后缀树的构建

广州市第二中学 代晨昕

摘 要

后缀树是处理字符串匹配问题的利器。本文主要介绍它的两种在线构建方法，并给出了一些应用。

1 前言

后缀数据结构，如后缀自动机和后缀树，在若干年以前进入了国内信息学竞赛选手们的视野，现在已经有了广泛的应用，算法竞赛界也已有许多介绍它们的文献。本文主要对后缀树建树方面作出了一些补充，介绍了后缀树的一些在线构建方法。

本文第二、三节主要讲述关于字符串和后缀树的一些基础定义。第四节介绍从后往前构造后缀树的方法。第五节介绍从前往后构造后缀树的方法。第六节介绍了一些例题。第七节总结全文。

2 基础定义

记构建后缀树的母串为 S，长度为 n。

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

令 $S[l,r]$ 表示 S 中第 l 个字符至第 r 个字符组成的字符串，称为 S 的一个子串。

记 $S[i,n]$ 为 S 的以 i 开头的后缀，$S[1,i]$ 为 S 的以 i 结尾的前缀。

若 str_1，str_2 为单一字符或字符串，令 $str_1 + str_2$ 表示将 str_1 和 str_2 拼接后得到的字符串。

如无特殊说明，本文默认字符串 S 的字符集为所有小写字母，即字符集大小为常数。

3 后缀树的定义

定义字符串 S 的**后缀 trie** 为将 S 的所有后缀插入至 trie 树中得到的字典树。在后缀 trie 中，节点 x 对应的字符串为从根节点走到 x 的路径上经过的字符拼接而成的字符串。记后

缀 trie 中所有对应 S 的某个后缀的节点为后缀节点。

容易看出后缀 trie 的优越性质：它的非根节点恰好能接受 S 的所有本质不同非空子串。但构建后缀 trie 的时空复杂度均为 $O(n^2)$，在很多情况下不能接受，所以我们引入后缀树的概念。

如果令后缀 trie 中所有拥有多于一个儿子的节点和后缀节点为关键点，定义只保留关键点，将非关键点形成的链压缩成一条边形成的压缩 trie 树为**后缀树 (Suffix Tree)**。

如果仅令后缀 trie 中所有拥有多于一个儿子的节点和叶结点为关键点，定义只保留关键点形成的压缩 trie 树为**隐式后缀树 (Implicit Suffix Tree)**。容易看出隐式后缀树为后缀树进一步压缩后得到的结果。

在后缀树和隐式后缀树中，每条边对应一个字符串；每个非根节点 x 对应了一个字符串集合，为从根节点走到 x 的父亲节点 fa_x 经过的字符串，拼接上 fa_x 至 x 的树边对应的字符串的任意一个非空前缀。

令 mx_x 表示 x 对应的字符串集合中最长的字符串，len_x 表示 mx_x 的长度。

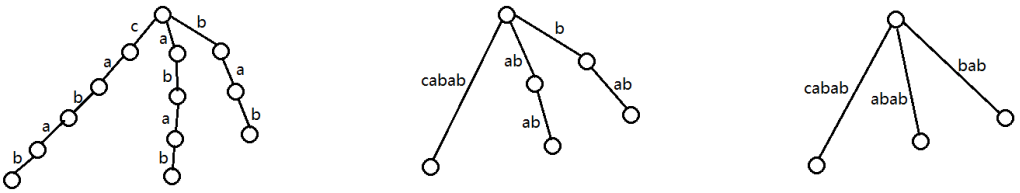

图 1: 字符串”cabab” 对应的后缀 trie、后缀树和隐式后缀树

引理 3.1：长度为 n 的字符串 S 构建的后缀树节点数不超过 $2n$。

证明. 考虑将 S 的后缀逐个插入至后缀 trie 中。从第二次插入开始，每次最多新增一个拥有多于一个儿子的节点和一个后缀节点，所以后缀 trie 中关键点个数最多为 $2n$ 个，得证。

