目 录

第一章　介绍

自 1950 年来，确定的有限自动机的最小化就一直在研究当中。简单来说就是找到一个唯一的（直至同构）最小的确定的有限自动机，它能接收与给定的确定的有限自动机相同的语言。解决这一问题的算法应用范围很广，从编译器构造到硬件电路的最小化都有它的身影。有了形式多样的应用程序，不同的表示形式的数量也在增加: 大多数教科书都有自己的变体，而时间复杂度最优的算法 (Hopcroft 的算法) 仍然晦涩难懂。

本文介绍了有限自动机最小化算法的有关分类。如下所示:

- 大多数教材的作者称他们的最小化算法由 Huffman 算法 [Huf54] 和 Moore 算法 [Moo56] 直接推导得到。不幸的是，大多数教材都展示了截然不同的算法 (比如 [AHO92, AHO88, Hop79, WW87]), 只有由 Aho 和 Ullman 发表的算法直接源自 [Huf54, Moo56]。

- 虽然大多数算法依赖于计算状态的等价关系，但伴随算法演示的许多解释并未明确提及算法是计算等价关系、它包含的状态划分还是它的补充。

- 算法之间的比较进一步受到呈现方式的巨大差异的阻碍——有时用命令式程序展示、有时使用函数式程序，但通常只作为描述性段落。

有限自动机构造算法的相关分类在 [Wat93] 中。

除了其中一个算法（Brzozowski 的算法）之外，其他所有算法依赖于确定等价[①]的自动机状态的集合。在第 2 章中讨论了不使用等价状态的算法。在第 3 章中给出了状态等价的定义和一些性质。计算等价状态的算法在第 4 章中给出。分类的主要结果被总结在结论部分 5 中。附录 A 和附录 B 给出了阅读本文所需的基本定义。与有限自动机相关的定义取自 [Wat93]。图 1.1 展示了最小化算法之间的关系。

[①]状态的等价稍后定义。

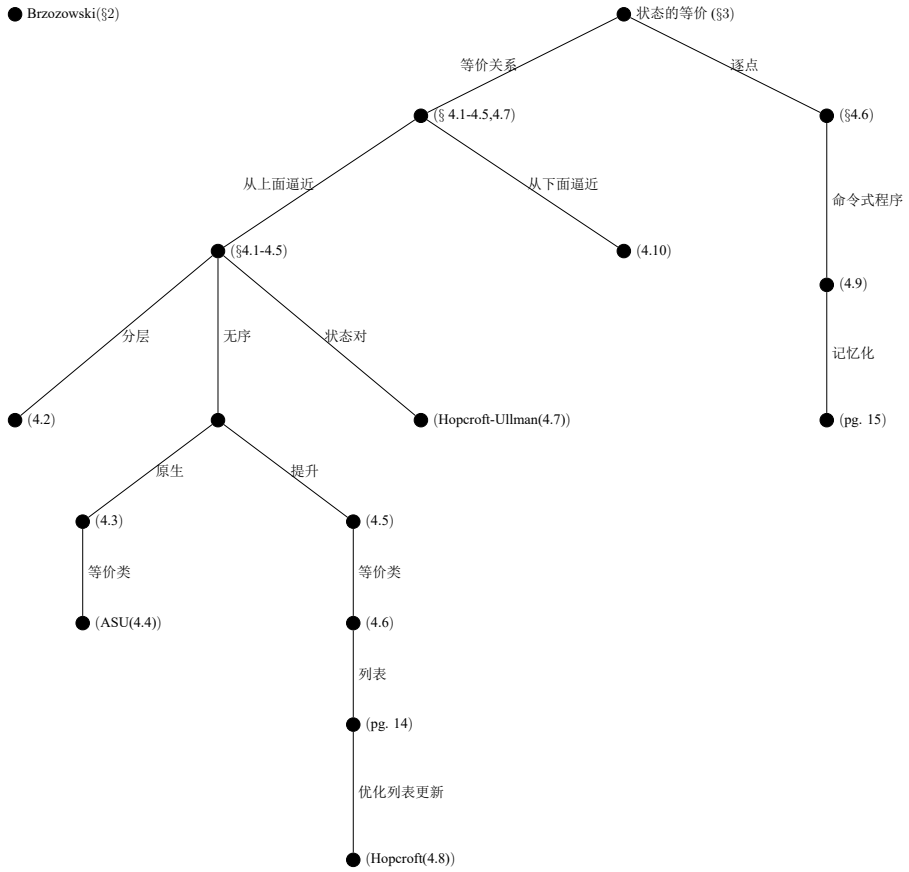

图 1.1

大多数最小化算法的主要计算在于确定等价的（或不等价的）状态，从而在状态上产生等价关系。在本文中，我们考虑以下最小化算法：

- Brzozowski（可能是非确定性的）有限自动机最小化算法在 [Brz62] 中提出。这个优雅的算法（第 2 节）最初是由 Brzozowski 提出，此后又在没有 Brzozowski 的“参与”的情况下被重新提出。给出了一个不含 ϵ-转移的有限自动机（可能不确定的），该算法生成一个最小的接受相同的语言的确定的有限自动机。

- 分层等价计算等价于 [WW87, Moo56, Bra88, Urb89] 中提出。算法（算法 4.2）是由状态等价的定点定义产生的近似序列所建议的直接实现。

- 等价性的无序计算。该算法（算法 4.3，未出现在文献中）计算等价关系；以任意顺序选择状态对（考虑等价性）。

- 等价类的无序计算发表在 [AHO88]。该算法（算法 4.4）是上述计算状态等价性的算法的一种更改。

图 1.1 ：有限自动机最小化算法之间的关系。Brzozowski 的最小化算法与其他算法无关，并作为一个单独的顶点出现。本文提出的每一个算法都作为树的一个顶点出现。对于本文中明确展示的每个算法，则在括号中标示出它在本文中出现的位置。对于没有明确展示的算法，给出了相应的页码。边表示解的细化（也即算法之间的关系）。边上面标记了细化的名称。

- 改进的等价的无序计算。这个算法 (算法 4.5，没有出现在文献中) 也以任意顺序计算等价关系。该算法是对其他无序算法的一个小改进。
- 改进了类的无序计算。该算法（算法 4.6，不在文献中）是上述用来计算状态的等价类的算法的修改。该算法用于 Hopcroft 最小化算法的推导。
- Hopcroft 和 Ullman 的算法在 [Hop79] 中提出。该算法（算法 4.7）计算不等价的（可区分的）关系。虽然它是基于 Huffinan [Huf54] 和 Moore [Moo56] 的算法，但该算法使用一些有趣的编码技术。
- Hopcroft 的算法在 [Hop71, Gri73] 提出。该算法（算法 4.8）是用于最小化的最有名的算法（在运行时间分析方面）。由于 Hopcroft 的原始陈述是难以理解的，所以本文的对该算法介绍基于 Gries 的文章。
- 等价的点态计算。该算法（算法 4.9，不在文献中）计算给定状态对的等价性。它借鉴了一些非自动机相关的技术，例如：类型的结构等价和函数式程序的记忆化。
- 计算等价性（相对于细化）。该算法与其他算法不同，该算法的中间结果可以用于构造较小的（虽然不是最小的）确定的有限自动机。

