浅谈压缩后缀自动机

江苏省常州高级中学 徐翊轩

摘 要

本文从一系列传统的信息学竞赛中的后缀数据结构出发，自然地引出了一种新型后缀数据结构：压缩后缀自动机，以及它的一个变体：对称压缩后缀自动机。对于两者，本文均详细地分析了它们的性质，并且给出了一种，在允许离线的情况下，时空复杂度均为线性的构造算法，同时，也介绍了它们的实际应用。

在当今的信息学竞赛界，同学们对后缀自动机的研究热情依旧很高，希望本文引入的压缩后缀自动机，及其背后的思考方式，能够起到抛砖引玉的作用，激发更多同学对后缀数据结构，乃至更广泛的问题的思考与探究。

1 前言

后缀数据结构是当前在算法竞赛中被广泛应用的一个体系。它包含后缀数组、后缀字典树、后缀自动机、后缀树等广为人知的内容，这些后缀数据结构能高效地处理许多关于字符串的问题。

如果从自动机理论的角度来理解这些后缀数据结构的关系，便能发现：后缀字典树最小化后，可以得到后缀自动机，而将其收缩后，将会得到后缀树。而如果同时对后缀字典树进行最小化和收缩，便得到了本文将要介绍的压缩后缀自动机。

由此，笔者对压缩后缀自动机展开了研究，并写作了本文。

本文的结构如下：

第 2 节介绍了一些有关字符串和自动机的基本定义。

第 3 节回顾了 OI 中的传统后缀数据结构，并介绍了它们之间的关系。

第 4 节详细介绍了压缩后缀自动机的定义、构造方式，及应用。

第 5 节介绍了其变体，对称压缩后缀自动机的定义、构造方式，及应用。

第 6 节对文章所涉及的内容进行了分析和总结。

2 基础定义

2.1 字符串基础

令 S 为一个字符串，用 $|S|$ 表示 S 中的字符个数，即 S 的**长度**。

若 $|S| = 0$，则称 S 为**空串**，记作 $S = \varnothing$。

记 $S[i]$ 表示 S 中的第 i 个字符，其中 $1 \le i \le |S|$。

记 $S[l, r]$ 表示 S 中的第 l 到第 r 个字符组成的字符串，称为 S 的一个**子串**。

若 $1 \le l \le r \le |S|$，则 $S[l, r]$ 是一个非空子串，否则，$S[l, r] = \varnothing$。

记 $Pre[i] = S[1, i]$，即 S 中第 i 个字符及其之前的部分，称为 S 的一个**前缀**。

记 $Suf[i] = S[i, |S|]$，即 S 中第 i 个字符及其之后的部分，称为 S 的一个**后缀**。

2.2 上下文

在本文中，我们所讨论的问题都是针对单一母串 S 的问题，因此，我们在此所规定的记号通常会略去字符串 S，而仅考虑将 S 的某个子串写入记号。

定义 2.2.1：对于字符串 S 的子串 $S[l, r]$，

定义其**上文** $LeftContext(S[l, r])$ 为：

$$S[\min\{x \mid \forall S[l, r] = S[y - (r - l), y], S[x, r] = S[y - (r - x), y]\}, r]$$

类似地，定义其**下文** $RightContext(S[l, r])$ 为：

$$S[l, \max\{x \mid \forall S[l, r] = S[y, y + (r - l)], S[l, x] = S[y, y + (x - l)]\}]$$

通俗地来讲，一个子串 $S[l, r]$ 的上文即为最长的，每当 $S[l, r]$ 在 S 中出现，其出现位置前方一定会出现的字符串与 $S[l, r]$ 拼接的结果；而 $S[l, r]$ 的下文即为最长的，$S[l, r]$ 与每当 $S[l, r]$ 在 S 中出现，其出现位置后方一定会出现的字符串拼接的结果。

在这样的表述下，不难发现

$$LeftContext(RightContext(S[l, r])) = RightContext(LeftContext(S[l, r]))$$

定义 2.2.2：对于子串 $S[l, r]$，定义其**上下文** $Context(S[l, r])$ 为：

$$LeftContext(RightContext(S[l, r]))$$

这样，一个子串的上下文即为最长的每当其在 S 中出现，便一定会出现的字符串。

定义 2.2.3：对于字符串 S 的子串 $S[l,r]$，

定义其**左集合** $Left(S[l,r])$ 为：

$$\{x \mid S[x, x+(r-l)] = S[l,r]\}$$

类似地，定义其**右集合** $Right(S[l,r])$ 为：

$$\{x \mid S[x-(r-l), x] = S[l,r]\}$$

可以发现，一个子串的左集合就是其在 S 中所有出现位置的左端点集合，而一个子串的右集合就是其在 S 中所有出现位置的右端点集合。

例 2.2：考虑字符串 $S = aabaaba$

则有

$$LeftContext(a) = a, LeftContext(aa) = aa, LeftContext(b) = aab$$

$$RightContext(a) = a, RightContext(aa) = aaba, RightContext(b) = ba$$

$$Context(a) = a, Context(aa) = aaba, Context(b) = aaba$$

对于 S 的子串 ab，其出现位置为 $\{[2,3],[5,6]\}$，因此

$$Left(ab) = \{2,5\}, Right(ab) = \{3,6\}$$

2.3 自动机

定义 2.3[1]：**确定有限状态自动机**（DFA）$\mathcal{A}$是由

*一个非空有限的**状态**集合Q

*一个输入**字母表**Σ（非空有限的字符集合）

*一个转移**函数**$\delta : Q \times \Sigma \to Q$（例如：$\delta(q,\sigma) = p, (p,q \in Q, \sigma \in \Sigma)$）

