班　级　1504012

学　号　15040120169

西安电子科技大学

本科毕业设计论文

题　　目　有限自动机 C++ 工具箱等价性和最小化类软件测试

学　　院　机电工程学院

专　　业　机械设计制造及其自动化

学生姓名　胡双朴

导师姓名　段江涛

毕业设计（论文）诚信声明书

本人声明：本人所提交的毕业论文《有限自动机 C++ 工具箱等价性和最小化类软件测试》是本人在指导教师指导下独立研究、写作成果，论文中所引用他人的无论以何种方式发布的文字、研究成果，均在论文中加以说明；有关教师、同学和其他人员对本文本的写作、修订提出过并为我在论文中加以采纳的意见、建议，均已在我的致谢辞中加以说明并深致谢意。

本文和资料若有不实之处，本人承担一切相关责任。

论文作者：______________（签字） 时间：2019 年 06 月 01 日

指导教师已阅：__________（签字） 时间：2019 年 06 月 01 日

摘要

本文对有限自动机 C++ 工具箱中的五个最小化算法进行功能测试。

其余不太重要的部分，会以其他内容进行展示，例如，诗歌、歌词等。

关键词：确定性有限自动机　最小化　算法

Abstract

This paper is just a sample example for the users in learning the XDU*thesis*. I will try my best to use the commands and environments which are involved by the XDU*thesis*. Also, the popular composition skills in figures, tables and equations will be elaborated.

In the part unimportant, I will show something others, such as poems and lyrics.

Key words: XDUthesis commands environments skills

目录

第一章　引言

自动机理论是许多科学的重要理论基础，从硬件电路的简化，到各式各样的编译器构造，到处都有着自动机理论的应用，而确定性有限自动机（Deterministic Finite Automata，DFA）的最小化是自动机理论的一个重要组成部分，研究确定性有限自动机的最小化对自动机理论的健全和发展有重要意义。

1.1　FIRE engine

FIRE engine[1] 是一个使用 C++ 语言实现有限自动机和正则表达式的类库。本文中也把“有限自动机 C++ 工具箱”称作“FIRE engine”。有限自动机 C++ 工具箱（FIRE engine）实现了论文[2,3] 中提及的大部分算法，其中也包括了 Hopcroft、Ullman 等人提出的著名的确定性有限自动机最小化算法，验证这些算法的正确性和有效性对研究其他最小化算法很有帮助。

1.2　国内外研究现状

自动机理论的发展已经有很长时间，有限状态自动机、确定性有限状态自动机及相关的理论出现的时间都比较早。国外已经有文章[3] 对一些著名的确定性有限自动机最小化算法做了详细的阐述。

以状态的等价为基础，通过分层逼近、逐点近似等方法来计算等价关系，并在此基础上衍生处其他最小化算法。文中提及的算法中，Brzozowski 的最小化算法不依赖于其他任何最小化算法，而其他著名的算法都是在已有算法基础上的改进[3]。

国内对最小化算法的研究的分类则更加详细一些。

第二章　预备知识

在进行本文的叙述之前，需要预先了解一些相关的知识和定义，以便于使用更加简洁的描述方式。

一般来说，我们使用状态转移图来表示一个自动机，如图2.1

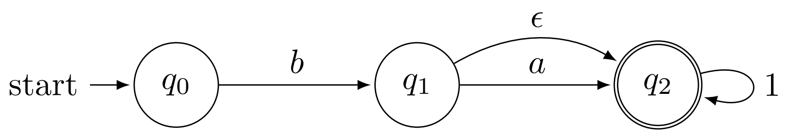


图 2.1　自动机状态转移图

图2.1中，自动机由 q_0 进入，接收字符“b”进入状态 q_1，状态 q_1 接收字符“a”进入状态 q_2，图中的两个同心圆圈住状态 q_2 表示结束状态（final state ）或者接受状态（accepted state ），意味自动机处理字符串到达此状态时，自动机就接受当前处理的字符串，状态 q_2 上有一个指向自己的箭头，意为接收字符“1”之后，仍然指向自己。而状态 q_1 经过 ϵ 转移到状态 q_2，意为状态 q_1 可以不接收任何字符即可转移到状态 q_2。图2.1中的自动机接受的字符串为 $ba(1)^* \cup b(1)^*$。

定义 2.1： 有限自动机[3]: 也称有限状态自动机（Finite automata，FA），有限自动机是一个 6 元组 (Q, V, T, E, S, F)，其中

- Q 是有限状态集;
- V 是一个字母表;
- $T \in \mathcal{P}(P \times V \times Q)$ 是一个转移关系;
- $E \in \mathcal{P}(Q \times Q)$ 是一个 ϵ-转移关系（空转移，不需要接收字符即可转移）;
- $S \subseteq Q$ 是开始状态集;
- $F \subseteq Q$ 是结束状态集;

字母表和函数 $\mathcal{P}$ 的定义分别在“定义A.2”和“惯例A.1”。

例 2.1： 我们可以用 $M = (\{q_0, q_1, q_2\}, \{b, a, 1, \epsilon\}, T, E, \{q_0\}, \{q_2\})$ 来指代图2.1中的自动机。其中，$T = \{(q_0 \times b \times q_1), (q_1 \times a \times q_2), (q_2 \times 1 \times q_2)\}$，$E = (q_1 \times q_2)$。

为了在某些情况下方便的展示状态之间的关系，也把状态转移图写成表格[4]，如图2.1中的自动机，写成表格2.1：

表 2.1 状态转移函数

状态说明	状态	输入字符			
		a	b	1	ϵ
开始状态 (start)	q_0	-	q_1	-	-
	q_1	q_2	-	-	q_2
结束状态 (accept)	q_2	-	-	q_2	-

下文中将表格2.1简称为转移函数。表格2.1的状态之间的关系与 $T = \{(q_0 \times b \times q_1), (q_1 \times a \times q_2), (q_2 \times 1 \times q_2)\}$，$E = (q_1 \times q_2)$ 一一对应。

定义 2.2 (确定性有限自动机)： 当且仅当

- 无多重初始状态;
- 无 ϵ 转移;
- 转移函数 $T \in Q \times V \longrightarrow \mathcal{P}(Q)$ 不将 $Q \times V$ 映射至多重状态。

时有限自动机 M 是确定性的。公式形式表达为式 2-1：

$$Det(M) \equiv (|S| \leq 1 \wedge \epsilon\text{-}free(E) \wedge (\forall q, a : q \in Q \wedge a \in V : |T(q, a)| \leq 1)) \tag{2-1}$$

下文中将确定性有限自动机简称为 DFA（Deterministic Finite Automata）。

第三章　等价关系和最小化

第四章　实例化类 DFA 对象

4.1　DFA 类

在本文中，我们仅仅对 DFA 应用最小化算法，FIRE engine 中的 DFA 类为应用最小化算法的类。FIRE engine 提供了多种方式构造一个 DFA，类 DFA 的部分实现如表 4.1

表 4.1　类 DFA

Class DFA			
名称	类型	属性	说明
Q	StatePool	protected	
S	StateSet	protected	$S \in Q$
F	StateSet	protected	$F \in Q$
T	DTransRel	protected	转移关系
DFA(const DFA_components& r);		public	构造函数
reconstruct(const DFA_components& r);	DFA&	public	
determinism() const;	DFA	public	
reverse();	DFA&	public	反转 DFA
min_Brzozowski();	DFA&	public	最小化算法
min_HopcroftUllman();	DFA&	public	最小化算法
min_dragon();	DFA&	public	最小化算法
min_Hopcroft();	DFA&	public	最小化算法
min_Watson();	DFA&	public	最小化算法
usefulf();	DFA&	public	
split(State,State,CharRange,StateEqRel&);	State	public	分割等价类

由表 4.1 可知，类 DFA 默认情况下，提供一个用“DFA_com-ponents”变量的引用来实例化 DFA 对象的方法。

除了使用“DFA_components”来实例化 DFA 对象之外，FIRE engine 还可以通过执行类 LBFA 的成员函数 LBFA::determinism() 来将 LBFA 对象转化成一个 DFA 对象，拥有同样功能的类还有类 FA 、类 RFA 和类 RBFA ，这四个类与类 DFA 都继承自类 FAabs 。

类 FAabs 及其派生类关系如图4.1

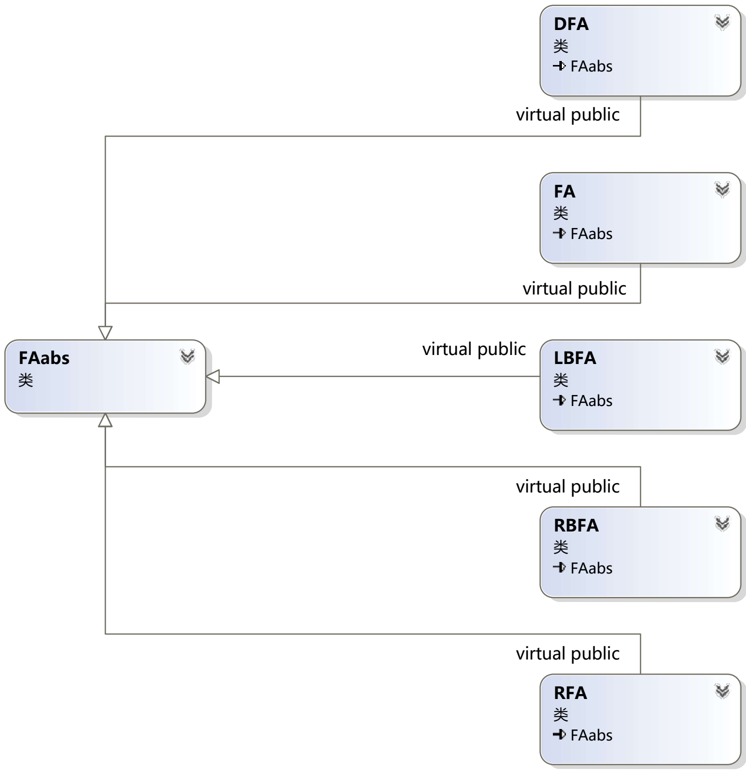


图 4.1 类 FAabs 及其派生类

4.2 DFA_components 结构体

从“DFA_components”构造一个 DFA 对象是最简单直接的方法，观察“DFA_components”的实现，如表4.2所示 由表4.2可知，结构体 DFA_components

