

小组成员：

班级	姓名	学号
2021211321	戴鑫旺	2021211016
2021211321	张一达	2021211005
2021211315	赵先哲	2021211004

成员分工：

工作	参与成员
代码	戴鑫旺、张一达
调试	张一达、赵先哲
文档	戴鑫旺、赵先哲

实验目的

编程实现NFA到DFA的转化，理解不同自动机的转化过程。

实验内容

有限状态自动机是描述控制过程有力工具。有限状态自动机有不同的类型，例如，确定有限状态自动机（DFA）和不确定有限状态自动机（NFA）。这些不同类型的自动机之间可以等价转化。我们在实际应用中，可以利用某种类型的自动机更加方便刻画实际系统，然后再利用等价转化算法实现不同类型的自动机转化。

本实验要求编程实现NFA到DFA的自动转化。输入自己设定的不确定有限自动机描述格式，输出对应的确定有限自动机。

实验环境描述

- Java软件开发工具：IntelliJ IDEA Ultimate 2023.1.2
- JDK版本：JDK 1.8.0_212
- 操作系统：Windows 11

实验设计思路

- 实现对DFA类、NFA类、DFA转移函数类、NFA转移函数类的成员定义
- 实现根据用户输入构建NFA，该过程须具有良好的交互性
- 实现根据NFA生成DFA，该过程的核心算法为子集构造法，需要实现的工具函数有求出某个状态的 ϵ 闭包、求出某个状态集合中所有状态经由某个字符转移后的状态集合、求出某个状态集合经由某个字符转移后的 ϵ 闭包
- 实现对生成的DFA的输出，输出DFA的状态集合、字母表、起始状态、终止状态、转移函数。为使结果更清晰，输出时将状态集合、起始状态、终止状态和转移函数中的状态用大写字母A-Z表示

核心算法

该实验中的核心方法为NFA类中的 `buildDFA()` 方法，该方法实现了由NFA到DFA的转换。

该方法中涉及到的工具方法有 `epsilonClosure()`、`move()` 和 `moveEpsilonClosure()`，首先对这三个工具方法进行解释。

epsilonClosure ()

```
public ArrayList<Integer> epsilonClosure(int state) {
    ArrayList<Integer> closure = new ArrayList<>();
    closure.add(state);
    for (NFATransition transition : this.transitions) {
        if (transition.from == state && transition.ch.equals("&")) {
            if (transition.from == transition.to) {
                continue;
            }
            closure.addAll(epsilonClosure(transition.to));
        }
    }
    return closure;
}
```

`epsilonClosure()` 实现了计算epsilon闭包。epsilon闭包是指从当前状态出发可以通过空转移到达的所有状态的集合，也就是包括当前状态在内，通过空转移能够到达的所有状态。

算法步骤：

1. 声明一个ArrayList closure用来存储epsilon闭包中的所有状态，将当前状态state添加到closure中。
2. 遍历NFA中的每个转移NFATransition，如果转移的起始状态为state且转移字符为"&"（即空字符），则执行以下步骤： a. 如果当前转移的终止状态和起始状态相同，则跳过该转移，避免产生无限递归。 b. 否则将epsilonClosure(transition.to)的结果添加到closure中，表示从当前状态可以通过空字符到达转移的终止状态和终止状态可以到达的状态都属于epsilon闭包。
3. 返回最终得到的epsilon闭包closure列表。

move ()

```
public ArrayList<Integer> move(ArrayList<Integer> states, String ch) {
    ArrayList<Integer> move = new ArrayList<>();
    for (Integer state : states) {
        for (NFATransition transition : this.transitions) {
            if (transition.from == state && transition.ch.equals(ch)) {
                move.add(transition.to);
            }
        }
    }
    return move;
}
```

