杭州电子科技大学

《编译原理课程实践》

实验报告

题　　目：正则表达式转最小 DFA 算法

学　　院：计算机

专　　业：计算机科学与技术

班　　级：22052312

学　　号：22050233

姓　　名：郑方昊

完成日期：2024.11.17

一、 实验目的

1. 掌握正规表达式与有限自动机的基本概念和转换方法。
2. 了解非确定有限自动机（NFA）的构建过程。
3. 熟悉编程实现正规表达式到 NFA 转换的算法。
4. 提高编程能力和算法设计的技能。
5. 掌握非确定有限自动机与确定有限自动机的基本概念及其转换方法。
6. 了解 NFA 到 DFA 转换过程中的子集构造算法。
7. 实现 NFA 到 DFA 的转换算法，并验证 DFA 的正确性。
8. 设计合理的数据结构，以便为后续 DFA 最小化实验任务做好准备。
9. 掌握确定有限自动机（DFA）的最小化原理和算法，尤其是 Hopcroft 算法。
10. 学习 DFA 状态等价性的判定方法，理解最小化过程中的分割和合并策略。
11. 实现 DFA 最小化算法，并验证最小化 DFA 的正确性。
12. 延续前两次实验的设计，确保数据结构能贯通整个自动机系列实验。
13. 提高算法优化和编程实现能力，增强对编译原理的理解。

二、 实验内容与实验要求

2.1 正则表达式转 NFA

2.1.1 实验内容：

1. 理论背景：正规表达式是一种用于描述词法单元的形式化表示法，而NFA 是一种用于词法分析的状态机。正规表达式可以通过算法转化为 NFA，从而实现对字符串的模式匹配。

2. 任务描述：实现正规表达式到 NFA 的转换算法，并验证生成的 NFA 对给定输入字符串的接受性。同时，设计适合 NFA 的数据结构，为后续 NFA 转 DFA、DFA 最小化等实验任务提供基础支持。

3. ** 实验步骤**：

（1）解析输入的正规表达式。

（2）构建对应的 NFA，包括处理基本符号、连接、并联（或操作）、闭包（星号操作）等运算。

（3）设计并实现合理的数据结构表示 NFA，如状态集合、转移关系、初始状态和接受状态。

（4）对 NFA 进行模拟，验证其是否接受给定的输入字符串。

4. 案例分析：给定一个简单的正规表达式（如 a(b|c)* ），手动推导其 NFA，并用程序实现自动生成 NFA 的过程。

2.1.2 实验要求：

1. 输入输出要求：

输入：正规表达式和多个测试字符串。

输出：生成的 NFA 状态集合及其转换关系，指明每个测试字符串是否被 NFA 接受。

2. 算法要求：

支持基本的正规表达式运算符，如连接（ ab ）、或（a|b ）、闭包（ a*` ）。

实现 Thompson 构造法，将正规表达式分解为基本操作，然后逐步合成 NFA。

3. 数据结构要求：

（1）设计合理的数据结构来表示 NFA（如图的表示方式），应包括状态集、状态转移表、初始状态和接受状态的表示。
（2）数据结构需具备扩展性，以便在后续实验中使用，如 NFA 到 DFA 的转换、DFA 的最小化。
（3）考虑实现状态的唯一标识符，支持对状态进行增删查操作的高效实现。
4. 程序要求：
使用C/C++、Java、Python 等语言编写程序，代码结构清晰，具备良好的注释。
提供详细的实验报告，包括算法设计、实现过程、测试结果和问题分析。

2.2 NFA 转 DFA

2.2.1 实验内容：

1. 理论背景：NFA 是一种可以处理多条路径的状态机，而DFA 是其确定版本，不存在多条路径。通过子集构造算法（Subset Construction），可以将 NFA 转换为等价的 DFA，从而实现字符串匹配的确定性处理。
2. 任务描述：实现将 NFA 转换为 DFA 的算法，并对转换后的 DFA 进行验证。同时，设计适合 DFA 的数据结构，使其兼容前一次实验的 NFA 数据结构。
3. 实验步骤：
（1）理解子集构造算法的原理，包括 ε -闭包的计算和状态集合的映射。
（2）利用子集构造算法，将 NFA 转换为 DFA。
（3）设计并实现 DFA 的数据结构，确保其能够表示状态集合、状态转换、初始状态和接受状态。
（4）验证 DFA 的正确性，对比DFA 与 NFA 在同一组测试输入上的匹配结果。

2.2.2 实验要求：

1. 输入输出要求
输入：一个 NFA（包括状态集、转换表、初始状态和接受状态集合）和多个测试字符串。
输出：生成的 DFA 状态集合及其转换关系，指明每个测试字符串是否被 DFA 接受。
2. 算法要求
实现子集构造算法，将 NFA 状态集合的子集映射为 DFA 的单个状态。处理 ε -闭包及其状态转换，生成对应的 DFA。
3. 数据结构要求
（1）在上一实验的基础上，设计 DFA 的数据结构，包含状态集合、转换关系、初始状态和接受状态集合的表示。
（2）确保数据结构可以支持后续的 DFA 最小化任务，便于后续实验任务的延续。
4. 程序要求
使用C/C++、Java、Python 等语言编写程序，代码结构清晰，具备良好的注释。
提供详细的实验报告，包括算法设计、实现过程、测试结果和问题分析。

2.3 DFA 最小化

2.3.1 实验内容：

1. 理论背景：DFA 最小化是将 DFA 状态数减少到最小的过程，通过合并等价状态，实现最优的状态机表示。Hopcroft 算法是求异法的一种高效实现，它通过维护状态的分割并使用快速查找机制来优化最小化过程。

2. 任务描述：实现 DFA 最小化算法，将给定的 DFA 简化为状态数最少的等价 DFA。验证最小化 DFA 的正确性，并对比最小化前后的状态数量。

3. 实验步骤

（1）理解 Hopcroft 算法的基本原理，包括状态等价的判定标准和状态合并的方法。

（2）实现 Hopcroft 算法，将原 DFA 简化为等价的最小化 DFA。

（3）设计合理的数据结构表示最小化后的 DFA，确保其与前两次实验的 NFA 和 DFA 数据结构保持一致。

（4）验证最小化 DFA 的正确性，确保其接受的语言与原 DFA 相同。

2.3.2 实验要求：

1. 输入输出要求

输入：一个 DFA（包括状态集合、状态转换表、初始状态和接受状态集合）。

输出：最小化后的 DFA 状态集合及其转换关系，指明最小化前后的状态数和状态转换关系。

2. 算法要求

实现 Hopcroft 算法，通过分割状态集合和快速查找机制来最小化 DFA。支持状态等价性判定及状态的合并操作。

3. 数据结构要求

（1）设计适合 Hopcroft 算法的高效数据结构，如用于记录状态分割的集合、合并后的状态转换表等。

（2）保持与前两次实验的数据结构一致，方便整个自动机系列实验的贯通实现。

4. 程序要求

使用C/C++、Java、Python 等语言编写程序，代码结构清晰，具备良好的注释。

提供详细的实验报告，包括算法设计、实现过程、测试结果和问题分析。

三、 设计方案与算法描述

3.1 正则表达式转 NFA

3.1.1 设计方案：

在此实验中，我使用一种常见的算法，即 Thompson 构造法，来将正则表达式（RE）转换为非确定性有限自动机（NFA）。通过构造状态和边来表示正则表达式的各个部分，并将基本操作（如拼接、选择和闭包）组合成完整的 NFA。并且还使用 graph 工具来进行可视化的实现。

输入：正则表达式字符串。

输出：对应的 NFA 结构，包括状态集合、转移关系、后缀表达式、起始状态和接受状态。

3.1.2 算法描述：

（1）字符处理（act_Elem）：

对于每个字符，创建一个由两个状态（起始状态和终止状态）组成的简单 NFA 单元。

使用该字符在这两个状态之间创建转换边。

（2）选择操作（act_Unit）：

对于正则表达式中的选择操作（如 a|b），我们需要引入一个新的起始状态和终止状态，并且通过从新起始状态到 a 和 b 分别转换，再从 a 和 b 到新终止状态的转换来实现。

（3）拼接操作（act_join）：

对于正则表达式中的拼接操作（如 ab），我们将一个 NFA 的终止状态与另一个 NFA 的起始状态连接，通过建立从第一个 NFA 的终止状态到第二个 NFA 的起始状态的 ε 转换来实现。

