OpenTopic 1：实现词法分析算法

张铭徐

南开大学计算机学院

2023.12.19

1. Introduction
2. Thompson 构造法
3. Powerset 构造法
4. Hopcroft 算法
5. 结果展示

实验目的

- 在编译原理中，词法分析阶段接受程序员写出的高级语言，通过正则表达式将语言分割为若干词素 lexeme，然后根据规则分配到各个单词 token 中，然后将整体分割的结果提供给语法分析器进行下一步的语法分析工作。我们在 yacc 编程中，只是使用了正则表达式进行模式匹配，但是具体内部是如何实现的，我们未曾可知，所以本次实现我们将实现 C++ 对于这部分内容的一个复现。

基本假设

- 无法直接输入 ε，将字符“?”视为 ε 进行处理。

基本假设

- 无法直接输入 ε，将字符“?”视为 ε 进行处理。
- 本次实现的 Thompson 构造法实际上并不足以解决我们遇到的所有情况，例如对于词法分析中涉及到的十进制整数正则表达式：*DECIMAL*$-?([1-9][0-9]*|0)$，就无法进行匹配，我们仅实现了词法分析算法的一个子集，包含连接，选择，**Kleene** 闭包这三种运算。

基本假设

- 无法直接输入 ε，将字符“?”视为 ε 进行处理。
- 本次实现的 Thompson 构造法实际上并不足以解决我们遇到的所有情况，例如对于词法分析中涉及到的十进制整数正则表达式：$DECIMAL-?([1-9][0-9]*|0)$，就无法进行匹配，我们仅实现了词法分析算法的一个子集，包含连接，选择，**Kleene** 闭包这三种运算。
- 在后面 NFA 图中，如果边上没有表转移字符，默认是一次 ε-转移。

自动机与有限自动机

- 自动机是一组状态转移关系，由一个状态集、一个输入符号集、一个转移函数、一个初始状态和一个接受状态集组成。如果将所有的状态以及转移关系都画出来，那么我们可以得到一张图，图上的节点代表状态；图中的边代表转移函数(转移代价)。

自动机与有限自动机

- 自动机是一组状态转移关系，由一个状态集、一个输入符号集、一个转移函数、一个初始状态和一个接受状态集组成。如果将所有的状态以及转移关系都画出来，那么我们可以得到一张图，图上的节点代表状态；图中的边代表转移函数(转移代价)。
- 有限自动机，是一种状态数有限的自动机，而根据其转移关系，又可以将其分为确定性有限状态自动机 (DFA) 以及非确定性有限状态自动机 (NFA)。DFA 由于是确定性的，所以在其状态转移图中，所有的边上都必须有代价，而 NFA 则边上可以为空，也即可以无条件转移。

自动机与有限自动机

- 自动机是一组状态转移关系，由一个状态集、一个输入符号集、一个转移函数、一个初始状态和一个接受状态集组成。如果将所有的状态以及转移关系都画出来，那么我们可以得到一张图，图上的节点代表状态；图中的边代表转移函数(转移代价)。
- 有限自动机，是一种状态数有限的自动机，而根据其转移关系，又可以将其分为确定性有限状态自动机 (DFA) 以及非确定性有限状态自动机 (NFA)。DFA 由于是确定性的，所以在其状态转移图中，所有的边上都必须有代价，而 NFA 则边上可以为空，也即可以无条件转移。
- 在本次实验中，实现 Thompson 构造法将正则表达式->NFA，Powerset 构造法实现 NFA->DFA，Hopcroft 法实现 DFA 最小化。

基本原理——基础构造

Thompson 构造法是自底向上构建 NFA 的过程，类似于表达式求值。一个大的正则表达式的 NFA 是由若干子 NFA 通过运算构造而来：

- 对于正则表达式的基本符号，我们可以构建以下 NFA：
  - 对于 ε（空串）：构建一个只有 ε-转移的 NFA。

基本原理——基础构造

Thompson 构造法是自底向上构建 NFA 的过程，类似于表达式求值。一个大的正则表达式的 NFA 是由若干子 NFA 通过运算构造而来：

- 对于正则表达式的基本符号，我们可以构建以下 NFA：
  - 对于 ε（空串）：构建一个只有 ε-转移的 NFA。
  - 对于任何字符 a：构建一个从起始状态到接受状态的 a 转移的 NFA。