`move()` 实现了计算从当前状态集合states出发，经过输入字符ch能够到达的所有状态的集合。

算法步骤：

1. 声明一个ArrayList move用来存储从当前状态集合states出发，经过输入字符ch能够到达的所有状态。
2. 遍历当前状态集合states中的每个状态state，遍历NFA中的每个转移NFATransition，如果转移的起始状态为state且转移字符为ch，则将转移的终止状态transition.to添加到move中，表示通过输入字符ch可以从当前状态state到达终止状态transition.to。
3. 返回最终得到的move列表。

moveEpsilonClosure ()

```
public ArrayList<Integer> moveEpsilonClosure(ArrayList<Integer> states, String ch) {  
    ArrayList<Integer> move = move(states, ch);  
    ArrayList<Integer> moveEpsilonClosure = new ArrayList<>();  
    for (Integer state : move) {  
        moveEpsilonClosure.addAll(epsilonClosure(state));  
    }  
    //删除重复元素并排序  
    HashSet<Integer> set = new HashSet<>(moveEpsilonClosure);  
    moveEpsilonClosure.clear();  
    moveEpsilonClosure.addAll(set);  
    Collections.sort(moveEpsilonClosure);  
    return moveEpsilonClosure;  
}
```

moveEpsilonClosure () 是对move和epsilonClosure两个方法的结合运用。moveEpsilonClosure是指从当前状态集合states出发，经过输入字符ch能够到达的所有状态的epsilon闭包。

算法步骤：

1. 调用move方法计算从当前状态集合states出发，经过输入字符ch能够到达的所有状态，返回结果存储在ArrayList move中。
2. 遍历move中的每个状态state，调用epsilonClosure方法计算state的epsilon闭包，将结果添加到ArrayList moveEpsilonClosure中。
3. 对moveEpsilonClosure进行去重和排序操作，得到最终的moveEpsilonClosure列表。
4. 返回最终得到的moveEpsilonClosure列表。

buildDFA ()

```
public DFA buildDFA() {  
    DFA dfa = new DFA();  
    //首先求出NFA的起始状态的ε闭包，作为DFA的起始状态  
    ArrayList<Integer> startClosure = epsilonClosure(this.start);  
    dfa.start = startClosure;  
    //将DFA的起始状态加入到DFA的状态集合中  
    dfa.states.add(startClosure);  
    //将NFA的字母表作为DFA的字母表  
    dfa.alphabet = this.alphabet;  
    //对于每个状态集合，求出经由每个字符转移后的ε闭包，作为DFA的转移函数，并将转移函数添加到DFA的转移函数集合中，并将转移函数的终点加入到DFA的状态集合  
    int dfaStateFinish = 0;  
    while (true){  
        //假如存在未遍历的状态集合，则继续遍历
```

```

//假如不存在未遍历的状态集合，则跳出循环
int dfaStateNumBefore = dfa.states.size();
for (int i = dfaStateFinish; i < dfaStateNumBefore; i++) {
    for (String ch : dfa.alphabet) {
        if (Objects.equals(ch, "&")) {
            continue;
        }
        ArrayList<Integer> moveEpsilonClosure =
moveEpsilonClosure(dfa.states.get(i), ch);
        if(!moveEpsilonClosure.isEmpty()){
            if (!dfa.transitions.contains(new
DFATransition(dfa.states.get(i), ch, moveEpsilonClosure))) {
                dfa.transitions.add(new
DFATransition(dfa.states.get(i), ch, moveEpsilonClosure));
                if (!dfa.states.contains(moveEpsilonClosure)) {
                    dfa.states.add(moveEpsilonClosure);
                }
            }
        }
    }
    dfaStateFinish = dfaStateNumBefore;
    int dfaStateNumAfter = dfa.states.size();
    if (dfaStateNumBefore == dfaStateNumAfter) {
        break;
    }
}

//求出DFA的终止状态集合
//对于每个状态集合，如果该状态集合中包含NFA的终止状态，则将该状态集合作为DFA的终止状态
集合
for (ArrayList<Integer> state : dfa.states) {
    for (Integer end : this.end) {
        if (state.contains(end)) {
            dfa.end.add(state);
            break;
        }
    }
}
return dfa;
}

```

buildDFA()实现了从NFA（非确定性有限状态自动机）构建DFA（确定性有限状态自动机）的算法，基本思路是先求出NFA的起始状态的 ϵ 闭包作为DFA的起始状态，然后对于每个DFA的状态集合，求出经由每个字符转移后的 ϵ 闭包，并将转移函数添加到DFA的转移函数集合中。最后，删除DFA状态集合中的空集，删除DFA转移函数中终点为null的转移函数，求出DFA的终止状态集合。

