**实验二 自动机实验：NFA转DFA并最小化**

# 实验目的

本实验旨在理解并掌握将问题中的实体转换为抽象模型的能力，通过设计确定有穷自动机 (DFA) 和非确定有穷自动机 (NFA) 描述的对象模型，学习实现这些自动机的基本操作。同时，本实验要求学生掌握将多个NFA合并的方法，以及通过子集构造法将NFA确定化为DFA，并对DFA进一步进行最小化。通过这些步骤，进一步加深对自动机理论的理解与应用，特别是在状态转移、等价状态消除等方面的复杂性处理。本实验需达到的效果如下：

1. 理解和掌握把问题中的实体转换成抽象模型中数据结构的能力，设计确定有穷自动机DFA和非确定有穷自动机NFA描述的对象模型或数据结构，实现DFA和NFA的基本操作（输入和输出）；
2. 掌握将多个NFA合并的方法；
3. 掌握将NFA确定化成DFA的方法；
4. 掌握将DFA最小化的方法。

# **问题描述**

## 实现功能

本实验需要实现以下功能：

1. 设计一个函数，实现将两个NFA合并；
2. 设计一个函数，将NFA确定化为一个DFA；
3. 设计一个函数，对DFA进行最小化；
4. 输入多个NFA：NFA描述存储在文本文件中，文件名作为命令行参数输入；
5. 输出合并、确定化以及最小化后的DFA到标准输出设备。

## **实现原理和方法**

1. NFA合并：通过构造新的初始状态并连接原始NFA的初始状态来完成，确保两个NFA的状态和状态转换均被保留并能够通过一个新初始状态进入原NFA状态。
2. NFA确定化：使用子集构造法将NFA的非确定状态转换为确定状态。通过从初始状态的epsilon闭包开始，逐步处理每个输入符号，生成确定状态的转换关系，直到所有可能的状态都被涵盖。
3. DFA最小化：通过划分集合细化来消除等价状态，将状态集划分为不同的组，不断细化直到无法进一步划分，得到最小化的DFA。

# **软件设计方法的选择**

本实验采用面向对象的设计方法，将自动机的各个组成部分抽象为类，以此实现NFA、DFA和最小化DFA的各种操作。开发使用了C++语言，标准：ISO C++20，利用Visual Studio作为开发环境。

在开发过程中，通过构建类模型实现对NFA和DFA的抽象，分别设计了NFA、DFA和MinimizedDFA类，并在这些类中实现合并、确定化和最小化的功能。每个阶段都涉及类模型的扩展和接口设计，确保代码具有良好的扩展性和模块化。

# **分析模型**

**对象模型（领域模型）**

1. NFA类：包含状态集、输入字母表、状态转换函数、初始状态和终止状态。提供合并功能以将两个NFA对象合并为一个。
2. DFA类：通过子集构造法确定化，使用扁平化的数据结构，将状态集表示为单一集合而非嵌套集合。提供确定化函数以将NFA对象转化为DFA。
3. MinimizedDFA类：通过划分集合的细化来最小化状态集，提供最小化函数以简化DFA对象。
4. 状态转换：所有自动机类（NFA、DFA、MinimizedDFA）都有各自的状态转换函数，以实现输入字母到目标状态的转换。状态转换函数使用映射 (map) 数据结构来保存。

**用例模型**

1. NFA合并：用户输入两个NFA对象，系统通过合并函数生成一个新的NFA对象。
2. NFA确定化为DFA：用户输入一个NFA对象，系统通过确定化函数生成一个DFA对象。
3. DFA最小化：用户输入一个DFA对象，系统通过最小化函数生成一个最小化的DFA对象。

**使用的数据结构**

1. set：用于存储状态集和输入字母表，以确保状态和字母的唯一性，并方便集合运算，如并集和交集。
2. map：用于存储状态转换函数，键是状态和输入符号的组合，值是目标状态集或单一状态，方便状态转换的高效查找。
3. queue：在NFA确定化为DFA的过程中用于追踪待处理的状态集，支持广度优先搜索，确保所有可能状态都能被遍历。
4. stack：在计算epsilon闭包时，用于追踪状态的深度优先搜索。
5. pair：用于表示状态和输入符号的组合，作为状态转换函数的键，以唯一确定状态之间的转换关系。