基本原理——基础构造

Thompson 构造法是自底向上构建 NFA 的过程，类似于表达式求值。一个大的正则表达式的 NFA 是由若干子 NFA 通过运算构造而来：

- 对于正则表达式的基本符号，我们可以构建以下 NFA：
  - 对于 ε（空串）：构建一个只有 ε-转移的 NFA。
  - 对于任何字符 a：构建一个从起始状态到接受状态的 a 转移的 NFA。
  - 上述转换为 NFA 示例如图1 以及2所示：

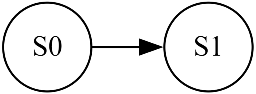


图 1: ε 转移示例

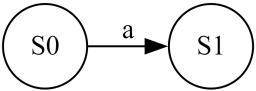


图 2: 字符转移示例

基本原理——组合构造 (连接)

Thompson 构造法是自底向上构建 NFA 的过程，类似于表达式求值。一个大的正则表达式的 NFA 是由若干子 NFA 通过运算构造而来：

- 对于三种基本运算，我们构建以下 NFA：
  - 连接：如果正则表达式是两个子表达式 r 和 s 的连接，即 rs，那么 r 的 NFA 的接受状态与 s 的 NFA 的起始状态通过 ε-转移连接。然后将 r 的接受状态与 s 的起始状态合并为一个状态 (采用 PPT 的方法 2)。

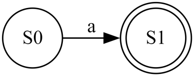


图 3: N(r) 示例

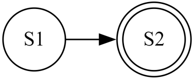


图 4: N(s) 示例

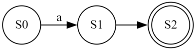


图 5: N(rs) 示例

基本原理——组合构造 (选择)

Thompson 构造法是自底向上构建 NFA 的过程，类似于表达式求值。一个大的正则表达式的 NFA 是由若干子 NFA 通过运算构造而来：

- 对于三种基本运算，我们构建以下 NFA：
  - 选择：如果正则表达式是 r 或 s，那么创建一个新的起始状态，并且通过 ε-转移连接到 r 和 s 的 NFA 的起始状态。同时，r 和 s 的 NFA 的接受状态都通过 ε-转移连接到一个新的接受状态。

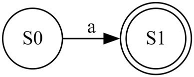


图 6: N(r) 示例

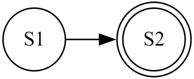


图 7: N(s) 示例

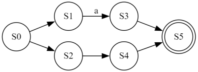


图 8: N(rs) 示例

基本原理——组合构造 (闭包)

Thompson 构造法是自底向上构建 NFA 的过程，类似于表达式求值。一个大的正则表达式的 NFA 是由若干子 NFA 通过运算构造而来：

- 对于三种基本运算，我们构建以下 NFA：
  - 闭包：如果正则表达式是 r^*，那么创建一个新的起始状态和一个新的接受状态，并且使用 ε-转移进行如下连接：新起始状态到 r 的 NFA 的起始状态、r 的 NFA 的接受状态到新接受状态、新起始状态到新接受状态（表示零次重复），以及新接受状态到 r 的 NFA 的起始状态（表示多次重复）。

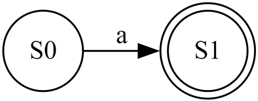


图 9: N(r) 示例

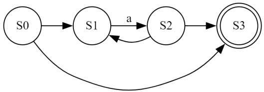


图 10: N($r*$) 示例

数据结构设计

- 考虑每一个 NFA 实际上都可以认为有一个开始状态和一个结束状态。

数据结构设计

- 考虑每一个 NFA 实际上都可以认为有一个开始状态和一个结束状态。
- 任意一个 NFA 可以设计为具有开始状态和结束状态的结构体 Fragment。

数据结构设计

- 考虑每一个 NFA 实际上都可以认为有一个开始状态和一个结束状态。
- 任意一个 NFA 可以设计为具有开始状态和结束状态的结构体 Fragment。
- 对于任意 NFA 的内部，可能有若干的状态连接，连接下一个边。

数据结构设计

设计对于任意的 NFA 为 Fragment，包含一个开始状态和一个结束状态，内部包含 State 节点包含若干状态的边。

```
const char EPSILON = '\0';
struct State {
    char symbol='\0';
    State* next;
    State* next2;
    State() : symbol(EPSILON), next(nullptr), next2(nullptr) {}
    State(char s, State* n1 = nullptr, State* n2 = nullptr)
    : symbol(s == '?' ? EPSILON : s), next(n1), next2(n2) {}
};
struct Fragment {
    State* start;
    State* accept;
};
```

