

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： 年 月 日

指导教师已阅：_____（签字） 时间： 年 月 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。有限自动机 C++ 工具箱对 Hopcroft 算法（由 Hopcroft 提出）的实现需要输入数据为完全的确定性有限自动机。而对于其他三个最小化算法 HopcroftUllman 算法（由 Hopcroft 和 Ullman 提出）、dragon 算法（由 Aho Sethi 和 Ullman 提出），Waston 算法（由 Bruce William Watson 提出）则没有明确要求数据数据是否是为完全的确定性有限自动机。除了 Brzozowski 的算法外，以上四个最小化算法都依赖于计算状态的等价关系（或可区分的关系），有限自动机 C++ 工具箱对 Brzozowski 算法的实现有一定的缺陷，在某些情况下会输出错误的结果。

本文还为有限自动机 C++ 工具箱增加了用来计算从开始状态可以到达的状态的集合的算法 (SReachable)、移除有限自动机中从开始状态不可到达的状态的算法 (usefuls) 和构造完全自动机的算法 (complete)。

关键词：确定性有限自动机 最小化 算法 状态等价

Abstract

This paper is just a sample example for the users in learning the *X_DUthesis*. I will try my best to use the commands and environments which are involved by the *X_DUthesis*. 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: XD_Uthesis commands environments skills

Abstract

目 录

第一章 引言	1
1.1 FIRE engine	1
1.2 国内外研究现状	1
1.3 本文的工作	2
第二章 预备知识	3
2.1 基本定义	3
2.2 有限自动机	4
2.2.1 有限自动机的性质	6
2.2.2 有限自动机的变换	9
2.3 等价性和最小化	10
2.3.1 最小化	11
第三章 类 DFA 及相关类的设计	13
3.1 DFA 类	13
3.2 DFA_components 结构体	14
3.2.1 StatePool 类	15
3.2.2 StateSet 类	15
3.2.3 DTransRel 类	16
3.3 实例化类 DFA 对象	17
第四章 测试内容	19
4.1 如何测试	19
4.2 无限循环	20
4.2.1 运行结果	20
4.2.2 错误原因	20
4.2.3 解决方法	21
4.3 函数 DFA::useful() 运行错误	22

4.3.1	运行结果	22
4.3.2	错误原因	23
4.3.3	解决方法	25
4.4	函数 <code>DFA::min_Hopcroft()</code> 运行错误	26
4.4.1	运行结果	27
4.4.2	错误原因	27
4.4.3	解决方法	27
4.5	测试结果汇总	28
4.5.1	不改变已经是最小的 DFA 接受的语言	28
4.5.2	最小化功能测试	30
4.6	<code>DFA::min_Hopcroft</code> 算法运行结果错误	32
4.6.1	运行结果	32
4.6.2	错误原因	34
4.6.3	解决方法	36
第五章 总结		41
致谢		43
参考文献		45
附录 A 自动机状态转移图		47
附录 B 算法迭代过程		53
附录 C 代码		55
C.1	实例化 DFA 对象示例代码	55
C.2	Hopcroft 算法代码	56
C.3	SReachable 实现代码	58
C.4	usefuls 实现代码	59
C.5	complete 实现代码	61
附录 D 类图		65

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，从自动机理论的诞生开始，自动机的最小化就一直是一个重要的关注点，因为自动机的状态数约少，意味着它使用的资源也会越少，这一点对资源敏感的自动机应用很重要。已经有很多人为此做了大量的工作。1954 年，Huffman 提出了经典的用于确定性有限自动机的最小化算法^[1]，随后更高效的最小化算法由 Hopcroft 和 Moore 提出。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^[2] 是 Stellenbosch 大学的 Bruce William Watson 教授使用 C++ 语言实现一个用于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（下称 FIRE engine）实现了论文^[3,4]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^[4]对一些著名的确定性有限自动机最小化算法，如 Hopcroft、Brzozowski、Ullman 等提出的最小化算法和这些算法之间的演化关系做出了详尽的阐述。

据称 Hopcroft 的算法有着最优的时间复杂度，此算法时间复杂度为 $\mathcal{O}(n \log$

$n)^{[5]}$ 。严格来说, Hopcroft 的算法的时间复杂度不仅仅与自动机的大小 $n^{\textcircled{1}}$ 有关, 还与自动机的输入的字母表的大小 k 有关, 其时间复杂度应为 $\mathcal{O}(kn \log n)$ 。Timo Knuutila 指出, 在自动机的输入的字母表的大小不固定时, Hopcroft 的原始算法的时间复杂度将不再是 $\mathcal{O}(kn \log n)^{[6]}$ 。

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, 并在此基础上衍生出其他最小化算法。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, 也不依赖于确定状态的等价, 而其他著名的算法的基础都是状态等价, 是在已有的算法上的改进^[4]。

国内对最小化算法的研究则针对一些更加详细的分类。比如在胡芙提出的超最小化算法, 用来把确信性树自动机转换为确定性有限自动机, 然后用等价状态合并和等价类分割, 最后得到差异有限的确定性树自动机^[7]。还有张锋的基于 Myhill-Nerode 原理推导出来的针对直觉模糊有限自动机和完备格上的直觉模糊有限自动机的最小化算法^[8]。对于自动机技术的实际应用, 孙丹丹在海量的 Web 信息中抽取用户需要的信息上应用了模糊树自动机技术, 并且提出模糊树自动机的构造算法和最小化算法^[9]。

1.3 本文的工作

第二章内容为确定性有限自动机最小化需要的预备知识, 用来帮助理解。第三章则介绍 FIRE engine 中对类 DFA 的实现和实例化类 DFA 所必须的其他的类, 给出实例化类 DFA 对象所需要的步骤。第四章则是在测试过程中遇到的问题和解决这些问题的步骤。

本文的主要使用 Visual Studio 2017 集成开发环境, 对 FIRE engine 实现的最小化算法进行功能测试, 验证 FIRE engine 的中的最小化算法的有效性。

^① n 为自动机的状态数, 也称自动机的大小。

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。

德国数学家在 1874 年提出集合论 (set theory)，经过长时间的发展和完善，集合论成为了数学的一个重要组成部分，并且已经渗透到许多领域。计算机科学出现并发展壮大的过程中，集合论也成为计算机理论的一个重要基础概念。

集合论的最基础的概念是集合，集合的一个非形式描述为：一定范围内确定的，并且彼此可以区分的对象汇集在一起形成的整体叫做集合 (set)^[10]。

2.1 基本定义

定义 2.1 (幂集^[4]): 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 。

例 2.1 (集合的幂集): 设集合 $A = \{a, b\}$ ，那么 $\mathcal{P}(A) = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$ ， $\mathcal{P}(A)$ 等价于 2^A 。

定义 2.2 (函数集^[4]): 对于集合 A 和 B ， $A \rightarrow B$ 代表所有从 A 到 B 的函数的集合。

定义 2.3 (笛卡尔积^[10]): 对于集合 A 和 B ， A 和 B 的笛卡尔积是一个集合。该集合满足

$$A \times B = \{(a, b) | a \in A \wedge b \in B\}$$

定义 2.4 (二元关系^[4,10]): 对于集合 A 和 B ， $C \subseteq A \times B$ 是 $A \rightarrow \mathcal{P}(B)$ 的二元关系。

定义 2.5 (关系的合成^[4,10]): 对于集合 A, B, C ，和关系 $\mathcal{R}_0 \subseteq A \times B, \mathcal{R}_1 \subseteq B \times C$ ， \mathcal{R}_0 和 \mathcal{R}_1 的合成定义为

$$\mathcal{R}_0 \circ \mathcal{R}_1 = A \rightarrow C = \{(a, c) | \exists (a, b) \in \mathcal{R}_0 \wedge (b, c) \in \mathcal{R}_2\}$$

定义 2.6 (等价关系的等价类^[4]): 对任何集合 A 上的等价关系 E , 我们使用 $[A]_E$ 代表等价类^① 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 2.7 (等价类的指数^[4]): 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 2.8 (字母表^[4]): 字母表是有限大小的非空集合。

定义 2.9 (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 (refinement)。

2.2 有限自动机

本小节中定义有限自动机、其性质和用于有限自动机的变换。本节内容大部分直接引用自^[4]。

注释 2.1 (状态转移图^[3]): 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

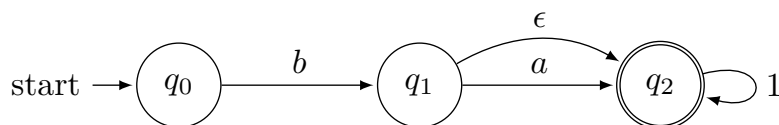


图 2.1 自动机状态转移图

图 2.1 中, 自动机由 q_0 进入, 接收字符 “b” 转移到状态 q_1 , 状态 q_1 接收字符 “a” 转移到状态 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 $\mathcal{L} = ba(1)^* \cup b(1)^*$ 。

^①等价关系和等价类稍后说明

定义 2.10 (有限自动机^[4]): 有限自动机 (Finite Automata, FA) 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2.8”和“定义 2.1”。

例 2.2: 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, $T = \{(q_0 \times b \times q_1), (q_1 \times a \times q_2), (q_2 \times 1 \times q_2)\}$, $E = (q_1 \times q_2)$ 。

注释 2.2: 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^[10], 如图2.1中的自动机, 写成表 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 (final)	q_2	-	-	q_2	-

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, “-” 表示没有相应的转移关系。

注释 2.3 (转移关系的表示): 为了更加简洁的描述转移关系, 用“ $T = (q_0 \times b \times q_1)$ ”来代表图 2.1 中的“状态 q_0 接收字符 ‘b’ 转移到状态 q_1 ”。

注释 2.4: 本文在不同情况下使用不同的形式来表示转移关系, 比如, 对于转移关系 $T = (q_0 \times b \times q_1)$, 也写成 $q_1 = T(q_0 \times b)$ 。对于图 2.1 中状态 q_0 和 q_2 之间的转移关系, 可以描述为 $T(q_0 \times w \times q_2)$ 或者 $q_2 = T(q_0 \times w)$, 其中, $w = ba$ 或 $w = b\epsilon$ 。

2.2.1 有限自动机的性质

本节定义有限自动机（Finite automata，下称 FA）的性质，为了使定义更加简洁，引进三个 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 2.11 (FA 的大小)： 定义一个 FA 的大小为 $|M| = |Q|$ 。

定义 2.12 (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 $g \in Q_0 \rightarrow Q_1$ ，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 2.13 (转移关系 T 的扩展)： 我们把 $T \in V \rightarrow \mathcal{P}(Q \times Q)$ 到 $T^* \in V^* \rightarrow \mathcal{P}(Q \times Q)$ 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。

定义 2.14 (左语言和右语言^[4])： 状态 (M) 中的左语言由函数 $\overleftarrow{\mathcal{L}}_M \in Q \rightarrow \mathcal{P}(V^*)$ 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 $\vec{\mathcal{L}}_M \in Q \longrightarrow \mathcal{P}(V^*)$ 给出, 其中

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

通常在没有歧义的时候移除下标 M 。

定义 2.15 (FA 的语言^[4]): 有限自动机的语言由函数 $\mathcal{L}_{FA} \in FA \longrightarrow \mathcal{P}(V^*)$ 给出, 该函数的定义为式 2-1:

$$\mathcal{L}_{FA}(M) = (\cup s, f : s \in S \wedge f \in F : T^*(s, f)) \quad (2-1)$$

例 2.3 (FA 的语言): 图 2.1 中的 FA 接受的语言为 $\mathcal{L} = \{ba(1)^* \cup b(1)^*\}$ 。下文中使用符号 \mathcal{L} 代替“FA 接受的语言”。

定义 2.16 (完全 FA (Complete)): 一个完全 FA 满足式 2-2 :

$$Complete(M) \equiv (\forall q, a : q \in Q \wedge a \in V : T(q, a) \neq \emptyset) \quad (2-2)$$

定义 2.17 (ϵ -free): 当且仅当 $E = \emptyset$ 时, M 是 ϵ -free 的。

定义 2.18 (可达状态^[3]): 可达状态分为“开始可达状态”和“可达结束状态” (M 中), 其中

- 开始可达状态: 从开始状态可以到达的状态, 写作 $SReachable(M)$;
- 可达结束状态: 能到达结束状态的状态, 写作 $FReachable(M)$;

可达状态定义为:

$$Reachable(M) = SReachable(M) \cap FReachable(M)$$

定义 2.19 (Start-useful 自动机^[4]): 一个 $Useful_s$ 自动机定义如下:

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

定义 2.20 (Final-useful 自动机^[4]): 一个 $Useful_f$ 自动机定义如下:

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

定义 2.21 (Useful 自动机^[4]): $Useful$ 自动机是一个只有可达状态的 FA:

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

例 2.4: 如图 2.2 所示, 其中

- 图 2.2(b) 为一个 $Useful$ FA;
- 图 2.2(a) 中状态 q_1 不是一个开始可达状态, 也称开始不可达状态 (start-unreachable);
- 图 2.2(a) 中状态 q_2 不是一个可达结束状态, 也称陷阱状态 (final-unreachable)^[10]。

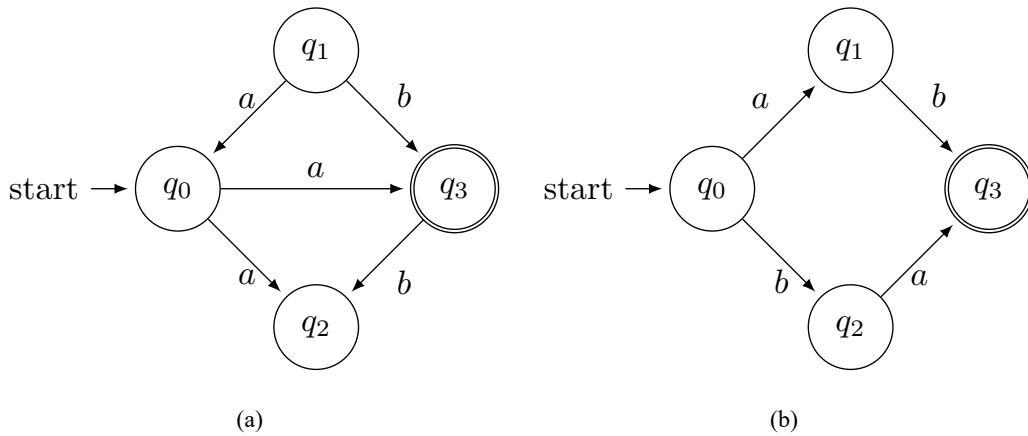


图 2.2

定义 2.22 (确定性有限自动机^[4]): 当且仅当

- 无多重开始状态;
- 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，称 M 为确定性有限自动机（Deterministic Finite Automata，下称 DFA）。公式形式表达为式 2-3：

$$Det(M) \equiv (|S| \leq 1 \wedge \epsilon\text{-free}(E) \wedge (\forall q, a : q \in Q \wedge a \in V : |T(q, a)| \leq 1)) \quad (2-3)$$

