以简化 DFA。采用检查状态是否为新状态的函数 checkINrawDFA(rawDFA, u)，最后使用 NFA 转换为 DFA 的主要函数 convertNFAToDFA(NFA)确保简化后的 DFA 与原始 NFA 等价。

4. 测试和验证

对设计的算法和实现进行测试和验证，确保能够正确地将 NFA 转换为 DFA，并且简化后的 DFA 与原始 NFA 等价。

2.2 程序核心算法

1. 计算 ε -closure 的算法

```python
# 计算ε-closure
def closure(setp, NFA):
    def recursive_closure(state, current_closure):
        for trans in NFA:
            if trans.src == state and trans.edge == '&' and trans.dst not in current_closure:
                current_closure.add(trans.dst)
                recursive_closure(trans.dst, current_closure)

    closure_set = set(setp)
    for state in setp:
        recursive_closure(state, closure_set)

    return ''.join(sorted(closure_set))
```

- 参数：setp：表示一个状态集合；NFA：状态转移函数列表，里面的元素是一个类，定义如下：

```python
# 状态转移函数类
class MovFn:
    def __init__(self, src, edge, dst):
        self.src = src  # 源状态
        self.edge = edge  # 转换条件
        self.dst = dst  # 目标状态
```

- 定义了一个内部函数 **recursive_closure**，递归地计算通过 **&** 转换可以到达的所有状态
- 使用一个集合 **closure_set** 存储当前的闭包状态，以避免重复状态
- 遍历传入的状态集，对于每个状态遍历 NFA 的状态转移函数，找出从当前状态经过多次空转换可达的所有状态并返回
- 返回值是一个新的状态集，表示状态集 setp 的 ε 闭包

2. 计算状态转移的函数 move

```python
# 给定状态t和输入字符a，根据已知的NFA来计算状态的转移
def move(t, a, NFA):
    temp = ""
    for i in range(len(t)):  # 遍历当前状态t里面的每个状态
        for j in range(len(NFA)):  # 遍历NFA里面的转移函数
            if t[i] == NFA[j].src and NFA[j].edge == a:  # 检查当前转移是否与状态 t 和输入字符 a 匹配
                if not check(temp, NFA[j].dst):  # 如果temp没有包含此状态
                    temp += NFA[j].dst  # 添加新状态到temp
    return temp
```

- 参数：t：当前状态集；a：输入字符；NFA：状态转移函数列表
- 遍历当前状态集，对于每个状态遍历 NFA 的状态转移函数，找出从当前状态通过传入的状态转移函数可以到达的全部状态
- 返回值是一个新的状态集，表示 t 里面所有的状态通过 a 可达的状态的并集

3. 通过 move 和 closure 来计算 DFA 的状态集

```python
# 获取输入字符集
sigma = set()
for transition in NFA:
    if transition.edge != '&':
        sigma.add(transition.edge)
sigma = list(sigma)

start = closure("0", NFA)
start = "".join(sorted(start))
rawDFA = []
rawDFA.append(start)
rawDFAflag = []
rawDFAflag.append(False)
while checkFlag(rawDFAflag) != -1:
    m = checkFlag(rawDFAflag)  # 第一个未处理的状态索引
    rawDFAflag[m] = True
    for i in range(len(sigma)):
        u = closure(move(rawDFA[m], sigma[i], NFA), NFA)  # 计算新状态的闭包
        u = "".join(sorted(u))  # 新状态u
        if u and not checkINrawDFA(rawDFA, u):  # 检查新状态是否可达以及是否为新状态
            rawDFA.append(u)
            rawDFAflag.append(False)

```

- 变量：Sigma：输入字符集；rawDFA：DFA 包含的所有状态；rawDFAflag：表示状态是否经过处理
- 首先令初始状态“0“作为 rawDFA 里面唯一一个元素并把它的 flag 设为 false，然后循环，直到所有的状态都被处理过，即通过状态转移找不出新的状态
  - 对于第一个未经处理的状态，找出它的 ε 闭包
  - 如果它可达并且是一个新的状态，就将其加入到 rawDFA
  - 并且将这个新状态的 flag 设为 false
- 当所有的状态都被处理完并且没有新状态产生的时候，就结束循环。核心思想是子集构造法

4. 得出最终 DFA 的状态转移函数

```python
DFA = []
DFA_temp = []
DFADic = {}

# 遍历每个DFA的状态
for i in range(len(rawDFA)):
    # 计算每个状态在所有输入字符上的转换结果
    transitions = {s: "".join(sorted(closure(move(rawDFA[i], s, NFA), NFA))) for s in sigma}
    # 打印当前状态及其对应的转换结果
    print(f'状态 {rawDFA[i]}: {transitions}')
    # 将当前状态添加到DFA列表中
    DFA.append(rawDFA[i])
```