第二章 Brzozowski 提出的算法

大多数最小化算法应用于 DFA。对于不确定的 FA，应该先应用子集构造，然后应用最小化算法。在本节中，我们将考虑在 (未知的) 算法之后应用子集构造 (带无用状态删除) 以生成最小 DFA 的可能性。我们现在构造这样的算法。（在本节中描述的算法也可用于通过用 $subset$ 替换函数 $subsetopt$ 来构造最小完全 DFA）。

令 $M_0 = (Q_0, V, T_0, \emptyset, S_0, F_0)$ 为一个 $\epsilon\text{-}free\ FA$，对其进行最小化，令 $M_2 = (Q_2, V, T_2, \emptyset, S_2, F_2)$ 为最小的 DFA，那么 $\mathcal{L}(M_0) = \mathcal{L}(M_2)$ ($Min(M_2)$ 当然也是，详见定义 B.19)。（对本节中剩下的部分我们使用 $Minimal$（性质 B.21））。由于最后构造子集，有一些中间 FA $M_1 = (Q_1, V, T_1, \emptyset, S_1, F_1)$，于是 $M_2 = useful \circ subsetopt(M_1)$。我们要求 M_1 是由 M_0 得到的，那么 $\mathcal{L}_{FA}(M_2) = \mathcal{L}_{FA}(M_1) = \mathcal{L}_{FA}(M_0)$。

从 $Minimal(M_2)$（性质 B.21）开始，我们需要：

$$(\forall p, q, : p \in Q_2 \wedge q \in Q_2 \wedge p \neq q : \overrightarrow{\mathcal{L}}(p) \neq \overrightarrow{\mathcal{L}}(q)) \wedge useful(M_2)$$

对与所有的状态 $q \in Q_2$，由于 $M_2 = useful_s \circ subsetopt(M_1)$，所以有 $q \in P(Q_1$。性质 B.25 的子集构造给出：

$$(\forall p : p \in Q_2 : \overrightarrow{\mathcal{L}}(p) = (\cup q : q \in Q_1 \wedge q \in p : \overrightarrow{\mathcal{L}}(q)))$$

我们需要 M_1 上充足的条件来确保 $Minimal(M_2)$。相应条件的推导如下:

$Minimal(M_2)$

$\equiv$ $Minimal$ 的定义 (性质 B.21)

$(\forall p, q, : p \in Q_2 \wedge q \in Q_2 \wedge p \neq q : \overrightarrow{\mathcal{L}}(p) \neq \overrightarrow{\mathcal{L}}(q)) \wedge useful(M_2)$

$\Leftarrow$ 性质 B.25; $M_2 = useful \circ subsetopt(M_1)$

$(\forall p, q, : p \in Q_2 \wedge q \in Q_2 \wedge p \neq q : \overrightarrow{\mathcal{L}}(p) \neq \overrightarrow{\mathcal{L}}(q) = \emptyset) \wedge useful(M_2)$

$\equiv$ Det' 的定义（性质 B.18）和 $useful_s, Useful_f$（备注 B.13）

$$
\begin{array}{ll}
 & Det'(M_1^R) \wedge Useful(M_1^R) \\
\Leftarrow & \quad Det'(M) \Leftarrow Det(M) \\
 & Det(M_1^R) \wedge Useful(M_1^R)
\end{array}
$$

M_1 需要的条件可以通过 $M_1 = R \circ useful_s \circ subsetopt \circ R(M_0)$（反转前缀）建立。

完全最小化算法（对于任何 ϵ-$free\ M_0 \in FA$ ）如下：

$$M_2 = useful_s \circ subsetopt \circ R \circ useful_s \circ subsetopt \circ R(M_0)$$

该算法最初由 Brzozowski 在 [Brz62] 中提出。最初 Jan van de Snepscheut 在他的博士论文 [VdS85] 中提出该算法时，对该算法的描述是模糊的。在该论文中，算法来源于一个教授的私人通讯——埃因霍芬理工大学的 Pereman, Pereman 最初在 Mirkin 的文章 [Mir66] 中找到该算法。虽然 Mirkin 引用了 Brzozowski [Brz64] 的论文，但 Mirkin 的作品是否受 Brzozowki 的最小化工作的影响尚不清楚。Jan van de Snepscheut 的新书 [VdS93] 描述了该算法，但既不提供该算法历史，也不提供该算法的引文（除了他自己的论文外）。

第三章 状态等价最小化

节中，我们限定在完全的 DFA 的最小化中。严格地说，这是一种书写方式上的便利，可以对最小化算法进行修改，使其适用于不完全 DFA。除非 DFA 的语言是 V^*，否则完全最小化 DFA 比不完全最小化 DFA 多一个状态 (sink)。令 $M = (Q, V, T, \emptyset, S, F)$ 为完全 DFA，这个特殊的 DFA 将会贯穿本节内容。我们也假设所有的 M 的所有状态都是开始状态可达的，也就是 $Useful_s(M)$。由于 M 是确定且完全的，我们也把转移关系 $T \in Q \times V \to Q$ 称为全函数，而不是 $T \in Q \times V \to \mathcal{P}(Q)$。

为了最小化 DFA M，我们计算等价关系 $E \subseteq Q \times Q$, 将其定义为

$$(p, q) \in E \equiv (\overrightarrow{\mathcal{L}}(p) = \overrightarrow{\mathcal{L}}(q))$$

由于这是一个等价关系，我们对无序状态对有兴趣。使用有序状态对比使用无序状态对更方便。

根据等价关系 E ，使用 *merge* 转换来对等价关系进行合并。

变换 3.1 (合并状态)： 对于任何满足 $H \in E$ 的等价关系 H, 函数 $merge$ 可以用来减少 DFA 的状态数[①] ，函数 $merge$ 定义为：

$$\begin{aligned} merge((Q, V, T, \emptyset, \{s\}, F), H) \;=\; & \textbf{let} \quad T' = \{([p]_H, a, [q]_H) : (p, a, q) \in T\} \\ & \textbf{in} \\ & \qquad ([Q]_H, V, T', \emptyset, \{[s]_H\}, [F]_H) \\ & \textbf{end} \end{aligned}$$

$merge$ 的定义独立于等价类的代表的选择。函数 $merge$ 拥有以下性质：

$$\mathcal{L}_{FA}(merge(M, H)) = \mathcal{L}_{FA}(M) \wedge |merge(M, H)| \leq |M| \wedge |merge(M, H)| = \sharp H$$

[①] 当 H 是状态上的恒等关系时，函数 $merge$ 不会减少状态的数量。

它保留了 $Complete, \epsilon\text{-}free, Useful, Det,$ 和 $minimal$ 等性质; 当然, $merge$ 仅定义在 $\epsilon\text{-}free$ 和 DFA 上。