*一个**开始状态**$s \in Q$

*一个**接受状态**的集合$F \subseteq Q$

所组成的多元组。因此一个 DFA 可以写成这样的形式：$\mathcal{A} = (Q, \Sigma, \delta, s, F)$。

简单来说，一个 DFA 可以看做是一张有限的有向图，点集为 Q，图中的每一条边上写有一个 Σ 中的字符。存在一个特殊的节点 s，称为开始状态，另外还有一系列特殊的节点 F，称为接受状态。在本文中，我们所研究的是能够接受一个字符串 S 的所有子串的自动机，因此，我们所谈论到的 DFA 均满足 $F = Q$。

[1]确定有限状态自动机的定义引用自参考文献[1]

3 OI 中的传统后缀数据结构

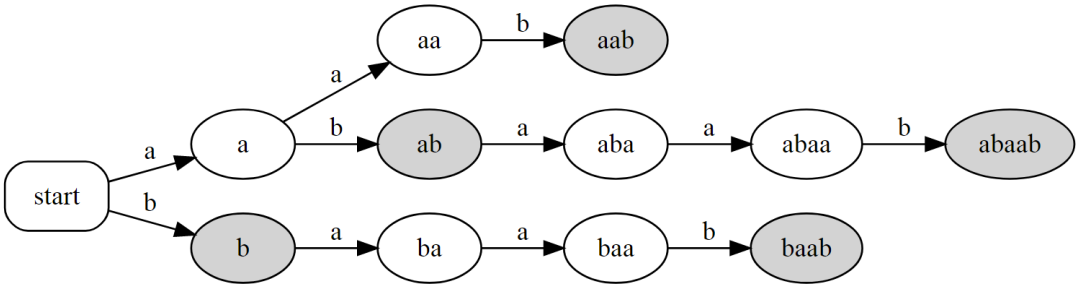
3.1 后缀字典树

为了得到一个能够接受所有 S 的子串，并且不会接受其它字符串的 DFA，不难想到对于 S 的所有后缀，建立一棵**字典树**[2]。由于字典树能够接受的字符串是集合中某一字符串的前缀，而 S 所有后缀的前缀恰好构成了 S 子串的集合，因此，这样的字典树能够不重不漏地接受所有 S 的子串。

定义 3.1：

对 $\{Suf[i] \mid i \in \{1,2,\ldots,|S|\}\}$ 建立字典树，称得到的 DFA 为**后缀字典树**。

例 3.1：字符串 $S = abaab$ 的后缀字典树如下图所示：



图中标记为灰色的节点代表的字符串对应了 S 的一个后缀。

可以发现，后缀字典树能够不重不漏地接受 S 的所有子串。

3.2 后缀自动机

在最坏情况下，后缀字典树的节点数和边数均可能达到 $O(|S|^2)$ 级别，在实际应用中，往往是难以接受的。为此，我们需要寻找更为高效的后缀数据结构。

前面提到，后缀字典树是一种 DFA，而 DFA 存在着一种**最小化**的概念[3]。

例如，在例 3.1 中，可以发现，节点 aab,baab,abaab 均没有出边。因此，可以认为，它们是等价的，从而将它们合并为同一个节点。类似地，节点 baa 和 abaa 均只有一条指向等价节点的，对应字符相同的边。因此，在这两个节点处的后续转移都是相同的，同样可以认为它们是等价的，而将它们合并为一个节点。

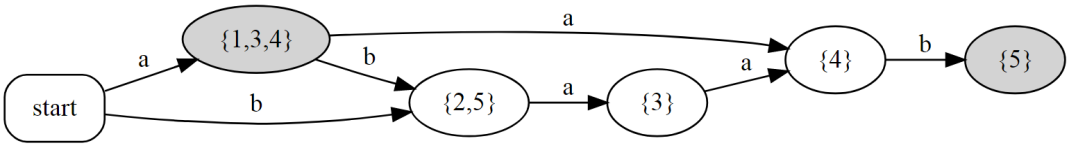
[2]字典树是一种用节点代表字符串，用边代表字符的数据结构，具体可见参考文献[8]

[3]有关内容可以参见参考文献[2]

定义 3.2：

将后缀字典树上 *Right* 集合相同的节点合并，称得到的 DFA 为**后缀自动机**。

例 3.2.1：字符串 $S = abaab$ 的后缀自动机如下图所示：



在上面的例子中，节点上标注的集合即为该节点所代表的 *Right* 集合，除了开始状态以外，其余的出度不是 1 的节点被标记为了灰色。

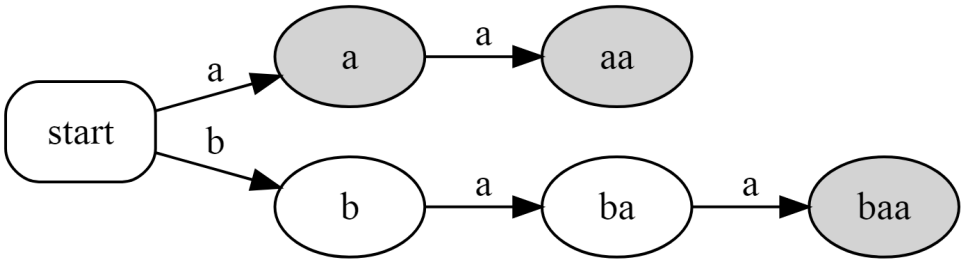
可以发现，沿着后缀自动机的边进行 DFS，能够还原出原本的后缀字典树。

在定义 3.2 中，我们并没有直接提及对后缀字典树的最小化。但事实上，考虑对后缀字典树最小化的含义，我们希望在不影响每个节点可以接受的字符串集合的同时，尽量减少 DFA 的节点数。在匹配子串的问题中，该过程几乎就是将对应字符串的 *Right* 集合相同的节点合并为了一个。这和定义 3.2 中对后缀自动机的定义是类似的。