且有 $DFA \subseteq FA$ 。

定义 2.23 (DFA 的最小化)： 满足以下条件时， $M \in DFA$ 是最小的：

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

函数 Min 仅定义在 DFA 上。一个最小的但是仍然完全的 DFA 的定义如下：

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

Min_c 仅定义在完全 DFA 上。

2.2.2 有限自动机的变换

变换 2.1 (FA 的反转 (reversal)^[4])： FA 反转由上标函数 $R \in FA \rightarrow FA$ 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 2.2 (移除开始状态不可达状态^[4])： 变换 $useful_s \in FA \rightarrow FA$ 移除开始状态不可达状态：

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

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 2.3 (子集构造^[4]): 函数 *subset* 把一个 ϵ -free FA 转换为一个 DFA (在子句 **let** $T' \in \mathcal{P}(Q) \times V \longrightarrow \mathcal{P}(\mathcal{P}(Q))$ 中):

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

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

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

有时候也把它说成“幂集”构造。

2.3 等价性和最小化

令 $M = \{Q, V, T, \emptyset, S, F\}$ 为一个完全 DFA, 并且 M 中所有状态都是开始可达的。

定义 2.24 (状态的等价^[4]): 当 M 中的状态 p, q 满足式 2-4 时, 称状态 p, q 等价。

$$(p, q) \in E \equiv (\vec{\mathcal{L}}(p) = \vec{\mathcal{L}}(q)) \quad (2-4)$$

其中, E 为等价关系 (Equivalence)。

定义 2.25 (状态的等价^[10]): 给出状态等价的另一种描述。对给定的 DFA $M = (Q, V, T, \emptyset, S, F)$, 如果 $\exists w \in V^*$, 使得 Q 中的两个状态 p 和 q , $T(q, w) \in F$ 和 $T(p, w)$ 有且仅有一个成立, 那么称 p 和 q 是可区分的, 否则称 p 和 q 等价, 写作 $p \equiv q$ 。

注释 2.5: 关系 D (Distinguishable) 是可区分的状态, E 和 D 满足 $D = \neg E$ 。

注释 2.6: $\forall p, q : p \in (Q \setminus F) \wedge q \in F$, 有 $p \neq q$ 。

2.3.1 最小化

最小化是对于一个给定的 DFA M ，找到一个与 M 接受同样的 \mathcal{L} 的且状态数最少的 DFA M_1 ($|M_1| \leq |M|$) 的过程，最小化算法用来计算 DFA 中的关系 D 和关系 E ，只需计算其中一个即可得到 M_1 ^[4]（本文仅针对 DFA 应用最小化算法）。

给出一种用于 DFA 的最小化算法^[8,10,11]，描述如算法 2.1 所示：

算法 2.1 经典算法

输入： DFA M

输出： 与 M 同构的最小的 DFA

```

1:  $\forall (p, q) \in (Q \setminus F) \times F$ ，标记状态对  $(p, q)$ 
2: do
3:   for all 未被标记的状态对  $(p, q)$  do
4:     for all  $\forall x \in V$  do
5:       if 状态对  $(T(p, x), T(q, x))$  被标记了  $\rightarrow$ 
6:         标记状态对  $(p, q)$ 
7:       end if
8:     end for
9:   end for
10: od 没有新的状态对被标记

```

定理 2.1： 算法 2.1 的时间复杂度为 $\mathcal{O}(n^2)$ ^[11]。

定理 2.2： 算法 2.1 移除不可达状态后的 DFA 为最小的 DFA^[10]。

例 2.5： 使用算法 2.1 对图 2.3 中的 DFA 进行最小化。

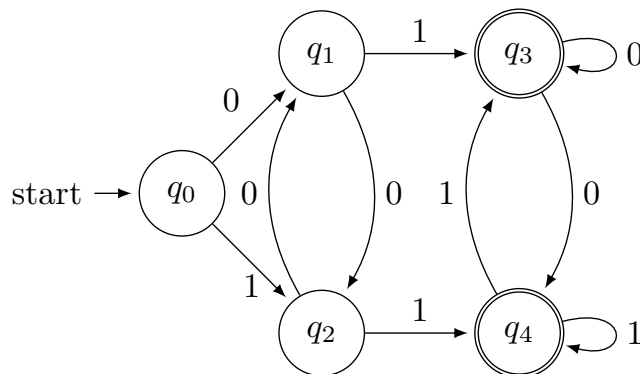


图 2.3 一个接受 $\mathcal{L} = \{0(0)^*1(0,1)^* \cup 1(0)^*1(0,1)^*\}$ 的 DFA

(1) 首先标记状态对 (q_0, q_3) , (q_0, q_4) , (q_1, q_3) , (q_1, q_4) , (q_2, q_3) , (q_2, q_4) 。剩下需要计算的状态对为 (q_0, q_1) , (q_0, q_2) , (q_1, q_2) , (q_3, q_4) ；

- (2) 对于状态对 (q_0, q_1) , $T(q_0, 0) = q_1$, $T(q_1, 0) = q_2$, 此时状态对 (q_1, q_2) 未被标记, 无操作; 再看 $T(q_0, 1) = q_2$, $T(q_1, 1) = q_3$, 此时 (q_0, q_3) 已经被标记, 那么标记状态对 (q_0, q_1) ;
- (3) 对于状态对 (q_0, q_2) , $T(q_0, 0) = q_1$, $T(q_2, 0) = q_1$, 不做处理, 再看 $T(q_0, 1) = q_2$, $T(q_2, 1) = q_4$, 此时状态对 (q_0, q_4) 已经被标记, 那么标记 (q_0, q_2) ;
- (4) 对于状态对 (q_1, q_2) , $T(q_1, 0) = q_2$, $T(q_2, 0) = q_1$, 不做处理, 再看 $T(q_1, 1) = q_3$, $T(q_2, 1) = q_4$, 此时状态对 (q_3, q_4) 未被标记, 无操作;
- (5) 对于状态对 (q_3, q_4) , $T(q_3, 0) = q_3$, $T(q_4, 0) = q_3$, 不做处理, 再看 $T(q_3, 1) = q_4$, $T(q_4, 1) = q_4$, 不做处理;
- (6) 第一次迭代结束, 此时还有状态对 (q_1, q_2) 和 (q_3, q_4) 未被标记。
- (7) 对于状态对 (q_1, q_2) , 根据第 (4) 步, 算法无操作;
- (8) 对于状态对 (q_3, q_4) , 根据第 (5) 步, 算法无操作;
- (9) 本次迭代未标记任何状态对, 算法结束, 此时还有状态对 (q_1, q_2) 和 (q_3, q_4) 未被标记。

算法结束后还有状态对 (q_1, q_2) 和 (q_3, q_4) 未被标记, 也就是说 $q_1 \equiv q_2$, $q_3 \equiv q_4$ 。此时的 DFA 为图 2.4。

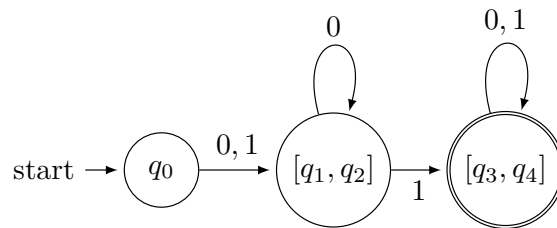


图 2.4 与图 2.3 同构的最小的 DFA

第三章 类 DFA 及相关类的设计

本章对类 DFA 做简要说明，说明如何实例化一个类 DFA 对象，作为第四章的测试的准备工作。类 DFA 为 FIRE engine 中应用最小化算法的类，类 FA、类 RFA、类 RBFA、类 LBFA 与类 DFA 都继承自类 FAabs，它们之间的关系如图 3.1 所示。

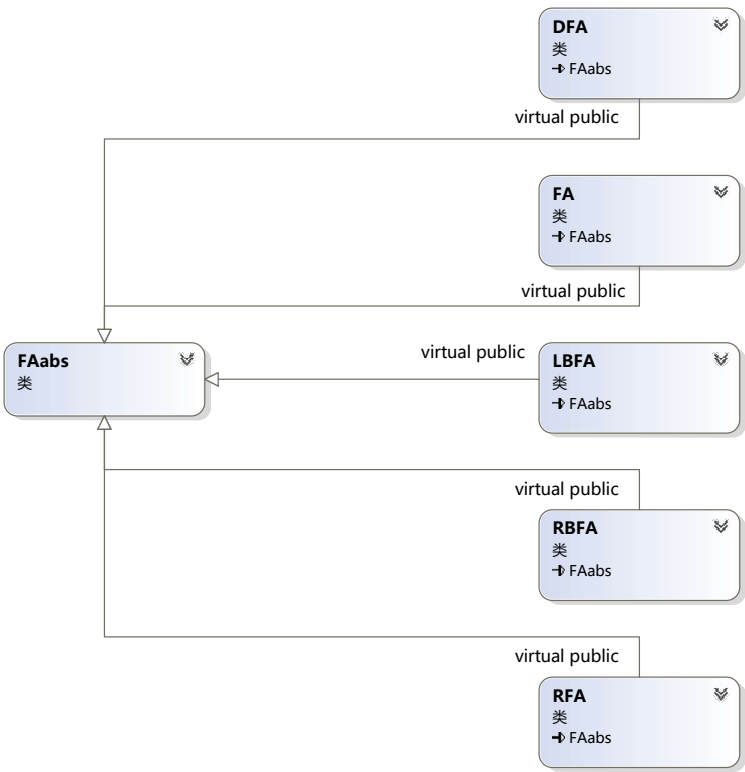


图 3.1 类 FAabs 及其派生类

3.1 DFA 类

在本文中，我们仅仅对 DFA 应用最小化算法，FIRE engine 提供了多种方式构造一个 DFA，要得到一个类 DFA 的对象，在已经实例化类 FA、类 RFA、类 RBFA 或类 LBFA 的情况下，可以通过执行他们的成员成员函数 “determinism()” 得到一个类 DFA 对象。

类 DFA 的部分实现如表 3.1

表 3.1 类 DFA

Class DFA			
名称	类型	属性	说明
Q	StatePool	protected	
S	StateSet	protected	$S \subseteq Q$
F	StateSet	protected	$F \subseteq Q$
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 3.1 可知，类 *DFA* 默认情况下，提供一个用 “DFA_components” 变量的引用来实例化 DFA 对象的方法。

3.2 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3.2所示

表 3.2 结构体 DFA_components

struct DFA_components		
名称	类型	说明
Q	StatePool	
S	StateSet	$S \subseteq Q$
F	StateSet	$F \subseteq Q$
T	DTransRel	转移关系

由表3.2可知，结构体 *DFA_components* 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 3.2 所示。

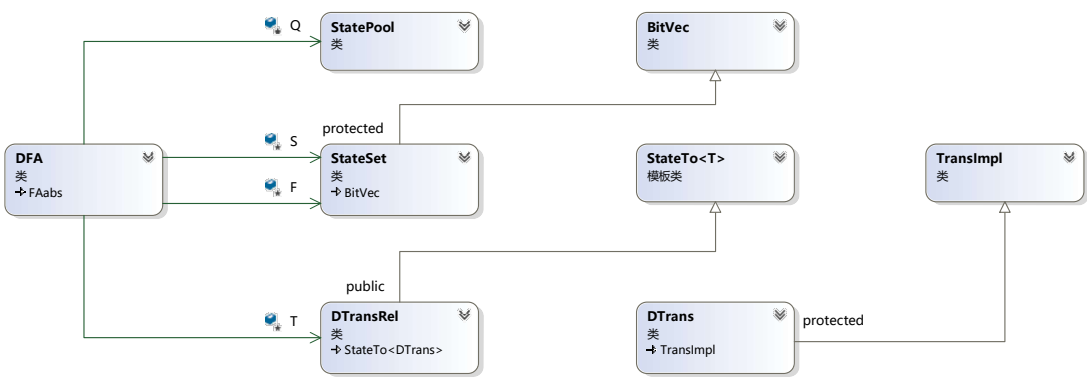


图 3.2 DFA 与其成员类及成员类的基类

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。

3.2.1 StatePool 类

StatePool 类的部分实现如表 3.3（文件 *StatePool.h*）：

表 3.3 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 3.3 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。

3.2.2 StateSet 类

类 StateSet 的部分声明如表 3.4（文件 *StateSet.h*）：

表 3.4 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	

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。

3.2.3 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 3.5 所示（文件 *DTransRel.h*）：

表 3.5 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()。

3.3 实例化类 DFA 对象

在文件 *DFA.cpp* 中有如下实现：

```

1 inline int DFA::class_invariant() const
2 {
3     return(Q.size() == S.domain()
4           && Q.size() == F.domain()
5           && Q.size() == T.domain()
6           && current < Q.size()
7           && S.size() <= 1);
8 }
```

代码 3.1 DFA.cpp

而文件 *DFA.h* 中的构造函数如下：

```

1 inline DFA::DFA(const DFA_components& r) :Q(r.Q), S(r.S), F(r.F), T(r.T)
2 {
3     current = Invalid;
4     assert(class_invariant());
5 }
```