为了计算关系 E, 需要函数 $\overrightarrow{\mathcal{L}}$ 的一个性质。

性质 3.2 (函数 $\overrightarrow{\mathcal{L}}$)： 函数 $\overrightarrow{\mathcal{L}}$ 满足

$$\overrightarrow{\mathcal{L}}(p) = (\cup a : a \in V : \{a\} \cdot \overrightarrow{\mathcal{L}}(T(p,a))) \cup (\textbf{ if } (p \in E) \textbf{ then } \{\epsilon\} \textbf{ else } \emptyset \textbf{ fi })$$

这允许我们给出状态等价的另一种描述。

定义 3.3 (状态的等价)： 等价关系 E 是最大细化条件下的划分关系：

$$(p,q) = E \equiv (p \in F \equiv q \in F) \wedge (\forall a : a \in V : (T(p,a), T(q,a)) \in E)$$

注释 3.4： 最大划分关系拥有最多的等价类数目。

注释 3.5： 定义 3.3 中的任何不动点都可以使用。为了最小化自动机，需要最大不动点。

性质 3.6 (近似 E)： 我们可以用连续逼近来计算最大不动点，E 的连续逼近如下 ($k \geq 0$):

$$(p,q) \in (E_{k+1} \equiv (p,q) \in E_k \wedge (\forall a : a \in V : (T(p,a), T(q,a)) \in E_k)$$

E_0 定义为：

$$(p,q) \in E_0 \equiv (p \in F \equiv q \in F)$$

E_0 的一个等价定义是 $E_0 = (Q \setminus F)^2 \cup F^2$。对于所有的 $K \geq 0$ 有 $E_{k+1} \in E_k$。

注释 3.7： 如果 E_k 是一个等价关系，那么 E_{k+1} 也是。E_0 是一个等价关系。

注释 3.8： 有一个直观的 E_k 的说明是有用的。当且仅当没有字符串 $w : |w| \leq k$ 满足 $w \in \overrightarrow{\mathcal{L}}(p) \not\equiv w \in \overrightarrow{\mathcal{L}}(q)$ 时，一个状态对 p, q 也可以说成 k-等价 (写作 $(p,q) \in E_k$) 。作为结果，当且仅当

- 两者都是最终态或者都不是最终态；
- $\forall a \in V, T(p,a)$和$T(q,a)$ 都是 (k-1)-等价 (根据 $\overrightarrow{\mathcal{L}}$ 和 T^* 的定义)；

时 p 和 q 都是 k-等价。

注释 3.9： E 的一个重要性质是，它也是包含于（集合包含而不是细化）定义 3.3 中等价性的最大的固定点。作为最大的不动点，E 可以用 $\subseteq$-下降关系序列来计算，从 $Q \times Q$ 开始，这样的序列不必只包含等价关系。在这样的近似序列中可能有比在上面给出的 E_k 序列有更多的步骤。幸运的是，每个这样的步骤通常都比从 E_k 计算 E_{k+1} 更容易计算。较容易计算这些（但较长）序列的一些算法在第 4.2-4.5 和第 4.7 节中给出。

所有先前已知的算法都是通过上面的连续逼近法（相对于 $\subseteq$）计算 E。第 4.7 节中的新算法通过逐次逼近来计算 E 。在这一节中，解释了这一点的实际重要性。

3.1 可区分性

通过先计算 E 的补集 $D = \neg E$ 来计算 E 是有可能的。关系 D (也叫做状态关系的可区分性) 定义为

$$(p, q) \in D \equiv (\overrightarrow{\mathcal{L}}(p) \neq \overrightarrow{\mathcal{L}}(q))$$

定义 3.10(状态的可区分性)： D 是一个方程中的最小 (在 $\subseteq$ 的条件下, 集合包含[①]) 划分关系

$$(p, q) \in D \equiv (p \in F \not\equiv q \in F) \wedge (\exists a : a \in V : (T(p, a)) \in D)$$

性质 3.11 (逼近 D)： 随等价关系 E，关系 D 可以通过连续逼近来计算 ($k \geq 0$)

$$(p, q) \in D_{k+1} \equiv (p, q) \in D_k \wedge (\exists a : a \in V : (T(p, a), T(q, a)) \in D_k)$$

有 $D_0 = \neg E_0 = ((Q \setminus F) \times F) \cup (F \times (Q \setminus F))$，对于所有的 $k \geq 0$, 有 $D_k = \neg E_k$，同时 $D_{k+1} \subseteq D_k$。

注释 3.12： 对于 E_k，有一个直观的 D_k 的说明是有用的。当且仅当没有字符串 $w : |w| \leq k$ 满足 $w \in \overrightarrow{\mathcal{L}}(p) \not\equiv w \in \overrightarrow{\mathcal{L}}(q)$ 一个状态对 p, q 也可以说成 k-*distinguished* (写作 $(p, q) \in E_k$) 。作为结果，当且仅当

[①]这里，$\subseteq$ 代表设置标准容量；因为 D 不一定是等价关系，所以细化并不适用。

- 其中一个是最终态而另外一个不是最终态；

- 存在 $a \in V$ such that $T(p,a)$ 和 $T(q,a)$ 是在 $(k-1)$-$distinguished$。

时 p 和 q 是 k-distinguished （一些作者也把它叫做 k-$distinguishable$）。

3.2　近似步骤数的上界

我们可以很容易的把一个近似步骤数的上界放进 E 的计算当中。

设 E_j 为定义 E 的方程的最大不动点，可以得到以下逼近步骤（where I_Q 是状态上的恒等关系）：

$$E_0 \subset E_1 \subset \cdots \subset E_j \in I_Q$$

近似序列中部分等价关系的指数是已知的：$\sharp I_Q = |Q|$且$\sharp E_0 \leq 2$，可以推导出：

$$\sharp E_0 < \sharp E_1 < \cdots < \sharp E_j \leq \sharp I_Q = |Q|$$

$\sharp E_0 = 0$ 时，E_0 是最大不动点。$\sharp E_1 = 1$ 时，要么所有状态都是终态，要么所有状态都不是终态。两种情况下 E_0 都是最大不动点。$\sharp E_0 = 2$ 时，$i + 2 \leq \sharp E_i$，由 $j + 2 \leq \sharp E_j \leq \sharp I_Q = |Q|$ 可得 $j \leq |Q| - 2$ 。这给出了计算（从 E_0 开始）最大不动点 E_j 的步骤 $(|Q| - 2)$ 的上界，最大值 0（使用性质 3.6 中的逼近序列）。

上界为 $E = E_{(|Q|-2)\mathbf{max}0}$。正如之后我们会见到的那样，它可以为算法带来效率上的提升。这个结果同样在 Wood [WW87, 引理 2.4.1] 中有所记录。此上界也通过逼近来计算 D 和 $[Q]_E$。