4 支持从后往前添加字符的后缀树构建

一个广为人知的结论是：后缀自动机的 *parent* 数组即为反串的后缀树 [1]，利用反串的后缀自动机来构建后缀树也成为了现在构建后缀树的主流算法。接下来将从后缀树的角度来描述此算法的过程。

4.1 一些定义

在后缀 trie 中，记 $trans_{x,c}$ 表示在节点 x 对应的字符串开头添加字符 c 后对应的节点，如果不存在设为空状态 $NULL$。

对 S 的某子串 str，令 $leftpos_{str}$ 表示 str 在 S 中出现位置的左端点集合。

由后缀树的定义推出：在后缀树上每个节点 x 对应的字符串集合的 $leftpos$ 集合相同，可以记为 $leftpos_x$，且不同的两个节点的 $leftpos$ 集合不相同。易得节点 x 对应的所有字符串开头添加字符 c 后得到的字符串的 $leftpos$ 集合都相同，在后缀树中一定对应同一节点 y，可以直接定义 $trans_{x,c} = y$。实际上，这就是反串的后缀自动机的转移边。

4.2 算法过程

考虑增量法，从后往前加入字符。假设已经维护好 $S[i+1,n]$ 的后缀树以及所有节点的 $trans$ 转移边，现在需要在后缀树中加入后缀 i。设 $S[i] = x$。

记录后缀 $i+1$ 对应的节点 $last$，新增后缀节点 np 表示 $S[i,n]$。现在，我们希望求出 $S[i,n]$ 在原树中出现过的最长前缀。由于 $S[i,n]$ 的前缀即为 $S[i+1,n]$ 的前缀在前端添加字符 x，所以找出 $last$ 的深度最深的祖先 p 使得 $trans_{p,x}$ 存在，那么 $x + mx_p$ 就是 $S[i,n]$ 在原树中出现过的最长的前缀。令 $q = trans_{p,x}$，有两种情况：

1.$mx_q = x + mx_p$，即 $len_q = len_p + 1$。此时在后缀树中会新增一条 np 至 q 的边，将 fa_{np} 设为 q 即可。

2.$len_q > len_p + 1$。此时在 q 与 fa_q 之间需要新建节点 nq，令 $len_{nq} = len_p + 1$，并将 fa_{nq} 设为 fa_q，fa_{np} 和 fa_q 设为 nq。同时，我们还需要维护 $trans$ 的变化。nq 的转移边可以直接复制 q 的转移边得到。同时，$trans_{p,x}$ 需要设为 nq；如果 $trans_{fa_p,x} = q$，也需要重新设为 nq ……从 p 点不断往根移动，不断修改转移边，直到发现当前节点 p_1 添加字符 x 不会转移到 q 为止。

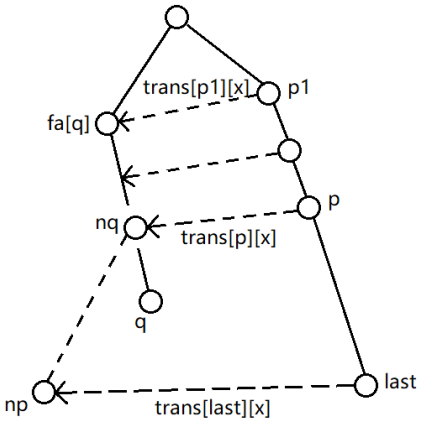

图 2: 情况 2 示意图

最后，从 $last$ 到 p 之间的节点也需要新增到 np 的转移，再将 $last$ 设为 np 即可。

4.3 算法时间复杂度证明

算法的正确性是显然的，但是时间复杂度仍有疑问。

引理 4.1：所有 *trans* 转移边的个数为 $O(n)$ 级别（事实上，也与字符集大小无关）。

即证反串后缀自动机转移边数量为 $O(n)$ 级别。证明可见 [2]，这里不再赘述。

考虑算法过程，除了情况 2 中“将连向 q 的转移边重新设为 nq”，其余步骤都伴随着新增节点或转移边，所以只需考虑这一步的时间复杂度即可。

设节点 x 的深度为 Dep_x。考虑 $Dep_{fa_{nq}}$ 和 Dep_{p_1}。由于 fa_{nq} 的任何祖先（不包括 *root*）一定可以由某个 p_1 的祖先（包括 *root*）通过前端添加字符 x 转移而来，所以有 $Dep_{fa_{nq}} \le Dep_{p_1}+1$。设转移边重定向进行了 k 次，那么有：

$$Dep_{np} = Dep_{fa_{nq}} + 2 \le Dep_{p_1} + 3 = Dep_p - k + 3 \le Dep_{last} - k + 3$$