（4）闭包操作（act_star）：
对于正则表达式中的闭包操作（如 a*），我们需要创建一个带有 ε 转换的循环结构：通过增加一个新的起始状态和终止状态，并添加适当的 ε 转换（从起始状态到终止状态、从终止状态回到起始状态）来表示零次或多次重复。
（5）后缀表达式处理：
使用 infixToPostfix 类将输入的中缀正则表达式转换为后缀表达式。这使得表达式处理更加简化，因为后缀表达式的运算顺序已经显式确定。
（6）NFA 构建：
根据后缀表达式中的每个元素（字符、选择、拼接、闭包等），逐步调用上述操作构建对应的 NFA 单元，最后将这些单元合成一个完整的 NFA。

3.2 正则表达式转 NFA

3.2.1 设计方案：
将非确定性有限自动机（NFA）转换为确定性有限自动机（DFA）。NFA 的状态之间可能没有明确的转换（即某些状态可能没有下一个状态或存在多个下一个状态），通过子集构造法我们能够将 NFA 的状态集合作为 DFA 的单个状态，从而避免了多路径的非确定性。
输入：对应的 NFA（包括状态集合、转移关系、初始状态和接受状态）。
输出：对应的 DFA（包括状态集合、转移关系、起始状态和接受状态）。
3.2.2 算法描述：
（1） ε 闭包计算（eClosure）：
计算一个给定状态集合的 ε-闭包。ε-闭包包括从该状态集合出发，通过 ε 转换可以到达的所有状态。此步骤是将 NFA 转换为 DFA 的关键,因为一个 DFA 的状态可以由多个 NFA 状态的集合表示。
（2）状态转移计算（move）：
对于给定的 DFA 状态集合和输入符号,通过检查 NFA 中每个状态的转移符号来计算下一个状态集合。
（3）DFA 状态集合：
初始状态是 NFA 起始状态的 ε-闭包。然后通过对每个符号的移动计算，生成所有可能的状态集合，这些状态集合作为 DFA 的新状态。
（4）DFA 转移关系：
对于每个 DFA 状态（由多个 NFA 状态组成），根据输入符号计算出所有可能的状态集合，并将每个状态集合作为 DFA 状态的转移。
（5）DFA 的接受状态：
如果 DFA 状态集合中包含 NFA 的任何接受状态，则该 DFA 状态也是接受状态。
（6）DFA 状态检查：
在构建 DFA 时，检查每个新状态是否已经存在于 DFA 状态集合中，避免状态重复，确保每个状态都是唯一的。

3.3 正则表达式转 NFA

3.3.1 设计方案：
DFA 最小化的目标是通过合并等价状态来减少 DFA 的状态数，使 DFA 更加简洁和高效。Hopcroft 算法是一种求异法的高效实现，通过分割状态集合和快速查找机制来优化最小化过程。
输入：对应的 DFA（包括状态集合、转移关系、起始状态和接受状态）。

输出：最小化后的 DFA（包括状态集合、转移关系、起始状态和接受状态）。

3.3.2 算法描述：

（1）初步分组：

将 DFA 的所有状态初步分为两个组：接受状态组和非接受状态组。

初始分组的状态可以通过 DFA 的接受状态和非接受状态进行区分。

（2）状态分割：

使用 Hopcroft 算法的核心思想，通过状态的转移关系进一步细分初步分组中的状态。

对于每个状态组，检查每个状态的转移符号对各个子组的影响。如果某些状态转移后到达的子组不同，则将这些状态拆分成不同的子组。

（3）重复细分：

重复状态细分过程，直到无法进一步细分为止。Hopcroft 算法使用一个高效的数据结构（如队列）来维护待处理的状态组，确保每次处理最有可能被细分的状态。

（4）最小化 DFA 的构建：

将每个状态组视为最小化后的 DFA 的新状态，原 DFA 中的转移关系根据合并后的状态组进行更新。对于每个状态组中的状态，确保所有状态之间的转换关系是一致的。

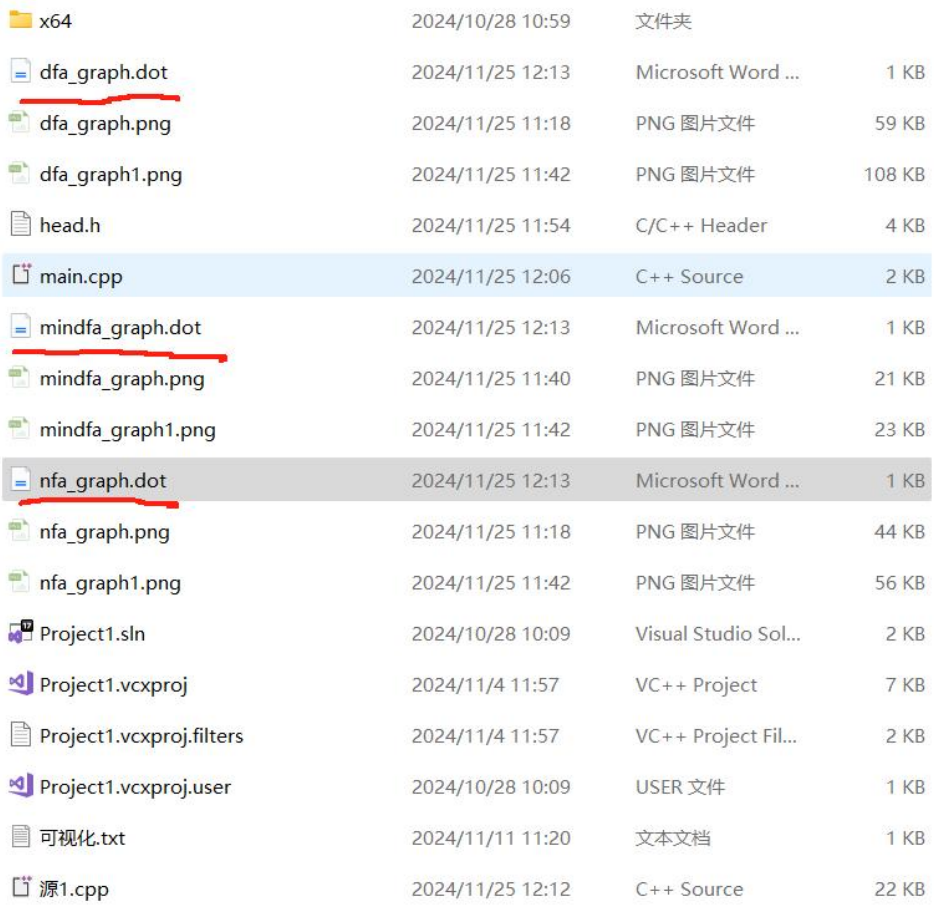
（5）输出最小化后的 DFA：

最终输出最小化后的 DFA，包括新状态集合、转移关系、起始状态和接受状态。

四、 测试结果

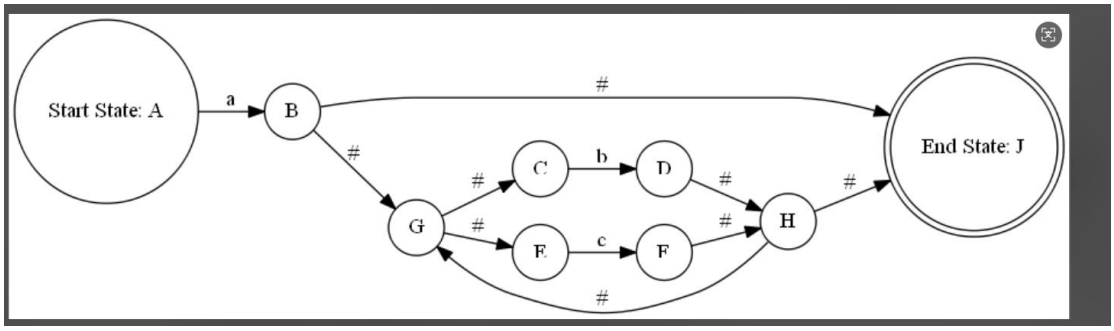
由于我把三个小实验是合并起来写了，所以我在下面统一展示，在本实验中我采取了两个测试案例