表 4.2 DFA_components

结构体 DFA_components		
名称	类型	说明
Q	StatePool	
S	StateSet	$S \in Q$
F	StateSet	$F \in Q$
T	DTransRel	转移关系

内包含 StatePool 变量 Q，StateSet 变量 S，DTransRel 变量 T，StateSet 变量 F。若需要声明一个 DFA_components 变量，则需要分别实例化 StatePool 、StateSet、DTransRel 对象。这几个类还分别继承自其他的类，如图 4.2 所示。

类 DFA 的成员变量“T”为一个 DTransRel 对象，公有继承自模板类 StateTo<T>，模板参数“T”为保护继承自类 TransImpl 的类 DTrans 。

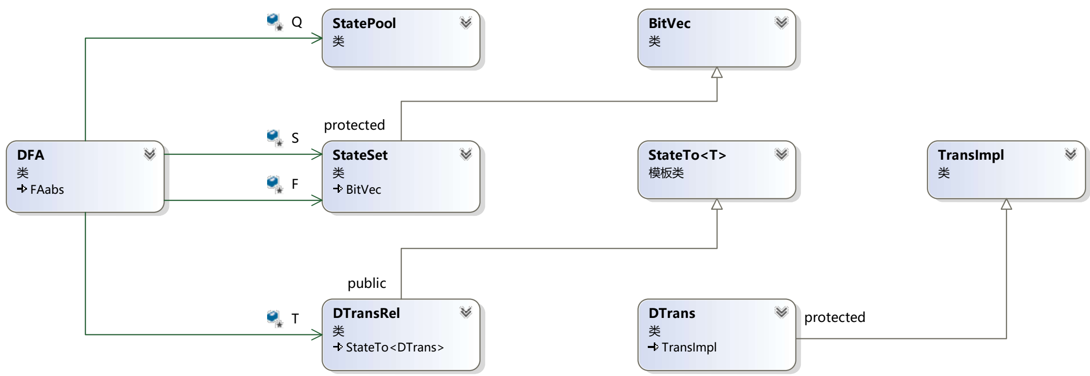


图 4.2　DFA 与其成员类及成员类的基类

4.2.1　StatePool 类

StatePool 类的部分实现如表 4.3 （文件 *StatePool.h*）:

表 4.3　类 StatePool

Class StatePool			
名称	类型	属性	说明
next	int	private	
StatePool(const StatePool& r);		public	构造函数
size() const;	int	public	
allocate();	State	public	
set_domain(const int r);	int	public	
contains(const State r) const;	int	public	

如表 4.3 所示，其中 StatePool::size() 和 StatePool::allocate() 为类 StatePool 两个比较重要的函数，前者为自动机 M 的大小，也即 $|M| = |Q|$。StatePool::allocate() 用来向 StatePool 增加新的状态。

4.2.2　StateSet 类

类 *StateSet* 的部分声明如表 4.4（文件 *StateSet.h*）:

由于在实例化过程 DFA 对象的过程中，只需要用到类 StateSet 的成员函数 StateSet::set_domain() 和 StateSet::add() 。类 StateSet 中有集合的交、并、差、补、判断是否为空集等功能，这些功能实际上由类 State 的父类 BitVec 提供，这里不再赘述类 BitVec 的内容。

表 4.4 类 StateSet

Class StateSet : protected BitVec			
名称	类型	属性	说明
StateSet();		public	构造函数
empty() const;	int	public	
size() const;	int	public	
complement();	StateSet&	public	求补集
add(const State r);	StateSet&	public	
remove(const State r);	StateSet&	public	
set_union(const StateSet& r);	StateSet&	public	求并集
intersection(const StateSet& r);	StateSet&	public	求交集
remove(const StateSet& r);	StateSet&	public	
smallest() const;	State	public	
set_domain(const int r);	void	public	

4.2.3 DTransRel 类

类 $DTransRel$ 是 DFA 的一个重要成员，它存储了 DFA 的状态转移关系，最小化算法中用到的等价类分割、等价状态合并等功能都与这个类息息相关。类 DTransRel 的部分实现如表 tab:Class-DTransRel 所示（文件 $DTransRel.h$）:

表 4.5 类 DTransRel

class DTransRel :public StateTo<DTrans>			
名称	类型	属性	说明
DTransRel();		public	构造函数
image(const State r, const char a) const;	State	public	
transition_on_range(State r, CharRange a) const;	State	public	
reverse_transition(const State r, const CharRange a) const;	StateSet	public	
labels_between(const State r, const StateSet& s) const;	CRSet	public	
out_labels(const State r) const;	CRSet	public	
reverse_closure(const StateSet& r) const;	StateSet	public	
add_transition(State p, CharRange a, State q);	DTransRel&	public	
set_domain(const int r);	void	public	

在实例化类 DFA 的过程中，类 DTransRel 需要用到的成员函数只有 DTransRel::set_domain() 和 DTransRel::add_transition()。

4.3　实例化类 DFA 对象

在文件 $DFA.cpp$ 中有如下实现:

```cpp
inline int DFA::class_invariant() const
{
    return(Q.size() == S.domain()
        && Q.size() == F.domain()
        && Q.size() == T.domain()
        && current < Q.size()
        && S.size() <= 1);
}
```

代码 4.1　DFA.cpp

而文件 $DFA.h$ 中的构造函数如下:

```cpp
inline DFA::DFA(const DFA_components& r) :Q(r.Q), S(r.S), F(r.F), T(r.T)
{
    current = Invalid;
    assert(class_invariant());
}
```

代码 4.2　DFA.h

由代码 4.1 和代码 4.2 可知，在语句“assert(class_invariant());”处，若函数“class_invariant()”返回值为“false”，那么程序将在此处中止[5]（下称 assert 中止）。由此可以知道，类 DFA 要求其成员变量满足

$$(Q.size() \equiv S.domain() \equiv T.domain) \wedge (S.size() \leq 1) \tag{4-1}$$

对于 $current \leq Q.size()$，“current”变量的值被初始化为“Invalid”，并且之后没有对其进行更改，所以不列入式4-1中。

在文件 $State.h$ 中有如下定义

```cpp
// Encode automata states as integers.
typedef signed int State;

// Invalid states mean something bad is about to happen.
const State Invalid = -1;
```

代码 4.3　State.h

由代码4.3可知，$State$ 类型实际上是整型，而“Invalid”为“-1”。

根据本节以上内容以及4.2节内容所说，可以实例化如图4.3的自动机。

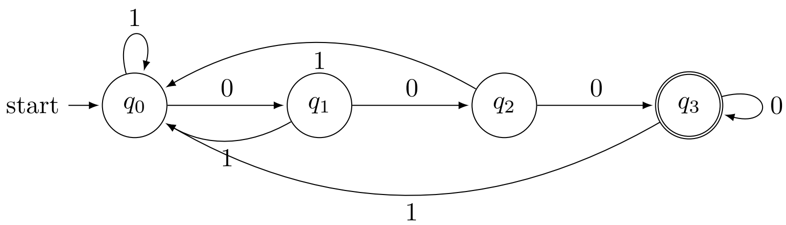


图 4.3 DFA 示例

图4.3中的自动机的转移函数如表4.6

表 4.6 图4.3状态转移函数

状态说明	状态	输入字符	
		0	1
开始状态 (start)	q_0	q_1	q_0
	q_1	q_2	q_0
	q_2	q_3	q_0
结束状态 (accept)	q_3	q_3	q_0

实例化 DFA 类如代码E.1:

代码 E.1 将在控制台输出如下信息:

```
DFA
Q = [0,4)
S = { 0 }
F = { 3 }
Transitions =
0->{ '0'->1  '1'->0 }
1->{ '0'->2  '1'->0 }
2->{ '0'->3  '1'->0 }
3->{ '0'->3  '1'->0 }

current = -1
```

代码 4.4 图4.3中自动机在 FIRE engine 中的表现形式

下文中将以表4.6的形式来描述一个自动机，以代码4.4的形式来表示自动机在 FIRE engine 中的展现形式。实例化一个 DFA 时以“实例化 DFA”代指，不用代码表示。

第五章　测试内容

本章内容为测试结果、测试过程中发现的问题、以及如何修复（部分）这些问题。

5.1　无限循环

实例化一个如表 5.1 的 DFA 的对象之后，调用函数 DFA::min_Watson()。（表 5.1 对应的状态转移图为图 C.1(a)，含有陷阱状态 q_5 ① ）

表 5.1　接受$\mathcal{L} = 0^*10^*$ 的自动机[4]

状态说明	状态	输入字符	
		0	1
开始状态 (start)	q_0	q_1	q_2
	q_1	q_0	q_3
结束状态 (accept)	q_2	q_4	q_5
结束状态 (accept)	q_3	q_4	q_5
结束状态 (accept)	q_4	q_4	q_5
陷阱状态 (sink)	q_5	q_5	q_5

5.1.1　运行结果

函数进入无限循环。

5.1.2　错误原因

单步调试发现进入无限循环的位置为 min-bww.cpp（124 行），为代码 5.1 中的“H.equivalize(p, q);”。

```
if (are_eq(p, q, S, H, Z))
{
    // p and q are equivalent.
    H.equivalize(p, q);
}
```

代码 5.1　min-bww.cpp

①进入此状态之后无法通过任何转移离开，如图 C.1(a) 中的状态q_5。

单步进入该函数，可以看到代码 5.2 （StateEqRel.cpp（42 行））

```
for (oldq->iter_start(i); !oldq->iter_end(i); oldq->iter_end(i))
{
    map(i) = newp;
}
```