建立基础构造

我们建立基础构造，对于字符（包括空）转移的 NFA。

```
Fragment charToFragment(char c) {
    State* s = new State{c, nullptr, nullptr};
    State* accept = new State{'\0', nullptr, nullptr};
    s->next = accept;
    return {s, accept};
}
```

连接运算

连接运算的主体是两个正则式 r 和 s，我们采用他们的正则式 $N(r)$ 以及 $N(s)$ 进行连接：

```
Fragment concatenateFragments(Fragment frag1, Fragment frag2) {
    frag1.accept->symbol = frag2.start->symbol;
    frag1.accept->next = frag2.start->next;
    frag1.accept->next2 = frag2.start->next2;
    delete frag2.start;
    Fragment newFrag = { frag1.start, frag2.accept };
    return newFrag;
}
```

选择运算

同理，选择运算的主题是两个正则式 r 和 s，我们采用他们的正则式 $N(r)$ 以及 $N(s)$ 进行选择：

```
Fragment unionFragments(Fragment f1, Fragment f2) {
    State* s = new State{'\0', f1.start, f2.start}; // '\0' 表示 边
    Fragment frag;
    frag.start = s;
    State* accept = new State{'\0', nullptr, nullptr};
    f1.accept->next = accept;
    f2.accept->next = accept;
    frag.accept = accept;
    return frag;
}
```

闭包运算

按照最开始我们在基础原理中所描述的那样，建立两个全新的节点，并建立对应的空转移。

```
Fragment closureFragment(Fragment f) {
    State* newStart = new State();
    State* newAccept = new State();
    newStart->next = f.start;
    f.accept->next = f.start;
    f.accept->next2 = newAccept;
    newStart->next2 = newAccept;
    Fragment frag;
    frag.start = newStart;
    frag.accept = newAccept;
    return frag;
}
```

读入字符流处理

需要对读入的数据流处理，调用上述的函数，类似于后缀表达式的处理过程，闭包为一元运算符，选择和连接为二元运算，使用 Fragment 栈存储对应的子 NFA。

- 循环遍历输入的正则表达式字符串。

读入字符流处理

需要对读入的数据流处理，调用上述的函数，类似于后缀表达式的处理过程，闭包为一元运算符，选择和连接为二元运算，使用 Fragment 栈存储对应的子 NFA。

- 循环遍历输入的正则表达式字符串。
- 正则表达式中的字符，将它们转换为对应的 NFA 片段。

读入字符流处理

需要对读入的数据流处理，调用上述的函数，类似于后缀表达式的处理过程，闭包为一元运算符，选择和连接为二元运算，使用 Fragment 栈存储对应的子 NFA。

- 循环遍历输入的正则表达式字符串。
- 正则表达式中的字符，将它们转换为对应的 NFA 片段。
- |：遇到选择符号时，表示正则表达式中的选择，把它推入操作符栈。

读入字符流处理

需要对读入的数据流处理，调用上述的函数，类似于后缀表达式的处理过程，闭包为一元运算符，选择和连接为二元运算，使用 Fragment 栈存储对应的子 NFA。

- 循环遍历输入的正则表达式字符串。
- 正则表达式中的字符，将它们转换为对应的 NFA 片段。
- |：遇到选择符号时，表示正则表达式中的选择，把它推入操作符栈。
- *：星号表示前一个字符或片段的零次或多次重复。在这里，它取出栈顶的片段并应用闭包操作，然后将结果推回栈中。

读入字符流处理

需要对读入的数据流处理，调用上述的函数，类似于后缀表达式的处理过程，闭包为一元运算符，选择和连接为二元运算，使用 Fragment 栈存储对应的子 NFA。

- 循环遍历输入的正则表达式字符串。
- 正则表达式中的字符，将它们转换为对应的 NFA 片段。
- |：遇到选择符号时，表示正则表达式中的选择，把它推入操作符栈。
- *：星号表示前一个字符或片段的零次或多次重复。在这里，它取出栈顶的片段并应用闭包操作，然后将结果推回栈中。
- (和)：处理括号，用于分组。左括号被推入操作符栈。遇到右括号时，执行括号内的操作，直到遇到左括号。将括号内的 NFA 视为一个整体，将整体处理后推入 Fragment 栈。