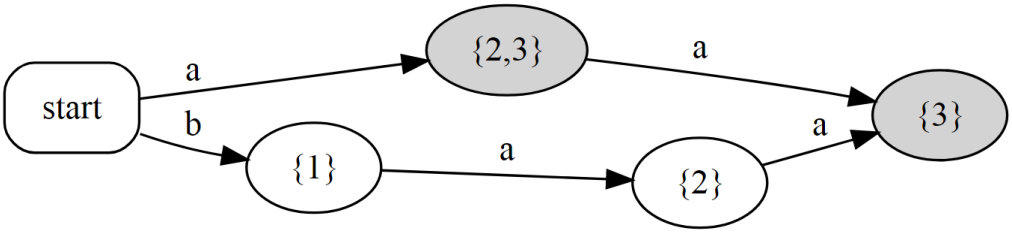
尽管如此，应当指出的是，后缀自动机并不等同于最小化的后缀字典树。

为了区分这两个概念，我们来看一个具体的例子。

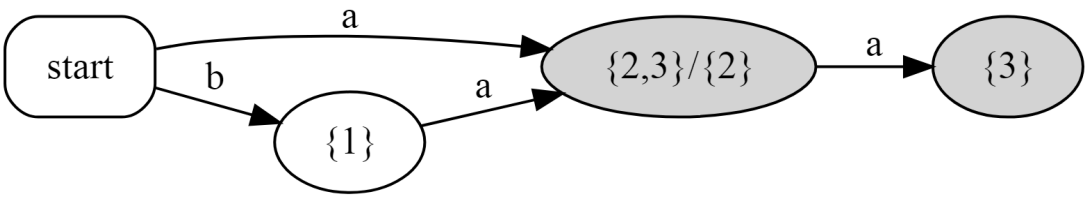
例 3.2.2：字符串 $S = baa$ 的后缀字典树如下图所示：



字符串 $S = baa$ 的后缀自动机如下图所示：



字符串 $S = baa$ 的最小化后缀字典树如下图所示：



可以发现， *Right* 集合分别为 $\{2,3\},\{2\}$ 的两个节点在最小化后缀字典树中被合并为了一个。事实上，由于一个以 $|S|$ 为右端点的出现位置右侧不再具有后续字符，如果两个 *Right* 集合仅仅相差 $|S|$ 这个元素，它们的后续转移是一样的。相反，如果两个 *Right* 集合的对称差包含除了 $|S|$ 以外的元素，它们的后续转移一定是不一样的，因此，在最小化后缀字典树中，这两个集合也会是不同的节点。

由此，我们可以看到，后缀自动机与最小化后缀字典树的唯一差别在于是否将满足对应 *Right* 集合的对称差为 $\{|S|\}$ 的节点合并起来。而如果我们在最小化时，将后缀字典树上代表一个后缀的节点，也即在图中标注为灰色的节点，看做与一般节点不同的节点，后缀自动机就和最小化后缀字典树的定义一致了。

在下文中，我们提及“后缀自动机是后缀字典树最小化的结果”时，其中“最小化”的含义均为在将后缀字典树上代表一个后缀的节点看做与一般节点不同的节点的情况下进行的最小化操作。

引理 3.2：对于 S 的任意两个子串 x,y，以下两者至少有一者成立：

$$Right(x) \cap Right(y) = \varnothing$$

$$Right(x) \subseteq Right(y) \ or \ Right(y) \subseteq Right(x)$$

证明：对于 $Right(x) \cap Right(y) \neq \varnothing$，一定存在 $s \in Right(x) \cap Right(y)$。

由此可知， x,y 均为 $Pre[s]$ 的一个后缀，从而 x,y 中有至少一者是另一者的后缀。

不失一般性地，令 x 为 y 的后缀，则有 $Right(y) \subseteq Right(x)$。

引理 3.2 的存在保证了后缀自动机的节点个数在 $O(|S|)$ 级别。事实上，一个后缀自动机至多具有 $2|S|-1$ 个节点和 $3|S|-4$ 条转移边[4]。正是因为后缀自动机优秀的时空复杂度，其在 OI 竞赛中的应用才能够如此广泛。

3.3 后缀树

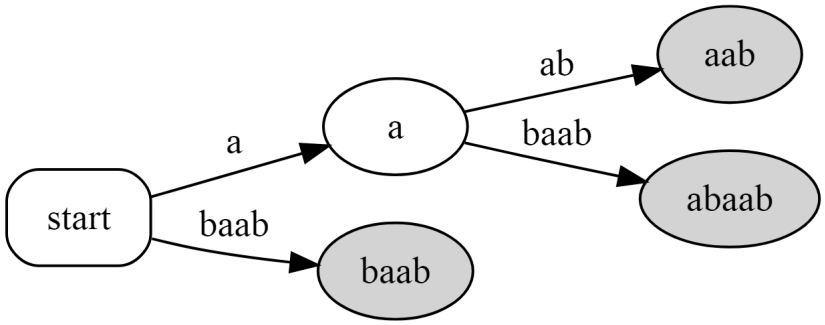
想要降低后缀字典树的时空复杂度，还有着另外一种方式。注意到后缀字典树实际上是由至多 $|S|$ 条从根节点出发的路径并起来的，因此，其叶子节点的个数应当不超过 $|S|$。那么，这些叶子节点形成的**虚树**[5]节点数应当在 $2|S|-1$ 以内。