# **设计模型**

**NFA 类：**

属性：

set<int> states：状态集

set<char> alphabet：输入字母表

map<pair<int, char>, set<int>> transitions：状态转换函数

int startState：初始状态

set<int> finalStates：终止状态集

方法：

static NFA merge(const NFA& nfa1, const NFA& nfa2)：合并两个NFA

static void printNFA(const NFA& nfa)：输出NFA的状态

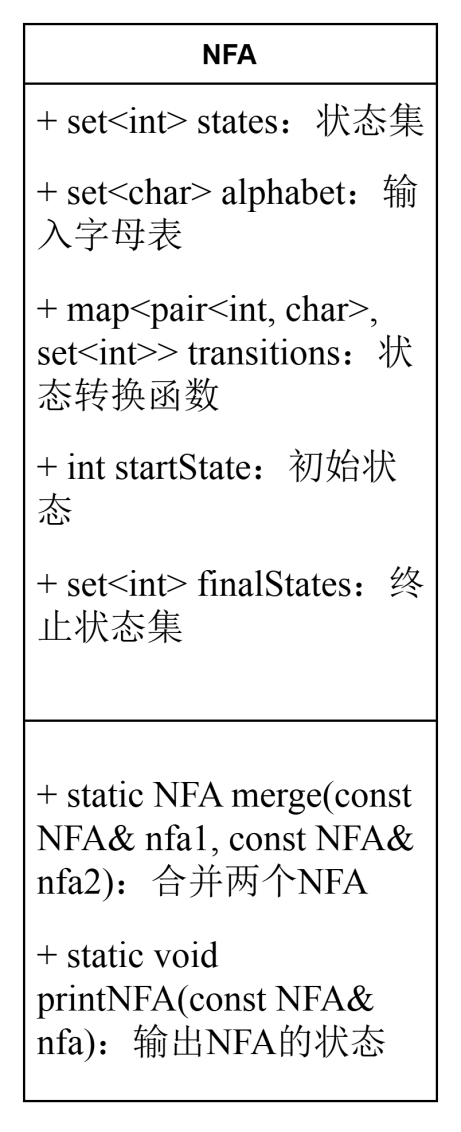


图1 NFA类图

**DFA 类（继承部分NFA属性）：**

继承自 NFA 的属性

新增属性：

set<int> dfaStates：DFA的状态集（扁平化后的状态）

map<pair<int, char>, int> transitions：扁平化后的状态转换函数

int startState：扁平化后的初始状态

set<int> finalStates：扁平化后的终止状态集

方法：