可视化是使用了 dot -Tpng nfa_graph.dot -o nfa_graph.png，将 dot 文件转化为图片

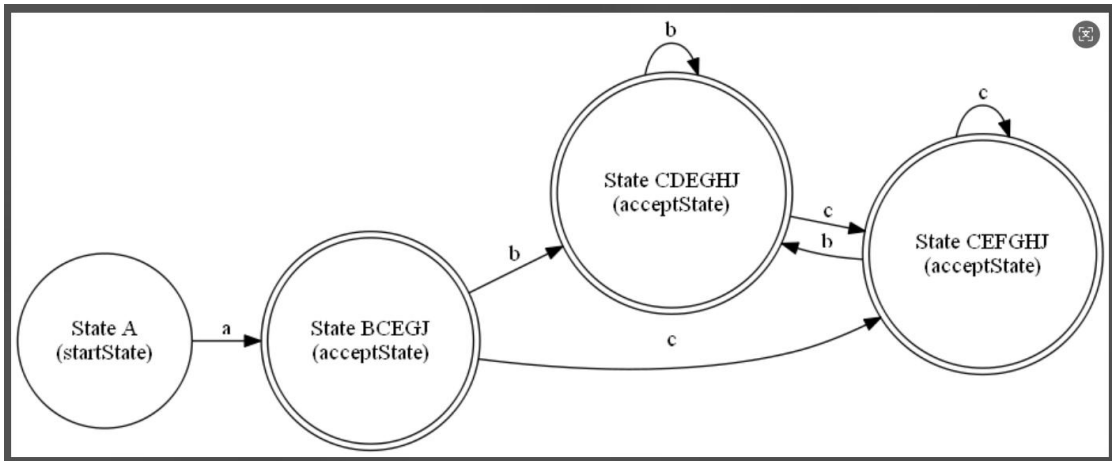


1. a(b|c)*

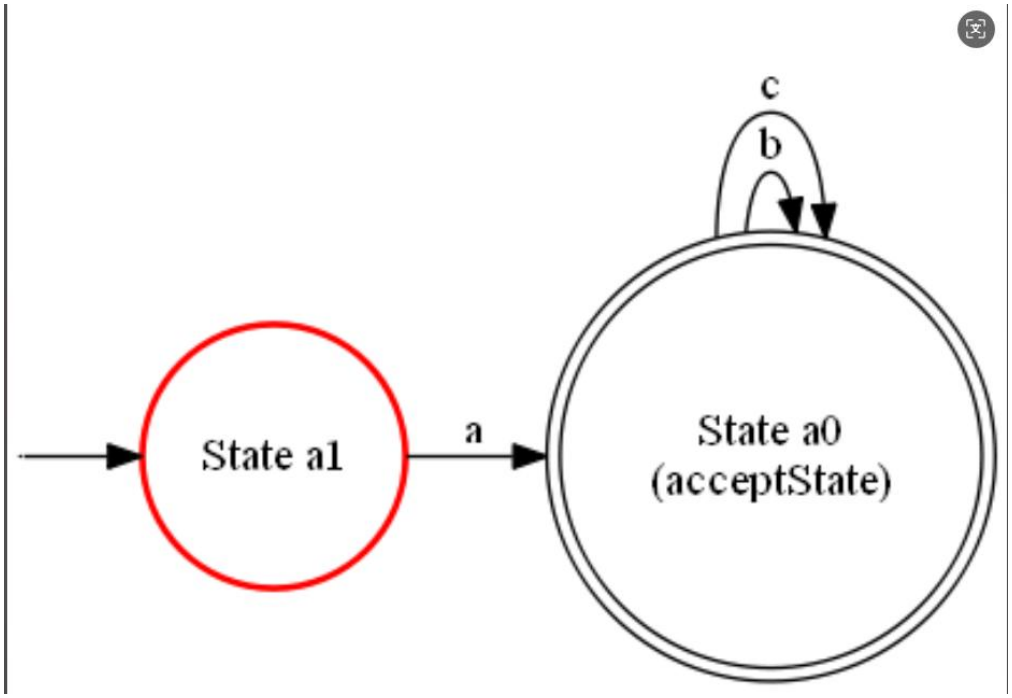
（1）NFA 可视化：



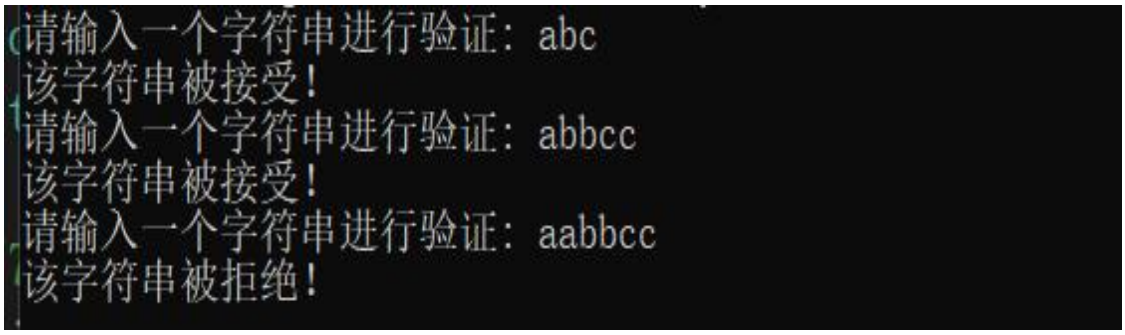
（2）DFA 可视化：



（3）最小化 DFA 可视化：



（4）字符串测试结果（abc,abbcc,aabbcc）：



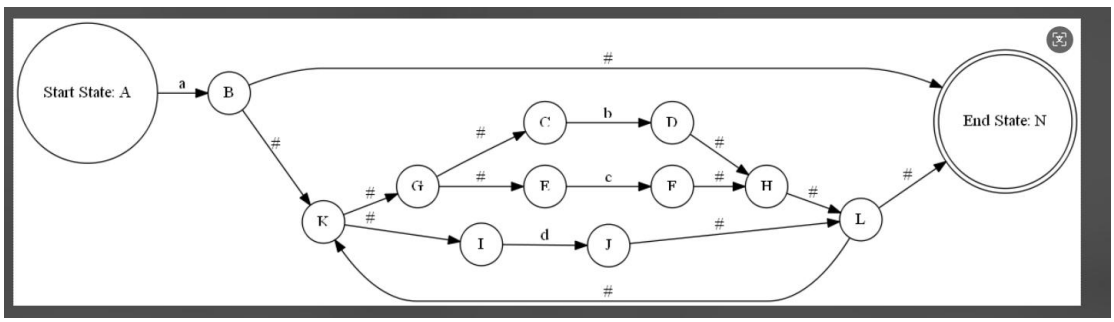
可以发现前两个字符串可以接受，最后一个 aabbcc 字符串不被接受

（5）正则表达式转 NFA，NFA 转 DFA 以及 DFA 最小化得全过程：

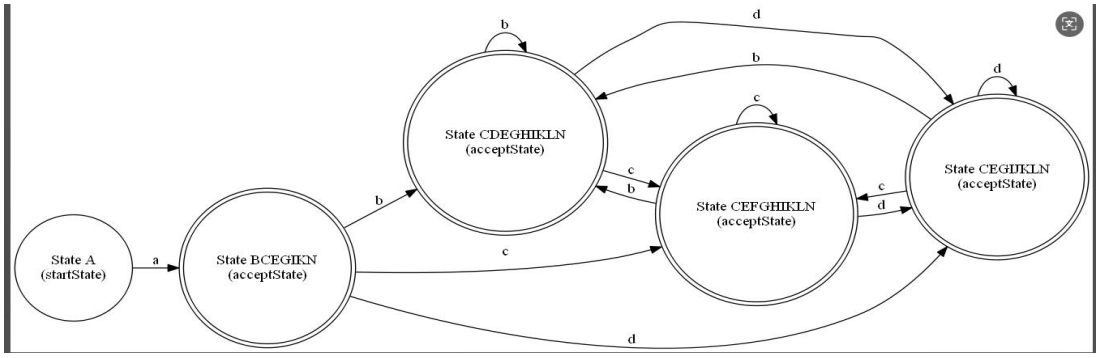
```
输入正则表达式：   （操作符：() * |;字符集：a~z A~Z）
a(b|c)*
加'+'后的表达式：a+(b|c)*
后缀表达式为：abc|*+
NFA States:
Start State: A
End State: J
NFA Transitions:
Edge 1: A --a--> B
Edge 2: C --b--> D
Edge 3: E --c--> F
Edge 4: G --#--> C
Edge 5: G --#--> E
Edge 6: D --#--> H
Edge 7: F --#--> H
Edge 8: B --#--> J
Edge 9: H --#--> G
Edge 10: B --#--> G
Edge 11: H --#--> J
End
NFA DOT file generated successfully.
DFA States:
State A (NFA States: A ) (Non-Accepting State)
State BCEGJ (NFA States: B C E G J ) (Accepting State)
State CDEGHJ (NFA States: C D E G H J ) (Accepting State)
State CEFGHJ (NFA States: C E F G H J ) (Accepting State)

DFA Transitions:
State A --a--> State BCEGJ
State BCEGJ --b--> State CDEGHJ
State BCEGJ --c--> State CEFGHJ
State CDEGHJ --b--> State CDEGHJ
State CDEGHJ --c--> State CEFGHJ
State CEFGHJ --b--> State CDEGHJ
State CEFGHJ --c--> State CEFGHJ
DFA DOT file generated successfully.
DFA 状态集：
状态：a0 (NFA States: BCEGJ CDEGHJ CEFGHJ ) (接受状态)
状态：a1 (NFA States: A ) (非接受状态)
DFA 转移关系：
a1 --a--> a0
a0 --b--> a0
a0 --c--> a0
DFA DOT file generated successfully.
```