代码 3.2 DFA.h

由代码 3.1 和代码 3.2 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^[12]（下称 assert 中止）。由此可以知道，类 DFA 要求其成员变量满足

$$(Q.size() \equiv S.domain() \equiv T.domain) \wedge (S.size() \leq 1) \quad (3-1)$$

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3-1中。

在文件 *State.h* 中有如下定义

```

1 // Encode automata states as integers.
2 typedef signed int State;
3
4 // Invalid states mean something bad is about to happen.
5 const State Invalid = -1;
```

代码 3.3 State.h

由代码3.3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3.2节内容所说，可以实例化如图3.3的自动机。

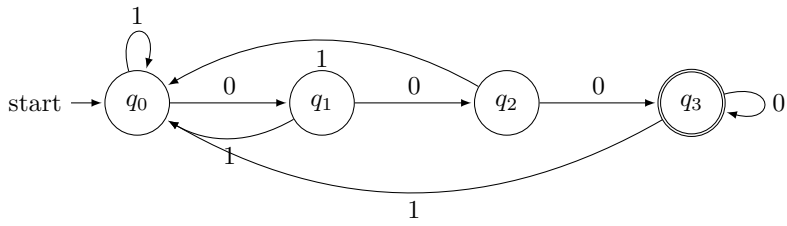


图 3.3 DFA 示例

图3.3中的自动机的转移函数如表3.6

表 3.6 图3.3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 (final)	q_3	q_3	q_0

实例化 *DFA* 类如代码C.1:

代码 C.1 将在控制台输出如下信息:

```
1 DFA
2 Q = [0,4)
3 S = { 0 }
4 F = { 3 }
5 Transitions =
6 0->{ '0'->1 '1'->0 }
7 1->{ '0'->2 '1'->0 }
8 2->{ '0'->3 '1'->0 }
9 3->{ '0'->3 '1'->0 }
10
11 current = -1
```

代码 3.4 图3.3中自动机在 FIRE engine 中的表现形式

下文中将以表 3.6 的形式来描述一个自动机，以代码 3.4 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。

第四章 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。表 4.1 中为本章概要，经过 4.2 节、4.3 节和 4.4 节的修改后，FIRE engine 中的最小化算法才能顺利执行。

表 4.1 本章内容概要

函数	运行结果	是否修复	内容	位置
DFA::min_Watson()	函数进入无限循环	是	分析并修复	4.2
DFA::usefulf()	函数运行中止	是	分析并修复	4.3
DFA::min_Hopcroft()	函数运行中止	是	分析并修复	4.4
最小化算法运行结果汇总	—————	—	———	4.5
DFA::min_Hopcroft()	函数功能异常	是	分析并修复	4.6

4.1 如何测试

以图 3.3 中的 DFA 为例，按照附录代码 C.1 所示，实例化一个类 DFA 的对象之后，通过执行最小化函数，观察他们的输出结果，列出转移函数，绘制自动机状态转移图。对比输入输出结果以得出结论。FIRE engine 中需要测试的五个最小化算法如表 4.2 所示。（表 4.2 中，Final-useful FA 可以通过执行类 DFA 的成员函数 DFA::usefulf() 得到）

表 4.2 Fire engine 中的五个最小化算法

算法	是否需要 Final-useful FA
DFA::min_Brzozowski()	否
DFA::min_Hopcroft()	是
DFA::min_HopcroftUllman()	是
DFA::min_dragon()	是
DFA::min_Watson()	是

例 4.1： 最小化函数执行示例如下

```

1 // 实例化 DFA 的对象(附录代码 C.1) , 得到 "dfa1"
2 dfa1.useful();
3 dfa1.min_Hopcroft();
4 std::cout << dfa1 << std::endl;

```

4.2 无限循环

实例化一个如表 4.3 的 DFA 的对象之后, 调用函数 `DFA::min_Watson()`。(表 4.3 对应的状态转移图为图 A.1(a), 含有陷阱状态 q_5)

表 4.3 接受 $\mathcal{L} = 0^*10^*$ 的自动机^[10]

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 (final)	q_2	q_4	q_5
结束状态 (final)	q_3	q_4	q_5
结束状态 (final)	q_4	q_4	q_5
陷阱状态 (sink)	q_5	q_5	q_5

4.2.1 运行结果

函数进入无限循环。

4.2.2 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 4.1 中的 “`H.equivalize(p, q);`”。

```

1 if (are_eq(p, q, S, H, Z))
2 {
3     // p and q are equivalent.
4     H.equivalize(p, q);
5 }

```

代码 4.1 min-bww.cpp

单步进入该函数, 可以看到代码 4.2 (`StateEqRel.cpp` (42 行))

```

1 for (oldq->iter_start(i); !oldq->iter_end(i); oldq->iter_end(i))
2 {
3     map(i) = newp;

```

4 }

代码 4.2 StateEqRel.cpp

for 循环的一般格式如代码 4.3

```

1 for (初始化循环变量; 循环条件; 迭代)
2 {
3     循环体
4 }
```

代码 4.3 for 循环的一般格式

在代码 4.2 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 4.4

```

1 // StateSet.h
2 // Is r the last State in an iteration sequence.
3 inline int StateSet::iter_end(State r) const
4 {
5     return(BitVec::iter_end(r));
6 }
7
8 // BitVec.h
9 // Is r the last set bit in an iteration sequence.
10 // if (r== -1) retrun 1; else return 0
11 inline int BitVec::iter_end(int r) const
12 {
13     return(r == -1);
14 }
```

代码 4.4 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。

4.2.3 解决方法

将代码 4.2 中的迭代“oldq->iter_end(i)”更改为“oldq->iter_next(i)”，更改后如代码 4.5，经过比对，更改后与原文^[2]相同。

```

1 for (oldq->iter_start(i); !oldq->iter_end(i); oldq->iter_next(i))
2 {
3     map(i) = newp;
4 }
```

代码 4.5 StateEqRel.cpp

更改后：函数“DFA::min_Watson();”不再陷入无限循环。

4.3 函数 DFA::useful() 运行错误

“DFA::useful()” 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 4.6

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

代码 4.6 DFA::useful()

以图 A.1(a) 为例，状态 q_5 即为非 “final-reachable” 状态（下称陷阱状态）。移除状态 q_5 之后如图 A.1(b)。图 A.1(b) 转移函数如表 4.4。

表 4.4 接受 $\mathcal{L} = 0^*10^*$ 的自动机^[10]

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 (final)	q_2	q_4	-
结束状态 (final)	q_3	q_4	-
结束状态 (final)	q_4	q_4	-

4.3.1 运行结果

执行函数 “DFA::useful()” 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 4.1 错误

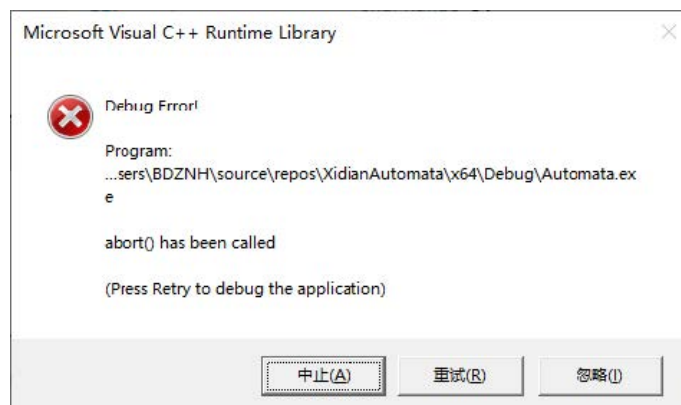


图 4.1 函数 DFA::useful() 错误提示

控制台提示如图 4.2

```
*****
Minimized by DFA::min_HopcroftUllman():
Useful required?: true
Assertion failed: 0 <= q, file c:\users\bdznh\source\repos\xidianautomata\automata\transimpl.cpp, line 79
```

图 4.2 函数 DFA::usefulf() 错误提示

可以确定函数未完成其定义功能。

4.3.2 错误原因

查看 TransImpl.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 4.7 TransImpl.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”^①, 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 4.8。

```
84 StateTo<State> newnames;
85 newnames.set_domain(Q.size());
86
87 // All components will be constructed into a special structure :
88 DFA_components ret;
89 State st;
90 for (st = 0; st < Q.size(); st++)
91 {
92     // If this is a Usefulf State, carry it over by giving it a name
93     // in the new DFA.
94     if (freachable.contains(st))
95     {
96         newnames.map(st) = ret.Q.allocate();
97     }
98 }
```

代码 4.8 DFA.cpp

在代码 4.8 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。

^①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 4.9

```

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
134 , b)));

```

代码 4.9 DFA.cpp

根据代码 4.7，可以知道程序中中止的地方为代码 4.9，133 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 4.10

```

108 // Compute the image of r, and CharRange it under *this.
109 inline State DTransRel::transition_on_range(const State r, const CharRange a)
110     const
111 {
112     assert(class_invariant());
113     assert(0 <= r && r < domain());
114     return lookup(r).range_transition(a);

```

代码 4.10 DTransRel.cpp

由代码 4.10 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 4.11

```

177 // The actual mapping function
178 // First, a const lookup operator.
179 template<class T>
180 inline const T& StateTo<T>::lookup(const State r) const
181 {
182     assert(class_invariant());
183     // First check that it's in bounds
184     assert(0 <= r && r < domain());
185     return data[r];
186 }

```

代码 4.11 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4.3中，状态 q_2 经过字符“1”将转移到状态 q_5 ，而在代码4.8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4.9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4.7中，State变量q的值为“-842150451”，导致程序在此处中止。

4.3.3 解决方法

如代码3.3所示，文件State.h中将无效状态设置为“Invalid”。在代码4.8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4.12

```

91 for (st = 0; st < Q.size(); st++)
92 {
93     // If this is a Useful State, carry it over by giving it a name
94     // in the new DFA.
95     if (freachable.contains(st))
96     {
97         newnames.map(st) = ret.Q.allocate();
98     }
99     else // 新增
100     { // 新增
101         newnames.map(st) = Invalid; // 新增
102     } // 新增
103 }
```

代码 4.12 更改后的 DFA.cpp

在代码4.9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4.13

```

133 State stdest; // 新增
134 stdest = newnames.lookup(T.transition_on_range(st, b)); // 新增
135
136 if (stprime != Invalid && stdest != Invalid) // 新增
137 { // 新增
138     ret.T.add_transition(stprime, b, stdest); // 修改
139 } // 新增
```

代码 4.13 更改后的 DFA.cpp

更改后函数DFA::useful()成功移除图A.1(a)中的陷阱状态 q_5 ，将图A.1(a)转换成图A.1(b)。更改后函数DFA::useful()成功移除图A.5(a)中的陷阱状态 q_1 ，将图A.5(a)转换成图A.5(b)。

4.4 函数 DFA::min_Hopcroft() 运行错误

实例化如表 4.5 所示的自动机。（含有开始不可达状态）

表 4.5 图A.9(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 (final)	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 (final)	q_8	q_5	q_4
开始不可达状态	q_9	q_7	q_5

函数 DFA::min_Hopcroft() 是 Hopcroft 算法在 FIRE engine 中的实现，Hopcroft 算法的描述如算法 4.1 所示^[4]

算法 4.1 Hopcroft 最小化算法

```

1:  $P := [Q]_{E_0}$ ;
2:  $L := (\text{if } (|F| \leq |Q \setminus F|) \text{ then } \{F\} \text{ else } \{Q \setminus F\} \text{ end if}) \times V$ ;
3:  $\{\text{恒有: } [Q]_E \subseteq P \subseteq [Q]_{E_0} \wedge L \subseteq (P \times V)\}$ 
4:  $\wedge (\forall Q_0, Q_1, a : Q_0 \in Q \wedge (Q_1, a) \in L : \neg \text{Splittable}(Q_0, Q_1, a)) \Rightarrow (P = [Q]_E)$ 
5: do  $L \neq \emptyset \longrightarrow$ 
6:   let  $Q_1, a : (Q_1, a) \in L$ ;
7:    $P_{old} := P$ ;
8:    $L := L \setminus \{(Q_1, a)\}$ ;
9:    $\{\text{恒有: } [Q]_E \subseteq P \subseteq P_{old}\}$ 
10:  for  $Q_0 : Q_0 \in P_{old} \wedge \text{Splittable}(Q_0, Q_1, a)$  do
11:     $Q'_0 := \{p : p \in Q_0 \wedge T(p, a) \in Q_1\}$ ;
12:     $P := P \setminus \{Q_0\} \cup \{Q_0 \setminus Q'_0, b\}$ ;
13:    for  $b : b \in V$  do
14:      if  $(Q_0, b) \in L \longrightarrow L := L \setminus \{(Q_0, b)\} \cup \{(Q'_0, b), (Q_0 \setminus Q'_0, b)\}$ 
15:      else if  $(Q_0, b) \notin L \longrightarrow$ 
16:         $L := L \cup (\text{if } (|Q'_0| \leq |Q_0 \setminus Q'_0|) \text{ then } \{(Q'_0, b)\} \text{ else } \{(Q_0 \setminus Q'_0, b)\} \text{ end if})$ ;
17:      end if
18:    end for
19:  end for
20:   $\{\text{恒有: } [Q]_E \subseteq P \subseteq P_{old} \wedge L \subseteq (P \times V)\}$ 
21: od  $\{P = [Q]_E\}$ 

```

4.4.1 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 4.14 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 4.14 CRSet.h

4.4.2 错误原因

以表 4.5 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 “0, 1”，所以传入参数为 “2” 时，超出了 `C` 的范围（数组从 0 开始计数），于是在语句 “`assert(!iter_end(it));`” 处导致程序中止。

表 4.5 所示的自动机在 `DFA::min_Hopcroft` 算法中迭代过程如附录表 B.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。

4.4.3 解决方法

函数 `DFA::min_Hopcroft()` 的主要功能实现部分如代码 4.15 所示。

```

100     L[q]--;
101     .....
102     for (repr.iter_start(p); !repr.iter_end(p); repr.iter_next(p))
103     {
104         .....
105         State r(split(p, q, C.iterator(L[q]), P));
106         .....
107     }