static DFA determinize(const NFA& nfa)：将NFA确定化为DFA

static void printDFA(const DFA& dfa)：输出DFA的状态

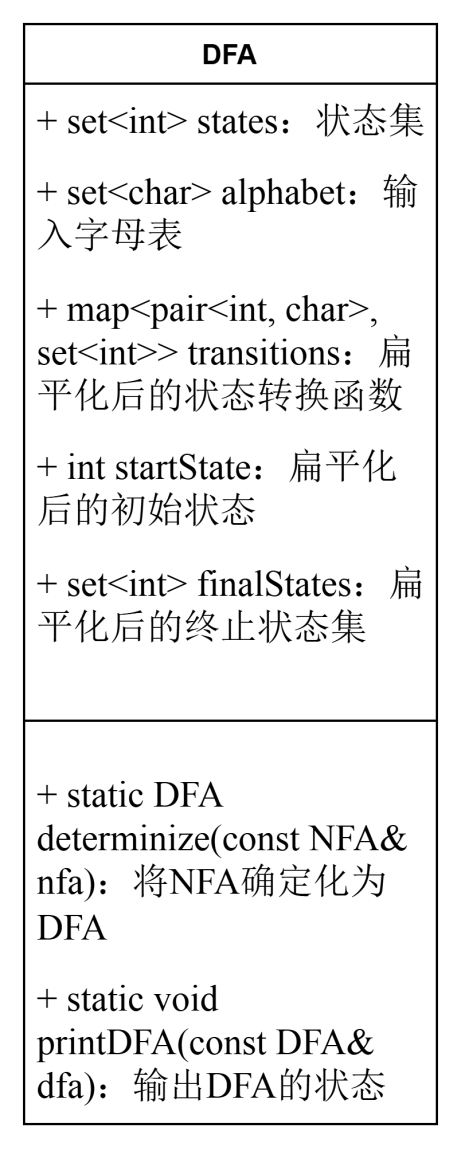


图2 DFA类图

**MinimizedDFA 类：**

属性：

set<int> minStates：最小化后的DFA状态集

map<pair<int, char>, int> minTransitions：最小化后的状态转换函数

int startState：最小化后的初始状态

set<int> finalStates：最小化后的终止状态集

方法：

static MinimizedDFA minimize(const DFA& dfa)：最小化DFA

static void printMinimizedDFA(const MinimizedDFA& dfa)：输出最小化后的DFA状态

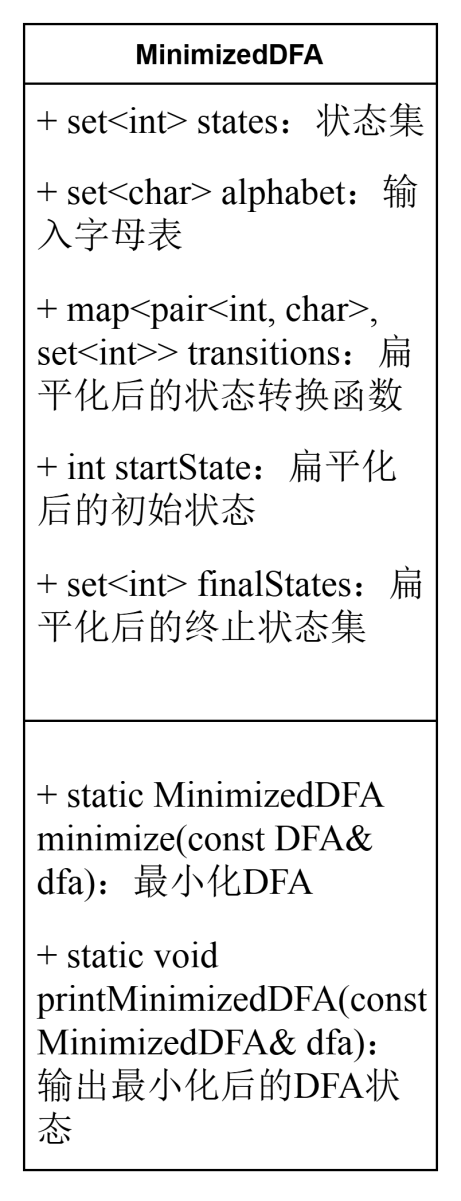


图3 MinimizedDFA 类图

三个类的交互图如图所示：

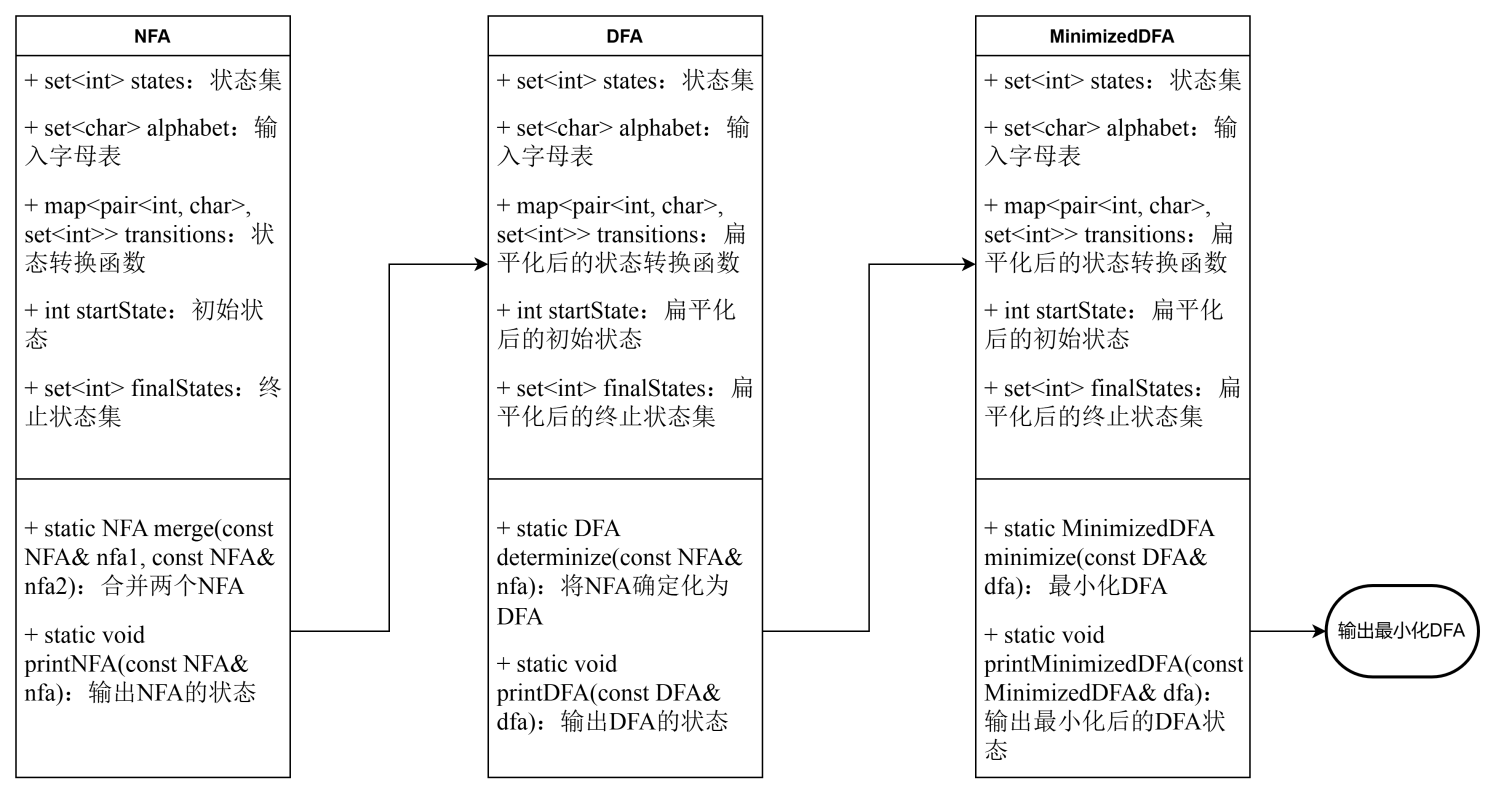