2. a(b|c|d)*

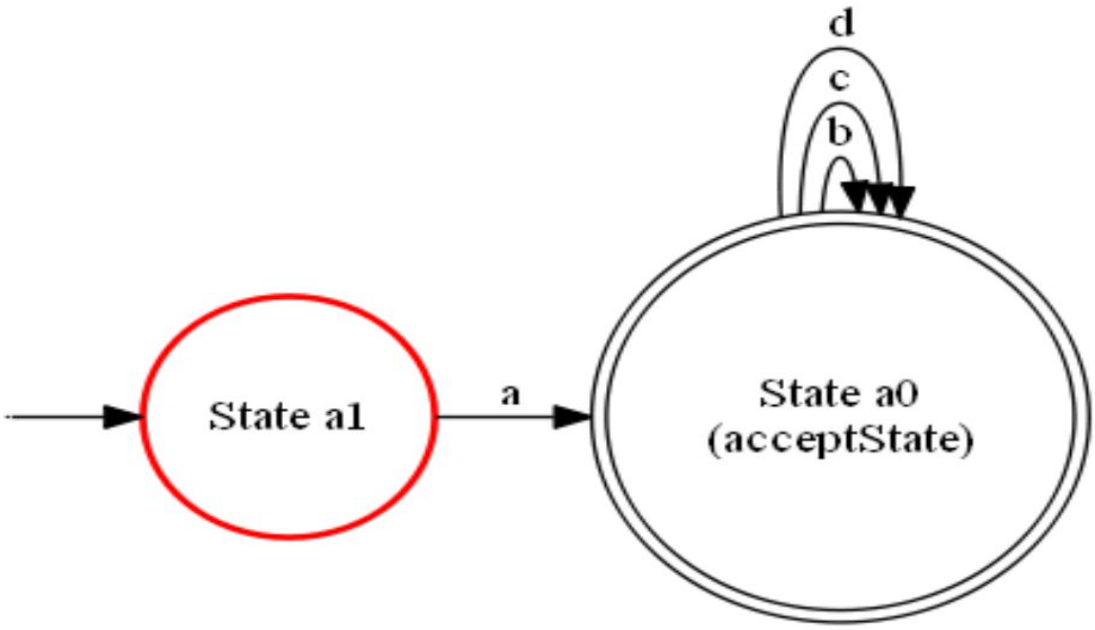
（1）NFA 可视化：



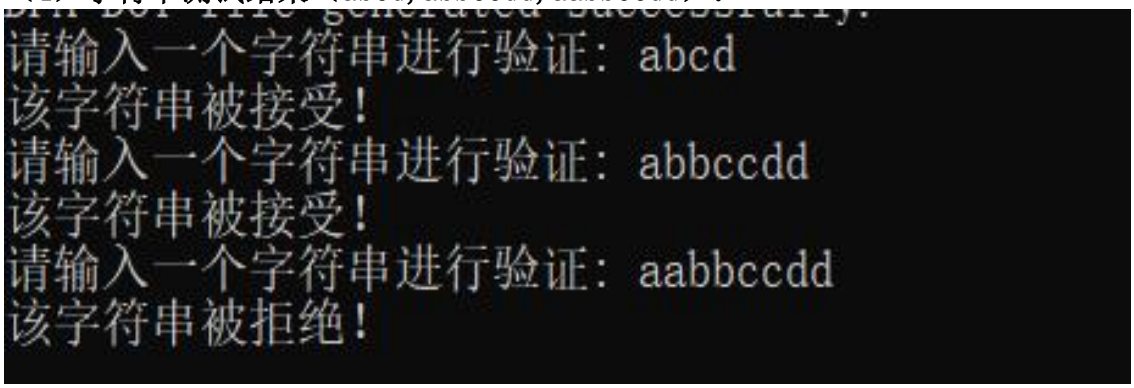
（2）DFA 可视化：



（3）最小化 DFA 可视化：



（4）字符串测试结果（abcd, abbccdd, aabbccdd）：



可以发现前两个字符串可以接受，最后一个 aabbccdd 字符串不被接受

（5）正则表达式转 NFA，NFA 转 DFA 以及 DFA 最小化得全过程：

```
输入正则表达式：   （操作符：() * |;字符集：a~z A~Z）
a(b|c|d)*
加'+'后的表达式：a+(b|c|d)*
后缀表达式为：abc|d|*+
NFA States:
Start State: A
End State: N
NFA Transitions:
Edge 1: A --a--> B
Edge 2: C --b--> D
Edge 3: E --c--> F
Edge 4: G --#--> C
Edge 5: G --#--> E
Edge 6: D --#--> H
Edge 7: F --#--> H
Edge 8: I --d--> J
Edge 9: K --#--> G
Edge 10: K --#--> I
Edge 11: H --#--> L
Edge 12: J --#--> L
Edge 13: B --#--> N
Edge 14: L --#--> K
Edge 15: B --#--> K
Edge 16: L --#--> N
End
NFA DOT file generated successfully.
DFA States:
State A (NFA States: A ) (Non-Accepting State)
State BCEGIKN (NFA States: B C E G I K N ) (Accepting State)
State CDEGHIKLN (NFA States: C D E G H I K L N ) (Accepting State)
State CEFGHIKLN (NFA States: C E F G H I K L N ) (Accepting State)
State CEGIJKLN (NFA States: C E G I J K L N ) (Accepting State)
```

```
DFA Transitions:
State A --a--> State BCEGIKN
State BCEGIKN --b--> State CDEGHIKLN
State BCEGIKN --c--> State CEFGHIKLN
State BCEGIKN --d--> State CEGIJKLN
State CDEGHIKLN --b--> State CDEGHIKLN
State CDEGHIKLN --c--> State CEFGHIKLN
State CDEGHIKLN --d--> State CEGIJKLN
State CEFGHIKLN --b--> State CDEGHIKLN
State CEFGHIKLN --c--> State CEFGHIKLN
State CEFGHIKLN --d--> State CEGIJKLN
State CEGIJKLN --b--> State CDEGHIKLN
State CEGIJKLN --c--> State CEFGHIKLN
State CEGIJKLN --d--> State CEGIJKLN
DFA DOT file generated successfully.
DFA 状态集：
状态: a0 (NFA States: BCEGIKN CDEGHIKLN CEFGHIKLN CEGIJKLN ) (接受状态)
状态: a1 (NFA States: A ) (非接受状态)
DFA 转移关系：
a1 --a--> a0
a0 --b--> a0
a0 --c--> a0
a0 --d--> a0
DFA DOT file generated successfully.
请输入一个字符串进行验证:
```

实验收获：

通过本次实验，我加深了对正则表达式、NFA 和 DFA 之间关系的理解，并掌握了如何从正则表达式构建 NFA、如何将 NFA 转化为 DFA 以及如何进行 DFA 最小化的全过程。特别是在学习 Thompson 构造法和子集构造法的过程中，我不仅掌握了常用的自动机构建技术，还通过实践理解了这些算法在解决实际问题中的应用。Hopcroft 算法的最小化处理让我深入理解了如何优化状态机，减少状态数量，从而提高匹配效率。通过不断调试和优化，我提升了自己