算法步骤：

1. 创建一个新的DFA对象
2. 调用epsilonClosure方法计算NFA的起始状态的 ϵ 闭包，将结果赋值给DFA的起始状态，并将其添加到DFA的状态集合中。
3. 将NFA的字母表作为DFA的字母表。
4. 对于每个状态集合，枚举DFA的字母表中的每个字符，调用moveEpsilonClosure方法计算经由该字符转移后的 ϵ 闭包，将其作为DFA的转移函数，并将转移函数的终点加入到DFA的状态集合中。

5. 对于每个状态集合，在其中查找是否包含NFA的终止状态，如果包含，则将该状态集合作为DFA的终止状态集合。
6. 返回构建完成的DFA对象。

代码结构

- class DFA
 - class DFATransition
 - DFA ()
 - printDFA ()
 - printDFA2 ()
- class NFA
 - class NFATransition
 - NFA ()
 - buildNFA ()
 - epsilonClosure ()
 - move ()
 - moveEpsilonClosure ()
 - buildDFA ()
- class NFA2DFA

程序中的类、方法和属性

```
class DFATransition {
    public ArrayList<Integer> from; //转移前状态
    public ArrayList<Integer> to; //转移后状态
    public String ch; //转移字符

    public DFATransition(ArrayList<Integer> from, String ch, ArrayList<Integer>
to) {
        this.from = from;
        this.to = to;
        this.ch = ch;
    }
}
```

```
class DFA {
    public ArrayList<Integer> start; //起始状态
    public ArrayList<ArrayList<Integer>> end; //终止状态
    public ArrayList<DFATransition> transitions; //转移函数
    public ArrayList<ArrayList<Integer>> states; //状态集合
    public ArrayList<String> alphabet; //字母表

    //构造函数
    public DFA()

    //打印DFA,包括状态集合、字母表、起始状态、终止状态、转移函数
    public void printDFA()
```

```

//打印DFA，要求将状态集中的状态用大写字母A-Z表示
//打印状态集合、字母表、起始状态、终止状态、转移函数
//打印时将状态集合、起始状态、终止状态和转移函数中的状态用大写字母A-Z表示
public void printDFA2()
}

```

```

class NFATransition {
    public int from; //转移前状态
    public int to; //转移后状态
    public String ch; //转移字符

    public NFATransition(int from, String ch, int to) {
        this.from = from;
        this.to = to;
        this.ch = ch;
    }
}

```

```

class NFA{
    public int start; //起始状态
    public ArrayList<Integer> end; //终止状态
    public ArrayList<NFATransition> transitions; //转移函数
    public ArrayList<Integer> states; //状态集合
    public ArrayList<String> alphabet; //字母表

    //构造函数
    public NFA()

    //根据输入构建NFA
    public NFA buildNFA(Scanner sc)

    //求出某个状态的ε闭包
    public ArrayList<Integer> epsilonClosure(int state)

    //求出某个状态集合中所有状态经由某个字符转移后的状态集合
    public ArrayList<Integer> move(ArrayList<Integer> states, String ch)

    //求出某个状态集合经由某个字符转移后的ε闭包
    public ArrayList<Integer> moveEpsilonClosure(ArrayList<Integer> states,
String ch)

    //由NFA构造DFA
    public DFA buildDFA()
}

```

```
//NFA2DFA类
class NFA2DFA{
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        NFA nfa = new NFA();
        nfa = nfa.buildNFA(sc);
        DFA dfa = nfa.buildDFA();
        dfa.printDFA2();
    }
}
```

程序的输入，输出以及执行效果

例一（来自课本例题）

例1 设 NFA $M=(Q,T,\delta,q_0,F)$,其中 $Q=\{q_0,q\},T=\{a,b\},F=\{q\},\delta$ 如表 3.3.1 所示,图3.3.1是 M 的状态转换图。找出等效的 DFA M_D 。

NFA的转移函数表:

表 3.3.1 δ 的转换函数表

状 态 \ 输 入	a	b
q_0	$\{q_0,q\}$	$\{q\}$
q	\varnothing	$\{q_0,q\}$

输入	输出	标准答案
请输入起始状态： 0 请输入终止状态个数： 1 请输入第1个终止状态： 1 请输入转移函数个数： 5 请输入第1个转移函数： 0 a 0 请输入第2个转移函数： 0 a 1 请输入第3个转移函数： 0 b 1 请输入第4个转移函数： 1 b 0 请输入第5个转移函数： 1 b 1	状态集合： A = 0 B = 0 1 C = 1 字母表： a b 起始状态： A 终止状态： B C 转移函数： A a B A b C B a B B b B C b B	<pre> graph LR start(()) --> q0(([q0])) q0 -- b --> q(([q])) q0 -- a --> q0q(([q0,q])) q -- b --> q0q q0q -- "a,b" --> q0q style start fill:none,stroke:none style q0 fill:none,stroke:none style q fill:none,stroke:none style q0q fill:none,stroke:none </pre>