图4 类的主要交互图

# **主要算法描述**

## **NFA合并算法**

**实现思路：**将两个NFA合并的关键在于如何处理它们的初始状态。首先需要合并两个NFA的状态集、输入字母表、状态转换函数和终止状态集。为了确保新NFA能够兼容原来的两个NFA，我们创建一个新的初始状态，通过epsilon转移（即空串转移）到原NFA的初始状态。

**具体实现：**使用集合 (set) 来存储状态集和输入字母表，这样可以确保状态的唯一性和高效的集合操作。使用映射 (map) 来存储状态转换函数，键是状态和输入符号的组合，值是目标状态集。创建一个新的初始状态，将其与原NFA的初始状态通过epsilon转移连接。合并终止状态集，使得新的NFA终止状态包含原两个NFA的终止状态。

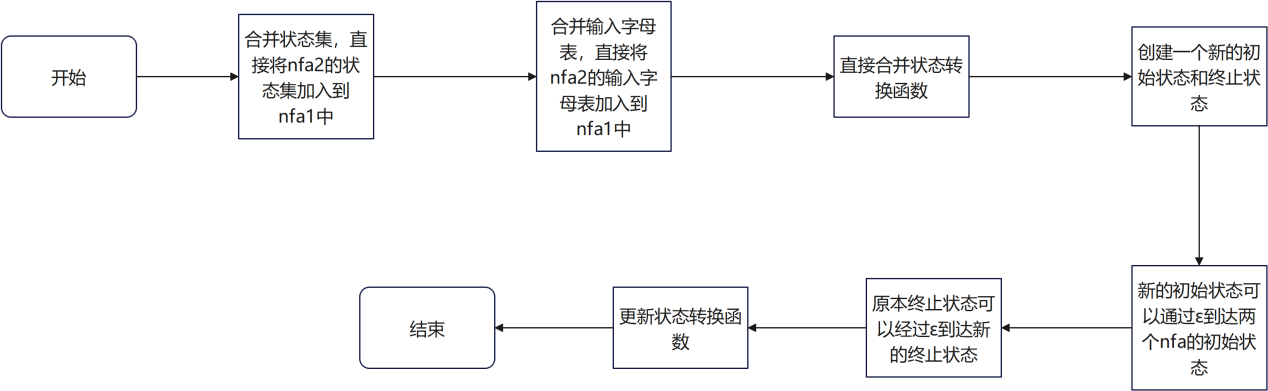


图5 NFA合并流程图

## **NFA确定化为DFA的子集构造法**

**实现思路：**确定化的核心在于将NFA的非确定状态集通过子集构造法转为DFA的确定状态。子集构造法的基本思想是通过追踪NFA中多个可能的状态集合，生成对应的DFA状态。这个过程从初始状态的epsilon闭包开始，不断根据输入符号进行状态转移，直到所有状态都被处理完。

