形式语言与自动机实验一：NFA 到 DFA 的转化 实验报告

毛子恒　　邹宇江　　王敏行　　曾嘉伟

北京邮电大学 计算机学院

日期：2021 年 4 月 11 日

小组成员

班级：2019211309	姓名：毛子恒	学号：2019211397	分工：代码 文档
班级：2019211309	姓名：邹宇江	学号：2019211416	分工：测试 文档
班级：2019211309	姓名：王敏行	学号：2019211410	分工：测试 文档
班级：2019211309	姓名：曾嘉伟	学号：2019211396	分工：测试 文档

目录

1 需求分析 **2**

1.1 题目描述 . . . 2
1.2 输入描述 . . . 2
1.3 输出描述 . . . 2
1.4 样例输入输出 . . . 2
1.5 样例解释 . . . 3

2 程序设计 **3**

2.1 环境 . . . 3
2.2 设计思路 . . . 3
2.3 核心算法伪代码 . . . 4

3 调试分析 **4**

3.1 算法复杂度分析 . . . 4
3.2 改进设想的经验和体会 . . . 4

4 测试结果 **5**

4.1 测试集 1 . . . 5
4.2 测试集 2 . . . 6
4.3 测试集 3 . . . 7
4.4 测试集 4 . . . 8

1 需求分析

1.1 题目描述

输入一个 NFA，输出等价转化出的 DFA。

1.2 输入描述

程序从标准输入中读入数据。

以下描述中**字符串**均特指不包含空格、回车等特殊字符的 ASCII 字符序列。**要求输入是一个合法的 NFA**。

第一行输入若干字符串，用空格分隔，表示 NFA 的状态集合，**要求输入不出现重复状态**，设集合大小为 n。

第二行输入若干字符串，用空格分隔，表示 NFA 的字母表，**要求输入不出现重复符号**，设字母表大小为 m，**注意**，如果输入的是 ε-NFA，则需要在本行最后输入字符串 [empty] 表示空串。

接下来的 $n \times m$ 行，每行若干个字符串，其中第 $(i-1) \times m + j$ 行表示第 i 个状态在输入第 j 个符号时转移到的状态集合，**要求输入不出现重复状态，且状态均包含在 NFA 的状态集合中**。如果状态集合为空，则以一个空行表示。

接下来的一行输入一个字符串，表示初始状态，**要求初始状态包含在 NFA 的状态集合中**。

最后一行输入若干字符串，用空格分隔，表示 NFA 的终止状态集合，**要求输入不出现重复状态，集合不为空，且终止状态包含在 NFA 的状态集合中**。

1.3 输出描述

程序向标准输出中输出等价的 DFA 的转移表。

1.4 样例输入输出

【输入】

```
p q r
0 1
q

q
q r


p
r
```

【输出】

见图 1。

	0	1
->{p}	{q}	{}
{q}	{q}	{q,r}
*{q,r}	{q}	{q,r}

图 1: 样例输出

1.5 样例解释

原 NFA 的转移表如**图 2**。

	0	1
→ p	{q}	ϕ
q	{q}	{q, r}
* r	ϕ	ϕ

图 2: 样例 NFA 的转移表

2 程序设计

2.1 环境

- macOS Big Sur 11.2.3
- gcc version 10.2.0
- C++11

2.2 设计思路

首先将状态和符号按照输入顺序从 0 开始编号，进而每一个状态集合都可以唯一对应一个二进制数。

以样例为例：状态 p, q, r 分别编号为 $0, 1, 2$，符号 $0, 1$ 分别编号为 $0, 1$。则状态集合 $\{p\}$ 可以表示为 001_2 即 1_{10}，状态集合 $\{q\}$ 可以表示为 010_2 即 2_{10}，状态集合 $\{q, r\}$ 可以表示为 110_2 即 6_{10}。

在状态字符串和状态序号之间需要建立双向映射，分别用 vector 和 unordered_map 实现。

进而 NFA 的状态转移可以用二维数组 $delta$ 表示，其中第一维下标表示状态序号，第二维下标表示字符序号，数组的值表示状态集合对应的二进制数。

例如 $delta[1][1] = 6$ 表示序号为 1 的状态 q 通过序号为 1 的符号 1 转移到状态集合 $\{q, r\}$。

通过子集构造法做等价转换，其过程无异于做深度优先搜索或者广度优先搜索，通过记录重复的状态，使得每个不同的状态最多遍历一次。而通过当前状态转移到下一个状态即对当前状态集合中每个元素的状态转移集合做并集，对于二进制数来说就是做或运算。

用二维数组 ans 表示 DFA 的状态转移表，其中每一行的第一个值存储原状态集合，其后的 m 个值依次为 m 种转移后的状态集合。

输出时，通过解析二进制数的每一位，依照映射输出对应的状态字符串。当出现初始状态和终止状态时加特殊标记，其中终止状态满足当前状态集合和原 NFA 的状态集合的交不为空集，即对应的二进制数做与运算不为 0。

此外，程序通过求解 ε-closure 的方法，将 ε-NFA 转换成 NFA。具体步骤参考代码注释。

2.3 核心算法伪代码

算法 1: 子集构造法

```
1 Function dfs(status)
2     将 status 标记为已访问;
3     在 ans 中增加新的一行，并将 status 记录为第一个元素;
4     foreach 符号 i do
5         nextstatus ← 0;
6         foreach 当前状态集合中的状态 j do
7             nextstatus ←nextstatus ∨ 状态 j 通过符号 i 转移到的状态集合;
8         将 nextstatus 加入 ans 的当前行;
9         if nextstatus 没有访问过 then dfs(nextstatus);
```

