

形式语言与自动机

实验报告

实验名称：NFA 到 DFA 的转化

课程名称：形式语言与自动机

姓名（学号）：叶淳瑜 2019212686

涂欣宇 2019212693

蒲昊龙 2019212968

李孟非 2019210933

学 院：计算机学院

专 业：计算机类

班 级：2019211318

指导教师：左兴权老师

二〇二一年 四月

1 实验概述

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。

本实验要求编程实现 NFA 到 DFA 的自动转化。小组成员编写了程序，允许用户输入不确定型有限自动机，输出对应的确定型有限自动机。

2 成员分工

叶淳瑜	项目规划、部分文档编写以及部分代码编写与调试工作
涂欣宇	核心代码编写工作
蒲昊龙	核心代码编写工作
李孟非	部分文档及报告编写、部分代码调试工作

3 环境描述

开发平台：Visual Studio Code

编程语言及标准：C++11

编译器：MinGW

4 设计思路及核心算法

整体框架简介：

代码分为三个模块，除了主函数有两个功能函数，分别如下：

- `string calculate(string state, string alphabet)`

返回 `state` 经过 `alphabet` 运算后的状态（`state` 可以是组合状态）。

- `void calculateStates(string state)`

查找当前的 `outputStates` 经过运算后是否会产生新的元素，并将新的元素加入到 `outputStates` 中。

功能函数具体实现：

在 `主函数` 中，先读入规定的输入格式，再调用 `calculateStates`，生成所有的 `outputStates`，不断更新 `outputStates` 中的状态数目，最后按照输出格式输出 dfa $M = (Q, T, \delta, q_0, F)$ 。

在 `calculateStates(string state)` 函数中，我们的先调用 `calculate` 函数，计算 `state` 经过 `inputAlphabet[i]` 运算后的所有组合状态（也可以只有一个状态），所得的 `temp` 即为从 NFA 转换成的等价的 DFA 状态集。

```
void calculateStates(string state)
{
    vector<string>::iterator j;

    for (int i = 0; i < numberOfLetters; i++)
    {
        string temp = calculate(state, inputAlphabets[i]);
        if (find(outputStates.begin(), outputStates.end(), temp) == outputStates.end())
        {
            outputStates.push_back(temp);
        }
    }
}
```

在 if 语句中，检查计算出的组合状态 temp（即当前 state 和第 i 个运算在 dfa 中表现出的结果）与已生成的 outputStats 是否有重复，如果不重复则将 temp 加入到 outputStates 里面。

在 `string calculate(string state, string alphabet)` 函数中，首先检查 state 是否是组合状态（我们规定，将组合的状态以“-”连接，例如 q0-q1-q2），以便能够读取有“-”连接的组合状态和单独的一个状态。

```
string calculate(string state, string alphabet)
{
    set<int> bla;
    if (state.find('-') == string::npos)
    {
        int count = 0;
        string temp = "";
        for (int i = 0; i < numberOfTransitions; i++)
        {
            if (transitionTable[i][0] == state && transitionTable[i][1] == alphabet && transitionTable[i][2] != "-")
            {
                bla.insert(stoi(transitionTable[i][2].substr(1, transitionTable[i][2].length() - 1)));
            }
        }
        set<int>::iterator it;
        for (it = bla.begin(); it != bla.end(); it++)
        {
            int f = *it;
            if (it == bla.begin())
            {
                temp += "q" + to_string(f);
            }
            else
            {
                temp += "-q" + to_string(f);
            }
        }
        return temp;
    }
}
```

而两者计算经过 alphabet 运算后的状态的算法一致，这里我们主要详细介绍在单独状态下的算法代码。

模块 1 根据我们输入的 delta 转移函数，实现了计算 state 在 alphabet 运算后的状态。substr 返回得到的状态 q_n 对应的 n，stoi 将字符串转换为整型 n，用 insert 函数将 n 输入进 bla 容器中。