```

代码 4.15 min-hop.cpp

在遍历 *repr*（等价类代表，见附录）之前，将当前需要处理的字符保存下来，更改后为代码 4.16

```

100     L[q]--;
101     CharRange c = C.iterator(L[q]);           // 新增
102     .....
103     for (repr.iter_start(p); !repr.iter_end(p); repr.iter_next(p))
104     {
105         .....
106         State r(split(p, q, c, P));           // 修改
107         .....
108     }

```

代码 4.16 min-hop.cpp

更改后函数 `DFA::min_Hopcroft()` 运行不再中止。

4.5 测试结果汇总

经过 4.2 节、4.3 节和 4.4 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。

4.5.1 不改变已经是最小的 DFA 接受的语言

接受同一个 \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^[10]。因此最小化算法有性质 4.1。

性质 4.1: 由定义 2.23 可知，对于 DFA M ， $\mathcal{L}(\text{Min}(M)) = \mathcal{L}(M)$ ，同样的，最小化算法不能使最小的 DFA 接受的 \mathcal{L} 发生改变。

注释 4.1: DFA 中的开始不可达（start-unreachable）状态和陷阱状态（final-unreachable）状态不影响 DFA 接受的 \mathcal{L} 。

表 4.6 一组最小的 DFA

序号	数据	属性	备注
1	图 A.5(a)	最小的	含有陷阱状态
2	图 A.5(b)	最小的	不含陷阱状态
5	图 A.6(a)	最小的	含有开始不可达状态
6	图 A.6(b)	最小的	不含开始不可达状态
7	图 A.7(a)	最小的	含有开始不可达状态
8	图 A.7(b)	最小的	不含开始不可达状态
3	图 A.9(a)	最小的	含有开始不可达状态
4	图 A.9(b)	最小的	不含开始不可达状态

对于性质 4.1，以表 4.6 中的一组最小的 DFA 为数据，测试结果如表 4.7 所示。

表 4.7 最小化算法对性质 4.1 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()	✓	✓	✓	✓	✓	✓	✓	✓

结果总结对比如下：

- 对于 DFA 中的陷阱状态，可以调用函数 “DFA::useful()” 函数去除。表 4.7 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::useful()”；
- 除了算法 “DFA::min_Brzozowski()” 外，其他最小化算法均不能去除 DFA 中的开始不可达状态（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，含有开始不可达状态，但是作为对照组且含有开始不可达状态的第 5 个和第 7 个数据没有发生 assert 中止，所以本文认为此算法的中止的问题不是 DFA 中开始不可达状态导致；

- 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。

4.5.2 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，以表 4.8 中的数据对五个最小化算法进行测试，结果如表 4.9 所示。

表 4.8 一组 DFA

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

表 4.9 最小化算法的定义功能的测试结果

算法	1	2	3	4	5	6	7
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()” 无论是否经过第 4.4 节的修改，均输出代码 4.17。

```
1 DFA
2 Q = [0,2)
3 S = { 0 }
4 F = { 1 }
5 Transitions =
6 0->{ ['a','b']->1 }
7 1->{ 'a'->1 }
8
9 current = -1
```


代码 4.17 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 4.10(a) 所示，代码 4.17 的转移函数如表 4.10(b) 所示

表 4.10 第 3 个数据在算法 “DFA::min_Hopcroft()” 中的输入输出对比

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

图 A.2(b) 原本是一个最小的 DFA，由表 4.10 可知，函数算法 “DFA::min_Hopcroft()” 已经改变了图 A.2(b) 的状态数，由 “3” 变到 “2”，如图 4.3 所示，已经不满足性质 4.1。由此可以认为 FIRE engine 中算法 “DFA::min_Hopcroft()” 未完成其定义功能。

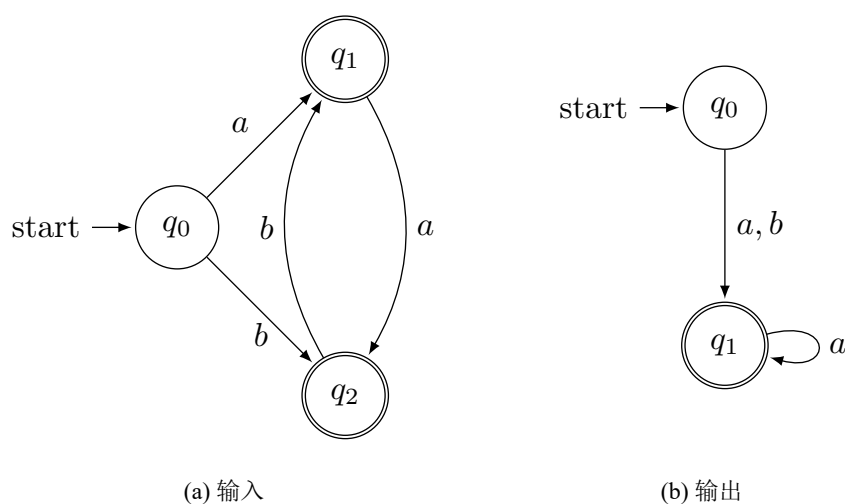


图 4.3 DFA::min_Hopcroft 对第 3 个数据输入输出对比

本小节总结如下

- FIRE engine 对于算法 “DFA::min_Brzozowski” 的实现仍然有不足之处；
- FIRE engine 对于算法 “DFA::min_HopcroftUllman()”、“DFA::min_dragon()”、“DFA::min_Watson()” 的实现可信度较高；

- 除了 DFA::min_Brzozowski() 算法，其他算法都不能移除 DFA 中的不可达状态；
- 移除 DFA::usefulf() 可以用来移除 DFA 中的陷阱状态。

4.6 DFA::min_Hopcroft 算法运行结果错误

4.6.1 运行结果

以图 A.8 中的五个最小的 DFA 为数据，测试结果统计如表 4.11 所示

表 4.11

算法	A.8(a)	A.8(b)	A.8(c)	A.8(d)	A.8(e)
DFA::min_Brzozowski()	✓	✓	✓	✓	✓
DFA::min_Hopcroft() (修改前)	中止	×	✓	中止	✓
DFA::min_Hopcroft() (修改后)	×	×	✓	×	✓
DFA::min_HopcroftUllman()	✓	✓	✓	✓	✓
DFA::min_dragon()	✓	✓	✓	✓	✓
DFA::min_Watson()	✓	✓	✓	✓	✓

对于图 A.8(a)，修改后 Hopcroft 算法输出如下代码 4.18，如图 4.4(a) 所示。

```
1 DFA
2 Q = [0,3)
3 S = { 0 }
4 F = { 2 }
5 Transitions =
6 0->{ 'a'->1 'b'->2 }
7 1->{ 'a'->2 }
8 2->{ 'b'->2 }
9
10 current = -1
```

代码 4.18 图 A.8(a) 输出

对于图 A.8(b)，无论是否修改，Hopcroft 均输出代码 4.19，如图 4.4(b) 所示。

```
1 DFA
2 Q = [0,2)
3 S = { 0 }
4 F = { 1 }
5 Transitions =
6 0->{ ['a','b']->1 }
7 1->{ 'b'->1 }
8
9 current = -1
```

代码 4.19 图 A.8(b) 输出

对于图 A.8(d) , 修改后 Hopcroft 输出代码 4.20, 如图 4.4(c) 所示。

```
1  DFA
2  Q = [0,3)
3  S = { 0 }
4  F = { 2 }
5  Transitions =
6  0->{ 'a'->1 }
7  1->{ 'b'->2 }
8  2->{ 'a'->2 }
9
10 current = -1
```

代码 4.20 图 A.8(d) 输出

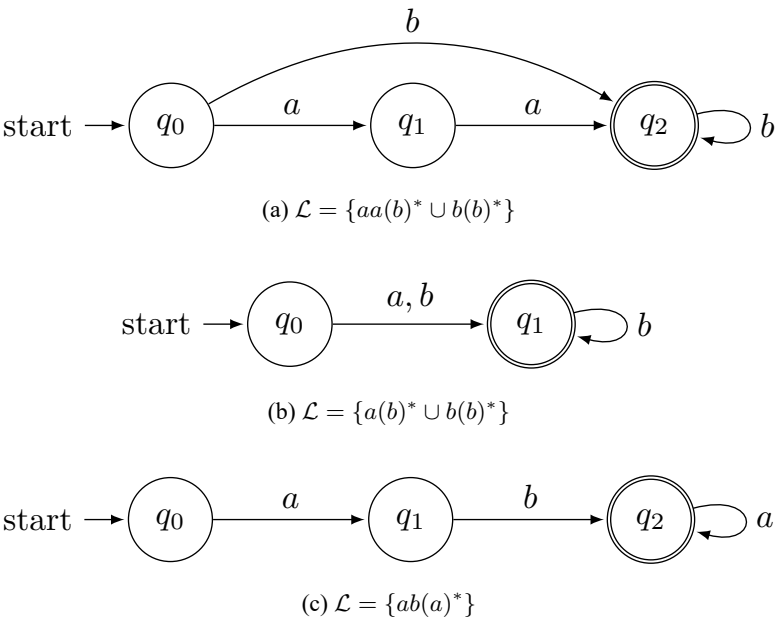


图 4.4 Hopcroft 算法的输出

表 4.12 为对比数据

表 4.12

数据	$ Q $	$ F $	$ F \leq Q \setminus F ?$	修改前结果	修改后结果
图 A.8(a)	5	3	否	中止	×
图 A.8(b)	3	2	否	×	×
图 A.8(c)	4	2	是	√	√
图 A.8(d)	5	3	否	中止	×
图 A.8(e)	5	2	是	√	√

4.6.2 错误原因

以表 4.8 中第三个数据, 也即图 A.2(b) 中的 DFA 为例, 此 DFA 在 DFA::min_Hopcroft 算法中的迭代过程如表 4.13 所示, 算法原型见算法 4.1:

表 4.13 图A.2(b)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		
1	前	0	0	1	b	-	1	0	0	{0,1}	{0},{1,2}
	后	0	0	1	b	-1	1	0	0	{0,1}	{0},{1,2}
2	前	1	0	1	b	-	1	0	0	{0,1}	{0},{1,2}
	后	1	0	1	b	-1	1	0	0	{0,1}	{0},{1,2}
3	前	0	0	1	a	-	0	0	0	{0,1}	{0},{1,2}
	后	0	0	1	a	-1	0	0	0	{0,1}	{0},{1,2}
4	前	1	0	1	a	-	0	0	0	{0,1}	{0},{1,2}
	后	1	0	1	a	-1	0	0	0	{0,1}	{0},{1,2}
5	-	-	-	-	-	-	-	-	-	-	——
	-	-	-	-	-	-	-	-	-	-	——

表 4.13 中各项意义见附录表 B.1。下面对 Hopcroft 算法进行分析:

- (1) Q 首先被分为两个等价类, $P = (Q \setminus F) \cup F = \{(0), (1, 2)\}$ 。因 $|Q \setminus F| \leq |F|$, 所以 $L = F \times V$ (算法 4.1, 语句 2);
- (2) 第一次迭代, 取出 $repr$ 第一个元素 0, 赋值给 p , 此时 $p = 0, q = 0, C = b$, 并且从 L 内移除与字符'b'相关的转移关系, $Splittable(p, q, c)$ 不成功^①, 不做操作;
- (3) 第二次迭代, 取出 $repr$ 第一个元素 1, 赋值给 p , 此时 $p = 1, q = 0, C = b$, $Splittable(p, q, c)$ 不成功, 不做操作;
- (4) 第三次迭代, 取出 $repr$ 第一个元素 0, 赋值给 p , 此时 $p = 0, q = 0, C = a$, 并且从 L 内移除与字符'b'相关的转移关系, $Splittable(p, q, c)$ 不成功, 不做操作;
- (5) 第四次迭代, 取出 $repr$ 第一个元素 1, 赋值给 p , 此时 $p = 1, q = 0, C = a$, $Splittable(p, q, c)$ 不成功, 不做操作;

^①即表 4.13 中 $r = -1$ 时, $Splittable(p, q, c)$ 不成功。

(6) 第五次迭代, 此时 L 内已经没有转移关系, 也就是此时 $L = \emptyset$ (算法 4.1, 语句 5), 迭代结束。此时 $P = \{\{0\}, \{1, 2\}\}$, 于是状态 1 与状态 2 被合并, 算法输出如图 4.3(b) 所示的结果。算法结束。

定义 4.1 ($Splittable^{[4]}$): $Splittable$ 的定义如下

$$Splittable(Q_0, Q_1, a) \equiv (\exists p, q : p \in Q_0 \wedge q \in Q_0 : (T(p, a) \in Q_1 \neq T(q, a) \in Q_1))$$

FIRE engine 对 $Splittable$ 的实现如算法 4.2 所示。

算法 4.2 DFA::split()

```

1: function SPLIT( $p, q, c, P$ )                                 $\triangleright p \in repr, c \in V, P := [A]_E$  (定义2.6)
                                                                 $\triangleright q$  为  $L$  中第一个不为 0 的元素的下标
2:    $part := \emptyset$ 
3:   for all  $s \in [p]$  do                                        $\triangleright [p]$  为状态  $p$  所在的等价类
4:     if  $T(s, c) \neq \emptyset \wedge T(s, c) \in [q] \rightarrow$ 
5:        $part := part \cup T(s, c)$ 
6:     end if
7:   end for
8:   if  $part \neq [p] \wedge part \neq \emptyset \rightarrow$ 
9:      $otherpiece := [p] \setminus part$ 
10:    if  $p \in part \rightarrow$ 
11:      return  $part$ 
12:    else
13:      return  $otherpiece$ 
14:    end if
15:  else
16:    return Invalid
17:  end if
18: end function

```

在本例中, 由于 $L = (Q \setminus F) \times V$, L 中第一个不为 0 的元素的下标是“0”(L 的意义见附录注释 B.1), q 的值即为 L 中第一个不为 0 的元素的下标, 此时 $q = 0$ 。依次取出 $repr = \{0, 1\}$ 中的元素, 对所有的 $c \in V$ 执行 DFA::split()。当 $p = 0 \wedge q = 0$ 时, $[p]$ 中不能通过任何字符转移到 $[q]$; 当 $p = 1 \wedge q = 0$ 时, $[p]$ 中不能通过任何字符转移到 $[q]$ 。此时算法运行结束, 但是等价类 $\{1, 2\}$ 此时还未被分割, 等价类 $\{1, 2\}$ 被合并, 成为新自动机 (图 4.3(b)) 中的 $\{1\}$, 于是 Hopcroft 算法输出了错误的结果。