代码 5.2 StateEqRel.cpp

for 循环的一般格式如代码 5.3

```
for (初始化循环变量; 循环条件; 迭代)
{
    循环体
}
```

代码 5.3 for 循环的一般格式

在代码 5.2 中循环变量为“i”，循环条件为“!oldq->iter_end(i);”，迭代为“oldq->iter_end(i)”。查看“iter_end()”函数实现如代码 5.4

```
// StateSet.h
// Is r the last State in an iteration sequence.
inline int StateSet::iter_end(State r) const
{
    return(BitVec::iter_end(r));
}

// BitVec.h
// Is r the last set bit in an iteration sequence.
// if (r== -1) retrun 1; else return 0
inline int BitVec::iter_end(int r) const
{
    return(r == -1);
}
```

代码 5.4 函数 iter_end() 的实现

可以看到函数“iter_end()”并未对参数“i”进行更改。于是程序在此处进入无限循环。

5.1.3 解决方法

将代码 5.2 中的迭代“oldq->iter_end(i)”更改为“oldq->iter_next(i)”，更改后如代码 5.5 ，经过比对，更改后与原文[1] 相同。

```
for (oldq->iter_start(i); !oldq->iter_end(i); oldq->iter_next(i))
{
    map(i) = newp;
}
```

代码 5.5 StateEqRel.cpp

更改后：函数“DFA::min_Watson();”不再陷入无限循环。

5.2　函数 DFA::usefulf() 运行错误

“DFA::usefulf()”函数为一个重要函数。用于去除有限自动机中的非“final-reachable”状态，在执行最小化算法前执行该函数，可以去除有限自动机中的非“final-reachable”状态，进而减少程序运行时间。其定义如代码 5.6

```
// Remove any States that cannot reach a final State.
// (This is a last step in minimization, since some of the min. algorithms
    may yield a DFA with a sink state.)
// Implement Remark 2.39  removing states that are not final - reachable.
DFA& usefulf();
```

代码 5.6　DFA::usefulf()

以图 C.1(a) 为例，状态 q_5 即为非“final-reachable”状态。移除状态 q_5 之后如图 C.1(b) 。图 C.1(b) 转移函数如表 5.2。

表 5.2　接受$\mathcal{L} = 0^*10^*$ 的自动机[4]

状态说明	状态	输入字符	
		0	1
开始状态 (start)	q_0	q_1	q_2
	q_1	q_0	q_3
结束状态 (accept)	q_2	q_4	-
结束状态 (accept)	q_3	q_4	-
结束状态 (accept)	q_4	q_4	-

5.2.1　运行结果

执行函数“DFA::usefulf()”后若状态 q_5 被去除，则函数定义功能正常执行。但是在实际的执行过程中，程序提示如图 5.1 错误

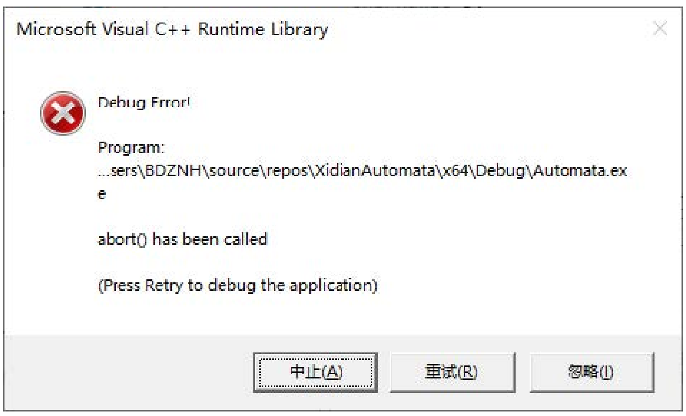


图 5.1　函数 DFA::usefulf() 错误提示

控制台提示如图 5.2

图 5.2 函数 DFA::usefulf() 错误提示

可以确定函数未完成其定义功能。

5.2.2 错误原因

查看 TransImple.cpp，79 行，代码如下

```
75 // Add a transition to the set.
76 TransImpl& TransImpl::add_transition(const CharRange a, const State q)
77 {
78     assert(a.class_invariant());
79     assert(0 <= q);
80     .........
81 }
```

代码 5.7 TransImple.cpp

在 79 行处打断点，单步调试至此处，可以看到 State 变量 q 的值为“-842150451”，对应的十六进制值为“0xFFFFFFFF”②，为常见的未初始化错误。此时表达式“0 <= q”不成立，返回值为“false”, 程序在此处中止。

查看函数 DFA::usefulf() 的实现，如代码 5.8 。

```
StateTo<State> newnames;
newnames.set_domain(Q.size());

// All components will be constructed into a special structure :
DFA_components ret;
State st;
for (st = 0; st < Q.size(); st++)
{
    // If this is a Usefulf State, carry it over by giving it a name
    // in the new DFA.
    if (freachable.contains(st))
    {
        newnames.map(st) = ret.Q.allocate();
    }
}
```

代码 5.8 DFA.cpp

在代码 5.8 中将“final-reachable ”状态保存到 StateTo<State> 变量 newnames 中，通过“ret.Q.allocate()”为状态命名新的状态名，作为新的自动机的状态名。

②调试时使用 64 位编译器产生的二进制文件，在 64 位系统下运行。

DFA_components 变量 ret 用于构建新的自动机，再看函数内构造新的自动机的主要实现部分，如代码 5.9

```
130 for (it = 0; !a.iter_end(it); it++)
131 {
132     b = a.iterator(it);
133     ret.T.add_transition(stprime, b, newnames.lookup(T.transition_on_range(st, b)));
134 }
```

代码 5.9　DFA.cpp

根据代码 5.7 ，可以知道程序中止的地方为代码 5.9，133 行。其中 State 变量为当前需要进行操作的状态，CharRange 变量 b 为当前状态转移输入字符。查看 T.transition_on_range(st, b)) 的实现，如代码 5.10

```
// Compute the image of r, and CharRange it under *this.
inline State DTransRel::transition_on_range(const State r, const CharRange a) const
{
    assert(class_invariant());
    assert(0 <= r && r < domain());
    return(lookup(r).range_transition(a));
}
```

代码 5.10　DTransRel.cpp

由代码 5.10 可知，T.transition_on_range(st, b)) 将返回原自动机中，状态 st 经过输入字符 b 转移之后的目标状态。

查看 newnames.lookup() 的实现，如代码 5.11

```
// The actual mapping function
// First, a const lookup operator.
template<class T>
inline const T& StateTo<T>::lookup(const State r) const
{
    assert(class_invariant());
    // First check that it's in bounds
    assert(0 <= r && r < domain());
    return(data[r]);
}
```

代码 5.11　StateTo.h

在本例中，模板类 StateTo<T> 的模板参数 “T” 为 State。则 newnames.lookup(T.transition_on_range(st, b)) 为原自动机中状态 st 经过字符 b 转移后的目标状态在新自动机中的状态。然后通过 ret.T.add_transition() 保存新的转移关系。对所有的状态进行以上操作之后，通过变量 ret 构造新的自动机。

经过单步调试发现，表5.1 中，状态 q_2 经过字符“1”将转移到状态 q_5，而在代码 5.8 中，状态 q_5 不满足“if (freachable.contains(st))”，所以状态 q_5 未被新的自动机保存，进而在代码 5.9 中，当 st 为状态 q_2 且 b 为字符“1”时，“newnames.lookup(T.transition_on_range(st, b))”将返回未经初始化的值“-842150451”，于是在代码 5.7 中，State 变量 q 的值为“-842150451”，导致程序在此处中止。

5.2.3 解决方法

如代码 4.3 所示，文件 State.h 中将无效状态设置为“Invalid”。在代码 5.8增加处理不满足条件“freachable.contains(st)”的状态的内容，将原自动机中的非“final-reachable”状态标记为“Invalid”，更改后为代码 5.12

```
for (st = 0; st < Q.size(); st++)
{
    // If this is a Usefulf State, carry it over by giving it a name
    // in the new DFA.
    if (freachable.contains(st))
    {
        newnames.map(st) = ret.Q.allocate();
    }
    else                            // 新增
    {                               // 新增
        newnames.map(st) = Invalid; // 新增
    }                               // 新增
}
```

代码 5.12 更改后的 DFA.cpp

在代码 5.9 中，在添加新的转移关系之前判断当前状态和目标状态是否都是有效状态，若都为有效状态，则添加新的转移关系。更改后为代码 5.13

```
    State stdest;                                          // 新增
    stdest = newnames.lookup(T.transition_on_range(st, b)); // 新增

    if (stprime != Invalid &&  stdest != Invalid)          // 新增
    {                                                      // 新增
        ret.T.add_transition(stprime, b, stdest));         // 修改
    }                                                      // 新增
```

代码 5.13 更改后的 DFA.cpp

更改后函数 DFA::usefulf() 成功移除图 C.1(a) 中的陷阱状态 q_5 ，将图 C.1(a) 转换成图 C.1(b)。更改后函数 DFA::usefulf() 成功移除图 C.4(a) 中的陷阱状态 q_1 ，将图 C.4(a) 转换成图 C.4(b) 。

5.3　函数 DFA::min_Hopcroft() 运行错误

实例化如表 5.3 所示的自动机。(含有初始不可达状态③（start-unreachable))

表 5.3　图C.5(a)的转移函数

状态说明	状态	输入字符	
		0	1
开始状态 (start)	q_0	q_1	q_2
	q_1	q_5	q_2
	q_2	q_3	q_6
	q_3	q_2	q_4
结束状态 (accept)	q_4	q_8	q_1
	q_5	q_1	q_6
	q_6	q_7	q_2
	q_7	q_6	q_8
结束状态 (accept)	q_8	q_5	q_4
初始不可达状态	q_9	q_7	q_5

函数 DFA::min_Hopcroft() 是 Hopcroft 算法在 FIRE engine 中的实现，Hopcroft 算法在文中[3] 描述如下:

Algorithm 4.8 (Hopcroft):