模块 2 用迭代器遍历 bla 这个集合，生成字符串 q_i-q_j 并返回 temp（以“-”连接的组合状态同理）。

```
else
{
    string temp = "";
    int count = 0;
    stringstream ss(state);
    string token;
    while (getline(ss, token, '-'))
    {
        for (int i = 0; i < numberOfTransitions; i++)
        {
            if (transitionTable[i][0] == token && transitionTable[i][1] == alphabet && transitionTable[i][2] != "-")
            {
                bla.insert(stoi(transitionTable[i][2].substr(1, transitionTable[i][2].length() - 1)));
            }
        }
    }
    set<int>::iterator it;
    for (it = bla.begin(); it != bla.end(); it++)
    {
        int f = *it;
        if (it == bla.begin())
        {
            temp += "q" + to_string(f);
        }
        else
        {
            temp += "-q" + to_string(f);
        }
    }
}
```

5 输入输出及执行效果

- 输入：input.txt 文件，格式如下

```
4,3
Q = {q0,q1,q2,q3}
I = {q0}
F = {q3}
Sigma = {a,b,c}
Delta(q0,a) = q1
Delta(q0,a) = q2
Delta(q1,b) = q1
Delta(q1,b) = q3
Delta(q1,c) = q2
Delta(q2,a) = q3
Delta(q3,b) = q2
```

其中，第一行是状态数目与输入字母表中字母数目；

第二行是 NFA 全部的状态集；

第三行是初始状态；

第四行是接受状态集；

第五行是字母表；

第六行及之后是 NFA 转换函数。

- 输出：output.txt 文件，格式如下：

```
Q = {q0,q1-q2,q3,q1-q3,q2,q1-q2-q3}
Init = {q0}
F = {q3,q1-q3,q1-q2-q3}
Sigma = {a,b,c}
Delta(q0, a) = {q1-q2}
Delta(q1-q2, a) = {q3}
Delta(q1-q2, b) = {q1-q3}
Delta(q1-q2, c) = {q2}
Delta(q3, b) = {q2}
Delta(q1-q3, b) = {q1-q2-q3}
Delta(q1-q3, c) = {q2}
Delta(q2, a) = {q3}
Delta(q1-q2-q3, a) = {q3}
Delta(q1-q2-q3, b) = {q1-q2-q3}
Delta(q1-q2-q3, c) = {q2}
```