通过对比总结可知, 当 $p : p \in F$, 对所有的 $c : c \in V$, 使得 $T(p, c) \notin (Q \setminus F)$, 且该 DFA 满足 $|F| > |Q \setminus F|$ 时, 也即式 4-1

$$(c, p : \forall c \in V \wedge p \in F \wedge T(p, c) \notin (Q \setminus F)) \wedge |F| > |Q \setminus F| \quad (4-1)$$

Hopcroft 算法有可能输出错误结果。

4.6.3 解决方法

由第 4.6.2 的内容，总结出以下两种解决方案：

- (1) 向 L 中增加二元关系 $F \times V$ ，也即更改算法 4.1 中语句 2 $L := (\text{if}(|F| \leq |Q \setminus F|) \text{then } \{F\} \text{else } \{Q \setminus F\} \text{end if}) \times V$ 为 $L := (Q \times V) \cup F \times V$ ；
- (2) 构造完全 DFA，使得对于满足关系 $|F| > |Q \setminus F|$ 的非完全 DFA 中，存在 $c \in V$ 使得 $p \in F \wedge T(p, c) \in (Q \setminus F)$ ；

具体做法

方案 1

对于方案 1，对 FIRE engine 的具体更改如下

```

1  if (F.size() <= (Q.size() - F.size())) //移除
2  {                                     //移除
3      repr.intersection(F);           //移除
4  }                                     //移除
5  else                                 //移除
6  {                                     //移除
7      repr.remove(F);                 //移除
8  }                                     //移除

```

代码 4.21 min-hop.cpp

Hopcroft 算法（算法 4.1）的正确性已经被证实过^[1,5,11]，该算法也是时间复杂度最优的版本，按照方案 1 更改将会导致算法的时间复杂度增加，并不符合算法本意。

方案 1 更改后的算法实际上是 Bruce William Watson 的论文中提及的未优化的算法，该算法如算法 4.3 所示。

注释 4.2： 对于算法 4.3 中语句 2，在算法的第一步，将 Q 分为两个等价类 $P = \{\{Q \setminus F\}, F\}$ ，对应方案 1 的更改方法。

算法 4.3 ^[4]

```

1:  $P := [Q]_{E_0}$ ;
2:  $L := P \times V$ ;
3: {恒有:  $[Q]_E \subseteq P \subseteq [Q]_{E_0} \wedge L \subseteq (P \times V)$ 
4:    $\wedge L \supseteq \{(Q_1, a) : (Q_1, a) \in (P \times V) \wedge (\exists Q_0 : Q_0 \in P : Splittable(Q_0, Q_1, a))\}$ 
5:    $\wedge (\forall Q_0, Q_1, a : Q_0 \in Q \wedge (Q_1, a) \in L : \neg Splittable(Q_0, Q_1, a)) \Rightarrow (P = [Q]_E)$ 
6: do  $L \neq \emptyset \longrightarrow$ 
7:   let  $Q_1, a : (Q_1, a) \in L$ ;
8:    $P_{old} := P$ ;
9:    $L := L \setminus \{(Q_1, a)\}$ ;
10:  {恒有:  $[Q]_E \subseteq P \subseteq P_{old}$ }
11:  for  $Q_0 : Q_0 \in P_{old} \wedge Splittable(Q_0, Q_1, a)$  do
12:     $Q'_0 := \{p : p \in Q_0 \wedge T(p, a) \in Q_1\}$ ;
13:     $P := P \setminus \{Q_0\} \cup \{Q_0 \setminus Q'_0, b\}$ ;
14:    for  $b : b \in V$  do
15:      if  $(Q_0, b) \in L \longrightarrow L := L \setminus \{(Q_0, b)\} \cup \{(Q'_0, b), (Q_0 \setminus Q'_0, b)\}$ 
16:      else if  $(Q_0, b) \notin L \longrightarrow L := L \cup \{(Q'_0, b), (Q_0 \setminus Q'_0, b)\}$ 
17:      end if
18:    end for
19:  end for
20:   $\{( \forall Q_0, Q_0 \in P : \neg Splittable(Q_0, Q_1, a) \}$ 
21: od  $\{P = [Q]_E\}$ 

```

并且 Bruce William Watson 也对 Hopcroft 算法（算法 4.1）中语句 2 进行了详细的说明，Bruce William Watson 已经指出，由于一开始就将 Q 分割成了 $P = [Q]_{E_0} = \{Q \setminus F, F\}$ ，所以只需要相对于 $(Q \setminus F, b)$ 或 (F, b) （对所有的 $b \in V$ ）进行等价类分割即可^[4,5]。所以方案 1 并不是一个比较好的方法。

按照方案 1 更改后 DFA::min-Hopcroft 通过样例测试，如表 4.14 所示。

表 4.14

算法	A.8(a)	A.8(b)	A.8(c)	A.8(d)	A.8(e)
DFA::min_Brzozowski()	✓	✓	✓	✓	✓
DFA::min_Hopcroft()（方案 1 修改前）	×	×	✓	×	✓
DFA::min_Hopcroft()（方案 1 修改后）	✓	✓	✓	✓	✓
DFA::min_HopcroftUllman()	✓	✓	✓	✓	✓
DFA::min_dragon()	✓	✓	✓	✓	✓
DFA::min_Watson()	✓	✓	✓	✓	✓

方案 2

FIRE engine 中没有实现构造完全 DFA 的算法。

本文提出一个用于在 FIRE engine 中构造一个完全 DFA 的算法，如算法 4.4 所示。

算法 4.4 构造完全 DFA 的算法**输入：** 一个 DFA $M = \{Q, V, T, E, S, F\}$ **输出：** 一个与 M 等价的完全 DFA $M' = \{Q', V, T', E, S, F\}$ ▷ M 与 M' 满足定义 2.12

```

1: function COMPLETE
2:    $flag := false;$ 
3:   for all  $p \in Q \wedge flag = false$  do
4:     for all  $c \in V \wedge flag = false$  do
5:       if  $T(p, c) = \emptyset \rightarrow flag := true$ 
6:       end if
7:     end for
8:   end for
9:   if  $flag = true \rightarrow$ 
10:      $usefulf(M);$                                 ▷ 移除  $M$  中的陷阱状态
11:      $usefuls(M);$                                 ▷ 移除  $M$  中的开始不可达状态
12:     for all  $p \in Q \wedge flag = false$  do
13:       for all  $c \in V \wedge flag = false$  do
14:         if  $T(p, c) \neq \emptyset \rightarrow T' := T' \cup T'(g(p) \times c \times g(T(p, c)))$ 
15:         else  $T' := T' \cup T'(g(p) \times c \times sink)$ 
16:         end if
17:       end for
18:     end for
19:   end if
20: end function

```

注释 4.3： 构造完全自动机本该仅当 $T(p, c) = \emptyset$ 时，增加转移关系 $T(p \times c \times sink)$ ，其中 p, c 满足 $p, c : p \in Q \wedge c \in V$ 。但是考虑到原本的 DFA 中可能有多个不可达状态，若仅仅构造完全 DFA，DFA 的大小仍然会变大，所以应该在构造完全 DFA 前，移除 DFA 中的不可达状态，然后再构造完全 DFA。这样构造出来的完全 DFA 仅比原本的 DFA 移除不可达状态后多一个状态（陷阱状态）。有利于缩短算法的运行时间。

公式表达形式为（若原 DFA 含不可达状态）

$$|Complete(useful(M))| - |useful(M)| = 1 \quad (4-2)$$

注释 4.4： 对于接受同一个 \mathcal{L} 的最小的但不完全 DFA M 和最小且完全的 DFA M_C ， M_C 比 M 多一个陷阱状态（Sink State）^[4]。

需要留意的是，FIRE engine 中仅实现了用于去除陷阱状态的函数 $DFA::usefulf()$ ，而没有实现用于去除开始状态的函数，虽然不可达状态并不会影响 DFA 接受的 \mathcal{L} 。但是不可达状态同样会影响算法的时间复杂度。

给出一个用于在 FIRE engine 内去除 DFA 的开始不可达状态的算法

算法 4.5 移除 DFA 中开始不可达状态的算法 *usefuls***输入：** 一个含有开始不可达状态的 DFA $M = \{Q, V, T, E, S, F\}$ **输出：** 一个与 M 等价的完全 DFA $M' = \{Q', V, T', E, S', F'\}$ ▷ M 与 M' 满足定义 2.12

```

1: function USEFULS
2:    $Q' := SReachable(M)$ 
3:   for all  $p \in Q'$  do
4:     for all  $c \in V$  do
5:       if  $T(p, c) \neq \emptyset \rightarrow$ 
6:          $T' := T' \cup T'(p \times c \times q)$ 
7:          $V' := V' \cup c$ 
8:       end if
9:     end for
10:  end for
11:   $S' := Q' \cap S$ 
12:   $F' := Q' \cap F$ 
13: end function

```

同样的，FIRE engine 中并没有实现 $SReachable()$ ，本文给出这样一个算法

算法 4.6 $SReachable$ **输入：** 一个 DFA $M = \{Q, V, T, E, S, F\}$ **输出：** $SReachable(M)$

```

1: function SREACHABLE
2:    $Q_1 := S$ 
3:    $Q_2 := \emptyset$ 
4:   do
5:      $Q_1 := Q_1 \cup Q_2$ 
6:     for all  $p \in Q_1$  do
7:       for all  $c \in V$  do
8:         if  $T(p, c) \neq \emptyset \rightarrow Q_2 := Q_2 \cup T(p, c)$ 
9:         end if
10:      end for
11:    end for
12:  od  $Q_1 = Q_2$ 
13:  return  $Q_2$ 
14: end function

```

代码实现

算法 4.4 的代码实现见附录 C.5。

算法 4.5 的代码实现见附录 C.4。

算法 4.6 的代码实现见附录 C.3。

第五章 总结

本文总结如下

- 对于 DFA 中的陷阱状态，可以使用函数 “DFA::usefulf()” 去除；
- 除了算法 DFA::min_Brzozowski()，其他算法都不能去除 DFA 中的开始不可达状态；
- 由于算法 DFA::min_Brzozowski() 在某些情况下，会改变最小的 DFA 接受的语言，所以本文认为 FIRE engine 中对 Brzozowski 算法的实现还有不足的地方；
- 对函数 DFA::usefulf() 的修改有较高可信度，详见第 4.3 节；
- 算法 DFA::min_Hopcroft() 严格来说不是算法的错误，而是数据不符合其要求，Bruce William Watson 在他的论文^[4] 第三节中基于完全 DFA 讨论了状态等价和最小化的相关内容原文为 “In this subsection, we restrict ourselves to considering minimization of *Complete* DFA's ”，但是在论文^[4] 第四节中描述最小化算法时没有提及限定在完全 DFA 上，再加之除了 Hopcroft 的算法（算法4.1）外，其他算法并没有输出错误结果，所以对判断算法的正确性有一定的误导。对于此算法的运行结果，本文认为第 4.6.3 节中的方案 2，也即构造完全自动机之后再进行最小化为比较合适的解决方法；
- 经过测试，算法 DFA::min_HopcroftUllman()、DFA::min_dragon() 和 DFA::min_Watson() 表现符合预期情况，本文认为这三个算法可以用于验证其他用于 DFA 最小化算法的正确性和有效性。
- 本文为 FIRE engine 新增加了函数 “DTransRel::closure()”，用于实现 *SReachable*。新增了函数 “DFA::usefulfs()”，用于移除 DFA 中的开始不可达状态。新增函数 “DFA::complete()”，用于构造一个完全 DFA。新增函数 “DFA::Complete()”，用于检查一个 DFA 是否是完全 DFA。

致 谢

本文是在段江涛老师的细心指导下完成的。从立题开始，到最终完成，他都给予了我非常多的关怀和帮助，及时纠正我在完成毕业设计期间的犯的错误，提出了非常宝贵的意见。

参考文献

- [1] Huffman D. The synthesis of sequential switching circuits[J]. Journal of the Franklin Institute, 1954, 257(3):161 - 190.
- [2] Watson B. The design and implementation of the fire engine: A c++ toolkit for finite automata and regular expressions[J]. 1994.
- [3] Watson B W. A taxonomy of finite automata construction algorithms[J]. 1993.
- [4] Watson B W. A taxonomy of finite automata minimization algorithms[J]. 1993.
- [5] Hopcroft J. An $n \log n$ algorithm for minimizing states in a finite automaton[M]//Theory of machines and computations. Elsevier, 1971: 189-196.
- [6] Knuutila T. Re-describing an algorithm by hopcroft[J]. Theoretical Computer Science, 2001, 250(1):333 - 363.
- [7] 胡芙, 黄兆华. 树自动机超最小化[J]. 南昌航空大学学报 (自然科学版), 2015(2):27-32.
- [8] 张锋. 两类有限自动机的最小化[D]. 四川师范大学, 2014.
- [9] 孙丹丹. 模糊树自动机的构造及最小化算法的研究[D]. 华东交通大学, 2014.
- [10] 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.
- [11] Yingjie X. Describing an $n \log n$ algorithm for minimizing states in deterministic finite automaton [M]. January, 2009.
- [12] 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 自动机状态转移图