[4]有关后缀自动机的更多性质，以及线性构造方式，可见参考文献 [3],[4]

[5]一种压缩节点之间链的数据结构，详见参考文献[9]

定义 3.3.1：对后缀字典树的叶子节点建立**虚树**，称得到的结果为**后缀树**。

例 3.3.1：字符串 $S = abaab$ 的后缀树如下图所示：



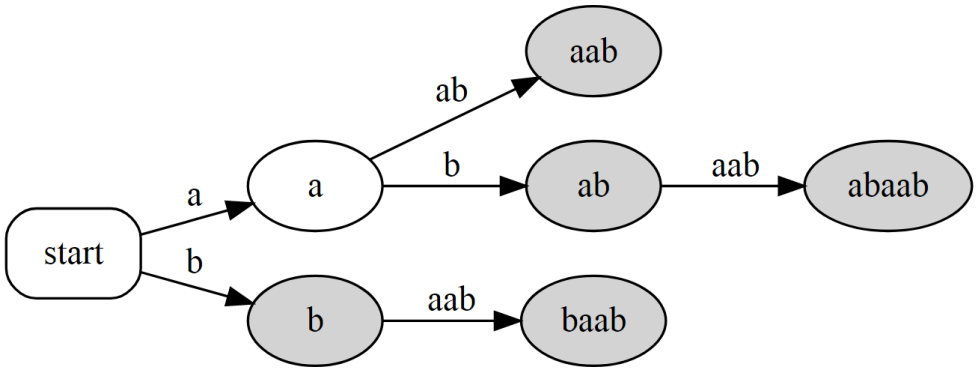
可以发现，建立后缀树的过程实际上是将后缀字典树上所有出度为 1 的节点收缩起来，从而达到减少后缀树节点数的效果。我们称该过程为对后缀字典树的**收缩**。

在例 3.3.1 中，一些原本代表着一个 S 的后缀的灰色节点同样被收缩简化了。而在实际处理问题的时候，我们经常会需要特别地考虑这些代表着一个后缀的节点，为此，我们可以用另一种方式定义后缀树。

定义 3.3.2：

对后缀字典树中代表原串后缀的节点建立**虚树**，称得到的结果为**完整后缀树**。

例 3.3.2：字符串 $S = abaab$ 的完整后缀树如下图所示：



与后缀树相比，完整后缀树将代表原串后缀的节点全部保留了下来。

需要注意的是，由于将一些路径压缩了起来，导致一条边上可能存在多个字符，后缀树实际上不能够算作是一种 DFA 。尽管如此，后缀树依然具有着优秀的线性时空复杂度[6]，以及相比于后缀自动机更加直观的形式。正因如此，后缀树同样是 OI 竞赛中，处理字符串的有力工具。

[6]有关后缀树的更多性质，以及线性构造方式，可见参考文献 [7]

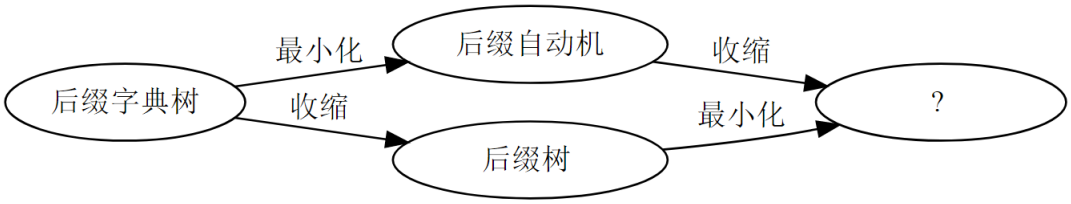
4 压缩后缀自动机

4.1 定义

对于后缀字典树进行**收缩**操作，将会得到后缀树。

对于后缀字典树进行**最小化**操作，将会得到后缀自动机。

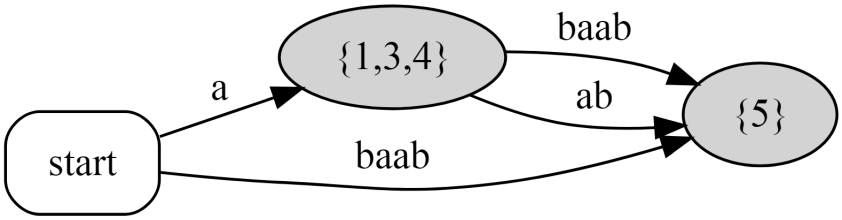
那么，如果同时对后缀字典树进行**收缩**操作和**最小化**操作[7]，将会得到什么呢？



定义 4.1.1：

同时对后缀字典树进行**收缩**操作和**最小化**操作，称得到的结果为**压缩后缀自动机**。

例 4.1.1：字符串 $S = abaab$ 的压缩后缀自动机如下图所示：



对比例 3.2.1 和例 3.3.1，可以发现，字符串 S 的压缩后缀自动机恰好是收缩了后缀自动机上所有出度为 1 的节点的结果，同时，如果在压缩后缀自动机上 DFS，将能够还原出字符串 S 的后缀树。