的编程能力、算法设计能力和调试技能，对自动机的设计与实现有了更深的认识。

实验挑战：
本次实验中最大的挑战是处理 NFA 到 DFA 转换过程中的状态爆炸问题。由于 NFA 中的状态可以通过多个路径进行转换，子集构造法将多个 NFA 状态集合并成一个 DFA 状态时，状态数目急剧增加，导致算法复杂度较高。此外，DFA 最小化过程中，如何高效地处理状态的细分与合并，以及如何确保 DFA 与原 NFA 等价，也是一个需要高度注意的问题。在实现 Hopcroft 算法时，如何合理组织数据结构以优化状态划分的效率，确保算法的高效性，也是我遇到的技术难题。

五、 源代码

#head.h

```
#include <iostream>
#include <stdio.h>
#include <cctype>
#include <stack>
#include <string>
#include <map>
#include <set>
#include <vector>
#include <iterator>
#include <fstream>
#include <queue>
#include <sstream>
#include <unordered_map>
#include <unordered_set>

using namespace std;

//NFA 的节点
struct node
{
    string nodeName;

};
//NFA 的边
struct edge
{
    node startName;   //起始点
    node endName;//目标点
    char tranSymbol;  //转换符号
```

```
};
//NFA 的组成单元，一个大的 NFA 单元可以是由很多小单元通过规则拼接起来
struct elem
{
    int edgeCount;    //边数
    edge edgeSet[100];    //该 NFA 拥有的边
    node startName;   //开始状态
    node endName; //结束状态
    char tranSymbol;
};

//创建新节点
node new_node();
//处理 a
elem act_Elem(char);
//处理 a|b
elem act_Unit(elem, elem);
//组成单元拷贝函数
void elem_copy(elem&, elem);
//处理 ab
elem act_join(elem, elem);
//处理 a*
elem act_star(elem);

void input(string&);

string add_join_symbol(string);    //两个单元拼接在一起相当于一个+

//infixToPostfix 类用于将中缀表达式（如 a|b）转换为后缀表达式（逆波兰表示法）
class infixToPostfix {
public:
    infixToPostfix(const string& infix_expression);

    int is_letter(char check);//判断字符是否为字母
    int ispFunc(char c);//获取操作符的栈内优先级
    int icpFunc(char c);//获取操作符的当前优先级
    void inToPost();
    string getResult();//获取转换后的后缀表达式

private:
    string infix;//存储原始的中缀表达式
    string postfix;//存储转换后的后缀表达式
    map<char, int> isp;//用于操作符优先级的映射（isp 是栈顶优先级，icp 是当前操作数优先级）
```

```
    map<char, int> icp;
};

elem express_to_NFA(string);//将正则表达式字符串转换为 NFA

void Display(elem);//显示 NFA 的状态和转移信息

void generateDotFile_NFA(const elem& nfa);//将 NFA 转换为 Dot 格式，供图形化工具（如
Graphviz）使用

// 定义 DFA 的状态
struct DFAState {
    set<string> nfaStates;//NFA 的状态集合
    set<string> originalStates;  // 记录原始 NFA 状态集合
    string stateName;//DFA 的状态名称
    bool   isAccept=false;//判断是否为接受状态
    // 比较两个 DFAState 是否相等，供 set 排序使用
    bool operator<(const DFAState& other) const {
        return stateName < other.stateName; // 根据状态名进行排序
    }
;

};
// 定义 DFA 的转换关系
struct DFATransition {
    DFAState fromState;
    DFAState toState;
    char transitionSymbol;
};
// 计算 NFA 状态的 ε 闭包
DFAState eClosure(const set<string>& nfaStates, elem nfa);

// 计算 DFA 的状态转移
DFAState move(const DFAState& dfaState, char transitionSymbol, elem nfa);

// 检查 DFA 状态是否在状态集合中
bool isDFAStateInVector(const vector<DFAState>& dfaStates, const DFAState& targetState);

//检查转换边是否在边集合中，比如 a->b 是否已经在集合中
bool isTransitionInVector(DFAState, DFAState, char, vector<DFATransition>);

//NFA 转换为 DFA
void buildDFAFromNFA(const elem& NFA_Elem, vector<DFAState>& dfaStates,
vector<DFATransition>& dfaTransitions);
```

```
// 显示 DFA 状态和转移关系
void displayDFA(const vector<DFAState>& dfaStates, const vector<DFATransition>&
dfaTransitions);

//生成 dot 文件
void generateDotFile_DFA(vector<DFAState>& dfaStates, vector<DFATransition>&
dfaTransitions);

bool areStatesEqual(const set<string>& s1, const set<string>& s2);

void minimizeDFA(vector<DFAState>& dfaStates, vector<DFATransition>& dfaTransitions);

void displayminDFA(const vector<DFAState>& dfaStates, const vector<DFATransition>&
dfaTransitions);

void generateDotFile_minDFA(vector<DFAState>& dfaStates, vector<DFATransition>&
dfaTransitions);

    bool simulateDFA(const vector<DFAState>& dfaStates, const vector<DFATransition>&
    dfaTransitions, const string& inputString);
```

#main.c

```
#include "head.h"

int main() {
    int i = 3;
    string Regular_Expression;
    elem NFA_Elem;
    //1.输入正则表达式
    input(Regular_Expression);
    if (Regular_Expression.length() > 1)    Regular_Expression =
add_join_symbol(Regular_Expression);
    infixToPostfix Solution(Regular_Expression);
    //2.中缀转后缀
    cout << "后缀表达式为：";
    Regular_Expression = Solution.getResult();
    cout << Regular_Expression << endl;
    //3.表达式转 NFA
    NFA_Elem = express_to_NFA(Regular_Expression);
    //4.打印 NFA
    Display(NFA_Elem);
    //5.生成 NFAdot 文件
```

```
    generateDotFile_NFA(NFA_Elem);

    // 6.初始化 DFA 状态集合和转换关系
    vector<DFAState> dfaStates; //用于存储所有的 DFA 状态
    vector<DFATransition> dfaTransitions; //用于存储 DFA 状态之间的转移
    set<string> nfaInitialStateSet;   //存储 NFA 的初始状态

    //7.从 NFA 构造 DFA
    buildDFAFromNFA(NFA_Elem, dfaStates, dfaTransitions);
    //8.显示 DFA
    displayDFA(dfaStates, dfaTransitions);

    //9.生成 DFAdot 文件
    generateDotFile_DFA(dfaStates, dfaTransitions);

    //10.进行 DFA 最小化
    minimizeDFA(dfaStates, dfaTransitions);

    //11.显示最小化后的 DFA
    displayminDFA(dfaStates, dfaTransitions);
    //12.生成 minDFAdot 文件
    generateDotFile_minDFA(dfaStates, dfaTransitions);
    // 13. 模拟 DFA 验证输入字符串的接受性
    while (i)
    {
        string inputString;
        cout << "请输入一个字符串进行验证: ";
        cin >> inputString;

        bool isAccepted = simulateDFA(dfaStates, dfaTransitions, inputString);
        if (isAccepted) {
            cout << "该字符串被接受！" << endl;
        }
        else {
            cout << "该字符串被拒绝！" << endl;
        }
        i--;
    }

    return 0;
}
```