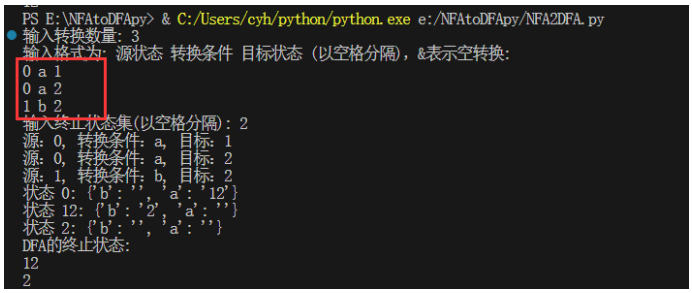
- 变量：DFA：状态列表；DFA_temp：用于存储(源,目的)二元组；DFADic：用于存储状态转移函数，键为(源，目的)，值为转移条件
- 遍历刚刚得出的 DFA 的所有状态（集），通过调用 closure(move(rawDFA[i], s, NFA), NFA) 来计算其在每个输入字符 s 上的转换结果，返回的结果是一个字符串，表示状态经过输入字符转换后的新状态
- 将这些转换结果以字典的形式存储在 transitions 中，其中键为输入字符，值为转换后的状态

- 最后将当前状态加入 DFA

5. 可视化：使用 graphviz
普通状态用圆圈表示，终止状态用双圆圈表示

三、程序的输入，输出，以及执行效果

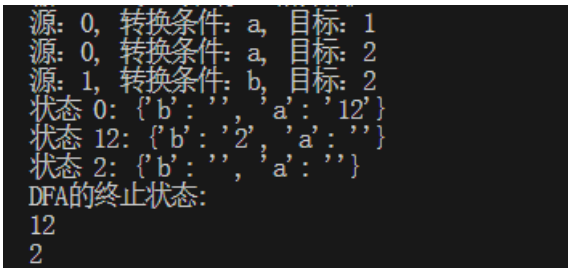
3.1 输入



根据提示，以源状态 转换条件 目标状态的格式输入状态和状态转移函数，其中要求状态是从 0 开始的数字，转换条件不限，用&表示空转换

然后还要根据提示，输入终止状态集

3.2 输出

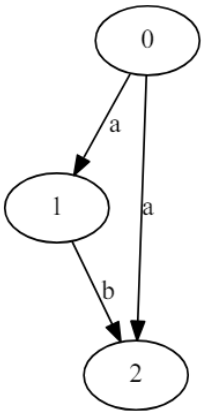


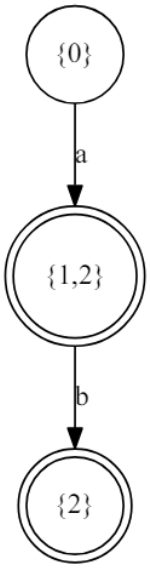
格式化输出输入的 NFA 的状态转移表

并且输出两个 pdf 文件，一个表示原始的 NFA：NFA.gv.pdf；一个表示最终生成的 DFA： DFA.gv.pdf

3.3 执行效果

显示输入的 NFA 和最终得出的 DFA：





3.4 更复杂的测试样例

1. 测试样例 1

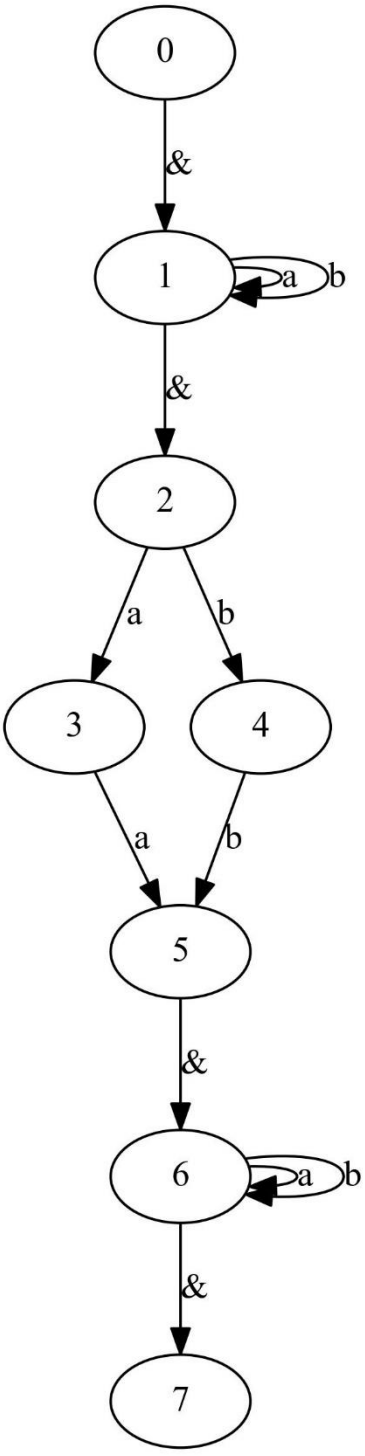
```
0 & 1
1 a 1
1 b 1
1 & 2
2 a 3
2 b 4
3 a 5
4 b 5
5 & 6
6 a 6
6 b 6
6 & 7
```