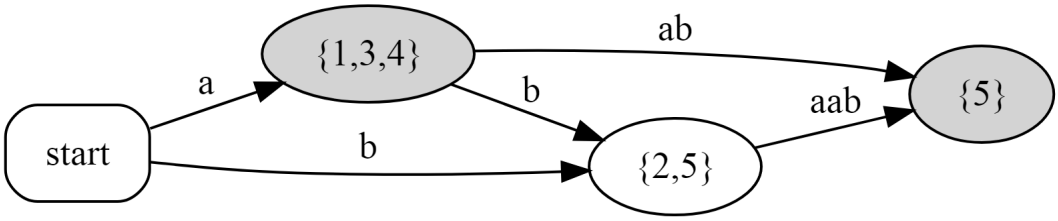
这也恰好符合了压缩后缀自动机同时对后缀字典树进行收缩和最小化操作的定义。

并且，我们也可以定义一种与完整后缀树对应的压缩后缀自动机。

定义 4.1.2：

对完整后缀树进行**最小化**操作，称得到的结果为**完整压缩后缀自动机**。

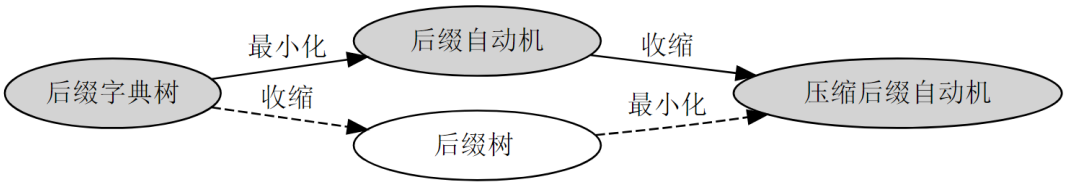
例 4.1.2：字符串 $S = abaab$ 的完整压缩后缀自动机如下图所示：



[7]对于一个 DFA，收缩和最小化操作的先后顺序是不影响结果的

4.2 构造方式

对于没有特殊性质的 DFA，最小化的时间复杂度往往是难以接受的，但是，收缩却很容易。因此，可以对已知的后缀自动机进行收缩操作，从而得到压缩后缀自动机。



例 4.2.1：考虑如何构造字符串 $S = abaab$ 的压缩后缀自动机。

如例 3.2.1 所示，首先构造 S 的后缀自动机，并将出度不为 1 的点标记为灰色。

按照后缀自动机的拓扑序处理未被标记的，出度为 1 的节点，计算出沿着这些节点的出边将会达到的第一个灰色节点。此后，依次处理开始状态以及灰色节点的每条出边，将其简化为连向沿着该出边将会达到的第一个灰色节点的边即可。

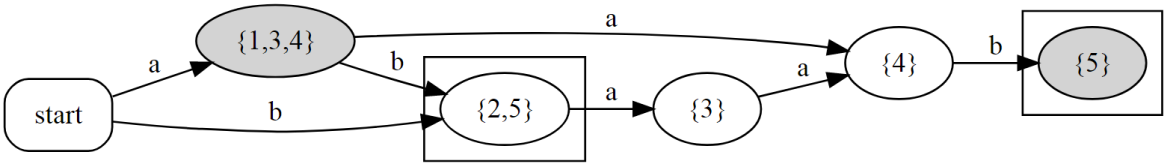
最终得到的结果如例 4.1.1 所示。

由于构造后缀自动机的时空复杂度均为 $O(|S|)$，并且收缩的过程时空复杂度同样为 $O(|S|)$，因此，**构造压缩后缀自动机的时空复杂度均为 $O(|S|)$**。

但是，在完整压缩后缀自动机的定义中，并没有直接与之对应的后缀自动机用来收缩。那么，如何在避免最小化操作的情况下构造字符串的完整压缩后缀自动机呢？

例 4.2.2：考虑如何构造字符串 $S = abaab$ 的完整压缩后缀自动机。

按照定义，在完整压缩后缀自动机上 DFS 应当能够还原出完整后缀树。也就是说，除了后缀自动机上出度不是 1 的节点，我们还应当将一些额外的节点标记为灰色，使得后缀树上代表原串后缀的节点能够被保留下来。



如上图，考虑将后缀自动机上对应 *Right* 集合包含 $|S|$ 的节点同样标记为灰色。

这样的节点在后缀字典树上必定代表了一个原串的后缀，因此，将它们标记为灰色后，再进行例 4.2.1 中的收缩操作，就可以得到完整后缀树最小化的结果，即完整压缩后缀自动机。最终得到的结果如例 4.1.2 所示。

可以发现，**构造完整压缩后缀自动机的时空复杂度同样均为 $O(|S|)$**。

4.3 性质与应用

考虑压缩后缀自动机的定义和构造方式，我们可以得到其下列基本性质：

性质 4.3.1：

压缩后缀自动机是后缀自动机**收缩**的结果。

各个节点在压缩后缀自动机上的所有入边上的字符串是其中最长者的后缀。

性质 4.3.2：

压缩后缀自动机是后缀树**最小化**的结果。

在压缩后缀自动机上 DFS，可以还原出后缀树。

为了更好地发挥压缩后缀自动机的优势，我们还需要证明一个压缩后缀自动机的重要性质。在实际应用压缩后缀自动机解决问题时，这一性质的存在往往尤为关键。

定理 4.3：

令 L_i 表示压缩后缀自动机第 i 个节点的所有入边中最长边的长度，则

$$\sum L_i = O(|S|)$$

证明： 考虑对后缀自动机收缩的过程。

对于后缀自动机上原有的一条边，有如下两种可能：

(1)、该边出现在了灰色节点的最长入边上，此时，其对 $\sum L_i$ 的贡献为 1。

(2)、该边没有出现在灰色节点的最长入边上，此时，其对 $\sum L_i$ 的贡献为 0。

因此，后缀自动机上原有的一条边至多对 $\sum L_i$ 产生 $O(1)$ 的贡献。

由引理 3.2，后缀自动机的边数为 $O(|S|)$ 级别，因此，$\sum L_i = O(|S|)$。

接下来，让我们来看一道具体的问题。

例 4.3 (Sasha and Swag Strings[8])：给定一个仅由小写字母组成的字符串 S $(1 \le |S| \le 10^5)$，求出其后缀树每条边上的字符串本质不同的子串的个数之和。