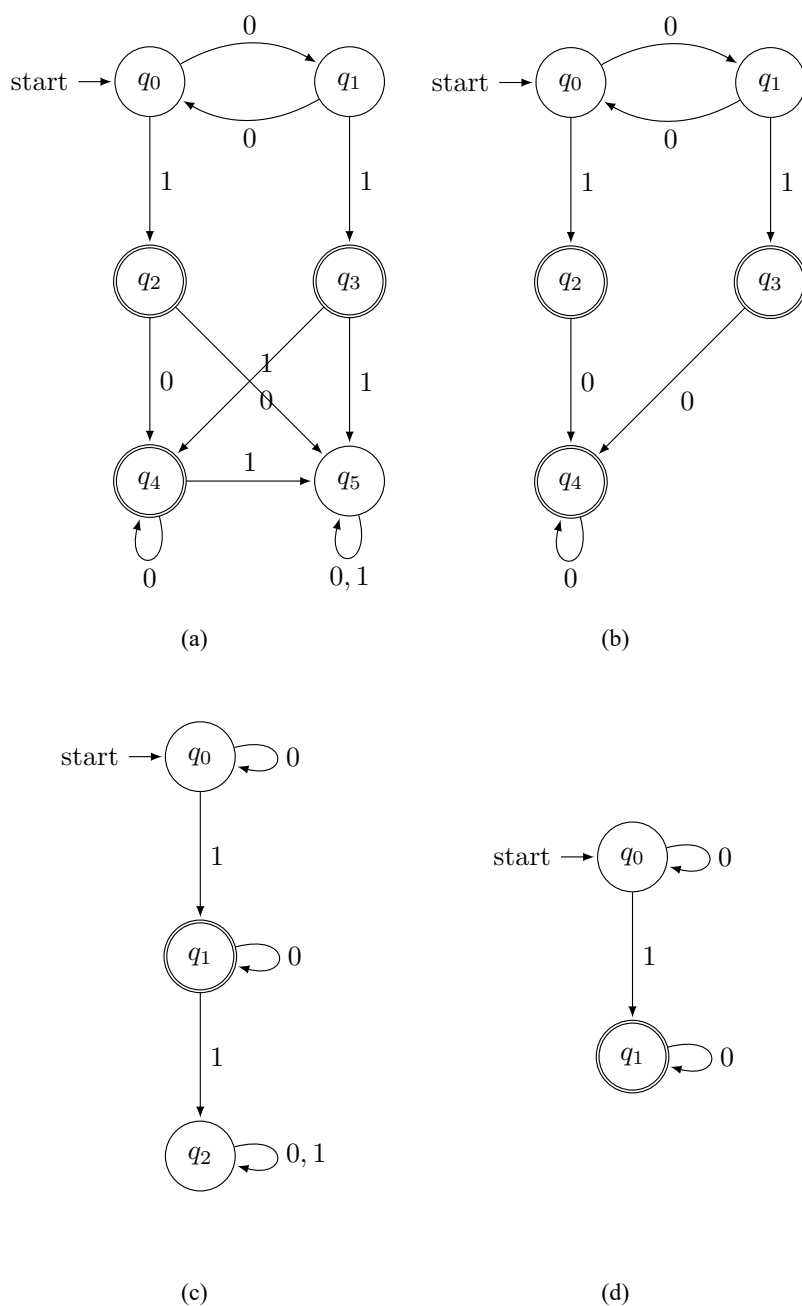


图 A.1 (a) 接受 $\mathcal{L}=0^*10^*$ 的自动机^[10] fig 5-4; (b) 图 (a) 去除非“final-reachable”状态 q_5 ; (c) 与图 (a) 的 *DFA* 同构的含有陷阱状态的最小 *DFA*; (d) 与图 (a) 的 *DFA* 同构的不含陷阱状态的最小 *DFA*。

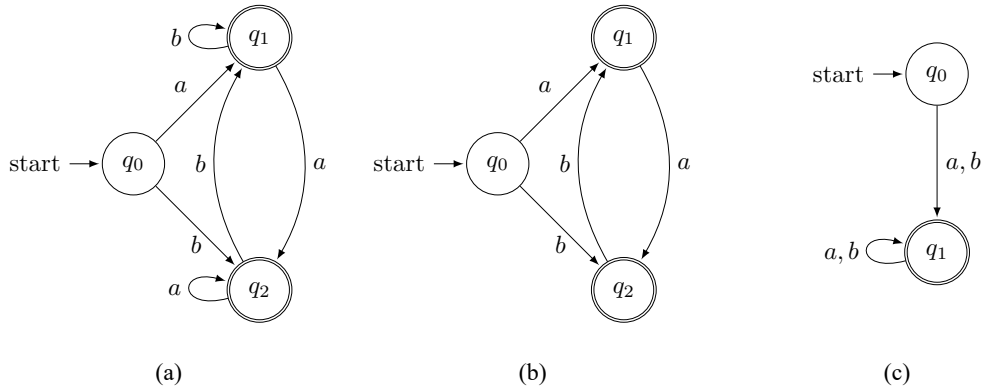


图 A.2 (a) 一个 DFA；(b) 图 (a) 去除转移关系 $T = \{(1 \times 'b' \times 1), (2 \times 'a' \times 2)\}$ 后的最小的 DFA；(c) 与图 (a) 同构的最小的 DFA

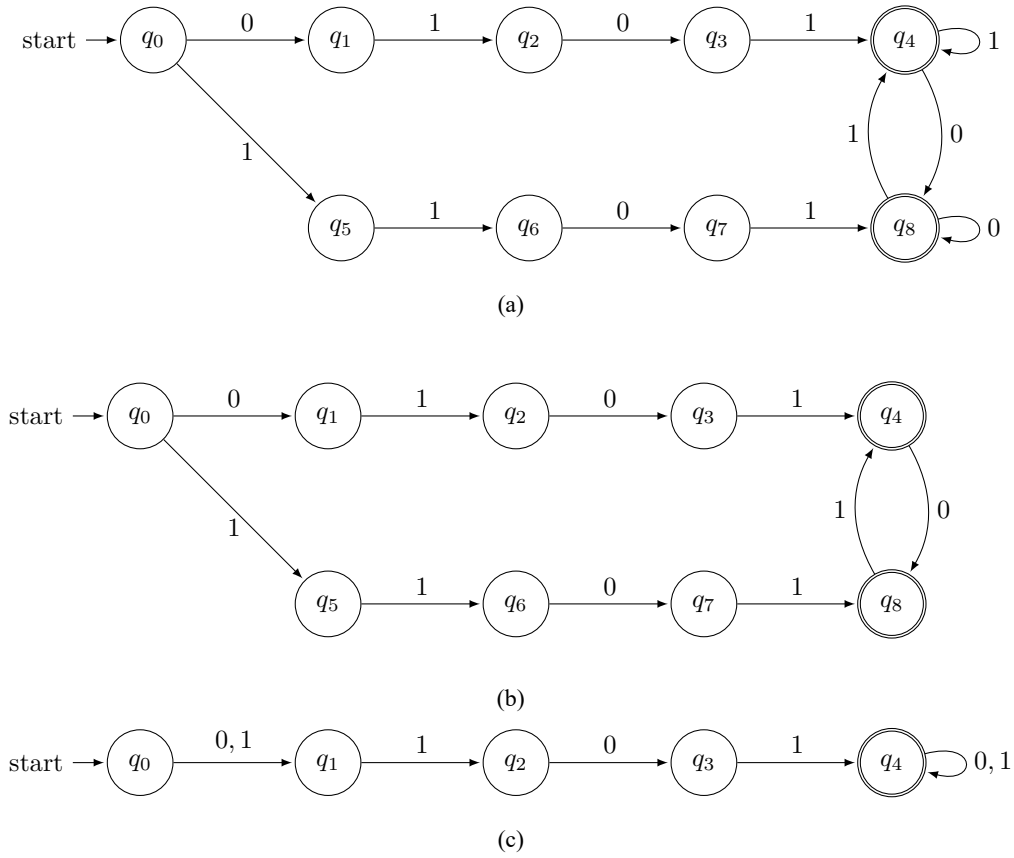


图 A.3 (a) 一个 DFA；(b) 图 (a) 去除转移关系 $T = \{(4 \times '1' \times 4), (8 \times '0' \times 8)\}$ 后的最小的 DFA；(c) 与图 (a) 同构的最小的 DFA

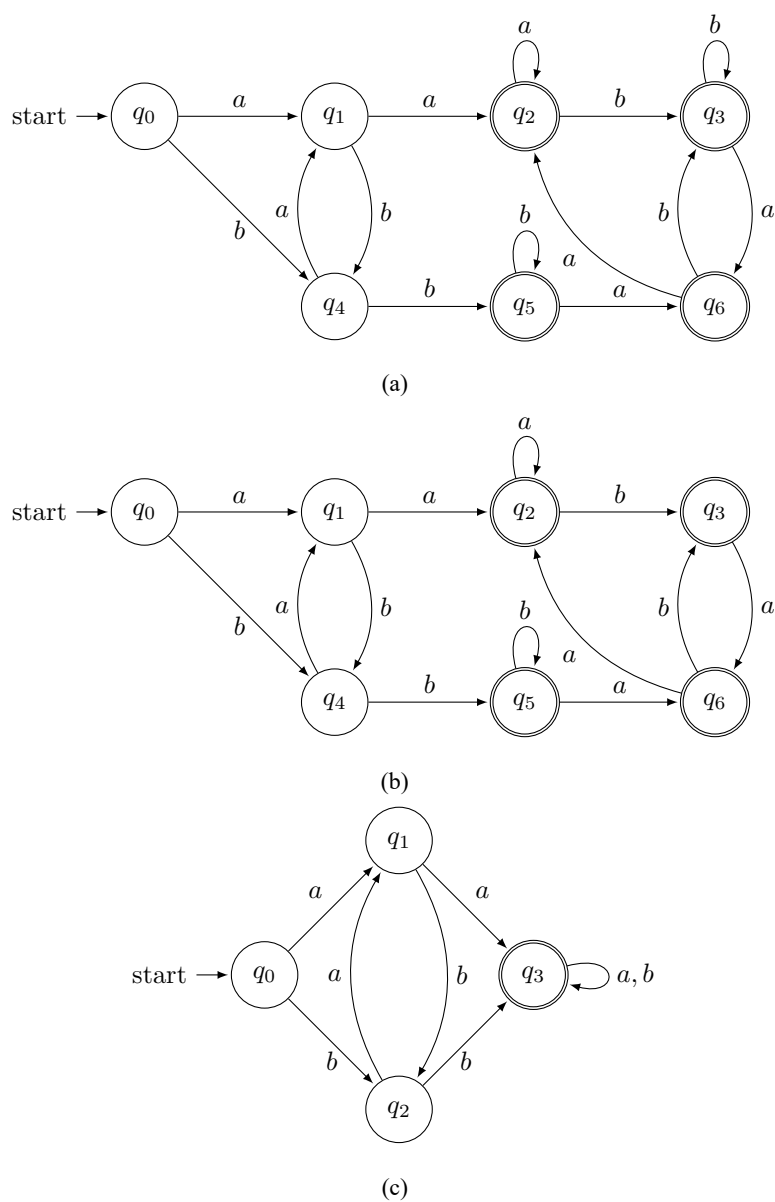


图 A.4 (a) 一个 DFA; (b) 图 (a) 去除转移关系 $T = \{(3 \times 'b' \times 3)\}$ 后的最小的 DFA; (c) 与图 (a) 同构的最小的 DFA

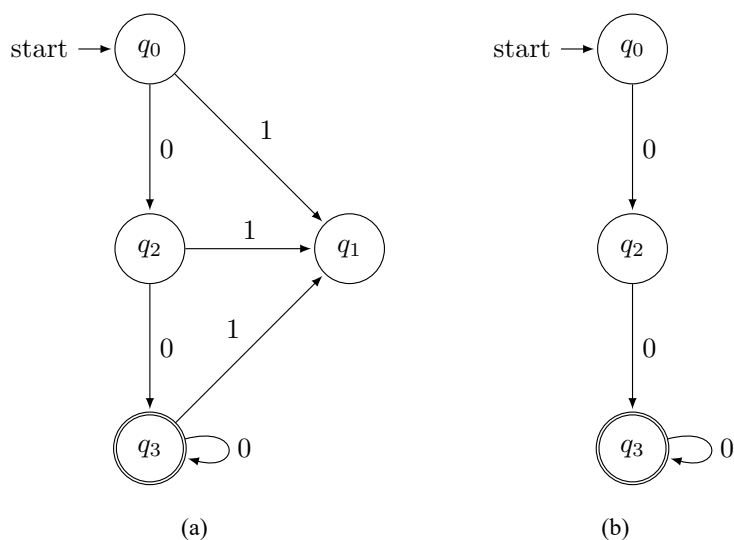


图 A.5 (a) 接受 $\mathcal{L}=000^*$ 的最小的 DFA; (b) 图 (a) 去除非 “final-reachable” 状态 q_1 ;

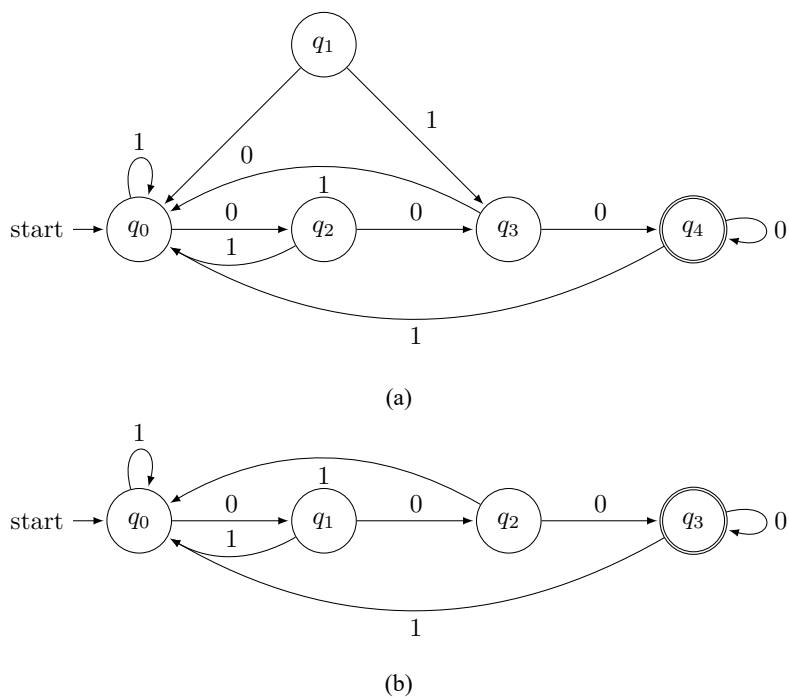
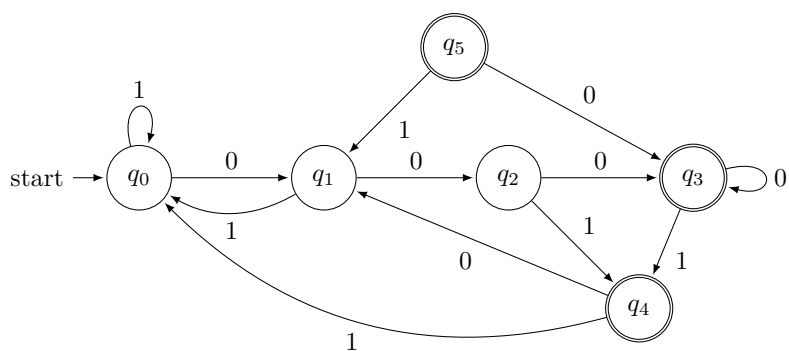
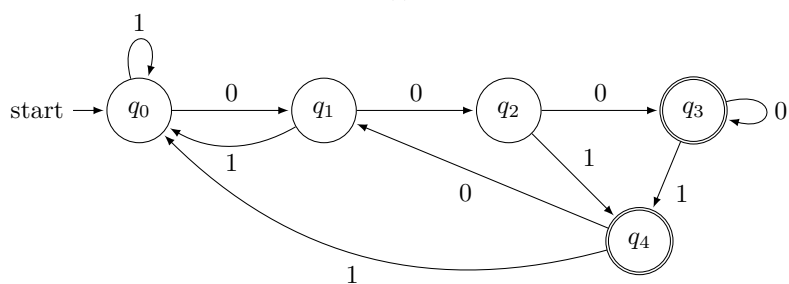