##源 1.c

```
#include "head.h"
```

```
int nodeNum = 0;

//创建新节点
node new_node()
{
    node newNode;
    newNode.nodeName = nodeNum + 65;//将名字用大写字母表示
    nodeNum++;
    return newNode;
}

//接收输入正规表达式
void input(string& RE)
{
    cout << "输入正则表达式：   （操作符：() * |;字符集：a~z A~Z）" << endl;
    cin >> RE;
}

//组成单元拷贝函数
void elem_copy(elem& dest, elem source)
{
    for (int i = 0; i < source.edgeCount; i++) {
        dest.edgeSet[dest.edgeCount + i] = source.edgeSet[i];
    }
    dest.edgeCount += source.edgeCount;
}

//处理 a
elem act_Elem(char c)
{
    //新节点
    node startNode = new_node();
    node endNode = new_node();

    //新边
    edge newEdge;
    newEdge.startName = startNode;
    newEdge.endName = endNode;
    newEdge.tranSymbol = c;

    //新 NFA 组成元素（小的 NFA 元素/单元）
    elem newElem;
    newElem.edgeCount = 0;//初始状态
```

```
    newElem.edgeSet[newElem.edgeCount++] = newEdge;
    newElem.startName = newElem.edgeSet[0].startName;
    newElem.endName = newElem.edgeSet[0].endName;

    return newElem;
}
//处理 a|b
elem act_Unit(elem fir, elem sec)
{
    elem newElem;
    newElem.edgeCount = 0;
    edge edge1, edge2, edge3, edge4;

    //获得新的状态节点
    node startNode = new_node();
    node endNode = new_node();

    //构建 e1（连接起点和 AB 的起始点 A）
    edge1.startName = startNode;
    edge1.endName = fir.startName;
    edge1.tranSymbol = '#';

    //构建 e2（连接起点和 CD 的起始点 C）
    edge2.startName = startNode;
    edge2.endName = sec.startName;
    edge2.tranSymbol = '#';

    //构建 e3（连接 AB 的终点和终点）
    edge3.startName = fir.endName;
    edge3.endName = endNode;
    edge3.tranSymbol = '#';

    //构建 e4（连接 CD 的终点和终点）
    edge4.startName = sec.endName;
    edge4.endName = endNode;
    edge4.tranSymbol = '#';

    //将 fir 和 sec 合并
    elem_copy(newElem, fir);
    elem_copy(newElem, sec);

    //新构建的 4 条边
    newElem.edgeSet[newElem.edgeCount++] = edge1;
    newElem.edgeSet[newElem.edgeCount++] = edge2;
```

```
    newElem.edgeSet[newElem.edgeCount++] = edge3;
    newElem.edgeSet[newElem.edgeCount++] = edge4;

    newElem.startName = startNode;
    newElem.endName = endNode;

    return newElem;
}
//处理 N(s)N(t)
elem act_join(elem fir, elem sec)
{
    //将 fir 的结束状态和 sec 的开始状态合并，将 sec 的边复制给 fir，将 fir 返回
    //将 sec 中所有以 StartState 开头的边全部修改
    for (int i = 0; i < sec.edgeCount; i++) {
        if (sec.edgeSet[i].startName.nodeName.compare(sec.startName.nodeName) == 0)
        {
            sec.edgeSet[i].startName = fir.endName; //该边 e1 的开始状态就是 N(t)的起始
状态
        }
        else if (sec.edgeSet[i].endName.nodeName.compare(sec.startName.nodeName) == 0) {
            sec.edgeSet[i].endName = fir.endName; //该边 e2 的结束状态就是 N(t)的起始状
态
        }
    }
    sec.startName = fir.endName;

    elem_copy(fir, sec);

    //将 fir 的结束状态更新为 sec 的结束状态
    fir.endName = sec.endName;
    return fir;
}
//处理 a*
elem act_star(elem Elem)
{
    elem newElem;
    newElem.edgeCount = 0;
    edge edge1, edge2, edge3, edge4;

    //获得新状态节点
    node startNode = new_node();
    node endNode = new_node();

    //e1
```

```
    edge1.startName = startNode;
    edge1.endName = endNode;
    edge1.tranSymbol = '#';    //闭包取空串

    //e2
    edge2.startName = Elem.endName;
    edge2.endName = Elem.startName;
    edge2.tranSymbol = '#';

    //e3
    edge3.startName = startNode;
    edge3.endName = Elem.startName;
    edge3.tranSymbol = '#';

    //e4
    edge4.startName = Elem.endName;
    edge4.endName = endNode;
    edge4.tranSymbol = '#';

    //构建单元
    elem_copy(newElem, Elem);

    //将新构建的四条边加入 EdgeSet
    newElem.edgeSet[newElem.edgeCount++] = edge1;
    newElem.edgeSet[newElem.edgeCount++] = edge2;
    newElem.edgeSet[newElem.edgeCount++] = edge3;
    newElem.edgeSet[newElem.edgeCount++] = edge4;

    //构建 NewElem 的启示状态和结束状态
    newElem.startName = startNode;
    newElem.endName = endNode;

    return newElem;
}
//判断是否有字母
int is_letter(char check) {
    if (check >= 'a' && check <= 'z' || check >= 'A' && check <= 'Z')
        return true;
    return false;
}
//添加连接符号
string add_join_symbol(string add_string)
{
    int length = add_string.size();
```

```
    int return_string_length = 0;
    char* return_string = new char[2 * length + 2];//最多是两倍
    char first, second;
    for (int i = 0; i < length - 1; i++)
    {
        first = add_string.at(i);
        second = add_string.at(i + 1);
        return_string[return_string_length++] = first;
        //要加的可能性如ab 、 *b 、 a( 、 )b 等情况
        //若第二个是字母、第一个不是'('、'|'都要添加
        if (first != '(' && first != '|' && is_letter(second))
        {
            return_string[return_string_length++] = '+';
        }
        //若第二个是'(',第一个不是'|'、'(',也要加
        else if (second == '(' && first != '|' && first != '(')
        {
            return_string[return_string_length++] = '+';
        }
    }
    //将最后一个字符写入second
    return_string[return_string_length++] = second;
    return_string[return_string_length] = '\0';
    string STRING(return_string);
    cout << "加'+'后的表达式：" << STRING << endl;
    return STRING;
}

//类里的各类元素定义
infixToPostfix::infixToPostfix(const string& infix_expression) : infix(infix_expression),
postfix("") {
    isp = { {'+', 3}, {'|', 5}, {'*', 7},  {'(', 1}, {')', 8}, {'#', 0} };
    icp = { {'+', 2}, {'|', 4}, {'*', 6}, {'(', 8}, {')', 1}, {'#', 0} };
}

int infixToPostfix::is_letter(char check) {
    if (check >= 'a' && check <= 'z' || check >= 'A' && check <= 'Z')
        return true;
    return false;
}

int infixToPostfix::ispFunc(char c) {
    int priority = isp.count(c) ? isp[c] : -1;
    if (priority == -1) {
```

```
        cerr << "error: 出现未知符号！" << endl;
        exit(1);  // 异常退出
    }
    return priority;
}

int infixToPostfix::icpFunc(char c) {
    int priority = icp.count(c) ? icp[c] : -1;
    if (priority == -1) {
        cerr << "error: 出现未知符号！" << endl;
        exit(1);  // 异常退出
    }
    return priority;
}

void infixToPostfix::inToPost() {
    string infixWithHash = infix + "#";
    stack<char> stack;
    int loc = 0;
    while (!stack.empty() || loc < infixWithHash.size()) {
        if (is_letter(infixWithHash[loc])) {
            postfix += infixWithHash[loc];
            loc++;
        }
        else {
            char c1 = (stack.empty()) ? '#' : stack.top();
            char c2 = infixWithHash[loc];
            if (ispFunc(c1) < icpFunc(c2)) { // 栈顶操作符优先级低于当前字符，将当前字符入栈
                stack.push(c2);
                loc++;
            }
            else if (ispFunc(c1) > icpFunc(c2)) {  // 栈顶操作符优先级高于当前字符，将栈顶操作符出栈并添加到后缀表达式
                postfix += c1;
                stack.pop();
            }
            else {
                if (c1 == '#' && c2 == '#') { // 遇到两个 #，表达式结束
                    break;
                }
                stack.pop(); //其中右括号遇到左括号时会抵消，左括号出栈，右括号不入栈
                loc++;
            }
```

```
        }
    }
}

string infixToPostfix::getResult() {
    postfix = ""; // 清空结果
    inToPost();
    return postfix;
}

//表达式转NFA处理函数,返回最终的NFA集合
elem express_to_NFA(string expression)
{
    int length = expression.size();
    char element;
    elem Elem, fir, sec;
    stack<elem> STACK;
    for (int i = 0; i < length; i++)
    {
        element = expression.at(i);
        switch (element)
        {
        case '|':
            sec = STACK.top();
            STACK.pop();
            fir = STACK.top();
            STACK.pop();
            Elem = act_Unit(fir, sec);
            STACK.push(Elem);
            break;
        case '*':
            fir = STACK.top();
            STACK.pop();
            Elem = act_star(fir);
            STACK.push(Elem);
            break;
        case '+':
            sec = STACK.top();
            STACK.pop();
            fir = STACK.top();
            STACK.pop();
            Elem = act_join(fir, sec);
            STACK.push(Elem);
            break;
```

```
            default:
                Elem = act_Elem(element);
                STACK.push(Elem);
            }
        }