3.3　E 的等价类

计算 $[Q]_E$: E 的等价类集合同样是可行的。为了分割 $[Q]_E$，我们由定义 3.3 出发，把 E 的等价类描述为最大等价关系：

$$
\begin{aligned}
& (\forall p, q : (p, q) \in E : (p \in F \equiv q \in F) \wedge (\forall a : a \in V : (T(p, a), T(q, a)) \in E)) \\
\equiv\ & E \text{ 中的成员关系定义;} \\
& (\forall p, q, a : (p, q) \in E \wedge a \in V : (p \in F \equiv q \in F) \wedge [T(p, a)]_E = [T(q, a)_E]) \\
\equiv\ & \{ \text{ 说明等价类 } Q_0, Q_1 \ \} \\
& (\forall Q_0, Q_1, a : Q_0 \in [Q]_E \wedge Q_1 \in [Q]_E \wedge a \in V : \\
& (\forall p, q : p \in Q_0 \wedge q \in Q_0 : (p \in F \equiv q \in F) \wedge T(p, a) \in Q_1 \equiv T(q, a) \in Q_1))
\end{aligned}
$$

定义 3.10 (函数 $Splittable$)： 为了使其更简洁，我们定义

$$Splittable(Q_0, Q_1, a) \equiv (\exists p, q : p \in Q_0 \wedge q \in Q_0 : (T(p, a) \in Q_1 \not\equiv T(q, a) \in Q_1))$$

使用 $Splittable, [Q]_E$ 是最大的划分 (在 $\sqsubseteq$ 的情况下)，那么 $[Q]_E \sqsubseteq [Q]_{E_0}$，且有

$$(\forall Q_0, Q_1, a : Q_0 \in [Q]_E \wedge Q_1 \in [Q]_E \wedge a \in V : \neg Splittable(Q_0, Q_1, a))$$

该描述会被用于 $[Q]_E$ 的计算。

第四章　计算 $E, D, [Q]_E$ 的算法

本节中，对计算 $E, D, [Q]_E$ 的算法进行叙述。一些算法以通用形式发表：计算 D 和 E。由于只需要 D 和 E 之中的一个（不是两个都需要），在实际使用中，可以更改通用算法来只计算其中一个。

4.1　通过分层逼近来计算 D 和 E

根据 E_k，E_{k+1} 的定义（D 也适用）很自然的引出下面计算 D 和 E 的算法（其中 k 是一个 ghost 变量，仅用于指定不变量）。

算法 4.1

$G, H = D_0, E_0$;
$G_{old}, H_{old}, k := \emptyset, Q \times Q, 0$;
{ 恒有：$G = D_k \wedge H = E_k$}
do $G \neq G_{old} \longrightarrow$
　　$\{G \neq G_{old} \wedge H \neq H_{old}\}$
　　$G_{old}, H_{old} := G, H$;
　　$G := (\cup p, q : (p, q) \in G_{old} \wedge (\exists a : a \in V : (T(p, a), T(q, a)) \in G_{old}) : \{(p, q)\})$;
　　$G := (\cup p, q : (p, q) \in H_{old} \wedge (\forall a : a \in V : (T(p, a), T(q, a)) \in H_{old}) : \{(p, q)\})$;
　　$\{G = \neg H\}$
　　$k := k + 1$
od $\{G = D \wedge H = E\}$

由于该算法计算序列 D_K 和 E_K，该算法被称为分层计算 D 和 E。循环中的 G 和 H 的更新可以由下面展示的程序中的另一个循环实现。

算法 4.2 可以分为两部分：一部分只计算 D，另外一部分只计算 E。只计算 E 的算法本质上是由 Wood 在 [WW87, pg.132] 发表的。据 Wood 称，此算法建立在 Moore [Moo56] 的基础之上。其时间复杂度为 $\mathcal{O}(|Q|^3)$ 。在 [Bra88] 中，Brauer 使用了一些编码技术提供了此算法的时间复杂度为 $\mathcal{O}(|Q|^2)$ 的版本，而后 Urbanek 在 [Urb89] 中提供了 Brauer 的空间优化版本。这里没有给出任何一个它们的变体。仅计算 D 的算法本文中没有提及。

只要稍稍多做一点工作，这个算法就可更改用来计算 $[Q]_E$。

算法 4.2

$G, H = D_0, E_0$;
$G_{old}, H_{old}, k := \emptyset, Q \times Q, 0$;
{ 恒有：$G = D_k \wedge H = E_k$}
do $G \neq G_{old} \longrightarrow$
    $\{G \neq G_{old} \wedge H \neq H_{old}\}$
    $G_{old}, H_{old} := G, H$;
    for $(p, q) : (p, q) \in H_{old}$ **do**
        if $(\exists a : a \in V : (T(p, a), T(q, a)) \in G_{old}) \longrightarrow$
            $G, H := G \cup (p, q), H \setminus \{(p, q)\}$
        else
            $(\exists a : a \in V : (T(p, a), T(q, a)) \in H_{old}) \longrightarrow$ skip
        end if
    end for
    $\{G = \neg H\}$
    $k := k + 1$
od $\{G = D \wedge H = E\}$

4.2 通过无序逼近来计算 D，E 和 $[Q]_E$

我们可以用任意顺序的状态对来计算 E，而不是计算每一个 E_k（按层计算 E）(如备注 3.9 所示)。使用下面的算法可以做到（也可以用来计算 D）:

算法 4.3

$G, H = D_0, E_0$;
{ 恒有：$G = \neg H \wedge G \subseteq D$}
do $(\exists p, q, a : a \in V \wedge (p, q) \in H : (T(p, a), T(q, a)) \in G) \longrightarrow$
    let $p, q : (p, q) \in H \wedge (\exists a : a \in V : (T(p, a), T(q, a)) \in G)$
    $\{(p, q) \in D\}$
    $G, H := G \cup (p, q), H \setminus (p, q)$
od $\{G = D \wedge H = E\}$

此算法可以分解为一个只计算 D 和一个只计算 E 的算法。在每个迭代步骤的最后，H 可能不是一个等价状态（也就是 $H \neq H^*$）——详见注释 3.9。可以通过在 **od** 前面添加一个等式来对这个算法稍作修改:

$$H := (\mathbf{MAX}_{\subseteq} J : J \subseteq H \wedge J = J^* : J); G := \neg H$$

等式的添加使得算法可以计算细化序列 E_k（详见备注 3.9）. 如果使用计算量化 **MAX** 的简便方法，那么这个等式可以提升这个算法的时间复杂度。本文中未提及此算法。

当我们把上面的这个算法转化来计算 $[Q]_E$，最终的算法如下，由 Aho, Sethi 和 Ullman 在 [AHO88, Alg.3.6] 中提出：

算法 4.4

1: $P : [Q]_{E_0}$;
2: { 恒有：$G = \neg H \wedge G \subseteq D$}
3: **do** $(\exists Q_0, Q_1, a : Q_0 \in P \wedge Q_1 \in P \wedge a \in V : Splittable(Q_0, Q_1, a)) \longrightarrow$
4: **let** $Q_0, Q_1, a : Splittable(Q_0, Q_1, a)$;
5: $Q'_0 := \{p : p \in Q_0 \wedge T(p, a) \in Q_1\}$;
6: $\{\neg Splittable(Q_0 \setminus Q'_0, Q_1, a) \wedge \neg Splittable(Q'_0, Q_1, a)\}$
7: $P := P \setminus \{Q_0\} \cup \{Q_0 \setminus Q'_0, Q'_0\}$
8: **od**
9: $\{(\forall Q_0, Q_1, a : Q_0 \in P \wedge Q_1 \in P \wedge a \in V : \neg Splittable(Q_0, Q_1, a))\}$
10: $\{P = [Q]_E\}$