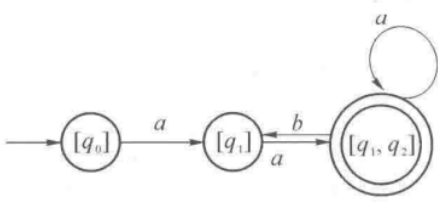
将输出与标准答案进行对照，结果一致

例二（来自课本例题）

例 2 设 NFA $M = (\{q_0, q_1, q_2\}, \{a, b\}, \delta, q_0, \{q_2\})$

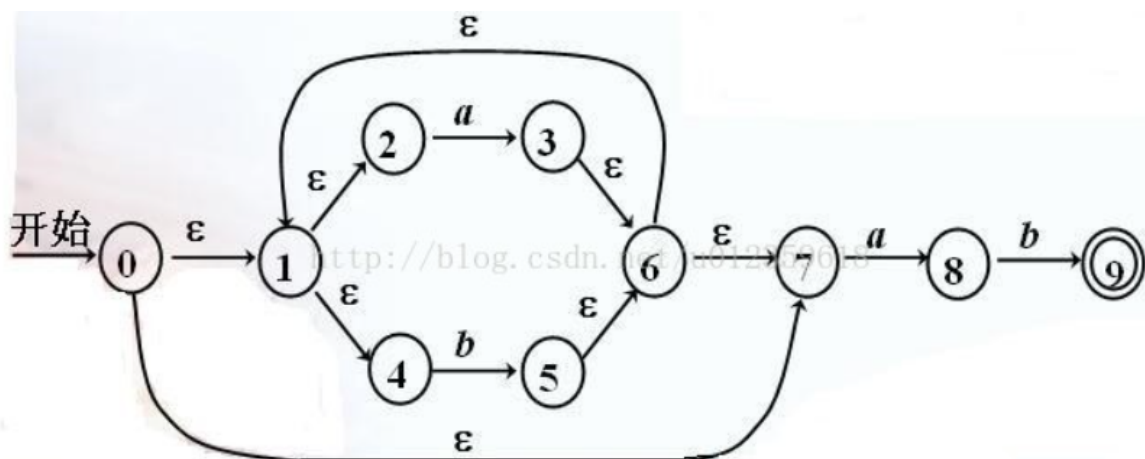
δ 的定义如下：

$$\begin{aligned}
 \delta(q_0, a) &= \{q_1\}, & \delta(q_0, b) &= \emptyset, \\
 \delta(q_1, a) &= \{q_1, q_2\}, & \delta(q_1, b) &= \emptyset, \\
 \delta(q_2, a) &= \emptyset, & \delta(q_2, b) &= \{q_1\}
 \end{aligned}$$

输入	输出	标准答案
请输入起始状态： 0 请输入终止状态个数： 1 请输入第1个终止状态： 2 请输入转移函数个数： 4 请输入第1个转移函数： 0 a 1 请输入第2个转移函数： 1 a 1 请输入第3个转移函数： 1 a 2 请输入第4个转移函数： 2 b 1	状态集合： $A = 0$ $B = 1$ $C = 1\ 2$ 字母表： $a\ b$ 起始状态： A 终止状态： C 转移函数： $A\ a\ B$ $B\ a\ C$ $C\ a\ C$ $C\ b\ B$	