        Elem = STACK.top();
        STACK.pop();

        return Elem;
}

//打印 NFA
void Display(elem Elem) {
        cout << "NFA States:" << endl;
        cout << "Start State: " << Elem.startName.nodeName << endl;
        cout << "End State: " << Elem.endName.nodeName << endl;

        cout << "NFA Transitions:" << endl;
        for (int i = 0; i < Elem.edgeCount; i++) {
            cout << "Edge " << i + 1 << ": ";
            cout << Elem.edgeSet[i].startName.nodeName << " --" << Elem.edgeSet[i].tranSymbol
<< "--> ";
            cout << Elem.edgeSet[i].endName.nodeName << endl;
        }

        cout << "End" << endl;
}

//生成 NFAdot 文件
void generateDotFile_NFA(const elem& nfa) {
        std::ofstream dotFile("nfa_graph.dot");

        if (dotFile.is_open()) {
            dotFile << "digraph NFA {\n";
            dotFile << "  rankdir=LR;  // 横向布局\n\n";
            dotFile << " node [shape = circle];   // 状态节点\n\n";

            dotFile << nfa.endName.nodeName << " [shape=doublecircle];\n";
            // 添加 NFA 状态
            dotFile << "  " << nfa.startName.nodeName << " [label=\"Start State: " <<
nfa.startName.nodeName << "\"];\n";
            dotFile << "  " << nfa.endName.nodeName << " [label=\"End State: " <<
nfa.endName.nodeName << "\"];\n";
```

```
        // 添加 NFA 转移
        for (int i = 0; i < nfa.edgeCount; i++) {
            const edge& currentEdge = nfa.edgeSet[i];
            dotFile << "  " << currentEdge.startName.nodeName << " -> " <<
currentEdge.endName.nodeName << " [label=\"" << currentEdge.tranSymbol << "\"];\n";
        }

        dotFile << "}\n";

        dotFile.close();
        std::cout << "NFA DOT file generated successfully.\n";
    }
    else {
        std::cerr << "Unable to open NFA DOT file.\n";
    }
}


// 计算 NFA 状态的 ε 闭包
DFAState eClosure(const set<string>& nfaStates, elem nfa) {
    DFAState eClosureState;
    eClosureState.nfaStates = nfaStates;

    stack<string> stateStack;

    // 初始化栈，将初始状态加入栈，最开始 nfaState 里只有 NFA_Elem.startName
    for (const string& nfaState_name : nfaStates) {
        stateStack.push(nfaState_name);
    }

    while (!stateStack.empty()) {
        string currentState = stateStack.top();
        stateStack.pop();

        // 遍历 NFA 的边
        for (int i = 0; i < nfa.edgeCount; i++) {
            edge currentEdge = nfa.edgeSet[i];

            // 如果边的起始状态是当前状态，并且边的转换符号是#，那么将目标状态加入 ε 闭
包
            if (currentEdge.startName.nodeName == currentState && currentEdge.tranSymbol
== '#') {
                // 检查目标状态是否已经在 ε 闭包中，避免重复添加
```

```
                if (eClosureState.nfaStates.find(currentEdge.endName.nodeName) ==
eClosureState.nfaStates.end()) {
                    eClosureState.nfaStates.insert(currentEdge.endName.nodeName);
                    // 将目标状态加入栈以便进一步处理
                    stateStack.push(currentEdge.endName.nodeName);
                }
            }
        }
    }

    // 为 ε 闭包分配一个唯一的名称
    for (const string& nfaState_name : eClosureState.nfaStates) {
        eClosureState.stateName += nfaState_name;
    }

    return eClosureState;
}

//move 函数
DFAState move(const DFAState& dfaState, char transitionSymbol, elem nfa) {
    DFAState nextState;

    // 遍历 DFAState 中的每个 NFA 状态
    for (const string& nfaState_name : dfaState.nfaStates) {
        // 在这里遍历所有 NFA 状态的边
        for (int i = 0; i < nfa.edgeCount; i++) {
            edge currentEdge = nfa.edgeSet[i];

            // 如果边的起始状态是当前状态，且边的转换符号等于输入符号，将目标状态加入
nextState
            if (currentEdge.startName.nodeName == nfaState_name &&
currentEdge.tranSymbol == transitionSymbol && currentEdge.tranSymbol != '#') {
                nextState.nfaStates.insert(currentEdge.endName.nodeName);
            }
        }
    }

    // 为 nextState 分配一个唯一的名称
    for (const string& nfaState_name : nextState.nfaStates) {
        nextState.stateName += nfaState_name;
    }

    return nextState;
}
```

```
// 检查 DFA 状态是否在状态集合中,即 dfaStates 里有没有找到 targetState
bool isDFAStateInVector(const vector<DFAState>& dfaStates, const DFAState& targetState) {
    for (const DFAState& state : dfaStates) {
        if (state.stateName == targetState.stateName) {
            return true; // 找到匹配的状态
        }
    }
    return false; // 没有找到匹配的状态
}

//检查转换边是否在边集合中，比如 a->b 是否已经在集合中
bool isTransitionInVector(DFAState dfaState, DFAState dfaNextState, char symbol,
vector<DFATransition> dfaTransitions)
{
    for (const DFATransition& transition : dfaTransitions) {
        if (transition.fromState.stateName == dfaState.stateName &&
dfaNextState.stateName == dfaNextState.stateName && symbol == transition.transitionSymbol)
{
            return true; //找到匹配的状态
        }
    }
    return false;
}

void buildDFAFromNFA(const elem& NFA_Elem, vector<DFAState>& dfaStates,
vector<DFATransition>& dfaTransitions) {
    // 初始化 DFA 状态集合和转换关系
    set<string> nfaInitialStateSet;
    nfaInitialStateSet.insert(NFA_Elem.startName.nodeName);
    DFAState dfaInitialState = eClosure(nfaInitialStateSet, NFA_Elem); // 计算 NFA 初始
状态的 ε 闭包

    // 判断是否包含 NFA 的 endName 状态来确定是否为接受状态
    for (const string& nfaState : dfaInitialState.nfaStates) {
        if (nfaState == NFA_Elem.endName.nodeName) {
            dfaInitialState.isAccept = true;  // 如果包含 endName，标记为接受状态
            break;
        }
    }

    dfaStates.push_back(dfaInitialState);

    // 开始构建 DFA
```

```
    for (int i = 0; i < dfaStates.size(); i++) {
        DFAState dfaState = dfaStates[i];
        for (int j = 0; j < NFA_Elem.edgeCount; j++) {
            char symbol = NFA_Elem.edgeSet[j].tranSymbol;
            DFAState nextState = move(dfaState, symbol, NFA_Elem);
            DFAState dfaNextState = eClosure(nextState.nfaStates, NFA_Elem);

            if (!nextState.nfaStates.empty()) {
                // 如果下一个状态不为空，且在 DFA 状态集合中还未添加，则加入 DFA 状态
集合
                if (!isDFAStateInVector(dfaStates, dfaNextState)) {
                    // 检查是否为接受状态
                    for (const string& nfaState : dfaNextState.nfaStates) {
                        if (nfaState == NFA_Elem.endName.nodeName) {
                            dfaNextState.isAccept = true;  // 如果包含 endName，标记
为接受状态
                            break;
                        }
                    }
                    dfaStates.push_back(dfaNextState);
                }

                // 对于边也要去重
                if (!isTransitionInVector(dfaState, dfaNextState, symbol,
dfaTransitions)) {
                    dfaTransitions.push_back({ dfaState, dfaNextState, symbol });
                }
            }
        }
    }
}

// 显示 DFA 状态和转移关系，包括起始和结束状态
void displayDFA(const vector<DFAState>& dfaStates, const vector<DFATransition>&
dfaTransitions) {
    cout << "DFA States:" << endl;

    for (const DFAState& state : dfaStates) {
        cout << "State " << state.stateName << " (NFA States: ";
        for (const string& nfaState_name : state.nfaStates) {
            cout << nfaState_name << " ";
        }
        cout << ")";
```

```
            // 显示是否是接受状态
            if (state.isAccept) {
                cout << " (Accepting State)";
            }
            else {
                cout << " (Non-Accepting State)";
            }

            cout << endl;
        }

        cout << "\nDFA Transitions:" << endl;
        for (const DFATransition& transition : dfaTransitions) {
            cout << "State " << transition.fromState.stateName << " --" <<
transition.transitionSymbol << "--> State " << transition.toState.stateName << endl;
        }
}

//生成 DFA 的 dot 文件
void generateDotFile_DFA(vector<DFAState>& dfaStates, vector<DFATransition>&
dfaTransitions) {
    std::ofstream dotFile("dfa_graph.dot");

    if (dotFile.is_open()) {
        dotFile << "digraph DFA {\n";
        dotFile << "  rankdir=LR;  // 横向布局\n\n";
        dotFile << " node [shape = circle];   // 初始状态\n\n";

        // 标记接受状态
        for (const auto& state : dfaStates) {
            if (state.isAccept) {