将每一步操作后的 Dep_{last} 设为势能函数。每次转移边重定向都伴随着 Dep 的减小，且 Dep 每次只会增加常数，通过势能分析可以得出这一步操作的时间复杂度是每次均摊 $O(1)$ 的。

综上所述，此算法构建后缀树的时间复杂度为 $O(n)$。

4.4 算法总结

当字符集大小视为常数时，本算法达到了 $O(n)$ 的时间复杂度下界，同时实现不算复杂，是离线构建后缀树的首选。然而，本算法只能支持在前端动态插入字符构造后缀树，在很多问题中，我们需要支持动态末端插入的构造方法，在下一节中将介绍另一种支持动态末端插入的后缀树构建算法——Ukkonen 算法 [3]。

5 支持从前往后添加字符的后缀树构建

Ukkonen 算法维护了 S 的隐式后缀树，并支持在末端插入字符。

称 S 的隐式后缀树为 $STree(S)$。

在 $STree(S)$ 中，每个叶结点对应了 S 的一个后缀。称在 $STree(S)$ 中不作为叶结点出现的后缀为 S 的隐式后缀，其余后缀为显式后缀。考虑隐式后缀具有什么样的性质。

引理 5.1：若 $S[i,n]$ 为 S 的隐式后缀，则对于 $\forall j > i$，$S[j,n]$ 也为 S 的隐式后缀。

证明. 由 $S[i,n]$ 为隐式后缀可知，存在字符 c 使得 $S[i,n]+c$ 为 S 的子串，所以 $S[j,n]+c$ 也为 S 的子串，由隐式后缀树的定义可知 $s[j,n]$ 也不作为叶结点出现。证毕。

所以，S 的隐式后缀一定为 S 的后缀中最短的那些。在算法流程中，我们同时维护 S 的最长隐式后缀 $S[k,n]$ 在树中的位置，用二元组 $(active, remain)$ 表示，意为：$S[k,n]$ 在树中所属节点的父亲为 $active$，位置为从 $active$ 往下走 $remain$ 步。

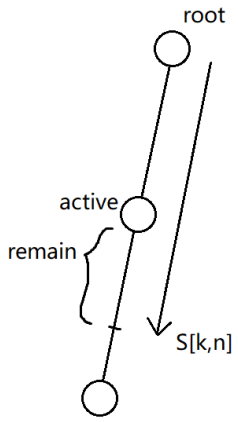

图 3: $(active, remain)$ 示意图

由于我们已经知道剩下的 $remain$ 个字符为 $S[n-remain+1,n]$，所以我们不需刻意记录 $S[k,n]$ 位于 $active$ 的哪一条出边上，只需 $remain$ 即可确定位置。

为了完成本算法，我们还需引入一个概念：后缀链接。

5.1 后缀链接

引理 5.2：对隐式后缀树中任意非叶非根节点 x，在树中存在另一非叶节点 y，使得 mx_y 为 mx_x 删去开头的字符。

证明. 令 str 表示 mx_x 删去开头字符形成的字符串。由隐式后缀树的定义可知，存在两个不同的字符 c_1, c_2，满足 $mx_x + c_1$ 与 $mx_x + c_2$ 均为 S 的子串。所以，$str + c_1$ 与 $str + c_2$ 也为 S 的子串，所以 str 在后缀 trie 中也对应了一个有分叉的关键点，即在 $STree$ 中存在 y 使得 mx_y=str。证毕。

由引理 5.2，我们可以对 $STree$ 中的所有非根非叶节点 x，定义 $Link_x = y$，$Link_x$ 称为 x 的**后缀链接 (Suffix Link)**。