此算法时间复杂度为 $\mathcal{O}(|Q|^2)$。

4.3　通过无序逼近更高效的计算 D 和 E

我们提出另外一种考虑任意顺序状态对的算法。该算法（也可以计算 D）由两个嵌套循环组成：

算法 4.5

1: $G, H = D_0, E_0$;
2: { 恒有：$G = \neg H \wedge G \subseteq D$}
3: **do** $(\exists p, q, a : a \in V \wedge (p, q) \in H : (T(p, a), T(q, a)) \in G) \longrightarrow$
4: **let** $p, a : p \in Q \wedge a \in V \wedge (\exists q : (p, q) \in H : (T(p, a), T(q, a)) \in G)$;
5: **for** $q : (q, p) \in H \wedge (T(p, a), T(q, a)) \in G$ **do**
6: $G, H := G \cup \{(p, q)\}, H \setminus \{(p, q)\}$
7: **end for**
8: **od** $\{G = D \wedge H = E\}$

与算法 4.3 一样，在每个外部迭代步骤的最后，可能有 $H \neq H^*$。可以像算法 4.3 那样给 H 赋值来解决这个问题。本文中没有此算法。它同样也可以更改后只用于计算 D 或 E。

更改上面的算法来计算 $[Q]_E$ 特别有趣。更改后的算法将会在 4.5 小节中用于推导由 Hopcroft 提出的算法，这个算法是用于 FA 最小化的最有名的算法。这个算法是（其中变量 P_{old} 仅用于不变量）：

算法 4.6

1: $P : [Q]_{E_0}$;
2: { 恒有：$[Q]_E \sqsubseteq p \sqsubseteq [Q]_{E_0}$}
3: **do** $(\exists Q_1, a : Q_1 \in P \wedge a \in V : (\exists Q_0 : Q_0 \in P : Splittable(Q_0, Q_1, a))) \longrightarrow$
4: **let** $Q_1, a : (\exists Q_0, Q_0 \in P : Splittable(Q_0, Q_1, a))$
5: $P_{old} := P$;
6: {恒有:$[Q]_E \sqsubseteq P \sqsubseteq P_{old}$}
7: **for** $Q_0 : Q_0 \in P_{old} \wedge Splittable(Q_0, Q_1, a)$ **do**
8: $Q'_0 := \{p : p \in Q_0 \wedge T(p, a) \in Q_1\}$;
9: $P := P \setminus \{Q_0\} \cup \{Q_0 \setminus Q'_0, Q'_0\}$
10: **end for**
11: $(\forall Q_0 : Q_0 \in P : \neg Splittable(Q_0, Q_1, a))$
12: **od**
13: $\{(\forall Q_0, Q_1, a : Q_0 \in P \wedge Q_1 \in P \wedge a \in V : \neg Splittable(Q_0, Q_1, a))\}$
14: $\{P = [Q]_E\}$

内部循环“分割”每一个符合条件的等价类 Q_0，实际上，如果有 $\neg Splittable(Q_0, Q_1, a).)$，那么一些特殊的 Q_0 不会被分割。

4.4 Hopcroft 和 Ullman 的算法

从 D 的定义，可以知道，当且仅当 $p \in F \not\equiv q \in F$ 或有 $a \in V$，也就是 $(T(p, a), T(q, a)) \in D$。这构成了本小节中考虑的算法的基础。对于每对状态 (p, q)，我们将一组状态对 $L(p, q)$ 关联起来:

$$(r, s) \in L(p, q) \Rightarrow ((p, q) \in D \Rightarrow (r, s) \in D)$$

对每一对 (p, q) ($(p, q) \notin D_0$ ——已知 P 和 Q 不可区分)，进行以下操作：

- 如果存在 $a \in V$，那么可以知道 $(T(p, a), T(q, a)) \in D$，于是 $(p, q) \in D$。把 (p, q) 添加到 D 的近似中，伴随着 $L(p, q)$, , 对于每个 $(r, s) \in L(p, q)$，添加 $L(r, s)$, 对于每个 $(t, u) \in L(r, s)$, 添加 $L(t, u)$ 等等；
- 如果没有 $a \in V$ 使得 $T((p, a), T(q, a)) \in D$。因为 $L(T(p, b), T(q, b)) \in D \Rightarrow (p, q) \in D$，那么对于所有的 $b \in V$，我们把状态对 (p, q) 放进集合 $L(T(p, b), T(q, b))$。如果有一部分 $b \in V, L(T(p, b), T(q, b)) \in D$, 那么我们就把 $L(T(p, b), T(q, b))$ 放进关系 D 中 (包括 (p, q)) 。

在我们的算法展示中，给出的不变量不足证明算法的正确性，而是用于证明算法的工作原理。该算法是：

算法 4.7

for $(p,q) : (p,q) \in (Q \times Q)$ **do**
    $L(p,q) := \emptyset$
end for
$G := D_0$;
$\{$恒有:$G \subseteq D \wedge (\exists p,q : (p,q) \notin D_0 : (\exists r,s : (r,s) \in L(p,q) : (p,q) \in D \Rightarrow (r,s) \in D))\}$
for $(p,q) : (p,q) \notin D_0$ **do**
    if $(\exists a : a \in V : (T(p,a), T(q,a)) \in G) \longrightarrow$
        $A, B := \{(p,q)\}, \emptyset$;
        $\{$恒有:$A \subseteq D \wedge B \subseteq G \wedge A \cap B = \emptyset \wedge A \cup B = (\cup p,q : (p,q) \in B : L(p,q))\}$
        do $A \neq \longrightarrow$
            let$(r,s) : (r,s) \in A$;
            $G := G \cup \{(r,s)\}$;
            $A, B := A \setminus \{(r,s)\}, B \cup \{(r,s)\}$;
            $A := A \cup (L(r,s) \setminus B)$
        od
    else if $(\exists a : a \in V : (T(p,a), T(q,a)) \notin G) \longrightarrow$
        for $a \in V : T(p,a) \neq T(q,a)$ **do**
            $\{(T(p,a), T(q,a)) \in D \Rightarrow (p,q) \in D\}$
            $L(T(p,a), T(q,a)) := L(T(p,a), T(q,a)) \cup \{(p,q)\}$
        end for
    end if
end for$\{G = D\}$