                dotFile << "  " << state.stateName << " [shape=doublecircle];\n";
            }
        }

        // 添加 DFA 状态
        for (const auto& state : dfaStates) {
            dotFile << "  " << state.stateName;
            dotFile << " [label=\"State " << state.stateName;
            if (state.stateName == dfaStates.front().stateName) dotFile <<
"\\n(startState)";
            if (state.isAccept) dotFile << "\\n(acceptState)";
            dotFile << "\"];\n";
        }
```

```
        dotFile << "\n";

        // 添加 DFA 转移
        for (const auto& transition : dfaTransitions) {
            dotFile << "  " << transition.fromState.stateName << " -> " <<
transition.toState.stateName << " [label=\"" << transition.transitionSymbol << "\"];\n";
        }

        dotFile << "}\n";

        dotFile.close();
        std::cout << "DFA DOT file generated successfully.\n";
    }
    else {
        std::cerr << "Unable to open DOT file.\n";
    }
}

bool areStatesEqual(const set<string>& s1, const set<string>& s2) {
    return s1 == s2;  // 使用 == 运算符来检查集合是否相等
}

void minimizeDFA(vector<DFAState>& dfaStates, vector<DFATransition>& dfaTransitions) {
    unordered_map<string, set<DFAState>> stateGroups;
    set<DFAState> acceptStates;
    set<DFAState> rejectStates;

    // 将状态分为接受状态和非接受状态
    for (auto& state : dfaStates) {
        if (state.isAccept) {
            acceptStates.insert(state);
        }
        else {
            rejectStates.insert(state);
        }
    }

    // 初始状态分组：根据接受状态和非接受状态
    stateGroups["accept"] = acceptStates;
    stateGroups["reject"] = rejectStates;

    bool splitOccurred = true;
    while (splitOccurred) {
        splitOccurred = false;
```

```
        unordered_map<string, set<DFAState>> newStateGroups;

        // 对每个状态组进行细分
        for (auto& groupEntry : stateGroups) {
            auto& group = groupEntry.second;
            unordered_map<string, set<DFAState>> transitionGroups;

            for (auto& state : group) {
                map<char, string> transitionMap;
                for (auto& transition : dfaTransitions) {
                    if (transition.fromState.stateName == state.stateName) {
                        transitionMap[transition.transitionSymbol] =
transition.toState.stateName;
                    }
                }

                // 使用转移关系的唯一标识来分组
                string transitionKey;
                for (auto& entry : transitionMap) {
                    transitionKey += entry.first + entry.second;
                }

                transitionGroups[transitionKey].insert(state);
            }

            // 如果存在多个不同的转移组，说明需要继续细分
            if (transitionGroups.size() > 1) {
                splitOccurred = true;
            }

            // 将细分后的状态组更新到新的状态分组中
            for (auto& transitionGroup : transitionGroups) {
                newStateGroups[transitionGroup.first] = transitionGroup.second;
            }
        }

        // 更新状态分组
        stateGroups = newStateGroups;
    }

    // 生成新的 DFA 状态和记录原始状态集合
    vector<DFAState> newDFAStates;
    unordered_map<string, DFAState> stateMapping;
    int newStateCount = 0;
```

```
    for (auto& groupEntry : stateGroups) {
        DFAState newState;
        newState.stateName = "a" + to_string(newStateCount++);
        newState.isAccept = groupEntry.second.begin()->isAccept;

        // 记录原始状态集合
        for (auto& state : groupEntry.second) {
            newState.originalStates.insert(state.stateName);
        }

        newDFAStates.push_back(newState);

        for (auto& state : groupEntry.second) {
            stateMapping[state.stateName] = newState;
        }
    }

    // 生成新的 DFA 转移关系，确保没有重复的转移
    vector<DFATransition> newTransitions;
    unordered_set<string> addedTransitions;  // 用于记录已经添加的转移，避免重复

    for (auto& transition : dfaTransitions) {
        DFAState newFromState = stateMapping[transition.fromState.stateName];
        DFAState newToState = stateMapping[transition.toState.stateName];

        // 为转移生成唯一的标识
        string transitionKey = newFromState.stateName + "-" + string(1,
transition.transitionSymbol) + "->" + newToState.stateName;

        // 检查这个转移是否已经添加过
        if (addedTransitions.find(transitionKey) == addedTransitions.end()) {
            DFATransition newTransition = { newFromState, newToState,
transition.transitionSymbol };
            newTransitions.push_back(newTransition);
            addedTransitions.insert(transitionKey);  // 记录这个转移已经添加过
        }
    }

    // 最小化后的 DFA 状态和转换
    dfaStates = newDFAStates;
    dfaTransitions = newTransitions;
}
```

```
void displayminDFA(const vector<DFAState>& dfaStates, const vector<DFATransition>&
dfaTransitions) {
    cout << "DFA 状态集：\n";
    for (const auto& state : dfaStates) {
        cout << "状态: " << state.stateName;
        cout << " (NFA States: ";
        for (const auto& nfaState : state.originalStates) {
            cout << nfaState << " ";
        }
        cout << ")";
        cout << (state.isAccept ? " (接受状态)" : " (非接受状态)") << endl;
    }

    cout << "DFA 转移关系：\n";
    for (const auto& transition : dfaTransitions) {
        cout << transition.fromState.stateName << " --" << transition.transitionSymbol <<
"--> "
            << transition.toState.stateName << endl;
    }
}
//生成 DFA 的 dot 文件
void generateDotFile_minDFA(vector<DFAState>& dfaStates, vector<DFATransition>&
dfaTransitions) {
    std::ofstream dotFile("mindfa_graph.dot");

    if (dotFile.is_open()) {
        dotFile << "digraph DFA {\n";
        dotFile << "  rankdir=LR;  // 横向布局\n\n";
        dotFile << " node [shape = circle];   // 状态节点\n\n";

        // 标记接受状态
        for (const auto& state : dfaStates) {
            if (state.isAccept) {
                dotFile << "  " << state.stateName << " [shape=doublecircle];\n";
            }
        }

        // 添加 DFA 状态
        for (const auto& state : dfaStates) {
            dotFile << "  " << state.stateName;
            dotFile << " [label=\"State " << state.stateName;
            if (state.isAccept) dotFile << "\\n(acceptState)";
            dotFile << "\"];\n";
        }
```

```
        dotFile << "\n";

        // 标记第二个状态 (即 dfaStates[1]) 为起始状态
        // 假设第二个状态为 a1
        if (dfaStates.size() > 1) {
            dotFile << "  " << dfaStates[1].stateName << " [shape=circle, style=bold,
color=red]; // 起始状态\n";
        }

        // 添加 DFA 转移
        for (const auto& transition : dfaTransitions) {
            dotFile << "  " << transition.fromState.stateName << " -> "
                << transition.toState.stateName
                << " [label=\"" << transition.transitionSymbol << "\"];\n";
        }

        // 定义起始状态（箭头指向第二个状态 a1）
        if (dfaStates.size() > 1) {
            dotFile << "  start [shape=point, width=0];\n";
            dotFile << "  start -> " << dfaStates[1].stateName << ";\n";
        }

        dotFile << "}\n";

        dotFile.close();
        std::cout << "DFA DOT file generated successfully.\n";
    }
    else {
        std::cerr << "Unable to open DOT file.\n";
    }
}

// DFA 模拟
bool simulateDFA(const vector<DFAState>& dfaStates, const vector<DFATransition>&
dfaTransitions, const string& inputString) {
    // 将初始状态改为第二个状态，注意索引从 0 开始，所以下标 1 是第二个状态
    DFAState currentState = dfaStates[1];  // 初始状态改为第二个状态

    // 遍历输入字符串的每个字符
    for (char ch : inputString) {
        bool foundTransition = false;

        // 查找当前状态的转移
        for (const auto& transition : dfaTransitions) {
```

```
            if (transition.fromState.stateName == currentState.stateName &&
transition.transitionSymbol == ch) {
                currentState = transition.toState;  // 更新当前状态
                foundTransition = true;
                break;
            }
        }

        // 如果找不到转移，则字符串不能被接受
        if (!foundTransition) {
            return false;  // 没有有效的转移，拒绝输入字符串
        }
    }

    // 如果最后状态是接受状态，则接受输入
    return currentState.isAccept;
}
```