考虑这个问题的一种传统解法，注意到后缀树每条边上的字符串都是 S 的一个子串，问题可以被转化为对于 $O(|S|)$ 个 S 的子串，分别求出其本质不同的子串的个数。

对于转化后的问题，在参考文献 [5] 中，介绍了一种用 LinkCutTree 维护后缀树，并用线段树维护答案的离线解法。其时间复杂度为 $O(|S|Log^2|S|)$。

但是，这个解法需要应用大量的高级数据结构，实际实现的代码难度较大。同时，其时间复杂度也很高，只是刚好可以通过 10^5 级别的数据范围。接下来，我们将介绍本题的一种应用压缩后缀自动机的，更为优秀的解法。

由性质 4.3.2，我们知道，压缩后缀自动机是后缀树最小化的结果。因此，只需要求出压缩后缀自动机每条边上的字符串本质不同的子串的个数，就可以得到后缀树每条边上的字符串本质不同的子串的个数。

考虑对于各个节点 i，分别求出其所有入边上的字符串本质不同的子串的个数。

[8]题目来源：Petrozavodsk Summer 2015. Moscow IPT Contest

由性质 4.3.1 ，这些字符串都是其中最长的字符串的后缀。

传统的求解字符串本质不同的子串的个数的方式一般是借助后缀树，或是后缀自动机，并且，是能够支持在字符串的一个方向上添加字符，同时动态维护其本质不同的子串的个数的。因此，我们可以在 $O(L_i)$ 的时间内求出点 i 的所有入边上的字符串的本质不同子串的个数。

对于所有节点，求解其入边上的字符串的本质不同子串的个数，总时间复杂度为

$$O(\sum L_i)$$

由定理 4.3 ，有

$$O(\sum L_i) = O(|S|)$$

由此，我们得到了本题的一个时空复杂度均为 $O(|S|)$ 的解法。同时，相比于前文提到的算法，运用压缩后缀自动机的算法更加易于实现。

可见，压缩后缀自动机不仅容易实现，其功能也十分强大。

5 对称压缩后缀自动机

5.1 定义

在例 3.2.1 中，我们使用输入字符串 T 的 *Right* 集合来描述后缀自动机上的节点。

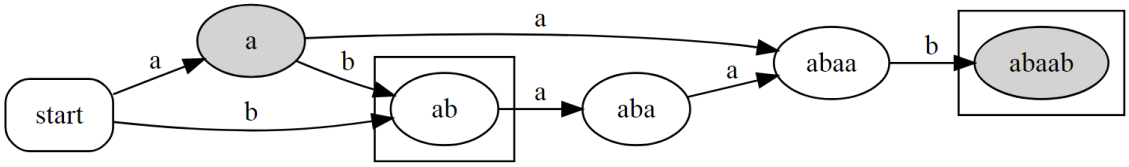
考虑可以被同一节点接受的字符串 T 的集合 $\{T_i\}$ ，令其中最长的字符串为 T_{Max} ，则对于任意 $T \in \{T_i\}$ ， T 应当为 T_{Max} 的后缀[9]。

并且，考虑 *Right* 集合的含义，我们可以进一步得到：

$$\forall T \in \{T_i\}, LeftContext(T) = T_{Max}$$

这意味着，我们可以用一个具体的字符串 T_{Max} 来描述后缀自动机上的节点。

例 5.1.1：字符串 $S = abaab$ 的后缀自动机如下图所示：



由此，我们也可以看到，相比于后缀字典树，后缀自动机保留了满足

$$LeftContext(T) = T$$

的字符串 T 对应的节点。

[9]这同样是性质4.3.1成立的原因

后缀自动机上出度为 1 的，且对应 *Right* 集合不包含 $|S|$ 的节点具有唯一的出边，并且不代表原串的一个后缀。这意味着其对应的字符串 T 满足

$$RightContext(T) \neq T$$

而在完整压缩后缀自动机的构造过程中，这些节点均被收缩了。

这意味着完整压缩后缀自动机上的每一个节点对应的字符串 T 应当同时满足

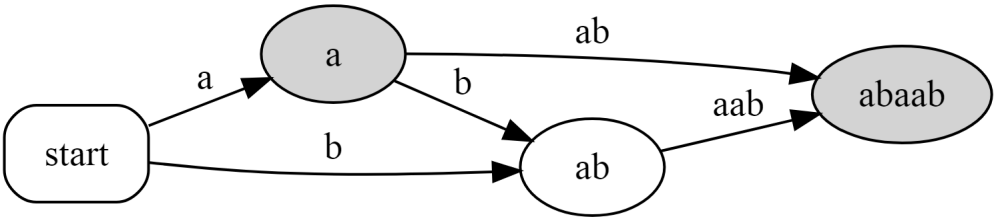
$$LeftContext(T) = T, RightContext(T) = T$$