此算法的时间复杂度为 $\mathcal{O}(|Q|^2)$，Hopcroft 在 [Hop79, Fig.3.8] 提出。在 [Hop79] 中，它也对 Huffman [Huf54] 和 Moore [Moo56] 有所贡献。Hopcroft 和 Huffman 把 L 描述为将每个状态对映射到一个状态对列表。这里不需要这个列表的数据类型，而是需要一个集合。

更改上面的算法来计算 E 是有可能的。但是本文中未提及。

4.5　Hopcroft 的计算 $[Q]_E$ 的高效算法

我们由 Hopcroft 推导出一个高效的算法。这个算法也已经由 Gries [Gri73] 推导得到。这个算法也是目前所有 DFA 最小化算法中时间复杂度最低的算法。

从算法 4.6 开始。回想一下，内部循环将每个等价类 Q_0 相对于 (Q_1, a) 进行“splits”。观察结果（来自 Hopcroft ）是，一旦所有等价类都对一个特定的 $(Q1, a)$ 进行了分割，那么在外部重复的后续迭代步骤中就不需要对相同的 $(Q1, a)$ 进行任何等价类的分割 [Hop71, pp.190-191], [Gri73, 引理 5]。我们可以使用这个事实来维护这样一组 (等价类，字母符号) 对 L。然后我们将对等价类进行相对于 L 的元素

的分割。在这个算法的最初表示 [Hop71, Gri73] 中，L 是一个列表。由于没有必要这样做，我们将 L 的类型保留为一个集合。

```
1: P := [Q]_{E_0};
2: L := P × V;
3: {恒有：[Q]_E ⊑ P ⊑ [Q]_{E_0} ∧ L ⊆ (P × V)
4:    ∧L ⊇ {(Q_1, a) : (Q_1, a) ∈ (P × V) ∧ (∃Q_0 : Q_0 ∈ P : Splittable(Q_0, Q_1, a))}
5:    ∧(∀Q_0, Q_1, a : Q_0 ∈ Q ∧ (Q_1, a) ∈ L : ¬Splittable(Q_0, Q_1, a)) ⇒ (P = [Q]_E)}
6: do L ≠ ∅ ⟶
7:     let Q_1, a : (Q_1, a) ∈ L;
8:     P_old := P;
9:     L := L \ {(Q_1, a)};
10:    {恒有:[Q]_E ⊑ P ⊑ P_old}
11:    for Q_0 : Q_0 ∈ P_old ∧ Splittable(Q_0, Q_1, a) do
12:        Q'_0 := {p : p ∈ Q_0 ∧ T(p, a) ∈ Q_1};
13:        P := P \ {Q_0} ∪ {Q_0 \ Q'_0, b};
14:        for b : b ∈ V do
15:            if (Q_0, b) ∈ L ⟶ L := L \ {(Q_0, b)} ∪ {(Q'_0, b), (Q_0, \Q'_0, b)}
16:            else if (Q_0, b) ∉ L ⟶ L := L ∪ {(Q'_0, b), (Q_0, \Q'_0, b)}
17:            end if
18:        end for
19:    end for
20:    {(∀Q_0, Q_0 ∈ P : ¬Splittable(Q_0, Q_1, a))}
21: od {P = [Q]_E}
```

最内部的 L 的更新是很繁琐的，将会用来推导 Hopcroft 和 Gries 的算法。在集合 L 的更新中，如果 $(Q_0, b) \in L$(某些$b \in V$) 且 Q_0 已经被分割成 $Q_0 \setminus Q'_0$ 和 Q'_0，那么 L 中 (Q_0, b) 会被 $(Q_0 \setminus Q'_0, b)$ 和 (Q'_0, b) 替换。

对于 Hopcroft 的另一个观察是 $(Q_0, b), (Q'_0, b), (Q_0 \setminus Q'_0, b)$ 中任何两个的等价类的拆分与对于所有三个等价类的拆分相同 [Hop71, pp. 190-191],[Gri73, 引理 6]。出于效率的考虑，我们在集合 L 的更新中选择三个 $((Q_0, b), (Q'_0, b), (Q_0 \setminus Q'_0, b))$ 中最小的两个。如果 $(Q_0, b) \notin L$，那么对 (Q_0, b) 的分割就已经完成，我们把 (Q'_0, b) 或者 $(Q_0 \setminus Q'_0, b)$ (较小者) 添加到 L。另一方面，如果 $(Q_0, b) \in L$，那么分割还未完成，从 L 中移除 (Q_0, b)，然后添加 (Q'_0, b) 或者 $(Q_0 \setminus Q'_0, b)$ 。

最后，我们观察到，从 $P = [Q]_{E_0} = \{Q \setminus F, F\}$ 开始，就已经完成了对 Q 的分割。也就是说，我们只需要对 $(Q \setminus F, b)$ 或者 (F, b) 进行分割即可 (对所有的 $b \in V$)[Hop71, pp. 190-191],[Gri73, 引理 6]。

这给出了算法:

算法 4.8 Hopcroft

```
1: P := [Q]_{E_0};
2: L := (if (|F| ≤ |Q \ F|)then {F}else {Q \ F}end if ) × V;
3: {恒有：[Q]_E ⊑ P ⊑ [Q]_{E_0} ∧ L ⊆ (P × V)
4:    ∧(∀Q_0, Q_1, a : Q_0 ∈ Q ∧ (Q_1, a) ∈ L : ¬Splittable(Q_0, Q_1, a)) ⇒ (P = [Q]_E)}
5: do L ≠ ∅ ⟶
6:     let Q_1, a : (Q_1, a) ∈ L;
7:     P_old := P;
8:     L := L \ {(Q_1, a)};
9:     {恒有:[Q]_E ⊑ P ⊑ P_old}
10:    for Q_0 : Q_0 ∈ P_old ∧ Splittable(Q_0, Q_1, a) do
11:        Q'_0 := {p : p ∈ Q_0 ∧ T(p, a) ∈ Q_1};
12:        P := P \ {Q_0} ∪ {Q_0 \ Q'_0, b};
13:        for b : b ∈ V do
14:            if (Q_0, b) ∈ L ⟶ L := L \ {(Q_0, b)} ∪ {(Q'_0, b), (Q_0, \Q'_0, b)}
15:            else if (Q_0, b) ∉ L ⟶
16:                L := L ∪ (if (|Q'_0| ≤ |Q_0 \ Q'_0|)then {(Q'_0, b)}else {(Q_0 \ Q'_0, b)}end if );
17:            end if
18:        end for
19:    end for
20:    {(∀Q_0, Q_0 ∈ P : ¬Splittable(Q_0, Q_1, a))}
21: od {P = [Q]_E}
```

不幸的是，这个算法的时间复杂度分析是非常复杂的，这里不做讨论。Hopcroft 和 Gries 在 [Gri73, Hop71] 中提及其时间复杂度为 $\mathcal{O}(|Q|log|Q|)$。

4.6　计算 $(p, q) \in E$

从确定两种类型的结构等价性问题出发，将 E 的定义转化为函数式程序，可以递归地计算出两种状态的等价性。如果将未修改的定义直接用于函数式程序，则有可能程序无法正常结束。为了使函数式程序工作，它需要有两个状态和第三个参数。