输入和输出的细节参考代码中的注释。通过设定 PRINT_WIDTH 常量改变输出表格的宽度。

3 调试分析

3.1 算法复杂度分析

一般情况下，构建出的 DFA 与 NFA 的规模大体相当，所以算法的时间复杂度约为 $O(n^2m)$，空间复杂度约为 $O(nm)$。

在最坏情况下，算法需要遍历所有状态集合，此时时间复杂度为 $O(2^nnm)$，空间复杂度为 $O(2^nm)$。

3.2 改进设想的经验和体会

输入输出采用 C++ 标准 IO 流，增加可读性。灵活应用 C++ 标准库，节省编程复杂度和空间。

由于期望输出的 n 较小，范围假定在 64 之内，所以表示状态集合的数字用`unsigned long long`类型。当 n 较大时程序可能很难在数秒的时间内运行完毕，并且此时需要用 `bitset` 存储状态集合，但对时间复杂度和空间复杂度没有太大影响。

如果输入输出、数据结构采用常规 C 的方法，可能会有常数级别的优化，但是会大大增加编程复杂度。

由于 NFA 不合法的情况太多，难以一一判断，所以默认输入的是合法的 NFA，没有做过多处理。

4 测试结果

4.1 测试集 1

4.1.1 输入

```
q0 q1 q2 q3
a b
q0 q1
q0
q2
q2
q3

q3
q3
q0
q3
```

4.1.2 解释

$M=(\{q_0,q_1,q_2,q_3\},(a,b),\delta,q_0,\{q_3\})$，其中 δ 如下：

$$\begin{aligned}
\delta(q_0,a)&=\{q_0,q_1\} & \delta(q_0,b)&=\{q_0\}\\
\delta(q_1,a)&=\{q_2\} & \delta(q_1,b)&=\{q_2\}\\
\delta(q_2,a)&=\{q_3\} & \delta(q_2,b)&=\varnothing\\
\delta(q_3,a)&=\{q_3\} & \delta(q_3,b)&=\{q_3\}
\end{aligned}$$

4.1.3 输出

	a	b
->{q0}	{q0,q1}	{q0}
{q0,q1}	{q0,q1,q2}	{q0,q2}
{q0,q1,q2}	{q0,q1,q2,q3}	{q0,q2}
*{q0,q1,q2,q3}	{q0,q1,q2,q3}	{q0,q2,q3}
*{q0,q2,q3}	{q0,q1,q3}	{q0,q3}
*{q0,q1,q3}	{q0,q1,q2,q3}	{q0,q2,q3}
*{q0,q3}	{q0,q1,q3}	{q0,q3}
{q0,q2}	{q0,q1,q3}	{q0}

4.2 测试集 2

4.2.1 输入

```
q0 q1 q2 q3
a b
q1 q3
q1
q2
q1 q2
q3
q0

q0
q0
q1 q3
```

4.2.2 解释

$M=(\{q_0,q_1,q_2,q_3\},(a,b),\delta,q_0,\{q_1,q_3\})$，其中 δ 如下：

$$\begin{array}{ll} \delta(q_0,a)=\{q_1,q_3\} & \delta(q_0,b)=\{q_1\} \\ \delta(q_1,a)=\{q_2\} & \delta(q_1,b)=\{q_1,q_2\} \\ \delta(q_2,a)=\{q_3\} & \delta(q_2,b)=\{q_0\} \\ \delta(q_3,a)=\varnothing & \delta(q_3,b)=\{q_0\} \end{array}$$

4.2.3 输出

	a	b
->{q0}	{q1,q3}	{q1}
*{q1,q3}	{q2}	{q0,q1,q2}
{q2}	{q3}	{q0}
*{q3}	{}	{q0}
*{q0,q1,q2}	{q1,q2,q3}	{q0,q1,q2}
*{q1,q2,q3}	{q2,q3}	{q0,q1,q2}
*{q2,q3}	{q3}	{q0}
*{q1}	{q2}	{q1,q2}
*{q1,q2}	{q2,q3}	{q0,q1,q2}

4.3 测试集 3

4.3.1 输入

```
q0 q1 q2
0 1 2 [empty]
q0


q1

q1

q2


q2

q0
q2
```

4.3.2 输出

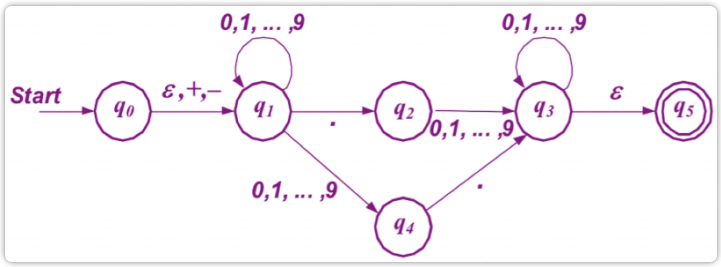
	0	1	2
->{q0}	{q0,q1,q2}	{q1,q2}	{q2}
*{q0,q1,q2}	{q0,q1,q2}	{q1,q2}	{q2}
*{q1,q2}	{}	{q1,q2}	{q2}
*{q2}	{}	{}	{q2}

4.4 测试集 4

4.4.1 输入

```
q0 q1 q2 q3 q4 q5
+ * 0 [empty]
q1


q1

q2
q1 q4



q3



q3
q5

q3









q0
q5
```

4.4.2 解释



4.4.3 输出

	+	*	0
->{q0}	{q1}	{q2}	{q1,q4}
{q1}	{}	{q2}	{q1,q4}
{q2}	{}	{}	{q3,q5}
*{q3,q5}	{}	{}	{q3,q5}
{q1,q4}	{}	{q2,q3,q5}	{q1,q4}
*{q2,q3,q5}	{}	{}	{q3,q5}