其中，第一行是 DFA 所有状态集；

第二行是初始状态；

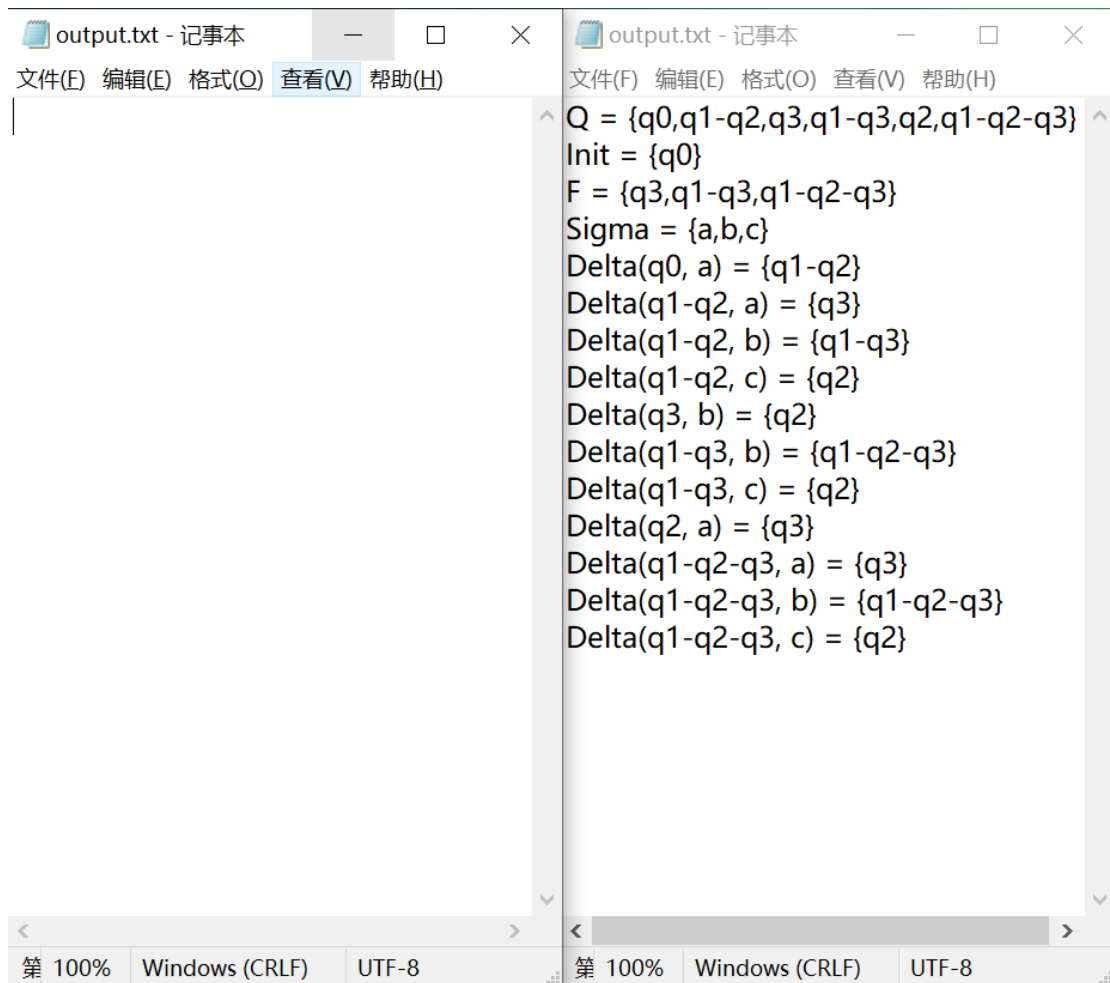
第三行是接受状态集；

第四行是字母表；

第五行及之后是 DFA 状态转移函数。

- 执行效果

编译 cpp 文件并运行，结果被写入 output.txt 文件中。



```
Q = {q0,q1-q2,q3,q1-q3,q2,q1-q2-q3}
Init = {q0}
F = {q3,q1-q3,q1-q2-q3}
Sigma = {a,b,c}
Delta(q0, a) = {q1-q2}
Delta(q1-q2, a) = {q3}
Delta(q1-q2, b) = {q1-q3}
Delta(q1-q2, c) = {q2}
Delta(q3, b) = {q2}
Delta(q1-q3, b) = {q1-q2-q3}
Delta(q1-q3, c) = {q2}
Delta(q2, a) = {q3}
Delta(q1-q2-q3, a) = {q3}
Delta(q1-q2-q3, b) = {q1-q2-q3}
Delta(q1-q2-q3, c) = {q2}
```

执行前

执行后

6. 改进思路和方法

- 鲁棒性

程序缺少对一些错误输入的检验，可以加入一些检验函数，来判断输入是否合法，增加程序的鲁棒性。

- 效率问题

在 `calculateStates` 函数的计算过程中，每次循环都要把状态转移表重新遍历一遍，影响了程序的运行效率。可以考虑再对每个初始状态都生成各自状态转移表，具体实现可以用 `struct` 实现。

● 局限性

输入状态集只能是 q 加数字的形式，这虽然简化了程序设计，但限制了 NFA 的功能性与兼容性。可以使用字符串来为 state 命名，再用 map 将每个 state 与一个数字对应起来，方便计算。

并且程序仅仅限于不含空转移的 NFA 转换，之后可以在深入研究 ϵ -NFA 到 DFA 的转换。

● 图形化

使用 Python 的 networkx, pydot 库，编写脚本，并配合 graphviz 可视化图形工具画出 NFA 转换成的 DFA 的状态转移图，可以帮助使用者更加直观地读懂有限状态机。