读入字符流处理

需要对读入的数据流处理，调用上述的函数，类似于后缀表达式的处理过程，闭包为一元运算符，选择和连接为二元运算，使用 Fragment 栈存储对应的子 NFA。

- 循环遍历输入的正则表达式字符串。
- 正则表达式中的字符，将它们转换为对应的 NFA 片段。
- |：遇到选择符号时，表示正则表达式中的选择，把它推入操作符栈。
- *：星号表示前一个字符或片段的零次或多次重复。在这里，它取出栈顶的片段并应用闭包操作，然后将结果推回栈中。
- (和)：处理括号，用于分组。左括号被推入操作符栈。遇到右括号时，执行括号内的操作，直到遇到左括号。将括号内的 NFA 视为一个整体，将整体处理后推入 Fragment 栈。
- 最后栈中包含有若干 NFA 片段，通过连接将其连接为一个 NFA 返回。

Motivation

- 首先考虑 NFA 和 DFA 的区别，实际上，NFA 无非是有很多的冗余状态，经历了很多的空转移，如果有很多空转移，那么我们可能对于一个状态而言，就会有很多的路径可走，无形之中增大了搜索空间的大小，我们想要将这些冗余的状态都进行合并，构造出一个没有任何空转移的图。

Motivation

- 首先考虑 NFA 和 DFA 的区别，实际上，NFA 无非是有很多的冗余状态，经历了很多的空转移，如果有很多空转移，那么我们可能对于一个状态而言，就会有很多的路径可走，无形之中增大了搜索空间的大小，我们想要将这些冗余的状态都进行合并，构造出一个没有任何空转移的图。
- Powerset 构造法是一种将非确定性有限自动机（NFA）转换为等价的确定性有限自动机（DFA）的技术。这种构造的核心思想是考虑 NFA 在任意给定输入时可能同时处于的所有状态，并将这些状态的集合视为 DFA 的一个状态。

基本原理

- 初始状态：DFA 的初始状态是由 NFA 的初始状态经过 ϵ-闭包得到的状态集。

基本原理

- 初始状态：DFA 的初始状态是由 NFA 的初始状态经过 ϵ-闭包得到的状态集。
- 状态合并：对于每一个状态集合 S 和每一个输入符号 a，计算从 S 中的所有状态通过符号 a 以及可能的后续 ϵ-转移所能到达的所有状态。这个计算结果是一个新的状态集合，它代表了 DFA 在读取符号 a 后应该转移到的状态。

基本原理

- 初始状态：DFA 的初始状态是由 NFA 的初始状态经过 ϵ-闭包得到的状态集。
- 状态合并：对于每一个状态集合 S 和每一个输入符号 a，计算从 S 中的所有状态通过符号 a 以及可能的后续 ϵ-转移所能到达的所有状态。这个计算结果是一个新的状态集合，它代表了 DFA 在读取符号 a 后应该转移到的状态。
- 接受状态：如果 NFA 的一个接受状态包含在某个状态集合 S 中，那么 S 在 DFA 中也是一个接受状态。

基本原理

- 初始状态：DFA 的初始状态是由 NFA 的初始状态经过 ϵ-闭包得到的状态集。
- 状态合并：对于每一个状态集合 S 和每一个输入符号 a，计算从 S 中的所有状态通过符号 a 以及可能的后续 ϵ-转移所能到达的所有状态。这个计算结果是一个新的状态集合，它代表了 DFA 在读取符号 a 后应该转移到的状态。
- 接受状态：如果 NFA 的一个接受状态包含在某个状态集合 S 中，那么 S 在 DFA 中也是一个接受状态。
- 核心思想：将 **NFA** 中所有等价的状态合并，所有无条件转移变化为有条件转移。

数据结构设计

- DFA 有一个起始状态，但是可能有很多结束状态。

数据结构设计

- DFA 有一个起始状态，但是可能有很多结束状态。
- 我们不能简单的用起始状态和结束状态来设计一个 DFA。

数据结构设计

- DFA 有一个起始状态，但是可能有很多结束状态。
- 我们不能简单的用起始状态和结束状态来设计一个 DFA。
- 对于一个 DFA 的状态，是由若干 NFA 状态合并而来的。

数据结构设计

设计对于任意的 DFA，其中包含有若干 DFA 节点，DFA 节点中含有若干 NFA 节点，可以设计为：

```
struct DFAState {
    std::unordered_set<State*> nfaStates;
    std::unordered_map<char, DFAState*> transitions;
    bool isAcceptState = false;
};
class DFA {
public:
    std::vector<DFAState*> states;
    DFAState* startState = nullptr;
};
```