$P := [Q]_{E_0};$
$L := (\textbf{if } (|F| \le |Q \setminus F|) \textbf{ then } \{F\} \textbf{ else } \{Q \setminus F\} \textbf{ fi}) \times V;$
$\{\text{invariant: } [Q]_E \sqsubseteq P \sqsubseteq [Q]_{E_0} \wedge L \subseteq (P \times V)$
$\quad \wedge (\forall\, Q_0, Q_1, a : Q_0 \in Q \wedge (Q_1, a) \in L : \neg \mathit{Splittable}(Q_0, Q_1, a)) \Rightarrow (P = [Q]_E)\}$
$\textbf{do } L \neq \emptyset \longrightarrow$
$\quad \textbf{let } Q_1, a : (Q_1, a) \in L;$
$\quad P_{old} := P;$
$\quad L := L \setminus \{(Q_1, a)\};$
$\quad \{\text{invariant: } [Q]_E \sqsubseteq P \sqsubseteq P_{old}\}$
$\quad \textbf{for } Q_0 : Q_0 \in P_{old} \wedge \mathit{Splittable}(Q_0, Q_1, a) \textbf{ do}$
$\quad\quad Q_0' := \{p : p \in Q_0 \wedge T(p, a) \in Q_1\};$
$\quad\quad P := P \setminus \{Q_0\} \cup \{Q_0 \setminus Q_0', Q_0'\};$
$\quad\quad \textbf{for } b : b \in V \textbf{ do}$
$\quad\quad\quad \textbf{if } (Q_0, b) \in L \longrightarrow L := L \setminus \{(Q_0, b)\} \cup \{(Q_0', b), (Q_0 \setminus Q_0', b)\}$
$\quad\quad\quad [\!] \; (Q_0, b) \notin L \longrightarrow$
$\quad\quad\quad\quad L := L \cup (\textbf{if } (|Q_0'| \le |Q_0 \setminus Q_0'|) \textbf{ then } \{(Q_0', b)\} \textbf{ else } \{(Q_0 \setminus Q_0', b)\} \textbf{ fi})$
$\quad\quad\quad \textbf{fi}$
$\quad\quad \textbf{rof}$
$\quad \textbf{rof}$
$\quad \{(\forall\, Q_0 : Q_0 \in P : \neg \mathit{Splittable}(Q_0, Q_1, a))\}$
$\textbf{od}\{P = [Q]_E\}$

③从开始状态不可以到达的状态，如图 C.5(a) 中状态 q_9 。

5.3.1 运行结果

执行函数 DFA::min_Hopcroft()，程序在文件 CRSet.h，第 146 行的“assert(!iter_end(it))”处触发 assert 中止，如代码 5.14 所示。调用函数“CRSet::iterator()”的地方为文件 min-hop.cpp，第 103 行，“State r(split(p, q, C.iterator(L[q]), P));”的“C.iterator(L[q])”处。

```
142 // Fetch the it'th CharRange in *this.
143 inline const CharRange& CRSet::iterator(const int it) const
144 {
145     assert(class_invariant());
146     assert(!iter_end(it));
147     return(data[it]);
148 }
149
150 // Is there even in it'th CharRange?
151 inline int CRSet::iter_end(const int it) const
152 {
153     assert(class_invariant());
154     assert(0 <= it);
155     return(it >= in_use);
156 }
```

代码 5.14 CRSet.h

5.3.2 错误原因

以表 5.3 中实例化的自动机为例，CRSet 变量 C = { 0, 1}，在程序中止处打断点，调试至此处可看到 int 变量 it 为“2”，因 C 中只有两个值“1、2”，所以传入参数为“2”时，超出可以 C 的范围，于是在语句“assert(!iter_end(it));”处导致程序中止。

算法迭代过程如表 D.1 所示。

在第 11 次迭代中，DFA::split() 函数将等价类 {0,1,5} 分割为等价类 {{0},{1,5}}，此时 $p = 0, q = 0, r = 1$，满足条件 $|Q_0'| \leq |Q_0 \setminus Q_0'|$，执行 L[r] = L[p] =1，L[p] = C.size() = 2。在第 12 次迭代中，由于第 11 次迭代中 $p = q$ ，所以 L[q] 的值将被更改为“2”，于是在 C.iterator(L[q]) 处传入一个超过 C 的范围的值，程序触发 assert 中止。

5.3.3 解决方法

函数 DFA::min_Hopcroft() 的主要实现部分如代码 5.15。

```
    for (repr.iter_start(p); !repr.iter_end(p); repr.iter_next(p))
    {
        //分割等价类
        ……
    }
```

代码 5.15　min-hop.cpp

在分割等价类之前，判断 L[q] 是否等于“C.size()”，若是，则减去“1”，更改后为代码 5.16

```
    for (repr.iter_start(p); !repr.iter_end(p); repr.iter_next(p))
    {
        if(L[q] == C.size()) // 新增
        {                    // 新增
            L[q]--;          // 新增
        }                    // 新增
        //分割等价类
        ……
    }
```

代码 5.16　min-hop.cpp

更改后函数 DFA::min_Hopcroft() 运行不再中止。

5.4　测试结果汇总

经过 5.1 节、5.2 节和 5.3 节的修改，FIRE engine 中的最小化算法及最小化算法用到的帮助函数都可以成功运行。本节内容为测试正确的结果和一些未能解决的问题的汇总。

5.4.1　不改变已经是最小的 DFA 接受的 $\mathcal{L}$

接受一个 $\mathcal{L}$ 的最小的 DFA 是唯一的，最小的且接受相同 $\mathcal{L}$ 的 DFA 仅仅在状态命名上有差别[4]。因此最小化算法有性质 5.1 。

性质 5.1： 最小化算法不能使最小的 DFA 接受的 $\mathcal{L}$ 发生改变。

注释 5.1： DFA 中的初始不可达（start-unreachable）状态和陷阱状态（final-unreachable）状态不影响 DFA 接受的 $\mathcal{L}$。

对于性质 5.1 ，以表 5.4 中的一组最小的 DFA 为数据，测试结果表 5.5 所示。

表 5.4　一组最小的 DFA

序号	数据	属性	备注
1	图 C.4(a)	最小的	含有陷阱状态
2	图 C.4(b)	最小的	不含陷阱状态
3	图 C.5(a)	最小的	含有初始不可达状态
4	图 C.5(b)	最小的	不含初始不可达状态
5	图 C.6(a)	最小的	含有初始不可达状态
6	图 C.6(b)	最小的	不含初始不可达状态
7	图 C.7(a)	最小的	含有初始不可达状态
8	图 C.7(b)	最小的	不含初始不可达状态

表 5.5　最小化算法对性质 5.1 的测试结果

算法	1	2	3	4	5	6	7	8
DFA::min_Brzozowski()	√	√	×	×	✓	√	✓	√
DFA::min_Hopcroft()（修改前）	√	√	中止	√	√	√	√	√
DFA::min_Hopcroft()（修改后）	√	√	√	√	√	√	√	√
DFA::min_HopcroftUllman()	√	√	√	√	√	√	√	√
DFA::min_dragon()	√	√	√	√	√	√	√	√
DFA::min_Watson()	√	√	√	√	√	√	√	√

对表 5.5 中的测试结果有两个算法需要额外关注——“DFA::min_Brzozowski()”和“DFA::min_Hopcroft()”。

结果总结对比如下：

- 对于 DFA 中的陷阱状态，可以调用函数“DFA::usefulf()”函数去除。表 5.5 中在执行最小化函数之前，除了算法“DFA::min_Brzozowski”，都执行了函数“DFA::usefulf()”；
- 除了算法“DFA::min_Brzozowski()”外，其他最小化算法均不能去除 DFA 中的初始不可达状态（start-unreachable）。数据为第 5 个和第 7 个，以 ✓ 标示，而不是 √；
- 对于算法“DFA::min_Hopcroft()”的 assert 中止，在发生 assert 中止的第 3 个数据中，含有初始不可达状态，作为对照组且含有初始不可达状态的第 5 个和第 7 个数据确没有发生 assert 中止，所以本文认为此算法的中止的问题不是 DFA 中初始不可达状态导致；

- 对于第 3 个和第 4 个数据，算法“DFA::min_Brzozowski()”改变了 DFA，输入的 DFA 仅有 10/9 个状态，算法“DFA::min_Brzozowski()”输出了含有数百个状态的自动机。

5.4.2　最小化功能测试

本节内容为最小化算法对定义功能的测试汇总。

表 5.6　一组 DFA

序号	数据	属性	预期结果
1	图 C.1(a)		图 C.1(d)
2	图 C.2(a)		图 C.2(c)
3	图 C.2(b)	最小的	图 C.2(b)
4	图 C.3(a)		图 C.3(c)
5	图 C.3(b)	最小的	图 C.3(b)

表 5.7　最小化算法的定义功能的测试结果

算法	1	2	3	4	5
DFA::min_Brzozowski()	√	√	√	√	√
DFA::min_Hopcroft()（修改前）	√	√	×	中止	√
DFA::min_Hopcroft()（修改后）	√	√	×	√	√
DFA::min_HopcroftUllman()	√	√	√	√	√
DFA::min_dragon()	√	√	√	√	√
DFA::min_Watson()	√	√	√	√	√

对于第 3 个数据，算法“DFA::min_Hopcroft()”无论是否修改过，均输出代码 5.17 。

```
DFA
Q = [0,2)
S = { 0 }
F = { 1 }
Transitions =
0->{ ['a','b']->1 }
1->{ 'a'->1 }

current = -1
```

代码 5.17　第 3 个数据在算法“DFA::min_Hopcroft()”中的输出

第 3 个数据的转移函数如表 5.8(a) 所示，代码 5.17 的转移函数如表 5.8(b) 所示

表 5.8 第 3 个数据在算法“DFA::min_Hopcroft()”中的输入输出对比

(a) 输入

状态说明	状态	输入字符	
		a	b
开始状态 (start)	q_0	q_1	q_2
结束状态 (accept)	q_1	q_2	-
结束状态 (accept)	q_2	-	q_1

(b) 输出

状态说明	状态	输入字符	
		a	b
开始状态 (start)	q_0	q_1	q_1
结束状态 (accept)	q_1	q_1	-

图 C.2(b) 原本是一个最小的 DFA，由表 5.8 可知，函数算法“DFA::min_Hopcroft()”已经改变了图 C.2(b) 的状态数，由“3”变到“2”，已经不满足性质 5.1 。由此可以认为 FIRE engine 中算法“DFA::min_Hopcroft()”未完成其定义功能。

对于算法“DFA::min_Hopcroft()”和算法“DFA::min_Brzozowski”的额外测试见附录。

第六章　总结

本文总结如下

1. 对于 DFA 中的陷阱状态，可以使用函数“DFA::usefulf()”去除；
2. 除了算法 DFA::min_Brzozowski()，其他算法都不能去除 DFA 中的初始不可达状态（start-unreachable）；

致谢

虽为致谢环境，其实就是一个 Chapter，为啥这么费事？因为，致谢一章没有编号。直接使用 `\chapter*{}`的话，页眉又不符合工作手册要求，而且要往目录中添加该章节，还需要添加两行代码；为了简单快捷的设计出符合要求，又方便用户使用，只能借 `\backmatter` 模式和本模板自定义的 `\comtinuematter` 模式配合环境来做了。

参考文献

[1] Watson B. The design and implementation of the fire engine: a c++ toolkit for finite automata and regular expressions[J]. 1994.