**具体实现：**使用集合 (set) 存储DFA的状态，每个状态都是NFA状态的集合，这样可以表示多个NFA状态的组合。使用队列 (queue) 来维护待处理的状态集，以实现广度优先搜索，确保所有可能的状态都能被遍历到。使用映射 (map) 来表示状态转换函数，键是DFA状态和输入符号的组合，值是下一个状态集合。

利用epsilon闭包操作，确保每次状态转移后所有可能到达的状态都包含在内。

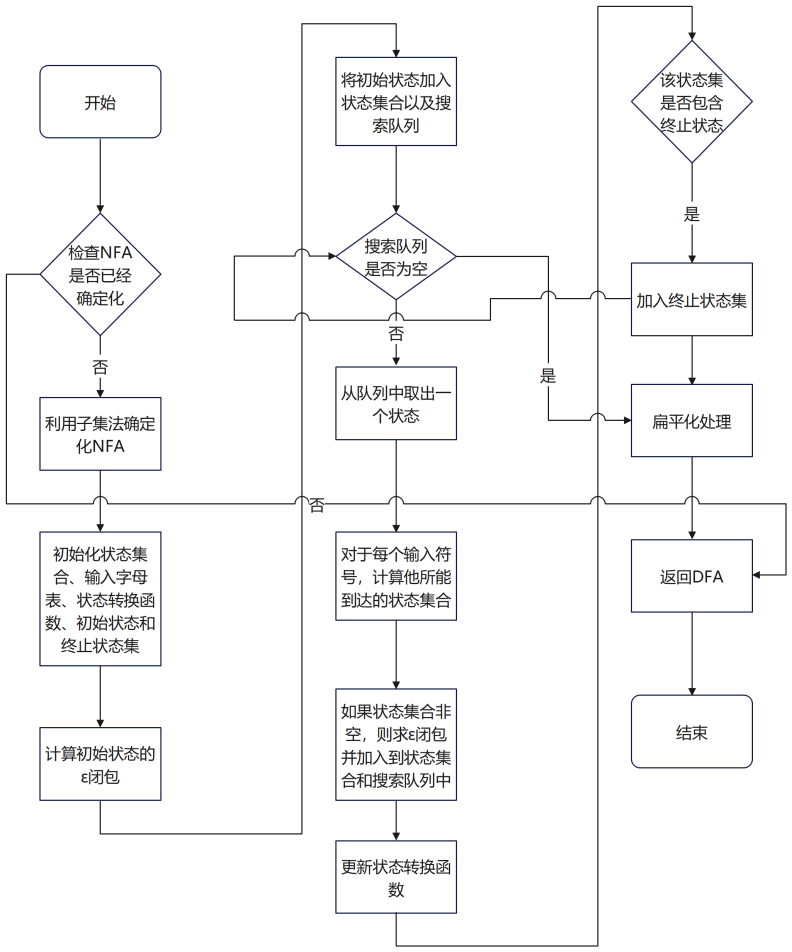


图6 DFA确定化流程图

## **DFA最小化算法**

**实现思路：**DFA最小化的目标是消除等价状态，从而减少状态的数量。最小化的主要方法是使用划分集合细化，将状态集划分为等价状态组。最初，将状态集划分为终态和非终态两组，然后根据输入符号不断细化这些划分，直到无法再细分为止。

**具体实现：**使用集合 (set) 来表示状态的划分，每个划分是一个状态组。使用映射 (map) 来跟踪每个状态组的转移，通过输入符号查找该状态组能到达的目标状态组。在细化过程中，利用标记 (signature) 来表示状态经过某个输入符号后所能到达的分区，确保等价状态被正确地分组。在每次细化时，将状态根据输入符号的转移情况进行重新划分，直到划分不再变化为止。