ε-闭包计算

算法关键在于将无条件转移的状态合并，计算任意 NFA 节点的 ε-闭包，使用 unorderset 保证求解闭包元素的唯一性：

```
std::unordered_set<State*> epsilonClosure(const std::unordered_set<State*>& states) {
    std::stack<State*> toProcess;
    std::unordered_set<State*> closureSet = states;
    for (auto s : states) {
        toProcess.push(s);
    }
    while (!toProcess.empty()) {
        State* current = toProcess.top();
        toProcess.pop();
        if (current && (!current->symbol || current->symbol == EPSILON)) {
            if (current->next && closureSet.insert(current->next).second) {
                toProcess.push(current->next);
            }
            if (current->next2 && closureSet.insert(current->next2).second) {
                toProcess.push(current->next2);
            }
        }
    }
    return closureSet;
}
```

NFA 转 DFA 算法思想

目的是将所有等价状态进行合并，所有空转移视为同一个状态。

- 对于开始状态，我们计算 ε-闭包，将其视为 DFA 的开始状态。

NFA 转 DFA 算法思想

目的是将所有等价状态进行合并，所有空转移视为同一个状态。

- 对于开始状态，我们计算 ε-闭包，将其视为 DFA 的开始状态。
- 使用队列处理每一个新的 DFA 状态，对于新 DFA 状态，模拟其内部 NFA 状态所有可能的字符转移，计算可能到达的状态集合 S，为保证唯一性，使用 unorderset 进行维护。

NFA 转 DFA 算法思想

目的是将所有等价状态进行合并，所有空转移视为同一个状态。

- 对于开始状态，我们计算 ε-闭包，将其视为 DFA 的开始状态。
- 使用队列处理每一个新的 DFA 状态，对于新 DFA 状态，模拟其内部 NFA 状态所有可能的字符转移，计算可能到达的状态集合 S，为保证唯一性，使用 unorderset 进行维护。
- 对于每个字符转移，计算状态集合 S 的 ε-闭包，如果对应闭包不存在，设置为一个全新的 DFA 状态，并加入队列，标记对应状态转移代价。

Motivation

在我们通过子集构造法得到 DFA 之后，虽然我们将所有的空转移都消除掉了，但是我们可以发现，有一些状态仍然是可以合并到一起的，为了消除掉冗余状态，我们需要二次合并，将所有等价的状态类都合并为一个状态，这个时候就需要我们 Hopcroft 算法实现它的作用了：Hopcroft 算法是一种用于最小化 DFA 的高效算法。通过该算法，可以从一个 DFA 得到一个等价的 DFA，使其具有最少的状态。

基本原理

Hopcroft 算法基于以下的事实：如果两个状态在它们对某些输入的响应上是不可区分的（即，它们对所有可能的输入字符串都有相同的行为），那么这两个状态就是等价的，可以被合并为一个状态。算法的步骤如下：

- 初始化：首先，将所有非接受状态放入一个集合，所有接受状态放入另一个集合。这是因为接受状态和非接受状态显然是可区分的。

基本原理

Hopcroft 算法基于以下的事实：如果两个状态在它们对某些输入的响应上是不可区分的（即，它们对所有可能的输入字符串都有相同的行为），那么这两个状态就是等价的，可以被合并为一个状态。算法的步骤如下：

- 初始化：首先，将所有非接受状态放入一个集合，所有接受状态放入另一个集合。这是因为接受状态和非接受状态显然是可区分的。
- 细分：考虑当前的状态集合。对于每个输入符号，如果该符号使得集合中的一些状态转移到其他集合中的状态，那么这些状态是可区分的，应该被分开。这会产生更小的、更细粒度的状态集合。

基本原理

Hopcroft 算法基于以下的事实：如果两个状态在它们对某些输入的响应上是不可区分的（即，它们对所有可能的输入字符串都有相同的行为），那么这两个状态就是等价的，可以被合并为一个状态。算法的步骤如下：

- 初始化：首先，将所有非接受状态放入一个集合，所有接受状态放入另一个集合。这是因为接受状态和非接受状态显然是可区分的。
- 细分：考虑当前的状态集合。对于每个输入符号，如果该符号使得集合中的一些状态转移到其他集合中的状态，那么这些状态是可区分的，应该被分开。这会产生更小的、更细粒度的状态集合。
- 迭代：反复应用上述细分过程，直到不再有新的集合产生。

基本原理

Hopcroft 算法基于以下的事实：如果两个状态在它们对某些输入的响应上是不可区分的（即，它们对所有可能的输入字符串都有相同的行为），那么这两个状态就是等价的，可以被合并为一个状态。算法的步骤如下：