[2] Watson B W. A taxonomy of finite automata construction algorithms[J]. 1993.

[3] Watson B W. A taxonomy of finite automata minimization algorithms[J]. 1993.

[4] 蒋宗礼, 姜守旭. 形式语言与自动机理论[M]. 三. 北京: 清华大学出版社, 2013: 140-156.

[5] Microsoft Docs. assert 函数输出的诊断[EB/OL]. https://docs.microsoft.com/zh-cn/cpp/c-runtime-library/reference/assert-macro-assert-wassert?view=vs-2019.

附录 A　一些基本定义

惯例 A.1 (幂集)： 对于任意集合 A，我们使用 $\mathcal{P}(A)$ 代表 A 的所有子集。$\mathcal{P}(A)$ 也叫做 A 的幂集。有时也写作 2^A。

惯例 A.2 (函数集)： 对于集合 A 和 B，$A \longrightarrow B$ 代表所有从 A 到 B 的函数的集合。而 $A \not\longrightarrow B$ 代表所有从 A 到 B 的“partial functions”。

注释 A.1： 对于集合 A, B, 关系 $\mathcal{C} \subseteq A \times B$，我们可以把 $\mathcal{C}$ 理解为函数 $\mathcal{C} \in A \longrightarrow \mathcal{P}(B)$。

惯例 A.3 (元组投影)： 对于 n 元组 $t = (x_1, x_2, \cdots, x_n)$，我们使用符号 $\pi_i(t)$ $(1 \leq i \leq n)$ 代表元组元素 x_i；我们使用符号 $\bar{\pi}_i(1 \leq i \leq n)$ 代表 $(n$-1) 元组 $(x_1, \cdots, x_{i-1}, x_{i+1}, \cdots, x_n)$。$\pi$ 和 $\bar{\pi}$ 自然扩展到元组集。

惯例 A.4 (组合关系)： 给出集合 A, B, C 和两个关系 $E \subseteq A \times B$ 和 $F \subseteq B \times C$，定义组合关系（插入操作符 $\circ$）:

$$E \circ F = \{(a, c) : (\exists b : b \in B : (a, b) \in E \wedge (b, c) \in F)\}$$

惯例 A.5 (等价关系的等价类)： 对任何集合 A 上的等价关系 E, 我们使用 $[A]_E$ 代表等价类集合，即:

$$[A]_E = \{[a]_E : a \in A\}$$

集合 $[A]_E$ 也叫做 A 的由 E 引出的“划分（partition）”。

定义 A.1 (等价类的指数)： 对于集合 A 上的等价关系 E, 定义 $\sharp E = |[A]_E|$。$\sharp E$ 也叫做 E 的“指数”。

定义 A.2 (字母表)： 字母表是有限大小的非空集合。

定义 A.3 (等价关系的细化)： 对于等价关系 E 和 E'（在集合 A 上），当且仅当 $E \subseteq E'$，E 是 E' 的“细化”。

定义 A.4 **(划分的细化关系 $\sqsubseteq$)：** 对于等价关系 E 和 E'（在集合 A 上)，当且仅当 $E \subseteq E'$，$[A]_E$ 也被称为 $[A]_{E'}$ 的细化（写作 $[A]_E \sqsubseteq [A]_{E'}$)。当且仅当 E 下的每一个等价类完全包含在 E' 下的某些等价类时，等价命题是 $[A]_E \sqsubseteq [A]_{E'}$ 。

定义 A.5 **(元组和关系反转)：** 对一个 n 元组 $(x_1, x_2, \cdots, x_n)$，定义反转为函数 R (后缀和上标)

$$(x_1, x_2, \cdots, x_n)^R = (x_n, \cdots, x_2, x_1)$$

给出一个集合元组 A，定义 $A^R = \{x^R : x \in A\}$。

附录 B　有限自动机

本节中我们定义有限自动机、其性质及其一些变化。大部分定义直接取自

定义 B.1 (**有限自动机** ($Finiteautomata, FA$))**：** 自动机是一个 6 元组 (Q, V, T, E, S, F)，其中:

- Q 是有限状态集;
- V 是一个字母表;
- $T \in \mathcal{P}(P \times V \times Q)$ 是一个转换关系;
- $E \in \mathcal{P}(Q \times Q)$ 是一个 ϵ-转换关系（空转换）;
- $S \subseteq Q$ 是开始状态集;
- $F \subseteq Q$ 是结束状态集;

字母表和函数 $\mathcal{P}$ 的定义分别在“定义A.2”和“惯例A.1”。

注释 B.1**：** 我们也会在状态转换关系的表示上采取一定的自由。例如, 我们也把转移关系写成 $T \in V \longrightarrow \mathcal{P}(Q \times Q), T \in Q \times Q \longrightarrow \mathcal{P}(V), T \in Q \times V \longrightarrow \mathcal{P}(Q), T \in Q \longrightarrow \mathcal{P}(V \times Q), E \in Q \longrightarrow \mathcal{P}(Q)$。每种情况下，$Q$ 的从左到右的顺序会是“preserved”; 例如, 函数 $T \in Q \longrightarrow \mathcal{P}(V \times Q)$ 定义为 $T(p) = \{(a, q) : (p, a, q) \in T\}$。所使用的签名将从上下文中清除。详见备注 A.3。“$\longrightarrow$”的定义出现在惯例 A.2。

由于本文中我们只考虑有限自动机，所以我们将会频繁的使用简化术语“自动机”。

B.1　有限自动机的性质

本小节将会定义一些有限自动机（下称 FA）的性质。为了使定义更加简洁明了, 我们引进三个特殊的 FA: $M = (Q, V, T, E, S, F)$, $M_0 = (Q_0, V_0, T_0, E_0, S_0, F_0)$, $M_1 = (Q_1, V_1, T_1, E_1, S_1, F_1)$。

定义 B.2 (FA 的大小)： 定义一个 FA 的大小为 $|M| = |Q|$。

定义 B.3 (FA 的同构 $\cong$)： 我们把同构定义为 FA 的等价关系。当且仅当 $V_0 = V_1$，并且存在双射 $g \in Q_0 \longrightarrow Q_1$ ，使得

- $T_1 = \{(g(p,q), a, g(q)) : (p,a,q) \in T_0\}$
- $E_1 = \{(g(p,q), a, g(q)) : (p,q) \in E_0\}$
- $S_1 = \{g(s) : s \in S_0\}$
- $F_1 = \{g(f) : f \in F_0\}$

时 M_0 和 M_1 是同构的（写作 $M_0 \cong M_1$）。

定义 B.4 (转移关系 T 的扩展)： 我们把 $T \in V \longrightarrow \mathcal{P}(Q \times Q)$ 到 $T^* \in V^* \longrightarrow \mathcal{P}(Q \times Q)$ 的转换关系以如下方式扩展:

$$T^*(\epsilon) = E^*$$

且对于 $(a \in V, w \in V^*)$ 有

$$T^*(aw) = E^* \circ T(a) \circ T^*(w)$$

操作符 $\circ$ 在惯例 A.5 中定义。这个定义也可以对称的表示。

注释 B.2： 有时候我们也使用把转移关系写成: $T^* \in Q \times Q \longrightarrow \mathcal{P}(V^*)$。

定义 B.5 (左语言和右语言)： 状态（M 中）的左语言由函数 $\overleftarrow{\mathcal{L}}_M \in Q \longrightarrow \mathcal{P}(V^*)$ 给出，其中:

$$\overleftarrow{\mathcal{L}}_M(q) = (\cup s : s \in S : T^*(s,q))$$

状态（M 中）的右语言由函数 $\overrightarrow{\mathcal{L}}_M \in Q \longrightarrow \mathcal{P}(V^*)$ 给出，其中

$$\overrightarrow{\mathcal{L}}_M(q) = (\cup f : f \in F : T^*(q,f))$$

通常在没有歧义的时候移除下标 M。

定义 B.6 (FA 的语言)： 有限自动机的语言由函数 $\mathcal{L}_{FA} \in FA \longrightarrow \mathcal{P}(V^*)$ 给出，该函数的定义为:

$$\mathcal{L}_{FA}(M) = (\cup s, f : s \in S \wedge f \in F : T^*(s,f))$$

定义 B.7 (完全自动机 ($Complete$))： 一个完全有限自动机满足:

$$Complete(M) \equiv (\forall q, a : q \in Q \wedge a \in V : T(q,a) \neq \emptyset)$$

定义 B.8 (ϵ-$free$)**：** 当且仅当 $E = \emptyset$ 时，M 是 ϵ-$free$ 的。

定义 B.9 (Start-useful **自动机**)**：** 一个 $Useful_s$ 有限自动机定义如下:

$Useful_s(M) \equiv (\forall q : q \in Q : \overleftarrow{\mathcal{L}}(q) \neq \emptyset)$