5.2 算法流程

为了构建 $STree(S)$，我们从前往后加入 S 中的字符。假设当前已经建出 $STree(S[1,m])$ 且维护好了后缀链接。$S[1,m]$ 的最长隐式后缀为 $S[k,m]$，在树中的位置为 $(active, remain)$。设 $S[m+1]=x$, 现在我们需要加入字符 x。

此时，$S[1,m]$ 的每一个后缀都需要在末尾添加字符 x。由于所有显式后缀都对应树中某个叶结点，它们只会让叶结点对应父边的长度增加 1，而不会改变树的形态，这很方便我们记录。所以，现在我们只用考虑隐式后缀末尾添加 x 对树的形态产生的影响。

首先考虑 $S[k,m]$，有两种情况：

1、$(active, remain)$ 位置不需要新增出边 x，即后缀树形态不会发生变化。由于 $S[k,m+1]$ 已经在后缀树中出现，所以对于 $j>k$，$S[j,m+1]$ 也会在后缀树中出现，此时只需将 $remain+1$，不需做任何修改。

2、$(active, remain)$ 需要新增出边 x。此时视情况可能需要对节点进行分裂。这时，对于 $j>k$，我们并不知道 $S[j,m]$ 会对后缀树形态造成什么影响，所以我们还需继续考虑 $S[k+1,m]$。考虑怎么求出 $S[k+1,m]$ 在后缀树中的位置：如果 $active$ 不为 $root$，可以利用后缀链接，令 $active = Link_{active}$；否则，令 $remain-1$ 即可。最后令 $k+1$，再次重复这个过程。

上述过程可以用 while 循环来完成，同时分裂节点时还需注意维护新增节点的后缀链接，下面给出完整修改过程：

维护最后一次新增出边的节点 $last$，初始设为 $NULL$。在循环过程中保证只有 $last$ 可能还未确定后缀链接。

按长度从长到短依次迭代当前未考虑的最长隐式后缀 $S[k,m]$，分两类讨论：

1、$S[k,m]+x$ 仍然出现在 $STree$ 中。此时，如果 $last$ 存在，由引理，$(active, remain)$ 位置必定恰好为 $STree$ 中某个非叶节点，将 $Link_{last}$ 设为当前节点并将 $remain+1$，即可退出迭代。

2、$S[k,m]+x$ 不出现在 $STree$ 中。此时可能需要在 $(active, remain)$ 位置分裂节点。记此时 $(active, remain)$ 对应的节点为 p。新增叶子结点 q，并添加一条从 p 连向 q 的边。显然，如果 $last$ 存在，那么 $Link_{last}=p$，再将 $last$ 设为 p 即可。

在每次迭代的最后，如果 $active$ 不为 $root$ 则将 $active$ 设为 $Link_{active}$，否则将 $remain-1$。

此时 $remain$ 可能超出了 $active$ 的对应出边的长度，所以需要在 $STree$ 中从 $active$ 往下搜索，直到 $remain$ 不超出范围为止。

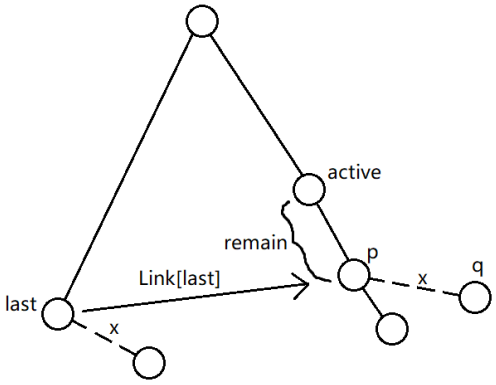

图 4: 过程示意图

至此，我们成功维护出了 $STree$ 形态的变化、所有后缀链接的变化和最长隐式后缀的变化。

5.3 算法时间复杂度证明

循环的每次运行都伴随着最长隐式后缀长度的减小，而最长隐式后缀长度只会增加 n 次，所以循环只会运行 $O(n)$ 次。

当 *remain* 超出 *active* 对应出边的长度时，我们会在 $STree$ 中往下搜索，但是每次移动都伴随着 *remain* 的减小，而 *remain* 总共只会增加 n 次，所以这一移动的时间复杂度为均摊 $O(1)$。