- 初始化：首先，将所有非接受状态放入一个集合，所有接受状态放入另一个集合。这是因为接受状态和非接受状态显然是可区分的。
- 细分：考虑当前的状态集合。对于每个输入符号，如果该符号使得集合中的一些状态转移到其他集合中的状态，那么这些状态是可区分的，应该被分开。这会产生更小的、更细粒度的状态集合。
- 迭代：反复应用上述细分过程，直到不再有新的集合产生。
- 合并：最后，每个集合中的所有状态都是等价的，可以被合并为单个状态。

数据结构设计

对于最小化 DFA，其数据结构应与 DFA 同构。DFA 最小化目的是将所有等价状态进行合并，不断区分所有的状态，直到不能够细分为止。对于所有的分区，尝试枚举进行细分：

```
struct DFAState {
    std::unordered_set<State*> nfaStates;
    std::unordered_map<char, DFAState*> transitions;
    bool isAcceptState = false;
};
class MinimizedDFA {
public:
    std::vector<DFAState*> states;
    DFAState* startState = nullptr;
};
```

NFA 转 DFA 算法思想

自顶向下分解大分区为小分区，而非在 DFA 的基础上对状态进行合并。

- 建立 inversePartition 存储当前分区情况。

NFA 转 DFA 算法思想

自顶向下分解大分区为小分区，而非在 DFA 的基础上对状态进行合并。

- 建立 inversePartition 存储当前分区情况。
- 枚举所有分区，以及当前分区中的所有状态。对于每一个状态，枚举所有可能的转移，并根据转移结果设置签名，使用 hash。

NFA 转 DFA 算法思想

自顶向下分解大分区为小分区，而非在 DFA 的基础上对状态进行合并。

- 建立 inversePartition 存储当前分区情况。
- 枚举所有分区，以及当前分区中的所有状态。对于每一个状态，枚举所有可能的转移，并根据转移结果设置签名，使用 hash。
- 使用 unorderset 维护唯一性。

NFA 转 DFA 算法思想

自顶向下分解大分区为小分区，而非在 DFA 的基础上对状态进行合并。

- 建立 inversePartition 存储当前分区情况。
- 枚举所有分区，以及当前分区中的所有状态。对于每一个状态，枚举所有可能的转移，并根据转移结果设置签名，使用 hash。
- 使用 unorderset 维护唯一性。
- newPartitions[i] 用于存储签名 (hash 值) 为 i 的状态集合。

NFA 转 DFA 算法思想

自顶向下分解大分区为小分区，而非在 DFA 的基础上对状态进行合并。

- 建立 inversePartition 存储当前分区情况。
- 枚举所有分区，以及当前分区中的所有状态。对于每一个状态，枚举所有可能的转移，并根据转移结果设置签名，使用 hash。
- 使用 unorderset 维护唯一性。
- newPartitions[i] 用于存储签名 (hash 值) 为 i 的状态集合。
- 如果 newPartitions.size > 1，说明在分区中，有部分元素转移不一致，需要将设置为全新分区。

DFA 最小化算法

```
while (changed) {
        changed = false;
        for (int i = 0; i <= count; i++) {
            if (inversePartition[i].size() <= 1) continue;
            std::unordered_map<int, std::unordered_set<DFAState*>> newPartitions;
            for (DFAState* state : inversePartition[i]) {
                int signature = 0;
                for (auto it = state->transitions.begin(); it != state->transitions.end(); ++it) {
                    char sym = it->first;
                    DFAState* nextState = it->second;
                    signature = (signature * 31 + partition[nextState]) ^ (sym + 127);
                }
                newPartitions[signature].insert(state);
            }
            if (newPartitions.size() > 1) {
                inversePartition.erase(i);
                for (auto it = newPartitions.begin(); it != newPartitions.end(); ++it) {
                    int sig = it->first;
                    std::unordered_set<DFAState*>& newSet = it->second;
                    count++;
                    inversePartition[count] = newSet;
                    for (DFAState* s : newSet) {
                        partition[s] = count;
                    }
                }
                changed = true;
            }
        }
    }
```