图 A.6 (a) 接受 $\mathcal{L} = \{x000|x \in \{0,1\}^*\}$ 的最小的 DFA; (b) 图 (a) 去除开始不可达状态 q_1 ;

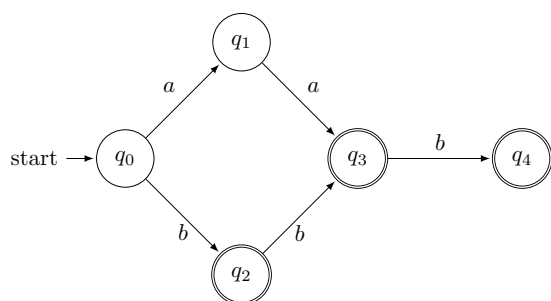
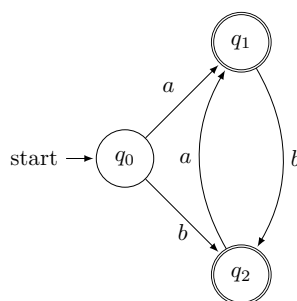


(a)



(b)

图 A.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) 去除开始不可达状态 q_5 ;

(a) $\mathcal{L} = \{aa \cup aab \cup b \cup bb \cup bbb\}$ 

(b)

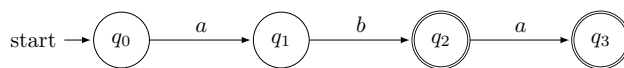
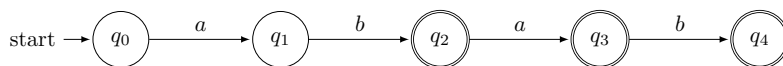
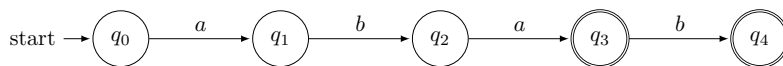
(c) $\mathcal{L} = \{ab \cup aba\}$ (d) $\mathcal{L} = \{ab \cup aba \cup abab\}$ (e) $\mathcal{L} = \{aba \cup abab\}$

图 A.8 一组最小的 DFA

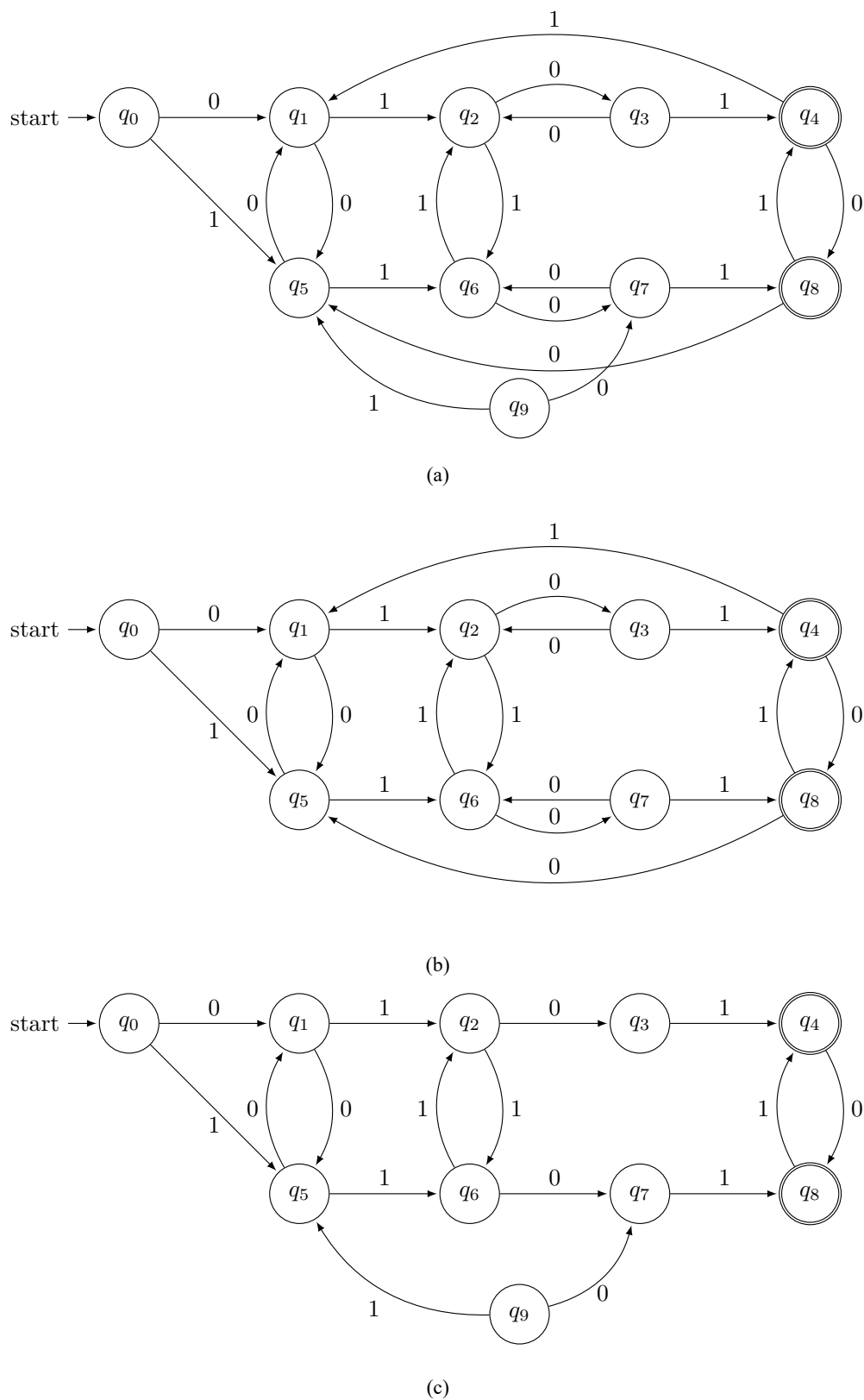


图 A.9 (a) 一个最小的 DFA; (b) 图 (a) 移除开始不可达状态 q_9 ; (c) 图 (a) 移除转移关系 $T = \{(4 \times' 1' \times 1), (8 \times' 0' \times 5), (3 \times' 0' \times 2), (7 \times' 0' \times 6)\}$ 后的 DFA。

附录 B 算法迭代过程

表 B.1 图 A.9(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	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	—

表 B.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 (FIRE engine 中无此变量, 为了方便描述算法迭代过程添加);
- “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;

注释 B.1: Hopcroft 算法 (算法 4.1) 中的 “ L ” 存储的是二元关系 $F \times V$ 或 $(Q \setminus F) \times V$, FIRE engine 中的 “ L ” 是一个一维整型数组, 其中第 i 个元素的值表示第 i 个状态还需要处理的字符数目。当 FIRE engine 中的 “ L ” 数组内所有元素的值全部为 0 时, 算法迭代结束。

附录 C 代码

C.1 实例化 DFA 对象示例代码

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

```
1 #include"DFA.h"
2 #include<iostream>
3 int main()
4 {
5     DFA_components dfa_com1;
6
7     // StateSet S 开始状态集
8     dfa_com1.S.set_domain(4);
9     dfa_com1.S.add(0);
10
11    // StateSet F 结束状态集
12    dfa_com1.F.set_domain(4);
13    dfa_com1.F.add(3);
14
15    // StatePool Q
16    int i = 10;
17    while (i-->0)
18    {
19        dfa_com1.Q.allocate();
20    }
21
22    // DTransRel T transition
23    dfa_com1.T.set_domain(4);
24    dfa_com1.T.add_transition(0, '0', 1);
25    dfa_com1.T.add_transition(1, '0', 2);
26    dfa_com1.T.add_transition(2, '0', 3);
27    dfa_com1.T.add_transition(3, '0', 3);
28    dfa_com1.T.add_transition(0, '1', 0);
29    dfa_com1.T.add_transition(1, '1', 0);
30    dfa_com1.T.add_transition(2, '1', 0);
31    dfa_com1.T.add_transition(3, '1', 0);
32
33    DFA dfa1(dfa_com1);
34    std::cout<<dfa1<<std::endl;
35    return 0;
36 }
```

C.2 Hopcroft 算法代码

本小节是经过第 4.4 节修改后的 Hopcroft 算法全文，位于文件 min-hop.cpp。

```

1 DFA& DFA::min_Hopcroft()
2 {
3     assert(class_invariant());
4
5     // This algorithm requires that the DFA not have any final unreachable
6     // State.
7     //assert(Useful());
8
9     State q;
10
11     // Keep track of the combination fo all the out labels of State's.
12     CRSet C;
13     for (q = 0; q < Q.size(); q++)
14     {
15         C.combine(T.out_labels(q));
16     }
17
18     // Encode set L as a mapping from State to [0,|C|] where:
19     //         if q is a representative of a class in the partition P, then
20     //         L(the abstract list) contains
21     //             ([q],C_0),([q],C_1),...,[q],C_(L(q)_1))
22     //         but not
23     //             ([q],C_(L(q))),...,[q],C_(|C|_1))
24     int *const L(new int[Q.size()]);
25     for (q = 0; q < Q.size(); q++)
26     {
27         L[q] = 0;
28     }
29
30     // Initialize P to be total equivalence relation.
31     StateEqRel P(Q.size());
32
33     // Now set P to be E_0.
34     P.split(F);
35
36     // Now, build the set of equivalentatives and initialize L.
37     StateSet repr(P.representatives());
38
39     if (F.size() <= (Q.size() - F.size()))
40     {
41         repr.intersection(F);
42     }
43     else
44     {
45         repr.remove(F);
46     }
47
48     // Do the final set up of L
49     for (repr.iter_start(q); !repr.iter_end(q); repr.iter_next(q))
50     {
51         L[q] = C.size();

```

```

52     }
53
54     // Use a break to get of this loop
55     while (1)
56     {
57         // Find the first pair in L that still needs processing.
58         for (q = 0; q < Q.size() && !L[q]; q++);
59
60         // It may be that we're at the end of the processing.
61         if (q == Q.size())
62         {
63             break;
64         }
65         else
66         {
67             // mark this element of L as processed.
68             L[q]--;
69             CharRange c = C.iterator(L[q]); // 增加
70
71             // Iterate over al eq. classes, and try to split them.
72             State p;
73             repr = P.representatives();
74
75             for (repr.iter_start(p); !repr.iter_end(p); repr.iter_next(p))
76             {
77
78                 // Now split [p] w.r.t (q,C_(L[q]))
79                 State r(split(p, q, c, P)); // 修改
80                 // r is the representative of the new split of the
81                 // eq. class that was represented by p.
82
83                 if (r != Invalid)
84                 {
85                     // p and r are the new representatives.
86                     // Now update L with the smallest of
87                     // [p] and [r]
88
89                     if (P.equiv_class(p).size() <= P.equiv_class(r).size())
90                     {
91                         L[r] = L[p];
92                         L[p] = C.size();
93                     }
94                     else
95                     {
96                         L[r] = C.size();
97                     } // if
98                 } // if
99             } // for
100         } // if
101     } // while
102
103     // L is no longer needed.
104     delete L;
105
106     // we can now use P to compress the DFA.