将输出与标准答案进行对照，结果一致

例三（来自网络）



输入	输出	标准答案
<p>请输入起始状态:</p> <p>0</p> <p>请输入终止状态个数:</p> <p>1</p> <p>请输入第1个终止状态:</p> <p>9</p> <p>请输入转移函数个数:</p> <p>12</p> <p>请输入第1个转移函数:</p> <p>0 & 1</p> <p>请输入第2个转移函数:</p> <p>0 & 7</p> <p>请输入第3个转移函数:</p> <p>1 & 2</p> <p>请输入第4个转移函数:</p> <p>1 & 4</p> <p>请输入第5个转移函数:</p> <p>2 a 3</p> <p>请输入第6个转移函数:</p> <p>3 & 6</p> <p>请输入第7个转移函数:</p> <p>4 b 5</p> <p>请输入第8个转移函数:</p> <p>5 & 6</p> <p>请输入第9个转移函数:</p> <p>6 & 1</p> <p>请输入第10个转移函数:</p> <p>6 & 7</p> <p>请输入第11个转移函数:</p> <p>7 a 8</p> <p>请输入第12个转移函数:</p> <p>8 b 9</p>	<p>状态集合:</p> <p>A = 0 1 2 4 7</p> <p>B = 1 2 3 4 6 7 8</p> <p>C = 1 2 4 5 6 7</p> <p>D = 1 2 4 5 6 7 9</p> <p>字母表:</p> <p>& a b</p> <p>起始状态:</p> <p>A</p> <p>终止状态:</p> <p>D</p> <p>转移函数:</p> <p>A a B</p> <p>A b C</p> <p>B a B</p> <p>B b D</p> <p>C a B</p> <p>C b C</p> <p>D a B</p> <p>D b C</p>	

将输出与标准答案进行对照，结果一致

实验中遇到的问题及解决方案

1. **初始时未定义良好的数据结构，导致后续的方法实现中出现问题。**在初始对NFA、DFA等类进行成员定义时，未考虑清楚两者在数据结构上的区别，定义了一个统一的转移函数类Transition作为两者的成员变量，在定义状态集合时也未选择合适的数据类型，导致后续的方法实现中对两者进行构造和赋值时存在问题。**解决方案：**分别定义两个转移函数类DFATransition和NFATransition作为DFA类和NFA类的成员变量，DFA类的状态集合选取 `ArrayList<ArrayList<Integer>>` 这一类型，NFA类的状态集合选取 `ArrayList<Integer>` 这一类型，以方便后续方法的编写。
2. **核心方法 `**buildDFA()` 中的循环中抛出 `**ConcurrentModificationException**` 异常。**在这一方法中，一个关键的步骤为：对于每个状态集合，求出经由每个字符转移后的 ϵ 闭包，作为DFA的转移函数,并将转移函数添加到DFA的转移函数集合中,并将转移函数的终点加入到DFA的状态集合。程序在集合的迭代过程尝试修改集合的结构，引起了并发修改的冲突。出错代码如下所示：

```
for (ArrayList<Integer> state : dfa.states) {
    for (String ch : dfa.alphabet) {
        ArrayList<Integer> moveEpsilonClosure = moveEpsilonClosure(state,
ch);

        if (!moveEpsilonClosure.isEmpty()) {
            dfa.transitions.add(new DFATransition(state, ch,
moveEpsilonClosure));
            if (!dfa.states.contains(moveEpsilonClosure)) {
                dfa.states.add(moveEpsilonClosure);
            }
        }
    }
}
```

解决方案：不使用for-each循环遍历状态集合，而是采用while循环与for循环嵌套的结构，添加整型变量记录已遍历的状态，实现对集合的遍历并并发修改集合，同时避免多次遍历同一状态。修改后代码如下：

```
while (true){
    //假如存在未遍历的状态集合，则继续遍历
    //假如不存在未遍历的状态集合，则跳出循环
    int dfaStateNumBefore = dfa.states.size();
    for (int i = dfaStateFinish; i < dfaStateNumBefore; i++) {
        for (String ch : dfa.alphabet) {
            if (Objects.equals(ch, "&")) {
                continue;
            }
            ArrayList<Integer> moveEpsilonClosure =
moveEpsilonClosure(dfa.states.get(i), ch);
            if (!dfa.transitions.contains(new
DFATransition(dfa.states.get(i), ch, moveEpsilonClosure))) {
                dfa.transitions.add(new DFATransition(dfa.states.get(i),
ch, moveEpsilonClosure));
                if (!dfa.states.contains(moveEpsilonClosure)) {
                    dfa.states.add(moveEpsilonClosure);
                }
            }
        }
    }
    dfaStateFinish = dfaStateNumBefore;
}
```

```
int dfaStateNumAfter = dfa.states.size();  
if (dfaStateNumBefore == dfaStateNumAfter) {  
    break;  
}  
}
```