综上所述，算法的时间复杂度为 $O(n)$。

5.4 算法总结

由于 Ukkonen 算法只能处理出 S 的隐式后缀树，而隐式后缀树在一些问题中的功能可能不如后缀树强大，所以在需要时，可以在 S 的末端添加一个从未出现过的字符 #，这时 S 的所有后缀可以和 $STree$ 的所有叶子一一对应。

Ukkonen 算法主要利用了后缀链接来优化每次寻找隐式后缀的过程，而且与上一节的算法相反，支持在字符串末端动态插入字符，时间复杂度也达到了 $O(n)$ 的下界。下面简单介绍上述算法的一些应用。

6 例题

6.1 例题一：SWERC 2015 Text Processor[1]

6.1.1 题目大意

给定一个长度为 n 的小写字母字符串 S，再给定 $1 \le w \le n$，对于所有 $i \in [1, n-w+1]$，求出 $S[i, i+w-1]$ 有多少本质不同字串。$n \le 10^5$。

6.1.2 题目分析

求区间本质不同子串个数是一个经典问题，可以利用后缀自动机构造后缀树的方法和动态树得到时间复杂度为 $O(nlog^2n)$ 的做法 [4]，实际上本题可以做到线性复杂度。

由于本质不同子串个数即为后缀 trie 节点数，如果我们对每个区间 $[i, i+w-1]$ 建出后缀树或隐式后缀树，求出边长总和即为答案。考虑利用 Ukkonen 算法维护隐式后缀树。由于本题区间长度固定，可以考虑采用滑动窗口，每次在末尾添加一个字符，或在开头删掉一个字符，维护当前隐式后缀树的边长总和。

在开头删掉一个字符即为删掉当前深度最深的叶子，可以用队列维护当前所有叶子，快速找到当前需要删除的节点 x。删去 x 以后，可能会使某个隐式后缀变为显式后缀，也可能会使 fa_x 的儿子节点个数减少至一个，需要压缩掉 fa_x 节点。在末尾添加字符时沿用 Ukkonen 算法。仔细维护当前 $(active, remain)$ 的位置和边长总和的变化即可，时间复杂度为 $O(n)$。

6.2 例题二：北大集训 2018 Day3 close

6.2.1 题目大意

维护一个字符串 S，支持双端插入，要求在线维护 S 的本质不同子串个数。

6.2.2 题目分析

令 n 为字符串 S 最后的总长。

出题人给出了利用平衡树维护后缀的时间复杂度为 $O(nlog^2n)$ 的做法 [5]，实际上本题同样可以优化至 $O(n)$[6]，下面简单介绍做法。

上文已经给出利用 *trans* 转移边和后缀链接 *Link* 维护插入的两种算法，容易想到同时维护 *trans* 和 *Link*，将两种算法结合。

[1] https://codeforces.com/gym/101128

由于末端插入时，S 的隐式后缀对后缀树的形态影响是难以预料的，所以仍然考虑维护 S 的隐式后缀树 $STree(S)$。首先需要解决的问题是：在隐式后缀树上，我们仍然可以类似的定义 *trans* 吗？

设 S 的最长隐式后缀为 $S[k,n]$。对 S 的一个子串 *str*，定义 $leftpos'_{str}$ 表示 $leftpos_{str}$ 在 $[1,k-1]$ 中的位置。类似后缀树，在 $STree$ 中有：节点 x 代表的字符串集合的 $leftpos'$ 相同，可以记为 $leftpos'_x$，且不同节点的 $leftpos'$ 集合不相同。x 代表的所有字符串 *str* 开头添加字符 c 后的 $leftpos'$ 集合相同的一个充分条件是：所有 *str* 的 $leftpos$ 集合在 $[1,k]$ 中出现的位置相同。所以，$trans_{x,c}$ 在 $STree$ 中几乎是良定义的，只有一种特殊情况：某些 $leftpos_{str}$ 中出现了位置 k，而某些不出现位置 k。这种节点在 $STree$ 中至多只有一个：$(active, remain)$ 所在的节点。