输出：

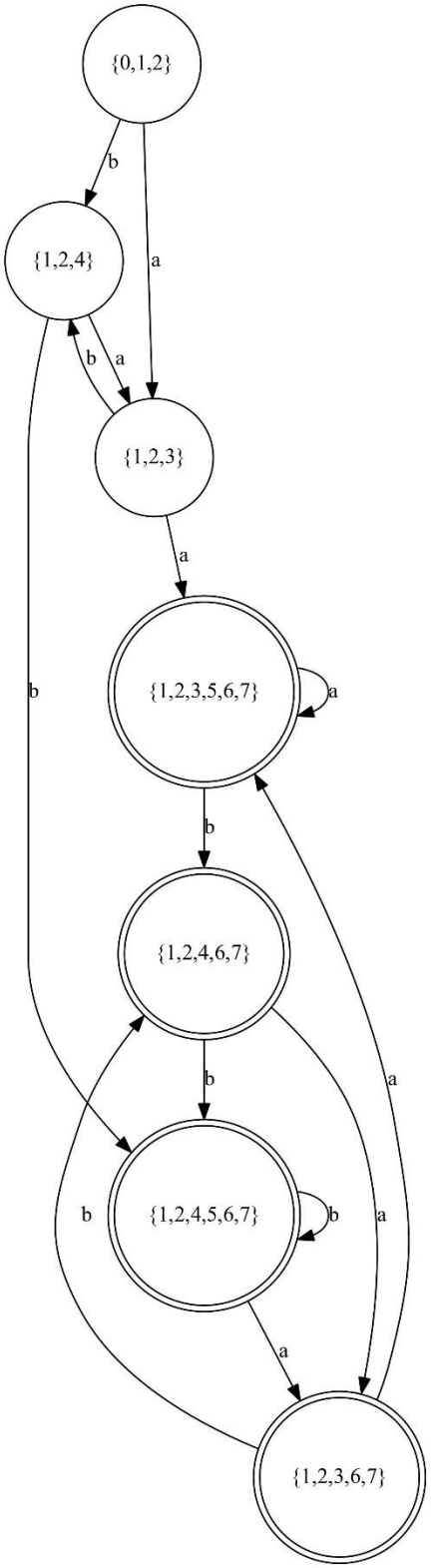
```
输入转换数量: 12
输入格式为: 源状态 转换条件 目标状态 (以空格分隔), &表示空转换:
0 & 1
1 a 1
1 b 1
1 & 2
2 a 3
2 b 4
3 a 5
4 b 5
5 & 6
6 a 6
6 b 6
6 & 7
输入终止状态集(以空格分隔): 7
源: 0, 转换条件: &, 目标: 1
源: 1, 转换条件: a, 目标: 1
源: 1, 转换条件: b, 目标: 1
源: 1, 转换条件: &, 目标: 2
源: 2, 转换条件: a, 目标: 3
源: 2, 转换条件: b, 目标: 4
源: 3, 转换条件: a, 目标: 5
源: 4, 转换条件: b, 目标: 5
源: 5, 转换条件: &, 目标: 6
源: 6, 转换条件: a, 目标: 6
源: 6, 转换条件: b, 目标: 6
源: 6, 转换条件: &, 目标: 7
状态 012: {'b': '124', 'a': '123'}
状态 124: {'b': '124567', 'a': '123'}
状态 123: {'b': '124', 'a': '123567'}
状态 124567: {'b': '124567', 'a': '12367'}
状态 123567: {'b': '12467', 'a': '123567'}
状态 12367: {'b': '12467', 'a': '123567'}
状态 12467: {'b': '124567', 'a': '12367'}
DFA的终止状态:
124567
123567
12367
12467
```