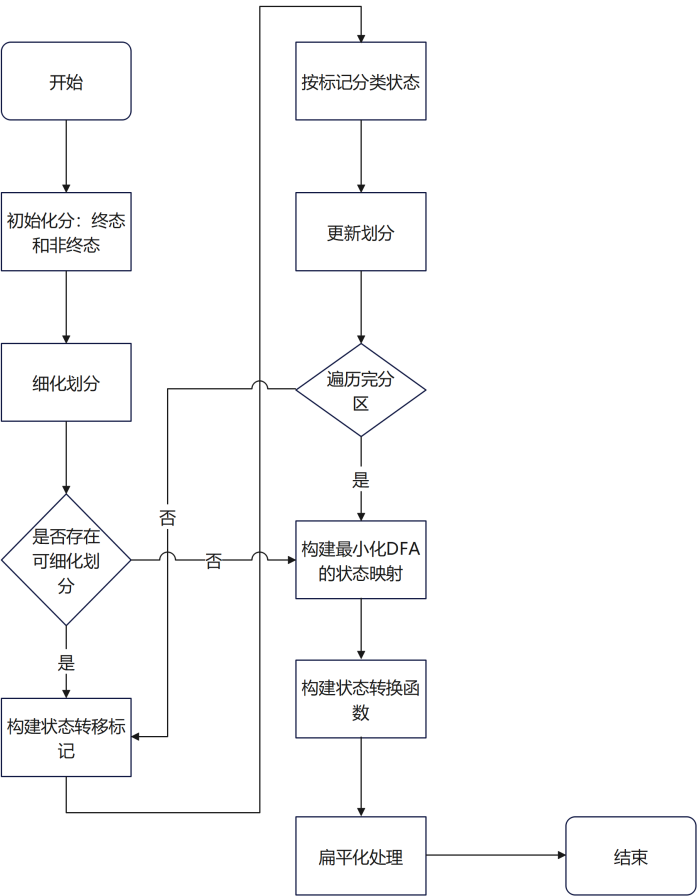


图7 DFA最小化算法流程图

## **扁平化处理算法**

**实现思路：**在NFA确定化为DFA的过程中，DFA中的每个状态通常是由多个NFA状态组合而成的集合，这使得状态表示变得复杂且难以操作。扁平化处理的目标是为每个状态集合分配一个唯一的整数编号，从而将DFA的状态集从集合形式简化为简单的整数形式，以便后续操作更加直观和高效。

**具体实现：**

1. **状态编号分配：**使用一个整数计数器 (stateCounter) 来为每个新的状态集合分配一个唯一的编号。使用映射 (map<set<int>, int> stateMapping)，用于将每个状态集合映射到一个唯一的整数标识符。当确定化过程中遇到一个新的状态集合时，检查该集合是否已经存在于 stateMapping 中。如果不存在，则使用 stateCounter 分配一个新的编号，并将其记录在 stateMapping 中，同时将 stateCounter 增加 1。
2. **状态转换的扁平化：**对于DFA的状态转换函数，原本的键是一个状态集合和输入符号的组合。为了进行扁平化处理，需要将状态集合替换为其对应的整数编号。建立一个新的映射 (map<pair<int, char>, int> flattenedTransitions)，其中键是扁平化后的状态编号与输入符号的组合，值是目标状态的编号。遍历原始状态转换函数，将每个状态集合键替换为其对应的整数编号，并更新到新的转换函数中。
3. **终态和初态的扁平化：**初态和终态集合同样需要进行扁平化处理。初态是由NFA的初始状态通过epsilon闭包得到的状态集合，因此在扁平化过程中，初态集合将被分配一个唯一的整数编号，成为DFA的初态。对于终态集合，遍历每个原始的DFA状态集合，若其中包含原NFA的终态，则将该集合映射为扁平化后的终态编号，并加入新的终态集合中。
4. **示例：**

假设在确定化过程中得到的DFA状态为 {1, 2, 3}，且尚未有编号分配，则通过 stateCounter 分配编号 0，并记录到 stateMapping 中：stateMapping[{1, 2, 3}] = 0。

如果后续遇到 {1, 2, 3}，则直接使用 stateMapping 中的编号 0，而无需重新分配。

对于状态转换，例如 {1, 2, 3} 在输入符号 'a' 下转移到 {4, 5}，经过扁平化后，转换函数将被表示为 (0, 'a') -> 1，其中 0 和 1 是对应状态集合 {1, 2, 3} 和 {4, 5} 的编号。