下面的程序与 [tE91] 中的类似，逐点计算 E；调用 $equiv(p, q, \emptyset)$ 确定状态 p 和状态 q 是否等价。它假设两个状态是等价的 (通过将这对状态放在 S 中，即第三个参数中)，直到出现其他情况。

```
1: function equiv(p, q, S)
2:     if {p, q} ∈ S ⟶ eq := true
3:     else if {p, q} ∉ S ⟶
4:         eq := (p ∈ F ≡ q ∈ F);
5:         eq := eq ∧ (∃a : a ∈ V : equiv(T(p, a), T(q, a), S ∪ {{p, q}}))
6:     end if
7:     return eq
8: end function
```

可以使用一个循环来实现

```
1: function equiv(p, q, S)
2:     if {p, q} ∈ S ⟶ eq := true
3:     else if {p, q} ∉ S ⟶
4:         eq := (p ∈ F ≡ q ∈ F);
5:         for a : a ∈ V do
6:             eq := eq ∧ equiv(T(p, a), T(q, a), S ∪ {{p, q}})
7:         end for
8:     end if
9:     return eq
10: end function
```

这个程序的正确性在 [tE91] 中已经展示。当然，在实际实现中，eq 可以用于该循环 (当 $eq \equiv false$ 时打断循环)。为了更加清楚明了，这里略去了优化。

有很多方法可以使这个程序更加有效率。回顾 3.2 节中的 $E = E_{|Q|-2}\mathbf{max}_0$。添加参数 k 到函数 $equiv$, 调用 $equiv(p, q, \emptyset, k)$ ， $(p, q) \in E_k$ 作为返回值。递归深度由 $(|q| - 2)\mathbf{max}0$ 决定。新函数是：

```
1: function equiv(p, q, S, k)
2:     if k = 0 ⟶ eq := (p ∈ F ≡ q ∈ F)
3:     else if k ≠ ∧{p, q} ∈ S ⟶ eq := true
4:     else if k ≠ ∧{p, q} ∉ S ⟶
5:         eq := (p ∈ F ≡ q ∈ F);
6:         for a : a ∈ V do
7:             eq := eq ∧ equiv(T(p, a), T(q, a), S ∪ {{p, q}})
8:         end for
9:     end if
10:    return eq
11: end function
```

第三个参数 S 是一个全局变量，在实际使用中提高算法的效率。作为结果，由于 $equiv$ 使用了全局变量，它不再是一个函数式程序。此变体的正确性已经在 [tE91] 中展示。假设 S 被初始化为 $\emptyset$ 。当 $S = \emptyset$ 时，调用 $equiv(p, q, \emptyset, k)$, $(p, q) \in E_k$ 为返回值。调用之后，$S = \emptyset$。

算法 4.9 E 的逐点计算

```
1: function equiv(p, q, S, k)
2:     if k = 0 ⟶ eq := (p ∈ F ≡ q ∈ F)
3:     else if k ≠ ∧{p, q} ∈ S ⟶ eq := true
4:     else if k ≠ ∧{p, q} ∉ S ⟶
5:         eq := (p ∈ F ≡ q ∈ F);
6:         S := S ∪ {{p, q}};
7:         for a : a ∈ V do
8:             eq := eq ∧ equiv(T(p, a), T(q, a), S ∪ {{p, q}})
9:         end for
10:        S := S \ {{p, q}}
11:    end if
12:    return eq
13: end function
```

实际使用中，$equiv$ 可以被记忆化来进一步降低时间复杂度。

本文未涉及此算法。

4.7　逼近计算 E

函数 $equiv$ 的最后一个版本可以用来计算 E 和 D （假设 I_Q 是状态上的恒等关系，S 是算法 4.9 中使用的全局变量）：

算法 4.10 计算 E from below

```
1: S, G, H := ∅, ∅, I_Q;
2: {恒有 : (G ∪ H) ⊆ (Q × Q) ∧ G ⊆ D ∧ H ⊆ E}
3: do G ∪ H ≠ Q × Q →
4:     let p, q : (p, q) ∈ ((Q × Q) \ (G ∪ H));
5:     if equiv(p, q, (|Q| − 2)max0) ⟶ H := H ∪ {(p, q)}
6:     else if ¬equiv(p, q, (|Q| − 2)max0) ⟶ G := G ∪ {(p, q)}
7:     end if
8: od {G = D ∧ H = E}
```

可以根据下面的做法来获得更进一步的效率提升：

- 把 G 初始化为 $G := ((Q \setminus F) \times F) \cup (F \times (Q \setminus F))$；
- 利用 $E = E^*$，显然 E 是对称的，可以将所需的计算量减半。H 可以在每个迭代步骤中由 $H := H^*$ 更新（实现的数据结构是易于实现 $*$-闭包的）
- 利用

$$
\begin{aligned}
(p,q) \notin E \;\Rightarrow\; & (\forall r,s : r \in Q \wedge s \in Q \\
& \quad \wedge (\exists w : w \in V^*(r,w) = p \wedge T^*(s,w) = q) : ((r,s) \notin E) \\
(p,q) \in E \;\Rightarrow\; & (\forall w : w \in V^* : (T^*(p,w), T^*(q,w)) \in E)
\end{aligned}
$$

如果 p,q 是可区分的，那么当 $w \in V^*$ and $T(r,w) = p \wedge T(s,w) = q$ 时，r,s 也是可区分的。

此算法有比 Hopcroft 的算法 [Hop71, Gri73] 更高的时间复杂度。对比其他所有算法，这个算法有一个显著的优点：即使函数 $equiv$ 从上面计算 E (关于 $\subseteq$, 细化) , 主程序从下面计算 (关于 $\subseteq$, 普通集合包含[①])。因此，任何计算 E 的中间结果都可以用于减小（至少部分减小）自动机的大小；其他所有算法都有不可用的中间结果。当最小化算法的运行时间由于某些原因（比如实时应用程序）受到限制时，该性质可以用于减小自动机的大小。

[①]这是集合包含，与细化相反，因为在计算过程中，中间结果 H 可能不是等价关系。

第五章　总结

最小化算法的总结如下：

- 给出了 Brzozowski 最小化算法的推导。这个推导比原始推导（Brzozowski）更容易理解，或者 van de Snepscheut 给出的推导更容易理解。给出了最小化算法的简史。希望解决其发现的一些错误归因。

- 等价性（关系 E）和可区分性（关系 D）作为某些方程的固定点的定义比许多教科书演示更容易理解。

- E 的不动点特征使得计算 E（或 D）所需的逼近步骤的数目的上界特别容易。这个上界后来被证明在确定一些算法的时间复杂度和逐点算法效率提升的上是有用的。

- 作为最大不动点的 E 的定义有助于识别所有（先前）已知的算法从上面计算出 E（关于细化）。因此，这些算法全都具有在最小化有限自动机的过程中不可用的中间结果。

- 在相同的框架下，我们成功地给出了所有已知的书籍上描述的算法。他们中的大多数被证明本质上是相同的，仅在它们的环结构上有微小的差异。一个例外是 Hopcroft 和 Ullman 的算法 [Hop79]，它具有明显不同的循环结构。本文提出的算法（带有不变量），比原来的表述更容易理解。我们的报告突出了这样一个事实，即算法中的主要数据结构不必是列表，一个集合就足够了。