```

```

107     compress(P);
108
109     assert(class_invariant());
110     return (*this);
111 }

```

代码 C.2 文件 min-hop.cpp

C.3 SReachable 实现代码

类 DTransRel 为 FIRE engine 里面类 DFA 中存储转移关系的类，但是这个类并没有实现 *SReachable*（见定义 2.18）。本文提出实现 *SReachable* 的算法（算法 4.6），本小节为代码实现。

本文为 FIRE engine 增加的 *SReachable* 功能声明为代码 C.3 中的 “StateSet closure(const StateSet& r) const;”，传入参数 $r = S$ 。

```

1 class DTransRel :public StateTo<DTrans>
2 {
3 public:
4     .....
5     // What are all States reachable from r?           // 新增
6     StateSet closure(const StateSet& r) const;         // 新增
7     .....
8 }

```

代码 C.3 文件 DTransRel.h

本文为 FIRE engine 增加的 *SReachable* 功能定义如代码 C.4 所示。

```

1 // What are all States reachable from r?
2 StateSet DTransRel::closure(const StateSet& r) const
3 {
4     StateSet result(r);
5     StateSet intermediate;
6     StateSet temp;
7     intermediate.set_domain(domain());
8     temp.set_domain(domain());
9     do
10    {
11        result.set_union(intermediate);
12        intermediate.set_union(result);
13        State st;
14        for (result.iter_start(st); !result.iter_end(st); result.iter_next(st))
15        {
16            StateSet temp = lookup(st).range(domain());
17            intermediate.set_union(temp);
18            temp.clear();
19        }
20    } while (result != intermediate);

```

```

21     return(result);
22 }

```

代码 C.4 文件 DTransRel.cpp

C.4 usefufs 实现代码

useful_s 用来移除 DFA 中的开始不可达状态（见变换 2.2），但是 FIRE engine 中并没有实现这个功能，本文提出相应的算法（见算法 4.5）。

注释 C.1: Bruce William Watson 在他的论文^[4] 中有对 *useful_s* 的描述，但是在 FIRE engine 内没有实现；对 *useful_f* 的描述出现在他的论文^[3] 注释 2.39，并且在 FIRE engine 中实现了这个功能，也即函数 DFA::usefulf()。

与上一节内容类似，在文件 DFA.h 内添加功能对应的函数的声明，如代码所示

```

1 class DFA : virtual public FAabs
2 {
3 public:
4     .....
5     // remove any States that cannot reach from a start State // 新增
6     DFA& usefufs();                                           // 新增
7     .....
8 }

```

代码 C.5 文件 DFA.h

在文件 DFA.cpp 中新增功能对应的函数的定义如下

```

1 DFA & DFA::usefufs()
2 {
3     assert(class_invariant());
4     StateSet sreachable(T.closure(S));
5     StateTo<State> newnames;
6     newnames.set_domain(Q.size());
7
8
9     // All components will be constructed into a special structure :
10    DFA_components ret;
11    State st;
12    for (st = 0; st < Q.size(); st++)
13    {
14        // If this is a Usefulf State, carry it over by giving it a name
15        // in the new DFA.
16        if (sreachable.contains(st))
17        {
18            newnames.map(st) = ret.Q.allocate();
19        }
20    }
21 }

```

```

20     else
21     {
22         newnames.map(st) = Invalid;
23     }
24 }
25
26
27 // It is possible that nothing needs to be done(ie.the all States were
28 // already F useful).
29 if (Q.size() != ret.Q.size())
30 {
31     ret.T.set_domain(ret.Q.size());
32     ret.F.set_domain(ret.Q.size());
33
34     CRSet a;
35     for (st = 0; st < Q.size(); st++)
36     {
37         // Only construct the transitions if st is final reachable.
38         if (sreachable.contains(st))
39         {
40             a = T.out_labels(st);
41             State stprime(newnames.lookup(st));
42
43             CharRange b;
44             int it;
45             // Construct the transitions.
46             for (it = 0; !a.iter_end(it); it++)
47             {
48                 b = a.iterator(it);
49                 State stdest;
50                 stdest = newnames.lookup(T.transition_on_range(st, b));
51                 if (stprime != Invalid && stdest != Invalid)
52                 {
53                     ret.T.add_transition(stprime, b, stdest);
54                 }
55             }
56             // This may be a final State.
57             if (F.contains(st)) ret.F.add(stprime);
58         }
59     }
60     ret.S.set_domain(ret.Q.size());
61
62     // Add a start State only if the original one was final reachable.
63     if (S.not_disjoint(sreachable)) ret.S.add(newnames.lookup(S.smallest
64 ()));
65     reconstruct(ret);
66     assert(class_invariant());
67
68     return(*this);
69 }

```

C.5 complete 实现代码

complete 用来构造完全 FA（见定义 2.16），由于本文的讨论仅针对 DFA，所以对该定义的实现也仅适用于 DFA。算法描述见算法 4.4。

向文件 DFA.h 内增加下内容

```

1 class DFA : virtual public FAabs
2 {
3 public:
4     .....
5     // construct a complete DFA.           // 新增
6     // see details Definition 2.15[3]       // 新增
7     DFA& complete();                       // 新增
8
9     // is *this a Complete DFA?           // 新增
10    int Complete()const;                   // 新增
11    .....
12 }
```

代码 C.7 文件 DFA.h

在文件 DFA.cpp 中新增功能对应的函数的定义如下

```

1 // construct a Complete DFA
2 DFA & DFA::complete()
3 {
4
5     if (!Complete())
6     {
7         return (*this);
8     }
9
10    usefulf();
11    usefuls();
12
13    DFA_components ret;
14
15    State q;
16
17    for (q = 0; q <= Q.size(); q++)
18    {
19        ret.Q.allocate();
20    }
21
22    State sink = ret.Q.size()-1;
23
24    CRSet C;
25    for (q = 0; q < Q.size(); q++)
26    {
27        C.combine(T.out_labels(q));
28    }
29
30    ret.S.set_domain(Q.size());
31    ret.F.set_domain(Q.size());
```

```

32
33     ret.S.set_union(S);
34     ret.F.set_union(F);
35
36     ret.S.set_domain(ret.Q.size());
37     ret.T.set_domain(ret.Q.size());
38     ret.F.set_domain(ret.Q.size());
39
40     for (q = 0; q < ret.Q.size(); q++)
41     {
42         CharRange c;
43         for (int i = 0; i < C.size(); i++)
44         {
45             c = C.iterator(i);
46             State stprime = q;
47             if (stprime == sink)
48             {
49                 ret.T.add_transition(stprime, c, sink);
50                 continue;
51             }
52             State stdest = T.transition_on_range(stprime, c);
53             if (stdest != Invalid)
54             {
55                 ret.T.add_transition(stprime, c, stdest);
56             }
57             else
58             {
59                 ret.T.add_transition(stprime, c, sink);
60             }
61         }
62     }
63
64     reconstruct(ret);
65     // TODO: 在此处插入 return 语句
66     return (*this);
67 }
68
69 int DFA::Complete() const
70 {
71     State q;
72     CRSet C;
73     for (q = 0; q < Q.size(); q++)
74     {
75         C.combine(T.out_labels(q));
76     }
77
78     for (q = 0; q < Q.size(); q++)
79     {
80         CharRange c;
81         for (int i = 0; i < C.size(); i++)
82         {
83             c = C.iterator(i);
84             if (T.transition_on_range(q, c) == Invalid)
85             {
86                 return 0;

```

```
87         }  
88     }  
89 }  
90 return 1;  
91 }
```

代码 C.8 文件 DFA.cpp

附录 D 类图

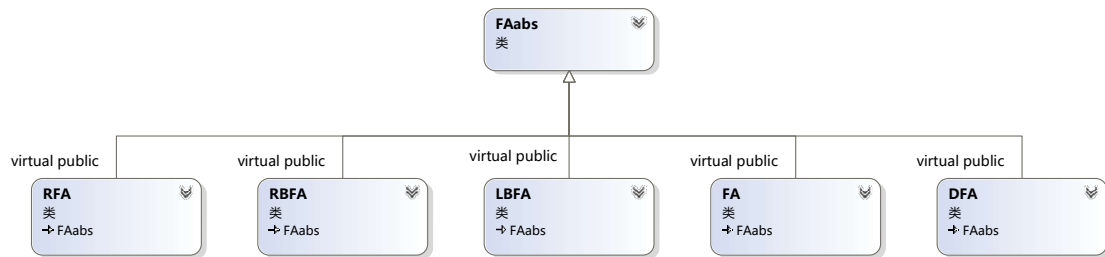


图 D.1 类 FAabs 及其派生类

箭头 \rightarrow 由派生类指向基类。箭头 \rightarrow 从一个类指向其成员变量的类型，箭头上标注了该成员变量的名称。

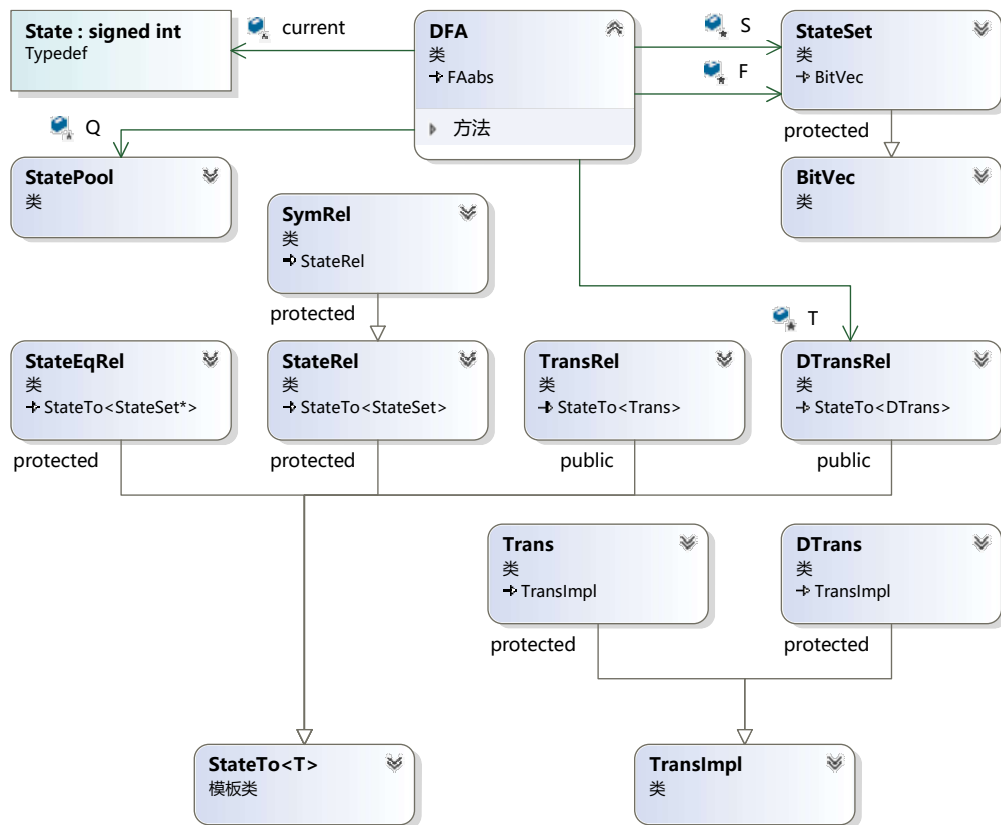


图 D.2 类 DFA 与其成员类及成员类的基类

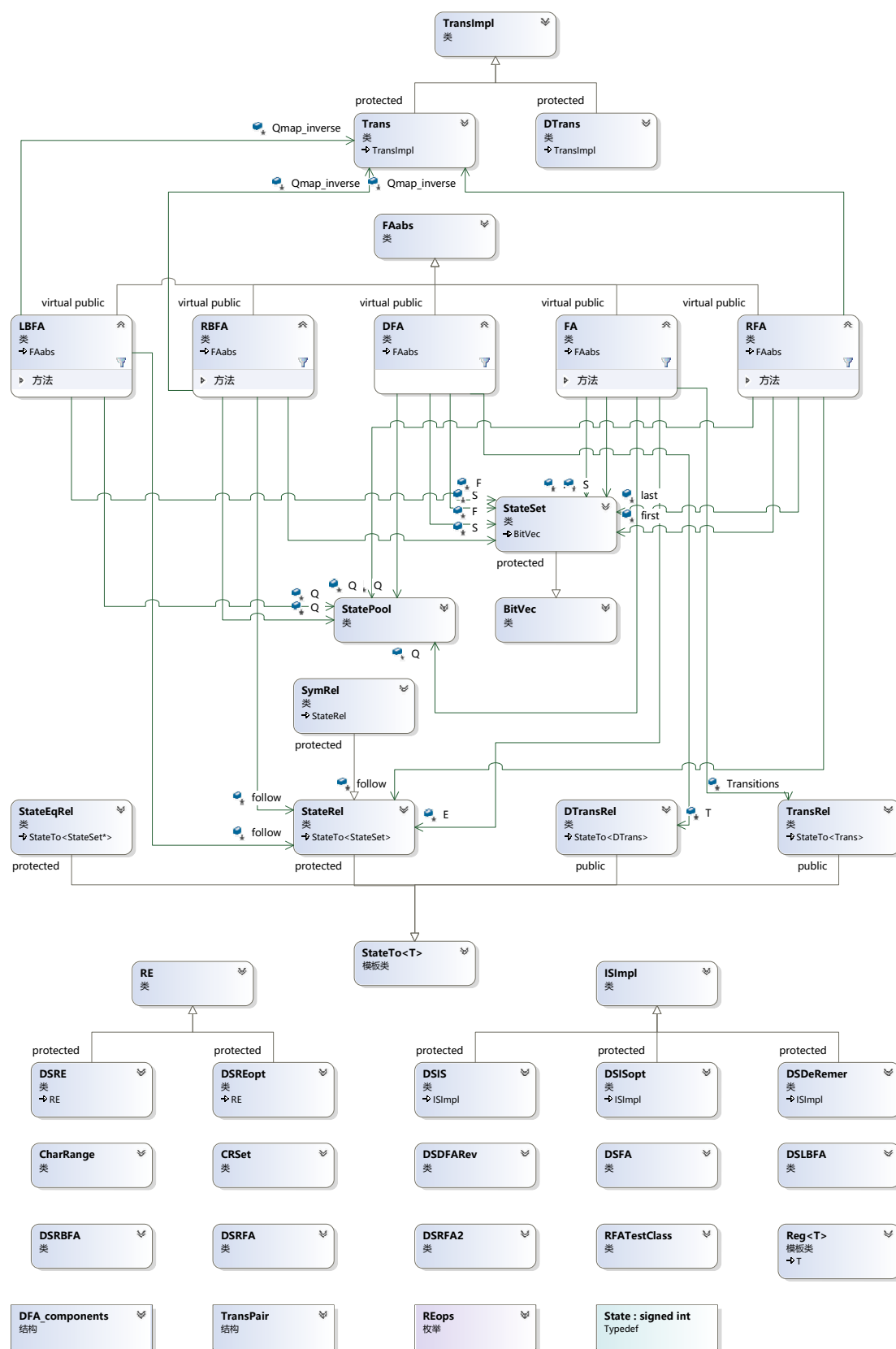


图 D.3 FIRE engine 的构成