也即

$$Context(T) = T$$

因此，我们也可以用满足 $Context(T) = T$ 的字符串 T 来描述一个节点。

例 5.1.2：字符串 $S = abaab$ 的完整压缩后缀自动机如下图所示：

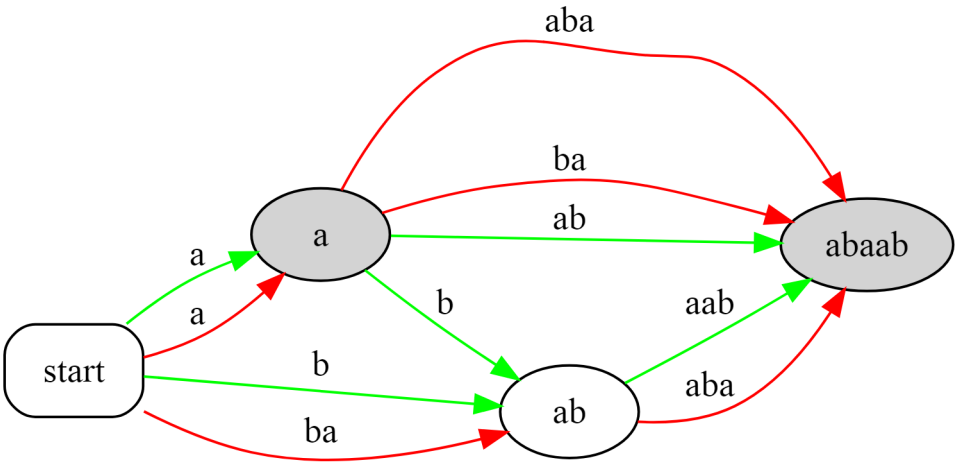


注意到满足 $Context(T) = T$ 的字符串 T 和在 S 的反串 S' 中满足 $Context(T') = T'$ 的字符串 T' 是一一对应的，我们可以同时对 S 的正反串建立完整压缩后缀自动机，并将对应的节点合并。

定义 5.1：

同时对字符串的正反串建立完整压缩后缀自动机，将对应的节点合并，并同时保留两组转移边，称得到的结果为**对称压缩后缀自动机**。

例 5.1.3：字符串 $S = abaab$ 的对称压缩后缀自动机如下图所示：



上图中，绿色的转移边是原串的转移边，代表向当前串的后侧添加字符，而红色的转移边是反串的转移边，代表向当前串的前侧添加字符。

5.2 构造方式

按照定义 5.1 ，我们可以通过对字符串的正反串建立完整压缩后缀自动机，再将对应的节点合并的方式来构造对称压缩后缀自动机。

那么，我们需要一个能够快速找到互为反串的子串的结构。

可以考虑使用字符串哈希[10]或是后缀数组[11]作为所需要的子串查询结构。

依照选择的子串查询结构的不同，构造对称压缩后缀自动机的时空复杂度为 $O(|S|)$ 或 $O(|S|Log|S|)$ 。

5.3 性质与应用

除了在 4.3 中提到的，压缩后缀自动机具有的性质以外，对称压缩后缀自动机还具有着**可以同时向两侧添加字符**的强大功能。接下来，让我们通过具体的例子来领略一下对称压缩后缀自动机的力量。

例 5.3.1 (子串查找[12])：给定长度为 N $(1 \le N \le 10^6)$ 的仅由小写字母组成的字符串 S ，字符串 T 初始为空串。要求在线地支持如下四种操作共 Q $(1 \le Q \le 10^6)$ 次：

操作 (1) ：给定字符 c ，令 $T = T + c$

操作 (2) ：给定字符 c ，令 $T = c + T$

操作 (3) ：给定整数 i ，将 T 赋值为第 i 次操作后得到的字符串

操作 (4) ：判断 T 是否为 S 的子串，如果是，求出其任意一个出现位置

对于没有操作 (2) 的问题，我们显然可以利用传统的后缀自动机来解决。

本题同时出现了向输入串的两端添加字符的操作，是传统的后缀自动机不能解决的。因此，考虑对字符串 S 建立对称压缩后缀自动机。考虑用 $Context(T)$ 对应的节点来表示字符串 T 。同时，维护 T 在 $Context(T)$ 中的出现位置，注意由于上下文的性质， T 在 $Context(T)$ 中的出现位置是唯一的。

对于操作 (1) ：

若 $RightContext(T) \neq T$ ，则 T 在 S 中出现时，

其后方一定会出现一个特定的字符，判断 c 是否为该字符。

若是，则 $Context(T + c) = Context(T)$ ，否则， $T + c$ 不是 S 的子串；

若 $RightContext(T) = T$ ，则 $Context(T + c) \neq Context(T)$ 。

找到在正向压缩后缀自动机中以字符 c 开头的出边。

[10]一种基于哈希思想判断字符串是否相等的算法，详见参考文献[10]

[11]一种广泛应用的处理字符串的数据结构，详见参考文献[6]

[12]题目来源：原创

若存在，则 $Context(T+c)$ 即为到达的节点，否则， $T+c$ 不是 S 的子串。

类似地，对于操作 (2)：

若 $LeftContext(T) \neq T$，则 T 在 S 中出现时，

其前方一定会出现一个特定的字符，判断 c 是否为该字符。

若是，则 $Context(c+T) = Context(T)$，否则， $c+T$ 不是 S 的子串；

若 $LeftContext(T) = T$，则 $Context(c+T) \neq Context(T)$。

找到在反向压缩后缀自动机中以字符 c 开头的出边。

若存在，则 $Context(c+T)$ 即为到达的节点，否则， $c+T$ 不是 S 的子串。

考虑操作 (3)，由于对于一个状态，我们仅仅需要维护 $Context(T)$ 对应的节点，以及 T 在 $Context(T)$ 中的出现位置，因此，只需要用数组记录下每次操作后的这些信息，就可以支持操作 (3)。

最后，对于操作 (4)，我们已经维护出了 $Context(T)$，以及 T 在 $Context(T)$ 中的出现位置。因此，判断 T 是否为 S 的子串即为判断是否存在 $Context(T)$ 对应的节点，而求出 $Context(T)$ 的任意一个出现位置，即可求出 T 的一个出现位置。