展示策略

- Graphviz 作为良好的工具，可建立对应的状态转移图。
- 定义了若干的处理函数，将原本的 NFA 类，DFA 类，MinimizeDFA 类都转化为了 dot 类型的文件。
- 对于每一个算法而言，我们都将给出 3 组测试样例测试程序的正确性，我们用三组样例分别测试 NFA，DFA，最小化 DFA 的正确性。

case 1 Thompson 构造法

使用 $a|(bce)|d*$ 测试正确性，为 a 与 bce 与 d 的闭包的一个选择，测试目前程序的优先级。

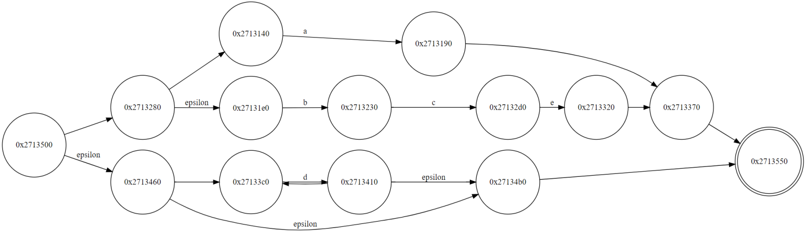
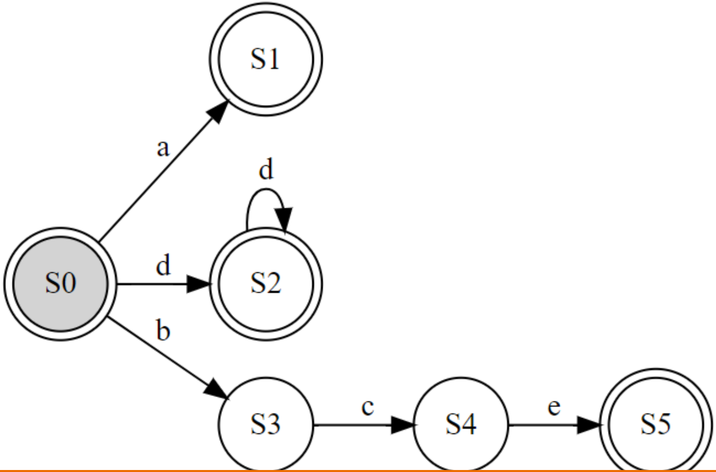


图 11: case1NFA

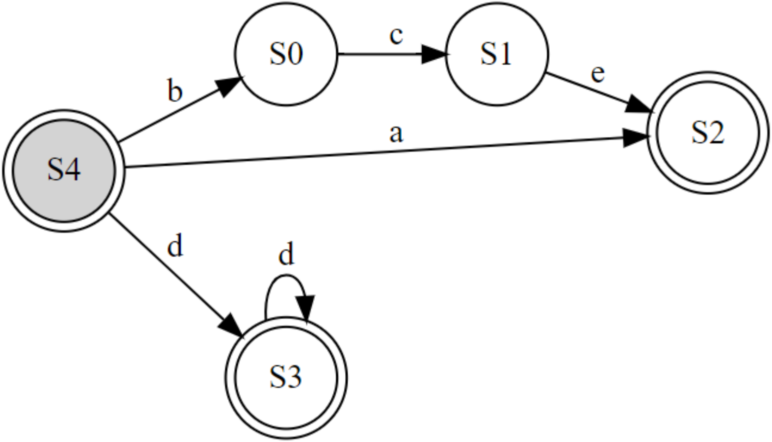
case 1 NFA->DFA

使用 $a|(bce)|d*$ 测试正确性，为 a 与 bce 与 d 的闭包的一个选择，测试目前程序的优先级。



case 1 DFA 最小化

使用 $a|(bce)|d*$ 测试正确性，为 a 与 bce 与 d 的闭包的一个选择，测试目前程序的优先级。



case 2 Thompson 构造法

使用正则表达式 $(d*(a|b))*|e$ 来测试括号的闭包的功能。

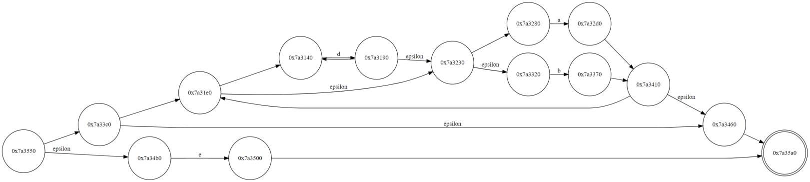
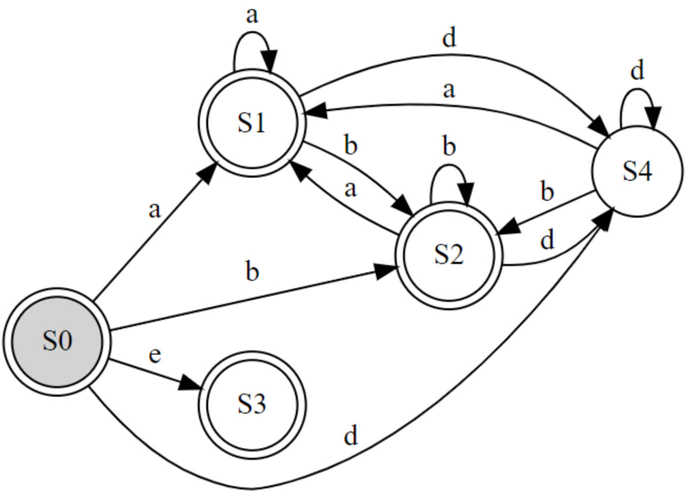