- Hopcroft 的最小化算法 [Hop71] 最初是以一种不太容易理解的方式呈现的。就像 Gries 的文章 [Gri73]。我们努力以清晰和精确的方式推导出该算法。本文中的介绍突出了两个重点：该算法的推导起点是一种易于理解的简单算法;，并且在 Hopcroft 和 Gries 的这种算法的呈现中使用列表数据结构是不必要的——可以使用集合。

- 本文提出了一些新的最小化算法，其中许多是已知算法的变体。有两个新算法（在第 4.6 和 4.7 节中给出）不是推导自任何已知算法，–并且在它们自己的版权中是重要的–。

  – 提出了一种以逐点法计算关系 E 的算法。该算法由一个用于确定结构等价类型的算法优化而来。在优化构成中某些技术扮演了重要角色：

    * 计算 E 所需的步骤数的上界被用来通过限制逐点计算 E 时需要考虑的状态对的数量来改进算法；
    * 算法的函数式程序部分的记忆化 (memoization) 用于减少冗余计算量。

  – 提出了一种新的计算 E 的算法。该算法利用 E 的逐点计算来构造和细化 E 的近似。因为计算是从下面，该算法的中间结果可用于（至少部分地）减小 DFA 的大小。这可用于 DFA 最小化时间量受限的应用中（如在实时应用中）。相反，所有（先前）已知的算法具有不可用的中间结果。

附录 A　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，$A \to B$ 代表所有从 A 到 B 的函数的集合。而 $A \not\to B$ 代表所有从 A 到 B 的“partial functions”。

注释 A.3： 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.4 (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.5 (关系的合成)： 给出集合 A, B, C 和两个关系 $E \subseteq A \times B$ 和 $F \subseteq B \times C$，定义关系的合成（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.6 (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.7 (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.8 (字母表)： 字母表是有限大小的非空集合。

定义 A.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”。

定义 A.10 (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.11 (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 (Finite automata,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.8”和“惯例A.2”。

注释 B.2： 我们也会在不同的情况下把转移关系写成不同的形式。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.3 (FA 的大小)： 定义一个 FA 的大小为 $|M| = |Q|$。

定义 B.4 (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.5 (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。

注释 B.6： 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.7 (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.8 (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.9 (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.10 (ϵ-$free$)： 当且仅当 $E = \emptyset$ 时，M 是 ϵ-$free$ 的。

定义 B.11 (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.12 (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.13： $Useful_s$ 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.14 (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.15 (确定的有限自动机 (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.16 (FA 的确定性)： DFA 代表所有确定的有限自动机的集合。我们把 $FA \backslash DFA$ 称为不确定的有限自动机 ($NDFA$,$nondeterministic\ \ finite\ \ automata$) 的集合。

惯例 B.17 (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 $T \in Q \times V \nrightarrow Q$（$\nrightarrow$ 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.18 (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.19 (DFA 的最小)： 满足以下条件时，$M \in DFA$ 是最小的：

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

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.20 (DFA 的最小化)： 根据 Myhill-Nerode 定理，一个 DFA M,$Min(M)$ 是唯一的最小的 DFA ，定理的相关介绍在 [Wat93]。

性质 B.21 (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 $Min(M) \Rightarrow Minimal(M)$。

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 $Min_{\mathcal{C}}(M) \Rightarrow Minimal_{\mathcal{C}}(M)$。

B.2　有限自动机的变换

变换 B.22 (FA 反转)： FA 反转由上标函数 $R \in FA \longrightarrow FA$ 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.23 (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：

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

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 B.24 (子集构造)： 函数 $subset$ 把一个 ϵ-$free\ FA$ 转换为一个 DFA (in the **let** clause $T' \in \mathcal{P}(Q) \times V \longrightarrow \mathcal{P}(\mathcal{P}(Q))$):

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

有时候也把它说成“幂集”构造。

性质 B.25 (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.26 (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

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

参考文献

[AHO88] AND J.D. Ullman AHO, A.V. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Co.,Reading, M.A, 1988.

[AHO92] AND J.D. Ullman AHO, A.V. *Foundations of the Computer Science*. Computer Science Press, New York, N.Y, 1992.

[Bra88] Wilfried Brauer. On minimizing finite automata. *Bulletin of the EATCS*, 35:113–116, 1988.

[Brz62] Janusz A Brzozowski. Canonical regular expressions and minimal state graphs for definite events. *Mathematical theory of Automata*, 12(6):529–561, 1962.

[Brz64] Janusz A Brzozowski. Derivatives of regular expressions. In *Journal of the ACM*. Citeseer, 1964.

[DDD+76] Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, Etats-Unis Informaticien, and Edsger Wybe Dijkstra. *A discipline of programming*, volume 1. prentice-hall Englewood Cliffs, 1976.

[Gri73] David Gries. Describing an algorithm by hopcroft. *Acta Informatica*, 2(2):97–109, 1973.

[Hop71] John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In *Theory of machines and computations*, pages 189–196. Elsevier, 1971.

[Hop79] J.D. Ullman Hopcroft, John E. *Introduction to Automata Theory, Languages, and computation*. Addison-Wesley Publishing Co., 1979.

[Huf54] David A Huffman. The synthesis of sequential switching circuits. *Journal of the franklin Institute*, 257(3):161–190, 1954.

[Mir66] Boris G Mirkin. On dual automata. *Cybernetics and Systems Analysis*, 2(1):6–9, 1966.

[Moo56] Edward F Moore. Gedanken-experiments on sequential machines. *Automata studies*, 34:129–153, 1956.

[Myh57] John R Myhill. Finite automata and the representation of events. 1957.

[Ner58] Anil Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958.

[RS59] Michael O Rabin and Dana Scott. Finite automata and their decision problems. *IBM journal of research and development*, 3(2):114–125, 1959.

[tE91] Hubertus Maria Martinus ten Eikelder. *Some Alogrithms to Decide the Equivalence of Recursive Types*. Eindhoven University of Technology, Department of Mathematics and Computing ⋯, 1991.

[Urb89] Friedrich J. Urbanek. On minimizing finite automata. *Bulletin of the EATCS*, 39:205–206, 1989.

[VdS85] Jan LA Van de Snepscheut. *Trace theory and VLSI design*. Number 200. Springer Science & Business Media, 1985.

[VdS93] Jan LA Van de Snepscheut. What is computing all about? In *What Computing Is All About*, pages 1–9. Springer, 1993.

[Wat93] Bruce William Watson. A taxonomy of finite automata construction algorithms. 1993.

[WW87] Derick Wood and Derrick Wood. Theory of computation. 1987.