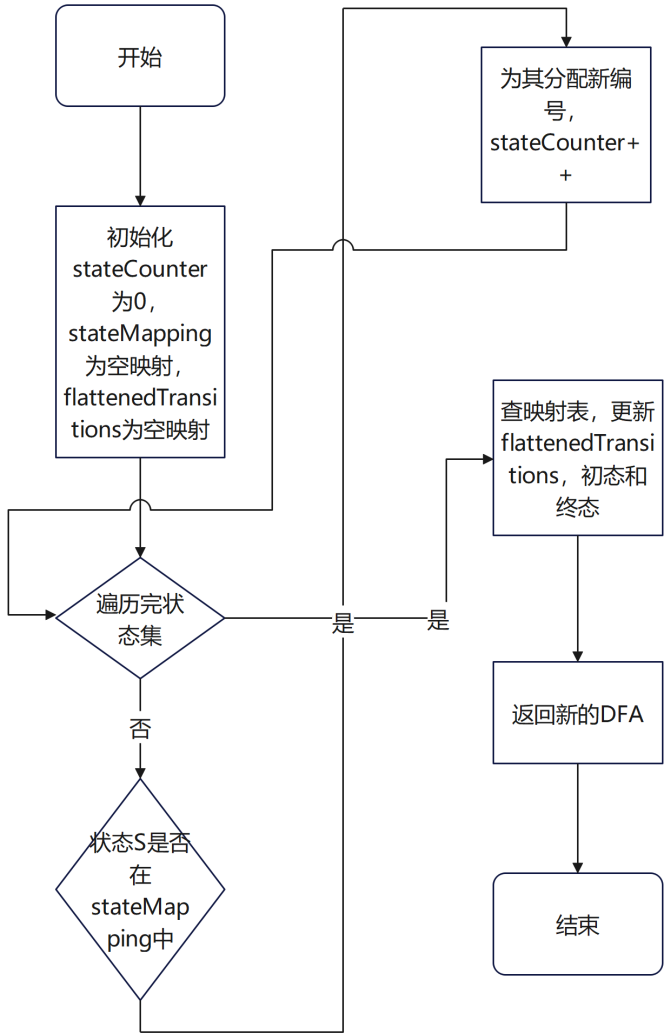


图8 扁平化处理流程图

# **测试数据与测试效果**

**测试样例1：**

**输入：**

1. 0 1 2 3 4 5
2. 0 1
3. 8
4. 0 -1 1
5. 1 1 3
6. 1 0 2
7. 3 -1 4
8. 4 1 4
9. 4 0 4
10. 4 -1 2
11. 2 -1 5
12. 0
13. 5

**输出：**

1. 成功打开文件
2. 合并后的NFA:
3. 状态集:0 1 2 3 4 5
4. 输入字母表:0 1
5. 状态转换函数:
6. (0, -1) -> 1
7. (1, 0) -> 2
8. (1, 1) -> 3
9. (2, -1) -> 5
10. (3, -1) -> 4
11. (4, -1) -> 2
12. (4, 0) -> 4
13. (4, 1) -> 4
14. 初始状态:0
15. 终止状态集:5
16. 确定化后的DFA:
17. 状态集:0 1 2 3
18. 输入字母表:0 1
19. 状态转换函数:
20. (0, 0) -> 3
21. (0, 1) -> 1
22. (1, 0) -> 2
23. (1, 1) -> 2
24. (2, 0) -> 2
25. (2, 1) -> 2
26. 初始状态:0
27. 终止状态集:1 2 3
28. 最小化后的DFA:
29. 状态集:0 1 2
30. 输入字母表:0 1
31. 状态转换函数:
32. (0, 0) -> 2
33. (0, 1) -> 1
34. (1, 0) -> 1
35. (1, 1) -> 1
36. 初始状态:0
37. 终止状态集:1 2

**测试样例2（输入已经确定化了的NFA）：**

**输入：**

1. 0 1 2 3 4 5
2. 0 1
3. 12
4. 0 1 1
5. 0 2 2
6. 1 1 1
7. 1 2 4
8. 2 1 1
9. 2 2 3
10. 3 1 3
11. 3 2 2
12. 4 1 0
13. 4 2 5
14. 5 1 5
15. 5 2 4
16. 0
17. 0 1