图 14: case1NFA

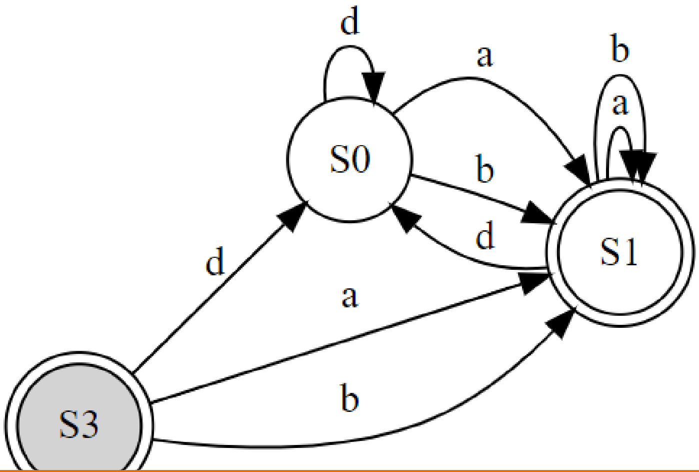
case 2 NFA->DFA

使用正则表达式 $(d*(a|b))*|e$ 来测试括号的闭包的功能。



case 2 DFA 最小化

使用正则表达式 $(d*(a|b))*|e$ 来测试括号的闭包的功能。



case 3 Thompson 构造法

使用正则表达式 $b*a((b|\epsilon)(a|b|\epsilon))$，也就是作业题目来测试空边的正确性。根据前面的约定，我们的输入是 $b*a((b|?)(a|b|?))$

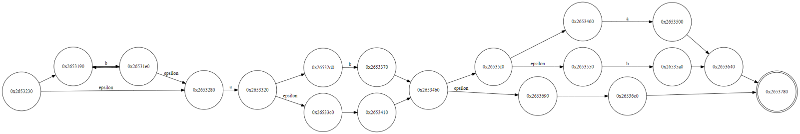


图 17: case1NFA

case 3 NFA->DFA

使用正则表达式 $b*a((b|\epsilon)(a|b|\epsilon))$，也就是作业题目来测试空边的正确性。根据前面的约定，我们的输入是 $b*a((b|?)(a|b|?))$

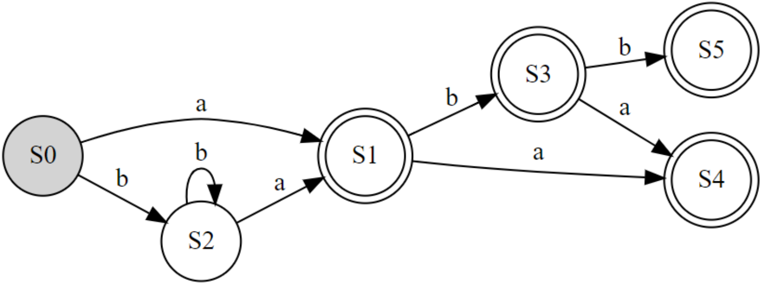


图 18: case1NFA

case 3 DFA 最小化

使用正则表达式 $b*a((b|\epsilon)(a|b|\epsilon))$，也就是作业题目来测试空边的正确性。根据前面的约定，我们的输入是 $b*a((b|?)(a|b|?))$

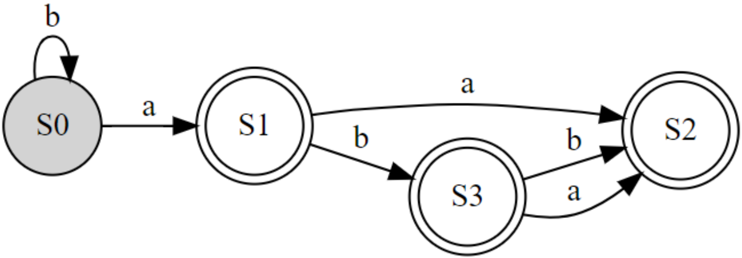


图 19: case1NFA