定义 B.10 (Final-useful **自动机**)**：** 一个 $Useful_f$ 有限自动机定义如下:

$Useful_f(M) \equiv (\forall q : q \in Q : \overrightarrow{\mathcal{L}}(q) \neq \emptyset)$

注释 B.3**：** $Useful_s$ 和 $Useful_f$ 与 FA 的反转密切相关（见变换 B.22），对所有的 $M \in FA$，有 $Useful_f(M) \equiv Useful_s(M^R)$。

定义 B.11 (Useful **自动机**)**：** $Useful$ 有限自动机是一个只有可达状态的有限自动机:

$Useful(M) \equiv Useful_s(M) \wedge Useful_f(M)$

性质 B.1 (**确定性有限自动机** (DFA))**：** 当且仅当

- 无多重初始状态;
- 无 ϵ 转移;
- 转移函数 $T \in Q \times V \longrightarrow \mathcal{P}(Q)$ 不将 $Q \times V$ 映射至多重状态。

时有限自动机 M 是确定性的。形式表达为:

$$Det(M) \equiv (|S| \leq 1 \wedge \epsilon - free(E) \wedge (\forall q, a : q \in Q \wedge a \in V : |T(q, a)| \leq 1))$$

定义 B.12 (FA **的确定性**)**：** DFA 代表所有确定性的有限自动机的集合。我们把 $FA \setminus DFA$ 称为非确定性有限自动机 ($NDFA$,$nondeterministic\ finite\ automata$) 的集合。

惯例 B.1 (DFA **的转换函数**)**：** 对于 $(Q, V, T, \emptyset, S, F) \in DFA$，我们考虑把转换函数记为 $T \in Q \times V \not\rightarrow Q$（$\not\rightarrow$ 的定义可以查看惯例 A.2）。当且仅当 DFA 是完全自动机的时候，转换函数是全函数。

性质 B.2 (弱确定性自动机)： 一些作者用比 Det 弱的确定性自动机的定义；使用左语言，定义如下：

$$Det'(M) \equiv (\forall q_0, q_1 : q_0 \in Q \wedge q_1 \in Q \wedge q_0 \neq q_1 : \overleftarrow{\mathcal{L}}(q_0) \cap \overleftarrow{\mathcal{L}}(q_1) = \emptyset)$$

很容易证明 $Det(M) \Rightarrow Det'(M)$。

定义 B.13 (DFA 的最小化)： 满足以下条件时，$M \in DFA$ 是最小化的：

$$Min(M) \equiv (\forall M' : M' \in DFA \wedge Complete(M') \wedge \mathcal{L}(M) = \mathcal{L}_{FA}(M') : |M| \leq |M'|)$$

Min 仅定义在 DFA 上。如果我们定义一个最小的但是仍然完全的 DFA，那么一些定义将会更加简单。它的定义如下：

$$Min_{\mathcal{C}}(M) \equiv (\forall M' : M' \in DFA \wedge Complete(M') \wedge \mathcal{L}_{FA}(M) = \mathcal{L}_{FA}(M') : |M| \leq |M'|)$$

$Min_{\mathcal{C}}$ 仅定义在完全 DFA 上。

定义 B.14 (DFA 的最小化)： 根据 Myhill-Nerode 定理，An M, such that Min(M)，是唯一的最小化 DFA，定理的相关介绍在[3]。

性质 B.3 (DFA 最小化的一个替代定义)： 为了最小化 DFA，使用定义（仅定义在 DFA 上)：

$$\begin{aligned} Minimal(Q, V, T, \emptyset, S, F) \equiv & (\forall q_0, q_1 : q_0 \in Q \wedge q_1 \in Q \wedge q_0 \neq q_1 : \overrightarrow{\mathcal{L}}(q_0) \neq \overrightarrow{\mathcal{L}}(q_1)) \\ & \wedge Useful(Q, V, T, \emptyset, S, F) \end{aligned}$$

有 $Minimal(M) \equiv Min(M)$（对所有 $M \in DFA$）。很容易证明 $Min(M) \Rightarrow Minimal(M)$。The reverse direction follows from the Myhill-Nerode 定理。

与 $Min_{\mathcal{C}}$ 相似的定义是（同样也只定义在 DFA 上)：

$$\begin{aligned} Minimal_{\mathcal{C}}(Q, V, T, \emptyset, S, F) \equiv & (\forall q_0, q_1 : q_0 \in Q \wedge q_1 \in Q \wedge q_0 \neq q_1 : \overrightarrow{\mathcal{L}}(q_0) \neq \overrightarrow{\mathcal{L}}(q_1)) \\ & \wedge Useful(Q, V, T, \emptyset, S, F) \end{aligned}$$

有 $Minimal_{\mathcal{C}}(M) \equiv Min_{\mathcal{C}}(M)$ 的性质（对于所有的 $M \in DFA$）。很容易证明 $Min_{\mathcal{C}}(M) \Rightarrow Minimal_{\mathcal{C}}(M)$。The reverse direction follows from the Myhill-Nerode 定理。

B.2　有限自动机的变换

变换 B.1 (FA 反转)： FA 反转由后缀（上标）函数 $R \in FA \longrightarrow FA$ 给出，它的定义如下:

$$(Q, V, T, S, F)^R = (Q, V, T^R, E^R, F, S)$$

函数 R 满足

$$(\forall M : M \in FA : (\mathcal{L}(M))^R = \mathcal{L}_{FA}(M^R))$$

变换 B.2 (移除开始状态不可达状态)： 变换 $useful_s \in FA \longrightarrow FA$ 移除开始状态不可达状态:

$$\begin{array}{rcll} useful_s(Q,V,T,E,S,F) & = & \textbf{let} & U = SReachable(Q,V,T,E,S,F) \\ & & \textbf{in} & \\ & & & (U, V, T \cap (U \times V \times U), E \cap (U \times U), S \cap U, F \cap U) \\ & & \textbf{end} & \end{array}$$

函数 $useful_s$ 满足

$$(\forall M : M \in FA : Useful_s(useful_s(M)) \wedge \mathcal{L}_{FA}(useful_s(M)) = \mathcal{L}_{FA}(M))$$

变换 B.3 (子集构造)： 函数 $subset$ 把一个 ϵ-$free$ FA 转换为一个 DFA (in the **let** clause $T' \in \mathcal{P}(Q) \times V \longrightarrow \mathcal{P}(\mathcal{P}(Q))$):

$$\begin{array}{rcll} subset(Q,V,T,\emptyset,S,F) & = & \textbf{let} & T'(U,a) = \{(q : q \in U : T(q,a))\} \\ & & & F' = \{U : U \in \mathcal{P}(Q) \wedge U \cap F \neq \emptyset\} \\ & & \textbf{in} & \\ & & & (\mathcal{P}(Q), V, T', \emptyset, \{S\}, F') \\ & & \textbf{end} & \end{array}$$

有时候也把它说成“幂集”构造。

性质 B.4 (子集构造)： 设 $M_0 = (Q_0, v, t_0, \emptyset, S_0, F_0)$ 和 $M_1 = subset(M_0)$ 为有限自动机。通过子集构造，状态集 M_1 成为 $\mathcal{P}(Q_0)$。有如下性质:

$$(\forall p : p \in \mathcal{P}(Q_0) : \overrightarrow{\mathcal{L}}_{M_1}(p) = (q : q \in p : \overrightarrow{\mathcal{L}}_{M_1}(q)))$$

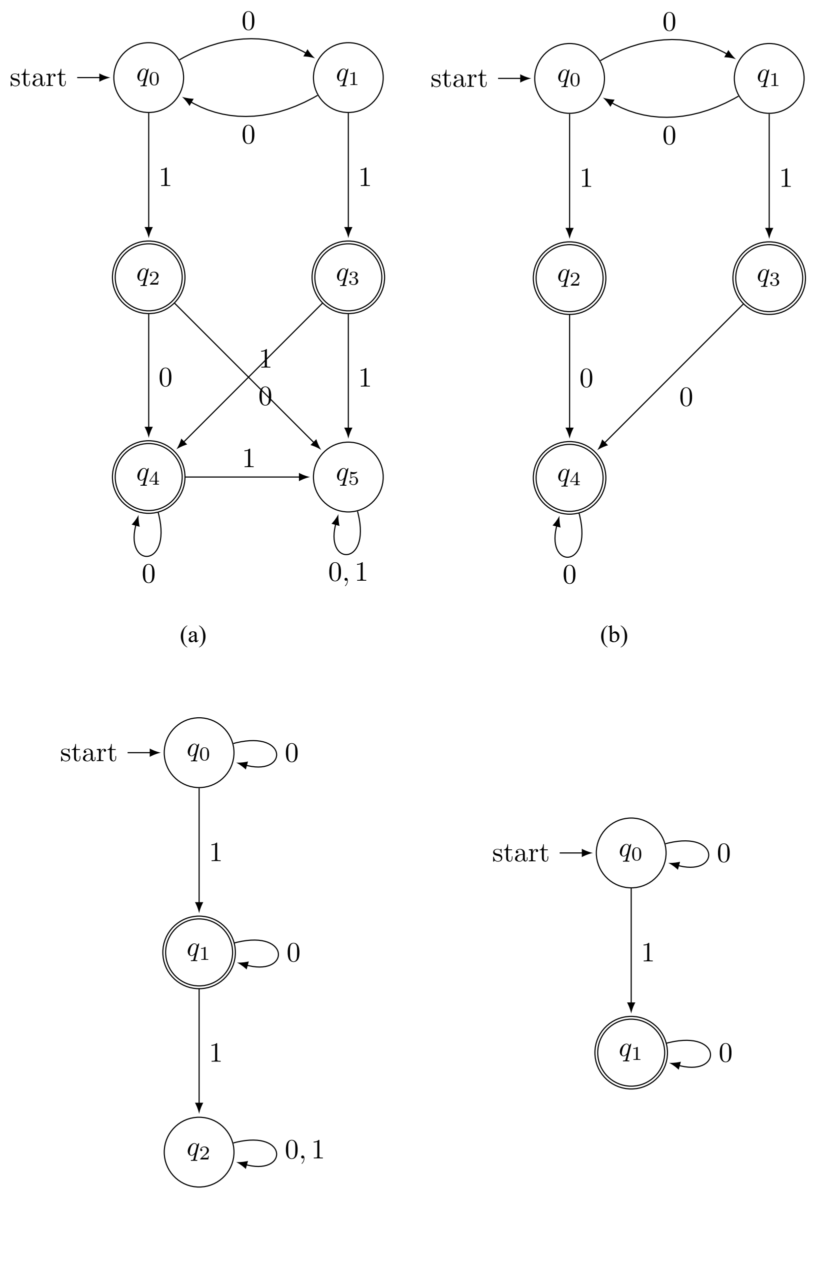
定义 B.15 (优化子集构造)： 函数 $subsetopt$ 把一个 $\epsilon\text{-}free\ FA$ 转换为一个 DFA 。此函数是 $subset$ 的一个优化版本:

$$\begin{aligned}
subset(Q, V, T, \emptyset, S, F) \;=\; & \textbf{let} \quad T'(U, a) = \{(q : q \in U : T(q, a))\} \\
& \qquad\;\; Q' = \mathcal{P}(Q) \setminus \{\emptyset\} \\
& \qquad\;\; F' = \{U : U \in \mathcal{P}(Q) \wedge U \cap F \neq \emptyset\} \\
& \textbf{in} \\
& \qquad\;\; (Q', V, T' \cap (Q' \times V \times Q'), \emptyset, \{S\}, F') \\
& \textbf{end}
\end{aligned}$$

除了性质 $\mathcal{L}_{FA}(subsetopt(M)) = \mathcal{L}(M)$（对所有的 $M \in FA$）之外，函数 $subsetopt$ 还满足

$$(\forall M : M \in FA \wedge \epsilon\text{-}free(M) : Det(subset(M)))$$

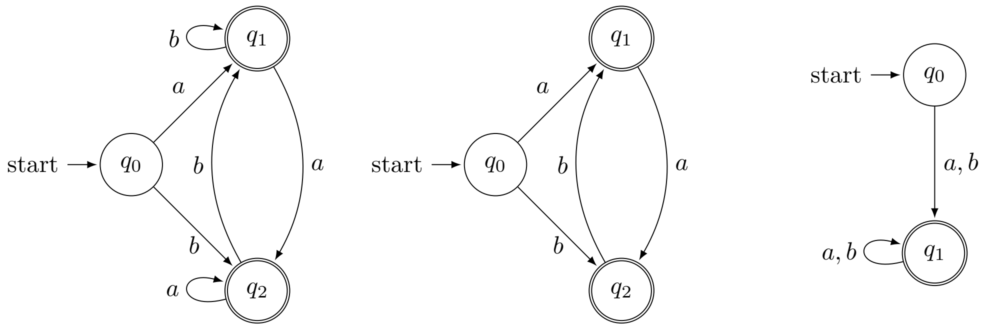
附录 C　自动机状态转移图



(a)　(b)

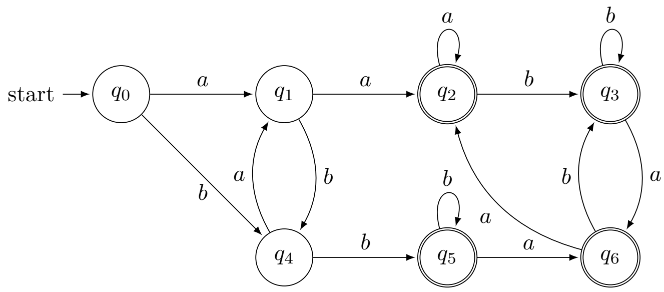
(c)　(d)

图 C.1　(a) 接受$\mathcal{L}$=0*10* 的自动机[4] fig 5-4；(b) 图 (a) 去除非“final-reachable”状态q_5；(c) 与图 (a) 的 DFA 同构的含有陷阱状态的最小 DFA；(d) 与图 (a) 的 DFA 同构的不含陷阱状态的最小 DFA。

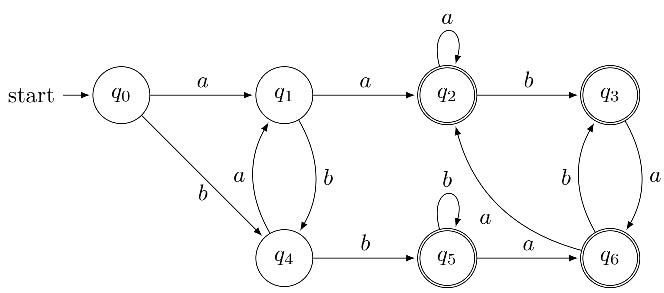


(a) (b) (c)

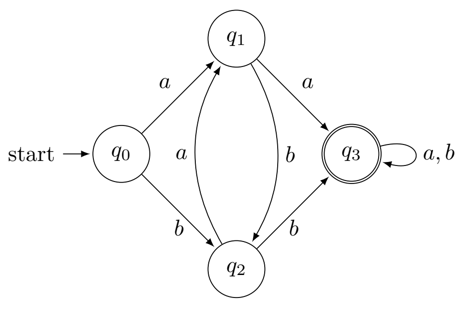
图 C.2 (a) 一个 DFA；(b) 图 (a) 去除转移关系 $T = \{(1 \times \text{'b'} \times 1), (2 \times \text{'a'} \times 2)\}$ 后的最小的 DFA ；(c) 与图 (a) 同构的最小的 DFA



(a)

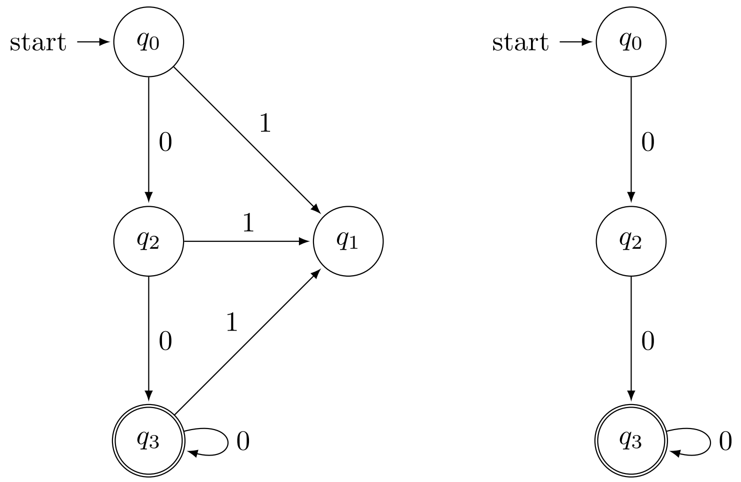


(b)



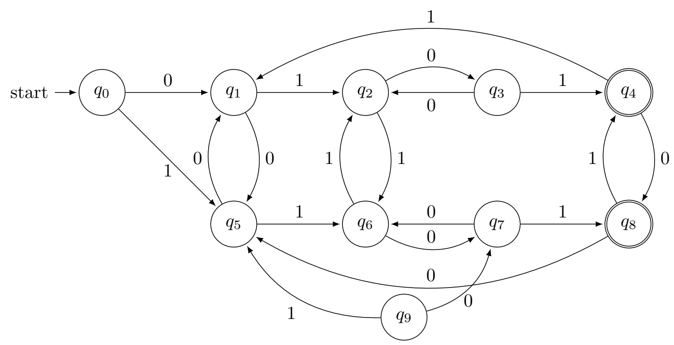
(c)

图 C.3 (a) 一个 DFA；(b) 图 (a) 去除转移关系 $T = \{(3 \times \text{'b'} \times 3)\}$ 后的最小的 DFA ；(c) 与图 (a) 同构的最小的 DFA

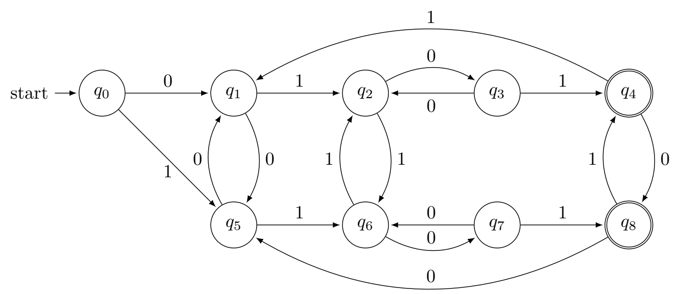


(a) (b)

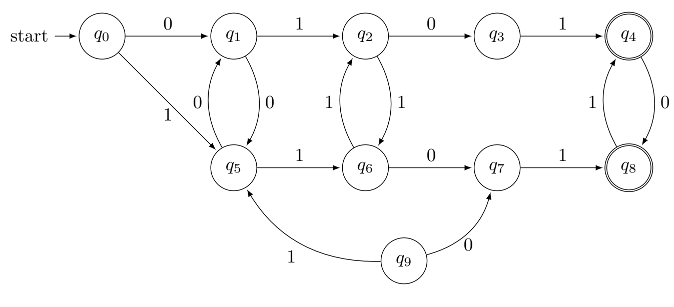
图 C.4　(a) 接受$\mathcal{L}$=000* 的最小的 DFA；(b) 图 (a) 去除非“final-reachable”状态q_1；



(a)



(b)



(c)

图 C.5　(a) 一个 DFA；(b) 图 (a) 去除“start-unreachable”状态q_9；(c) 图 (a) 移除转移关系 $T = \{(4 \times' 1' \times 1), (8 \times' 0' \times 5), (3 \times' 0' \times 2), (7 \times' 0' \times 6)\}$ 后的 DFA。