**输出：**

1. 成功打开文件
2. 合并后的NFA:
3. 状态集:0 1 2 3 4 5
4. 输入字母表:0 1
5. 状态转换函数:
6. (0, 1) -> 1
7. (0, 2) -> 2
8. (1, 1) -> 1
9. (1, 2) -> 4
10. (2, 1) -> 1
11. (2, 2) -> 3
12. (3, 1) -> 3
13. (3, 2) -> 2
14. (4, 1) -> 0
15. (4, 2) -> 5
16. (5, 1) -> 5
17. (5, 2) -> 4
18. 初始状态:0
19. 终止状态集:0 1
20. NFA已确定化，直接转换为DFA
21. 确定化后的DFA:
22. 状态集:0 1 2 3 4 5
23. 输入字母表:0 1
24. 状态转换函数:
25. (0, 1) -> 1
26. (0, 2) -> 2
27. (1, 1) -> 1
28. (1, 2) -> 4
29. (2, 1) -> 1
30. (2, 2) -> 3
31. (3, 1) -> 3
32. (3, 2) -> 2
33. (4, 1) -> 0
34. (4, 2) -> 5
35. (5, 1) -> 5
36. (5, 2) -> 4
37. 初始状态:0
38. 终止状态集:0 1
39. 最小化后的DFA:
40. 状态集:0 1 2
41. 输入字母表:0 1
42. 状态转换函数:
43. (0, 1) -> 0
44. (1, 1) -> 0
45. (2, 1) -> 2
46. 初始状态:0
47. 终止状态集:0

# **实验总结**

在本实验中，我成功实现了NFA到DFA的确定化和最小化操作，同时掌握了多个NFA合并的方法。实验过程中遇到的主要问题是如何处理NFA到DFA的状态扁平化，以及如何确保在最小化过程中正确地划分等价状态。通过对状态进行扁平化处理和使用子集构造法，我有效地解决了这些问题。

**收获与体会：**

1. **对自动机理论的深入理解：**本实验使我对NFA与DFA之间的关系有了更深入的理解，特别是在NFA确定化为DFA过程中，状态的复杂性如何通过子集构造法得到简化。了解了如何从多个状态组合形成确定状态，以及如何表示状态的转换和传递关系。
2. **编程实现的挑战与解决方案：**在实现子集构造法时，遇到的挑战主要在于如何处理状态集的组合和转换函数的有效存储与查找。在此过程中，通过使用集合和映射数据结构，我解决了NFA状态集合到DFA确定状态的映射问题，有效地减少了代码的复杂性并提高了算法的执行效率。
3. **DFA最小化的思维方法：**通过最小化DFA的过程，我学习了如何将状态划分为等价组，并不断细化这些组以达到最优状态数的目的。这一过程不仅加深了我对等价状态和状态划分概念的理解，也使我更加清楚如何将抽象理论应用于实际代码实现中。
4. **状态扁平化处理的重要性：**在NFA确定化过程中，DFA状态通常是由多个NFA状态组成的集合，这样的集合在后续处理中比较复杂。因此，进行扁平化处理至关重要，它将DFA状态集从集合形式简化为简单的整数形式，方便后续的最小化和输出操作。这一过程增强了我对数据表示和处理复杂性的认识。
5. **面向对象设计的优势：**通过面向对象的方法实现自动机的抽象，使得每一个自动机（NFA、DFA、MinimizedDFA）都有明确的类定义和职责划分，从而提高了代码的可读性和可维护性。在不同阶段扩展功能时，例如将NFA转化为DFA，或者最小化DFA，面向对象的设计使得这些功能的实现和集成变得更加容易。

**改进建议：**

1. **图形化工具的引入：**在处理较复杂的NFA时，可以考虑引入图形化工具来帮助理解状态的转移和合并过程。通过图形化的表示，可以更加直观地观察每一步操作的结果，帮助快速找到可能存在的错误。
2. **代码可视化与调试：**对于状态转换函数较为复杂的自动机，使用数据可视化工具可以更好地验证每个步骤的正确性和结果的合理性。此外，调试工具的使用可以帮助跟踪状态的变化，尤其是在DFA最小化过程中，便于定位问题所在。
3. **优化算法执行效率：**在处理大规模状态集时，子集构造法和划分细化方法的效率可能会成为瓶颈。因此，未来可以考虑优化这些算法的执行效率，例如通过引入并行处理来加速状态的划分和转换过程。