对于对称压缩后缀自动机上的每一个节点，我们可以在构造的时候预处理出其在 S 中的出现位置，从而支持操作 (4)。该解法的时间复杂度为 $O(N+Q)$，空间复杂度为 $O(N)$。可见对称压缩后缀自动机在处理双向添加字符类型的问题时的优势所在。

例 5.3.2 (Alice and Bob and A String[13])：

给定一个仅由小写字母组成的字符串 S $(1 \leq |S| \leq 10^5)$。

Alice 和 Bob 正在玩一个游戏，初始时，他们有一个 S 的子串 T。由 Alice 先手，两人轮流进行如下操作：在 T 的前后各添加一个字符，保证得到的结果仍然是 S 的一个子串。无法操作的玩家负。

假设 Alice 和 Bob 均采取使自己获胜的最优策略，则对于所有 S 的子串，请帮助 Alice 计算出，其中使她必胜的，字典序第 k $(1 \leq k \leq 10^{10})$ 小的 T，或者判断不存在 k 个使得 Alice 必胜的 T。

首先，考虑如何对一个给定的字符串 T，判断 Alice 是否能够取胜。

对于这个问题，最直观的想法应当是从长到短考虑每一个 S 的子串 T，记 dp_T 表示子串 T 是否为先手必胜态，如果 T 可以达到一个先手必败态，则 $dp_T = true$，否则，即 T 可以达到的状态均为先手必胜态， $dp_T = false$。

然而，该做法需要考虑 S 的所有子串，在 $|S|$ 达到 10^5 级别时是无法接受的。

考虑一个同时满足 $LeftContext(T) \neq T, RightContext(T) \neq T$ 的字符串 T。每当 T 在 S 中出现时，其前后必然会出现固定的字符，因此，对于当前串为 T 的状态，先手玩家只有唯一的选择。那么，我们可以按照唯一的选择推进游戏，直到 $LeftContext(T) = T, RightContext(T) = T$ 中的至少一者成立。

由定理 4.3，满足以上至少一者的字符串数量在 $O(|S|)$ 级别。

[13]题目来源：Petrozavodsk Summer 2018. Moscow IPT Contest

由此，建立对称压缩后缀自动机后，只需要在这样的字符串上进行上述DP即可。

接下来，考虑如何求出字典序第 k 小的先手必胜态。

由性质 4.3.2 ，若在压缩后缀自动机上 DFS ，我们可以还原出原串的后缀树。而后缀树恰好将 S 的所有子串按照字典序排好了序。因此，如果我们能够快速求出一条边上的必胜态总数，便只需要在压缩后缀自动机上 DFS ，就可以确定所求的答案在哪一条边上，从而确定答案。

由转移规则，可以发现，一条边上的必胜态总数可以通过对于节点 X ，维护满足 $Context(T) = LeftContext(T) = X$ ，或者 $Context(T) = RightContext(T) = X$ 的字符串 T 胜负情况的，奇偶长度分别的前缀和而快速计算。

由此，我们得到了问题的一个时间复杂度为 $O(\sigma \times |S|)$ 的解法，其中 $\sigma = 26$ 。

6 总结

本文从一系列传统的信息学竞赛中的后缀数据结构出发，自然地引出了一种新型后缀数据结构：压缩后缀自动机，以及它的一个变体：对称压缩后缀自动机。对于两者，本文均详细地分析了它们的性质，并且给出了一种，在允许离线的情况下，时空复杂度均为线性的构造算法，同时，也介绍了它们的实际应用。

在本文中，笔者利用大量的配图进行辅助讲解，语言也通俗易懂。笔者认为，即使是字符串基础较差的选手，阅读本文后，也能够有一定的收获。

值得一提的是，本文所介绍的压缩后缀自动机，实际上就是对大家熟知的后缀树、后缀自动机进一步考虑后的自然结果。这充分说明了，许多时候，解决新问题的方式往往藏于已有的工具中，只要我们勇于创新，积极思考，便能够将它们发现。

笔者注意到，在当今的信息学竞赛界，同学们对后缀自动机的研究热情依旧很高，希望本文引入的压缩后缀自动机，及其背后的思考方式，能够起到抛砖引玉的作用，激发更多同学对后缀数据结构，乃至更广泛的问题的思考与探究。

感谢

感谢中国计算机学会提供学习和交流的平台。

感谢江苏省常州高级中学的曹文老师，吴涛老师多年来给予我的关心和指导。

感谢郭晓旭前辈对本文做出的帮助与指导。

感谢集训队教练高闻远对我的帮助与指导。

感谢王修涵同学、杨俊昭同学和我校的杜伟桦同学为本文审稿。

感谢家人、朋友对我的支持与鼓励。

感谢帮助过我的老师、同学们。

参考文献

[1] 维基百科：确定有限状态自动机
https://zh.wikipedia.org/zh-hans/确定有限状态自动机

[2] 维基百科：确定有限状态自动机最小化
https://zh.wikipedia.org/zh-hans/确定有限状态自动机最小化

[3] 陈立杰《后缀自动机》2012 年 NOI WC 讲课课件

[4] 刘研绎《后缀自动机在字典树上的拓展》2015 国家集训队论文集

[5] 陈江伦《后缀树结点数 命题报告及一类区间问题的优化》2018 国家集训队论文集

[6] 罗穗骞《后缀数组——处理字符串的有力工具》 2009 国家集训队论文集

[7] 东南大学出版社《高级数据结构》

[8] 李煜东《算法竞赛进阶指南》

[9] OI Wiki《虚树》 https://oi-wiki.org/graph/virtual-tree/

[10] OI Wiki《字符串哈希》 https://oi-wiki.org/string/hash/