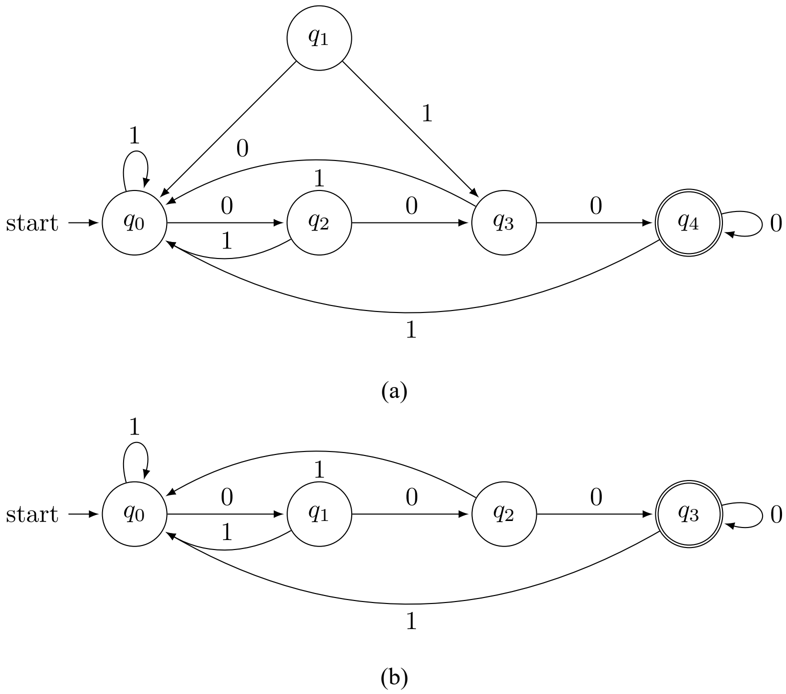


图 C.6 (a) 接受$\mathcal{L}=\{x000|x\in\{0,1\}^*\}$ 的最小的 DFA；(b) 图 (a) 去除初始不可达状态q_1；

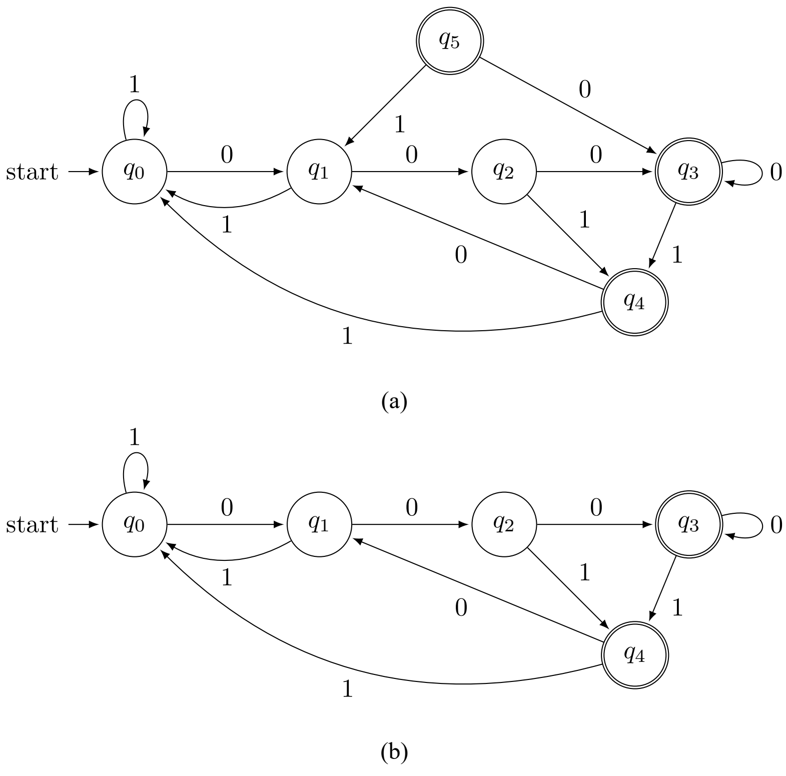


图 C.7 (a) 接受$\mathcal{L}=\{x000|x\in\{0,1\}^*\cup\{x001|x\in\{0,1\}^*\}$ 的最小的 DFA；(b) 图 (a) 去除初始不可达状态q_5；

附录 D 算法迭代过程

表 D.1 图C.5(a)的 DFA 在 DFA::min_Hopcroft 算法中的迭代过程

n	状态	p	q	$L[q]$	C	r	L										$repr$	P
							0	1	2	3	4	5	6	7	8	9		
1	前	0	4	1	1	-	0	0	0	0	1	0	0	0	0	0	{0,4}	{0,1,2,3,5,6,7,9},{4,8}
	后	0	4	1	1	3	0	0	0	2	1	0	0	0	0	0	{0,4}	{0,1,2,5,6,9},{3,7},{4,8}
2	前	4	4	1	1	-	0	0	0	2	1	0	0	0	0	0	{0,4}	{0,1,2,5,6,9},{3,7},{4,8}
	后	4	4	2	1	8	0	0	0	2	2	0	0	0	1	0	{0,4}	{0,1,2,5,6,9},{3,7},{4},{8}
3	前	0	3	1	1	-	0	0	0	1	2	0	0	0	1	0	{0,3,4,8}	{0,1,2,5,6,9},{3,7},{4},{8}
	后	0	3	1	1	-1	0	0	0	1	2	0	0	0	1	0	{0,3,4,8}	{0,1,2,5,6,9},{3,7},{4},{8}
4	前	3	3	1	1	-	0	0	0	1	2	0	0	0	1	0	{0,3,4,8}	{0,1,2,5,6,9},{3,7},{4},{8}
	后	3	3	1	1	-1	0	0	0	1	2	0	0	0	1	0	{0,3,4,8}	{0,1,2,5,6,9},{3,7},{4},{8}
5	前	4	3	1	1	-	0	0	0	1	2	0	0	0	1	0	{0,3,4,8}	{0,1,2,5,6,9},{3,7},{4},{8}
	后	4	3	1	1	-1	0	0	0	1	2	0	0	0	1	0	{0,3,4,8}	{0,1,2,5,6,9},{3,7},{4},{8}
6	前	8	3	1	1	-	0	0	0	1	2	0	0	0	1	0	{0,3,4,8}	{0,1,2,5,6,9},{3,7},{4},{8}
	后	8	3	1	1	-1	0	0	0	1	2	0	0	0	1	0	{0,3,4,8}	{0,1,2,5,6,9},{3,7},{4},{8}
7	前	0	3	0	0	-	0	0	0	0	2	0	0	0	1	0	{0,3,4,8}	{0,1,2,5,6,9},{3,7},{4},{8}
	后	0	3	0	0	2	2	0	0	0	2	0	0	0	1	0	{0,3,4,8}	{0,1,5},{2,6,9}{3,7},{4},{8}
8	前	3	3	0	0	-	2	0	0	0	2	0	0	0	1	0	{0,3,4,8}	{0,1,5},{2,6,9},{3,7},{4},{8}
	后	3	3	0	0	-1	2	0	0	0	2	0	0	0	1	0	{0,3,4,8}	{0,1,5},{2,6,9},{3,7},{4},{8}
9	前	4	3	0	0	-	2	0	0	0	2	0	0	0	1	0	{0,3,4,8}	{0,1,5},{2,6,9},{3,7},{4},{8}
	后	4	3	0	0	-1	2	0	0	0	2	0	0	0	1	0	{0,3,4,8}	{0,1,5},{2,6,9},{3,7},{4},{8}
10	前	8	3	0	0	-	2	0	0	0	2	0	0	0	1	0	{0,3,4,8}	{0,1,5},{2,6,9},{3,7},{4},{8}
	后	8	3	0	0	-1	2	0	0	0	2	0	0	0	1	0	{0,3,4,8}	{0,1,5},{2,6,9},{3,7},{4},{8}
11	前	0	0	1	1	-	1	0	0	0	2	0	0	0	1	0	{0,2,3,4,8}	{0,1,5},{2,6,9},{3,7},{4},{8}
	后	0	0	2	1	1	2	0	0	0	2	0	0	0	1	0	{0,2,3,4,8}	{0},{1,5},{2,6,9},{3,7},{4},{8}
12	前	2	0	2	-	-	-	-	-	-	-	-	-	-	-	-	-	——
	后	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	——

图 D.1 中的表头与 FIRE engine 中 Hopcroft 算法的实现中的变量一一对应。其中

- n 为迭代次数;
- “状态”意指进行等价类分割前和等价类分割后;
- p 为当前要进行等价类分割的等价类;

- q 为等价类分割的参数，$q \in Q$；
- L[q] 作为函数 C.iterator(L[q]) 的参数；
- C 为等价类分割的参数，$C \in V$；
- r 为分割开的等价类的代表；
- L 为 Hopcroft 算法中的 L；
- $repr$ 为等价类代表集合，其值为 P 中每个等价类中最小的状态的集合；
- P 为等价类集合；

附录 E　代码

一个实例化 DFA 类的例子如代码 E.1 所示。代码 E.1 中执行了 DFA::usefulf() 函数，去除多余的状态，让输出数据与原始数据相对应。代码 E.1 对应的 DFA 为图 C.6(b) 。

```
#include"DFA.h"
#include<iostream>
int main()
{
    DFA_components dfa_com1;

    // StateSet S  开始状态集
    dfa_com1.S.set_domain(10);
    dfa_com1.S.add(0);

    // StateSet F  结束状态集
    dfa_com1.F.set_domain(10);
    dfa_com1.F.add(3);

    // StatePool Q
    int i = 10;
    while (i--)
    {
        dfa_com1.Q.allocate();
    }

    // DTransRel T transition
    dfa_com1.T.set_domain(10);
    dfa_com1.T.add_transition(0, '0', 1);
    dfa_com1.T.add_transition(1, '0', 2);
    dfa_com1.T.add_transition(2, '0', 3);
    dfa_com1.T.add_transition(3, '0', 3);
    dfa_com1.T.add_transition(0, '1', 0);
    dfa_com1.T.add_transition(1, '1', 0);
    dfa_com1.T.add_transition(2, '1', 0);
    dfa_com1.T.add_transition(3, '1', 0);

    DFA dfa1(dfa_com1);
    dfa1.usefulf();

    std::cout<<dfa1<<std::endl;

    return 0;
}
```

代码 E.1　实例化 DFA 示例