所以，在 $STree$ 中，我们仍然可以类似地定义 *trans* 转移边：如果 x 代表的所有 *str* 在开头添加字符 c 后对应同一节点 y，那么令 $trans_{x,c}=y$，否则令 $trans_{x,c}=NULL$。只有在 $(active, remain)$ 所处的树边上需要特判：在这条边上，位于 $(active, remain)$ 上方的位置在前端添加字符 $S[k-1]$ 后，会转移到 $S[k-1,n]$ 对应节点。

现在，可以开始考虑算法流程。

6.2.3 算法流程

首先考虑在前端添加字符。沿用 4.2 中的算法流程，注意特判 $(active, remain)$ 位置的转移边即可。如果此时新增了分裂节点 nq，我们还需维护 $Link_{nq}$，显然 $Link_{nq}=p$。同时，可能 q 节点代表的显式后缀变为了隐式后缀，需要将连向 q 的转移边重定向为 nq。

然后考虑在末端插入字符。沿用 5.2 中的算法流程。此时我们还需维护新增的叶节点和分裂节点的转移边。

对于新增的叶子结点 q，会有到上一个加入的叶子结点 q' 的转移。同时，这个叶子结点代表了一个当前只出现了一次的后缀，它显然也只会有这一个转移。

对于新增的分裂节点，可以复制它的孩子的转移。

同时，对于上一个新增的分裂节点 *last*，可能会需要将一些转移边修改为它。可以发现，影响的所有节点恰好为从 $Link_{active'}$ 在 $STree$ 中寻找合适的 *active* 的位置往下搜索时遍历到的所有节点 ($active'$ 为上一次 *active* 所处的位置)。

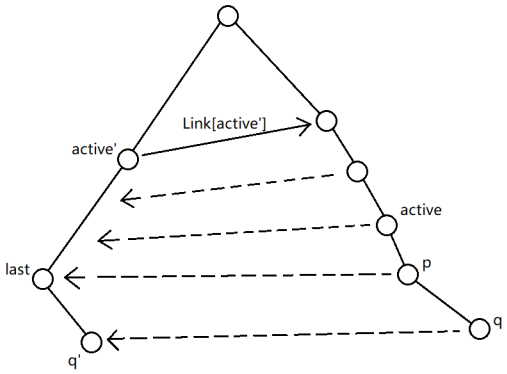

图 5: 虚线为需要修改的转移边

至此，我们完成了整个算法流程。本算法涉及到的操作均在前两种算法中出现过，将两者的时间复杂度证明结合，即可得出本算法的时间复杂度为 $O(n)$。

7 总结

在本文中，我们主要介绍了后缀树的动态前端插入、末端插入，并在例题中给出了将两种算法结合的支持动态双端插入的做法。

希望本文能激发读者的思考，让读者对后缀树和它的优美性质有更加深刻的理解，并带领读者领略到后缀树的魅力。

8 致谢

感谢中国计算机学会提供交流和学习的平台。
感谢国家级集训队高闻远教练的指导。
感谢林盛华老师、陈旭龙老师对我的支持与指导。
感谢赵雨扬前辈为本文提供思路。
感谢郑钧天学长、罗恺同学为本文验稿。
感谢马耀华同学为本文提供的帮助。
感谢给予过我帮助的老师与同学们。

参考文献

[1] 范浩强，《后缀自动机与线性构造后缀树》

[2] 陈立杰，《后缀自动机》，2012 冬令营营员交流。

[3] E. Ukkonen, Constructing suffix trees on‐line in linear time, in Algorithms,Software, Architecture. Information Processing 92, vol. I (J. van Leeuwen, ed.), Elsevier, 1992, pp. 484‐492.

[4] 陈江伦，《后缀树结点数》命题报告及一类区间问题的优化，IOI2018 中国国家候选队论文集

[5] https://zhuanlan.zhihu.com/p/51880239

[6] https://negiizhao.blog.uoj.ac/blog/4756