NFA的转移函数

DFA的转移函数



初始 NFA

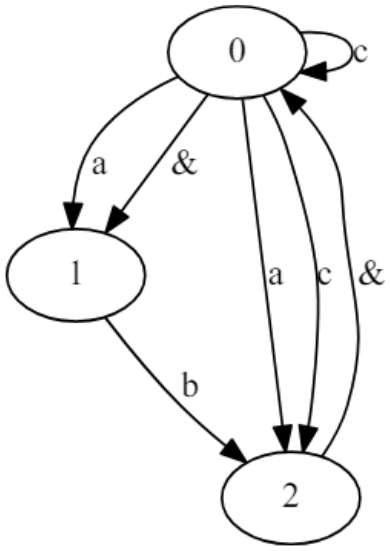


转换后的 DFA

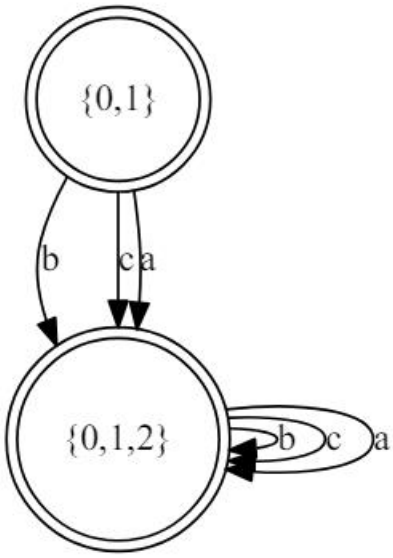
2. 测试样例 2：带空转换

0 a 1
0 a 2
0 c 0
0 c 2
0 & 1
1 b 2
2 & 0

```
PS E:\NFAtoDFApy> & C:/Users/cyh/python/python.exe e:/NFAtoDFApy/NFA2DFA.py
输入转换数量: 7
输入格式为: 源状态 转换条件 目标状态 (以空格分隔), &表示空转换:
0 a 1
0 a 2
0 c 0
0 c 2
0 & 1
1 b 2
2 & 0
输入终止状态集(以空格分隔): 1 2
源: 0, 转换条件: a, 目标: 1
源: 0, 转换条件: a, 目标: 2
源: 0, 转换条件: c, 目标: 0
源: 0, 转换条件: c, 目标: 2
源: 0, 转换条件: &, 目标: 1
源: 1, 转换条件: b, 目标: 2
源: 2, 转换条件: &, 目标: 0
状态 01: {'b': '012', 'c': '012', 'a': '012'}
状态 012: {'b': '012', 'c': '012', 'a': '012'}
DFA的终止状态:
01
012
```



NFA



DFA

四、改进思路和方法

4.1 数据结构优化

4.1.1 状态表示:

当前状态使用字符串拼接,这种方法在状态集合较小时可行,但随着集合大小增加,字符串比较和处理的效率会下降，尤其是在进行集合运算（如合并、求交集）时,对于大型 NFA/DFA 可能不够高效。

改进思路：考虑使用集合或自定义数据结构来表示状态集合，这样在状态合并和比较时会更快。

改进方法：使用 Python 的内置数据结构 set 来表示状态集合。集合提供了高效的成员检查、合并和差集等操作，非常适合表示状态集合。

4.1.2 NFA 存储:

原代码中，NFA 的状态转移是通过列表存储的，每项是一个自定义对象 MovFn，在查找特定转移时可能需要遍历整个列表。

改进思路：改进 NFA 的存储方式，例如使用字典来快速查找状态转移，减少遍历次数。

改进方法：使用字典来存储状态转移关系，以提高查找效率。字典的键可以是状态对（比如源状态和转换条件），值为目标状态，这样可以直接通过状态对查找目标状态，无需遍历。

4.2 算法效率优化

4.2.1 ε-闭包计算:

当状态数很大时，递归计算 ε-闭包容易导致栈溢出，可能不是最优选择

改进思路：可以考虑使用迭代避免栈溢出，并优化重复计算。

方法：可以尝试广度优先搜索(BFS)的迭代方法，通过队列管理待处理的状态。

4.2.2 状态压缩:

DFA 构建过程中可能会产生多个状态，它们对输入的响应规则完全相同，这增加了 DFA 的大小。

改进思路：在构建 DFA 时，如果发现两个状态的转换规则完全相同，可以合并这两个状态，进一步减小 DFA 的大小。

方法：通过比较两个状态对所有输入的转移结果，如果相同，则认为这两个状态等价，可以合并。或者可以考虑哈希表辅助。

4.3. 错误处理与输入验证

添加异常处理机制：比如捕获用户输入错误或文件操作异常。以避免程序意外中断。

4.4 可视化灵活性

动态图生成: 使用参数化的方式生成 Graphviz 图来增强可视化灵活性时，关键可以尝试对状态和转移逻辑的抽象。这样可以更灵活地控制图的展示细节。

五、小组分工

班级：2022211311 班

陈韵涵：核心算法代码、撰写文档设计思路及核心算法部分、输入输出部分、测试样例部分

毛杨帆：输入输出代码、撰写文档实现的功能描述与设计思路部分

艾宇婧：可视化实现、撰写文档改进思路和方法部分