回文树双端修改问题

李响
南京外国语学校，江苏 南京 210022
DOI: 10.61369/TACS.2025060006

摘　　要： 在回文树已有的单侧修改算法的基础上，本文独立提出了回文树双端修改算法，解决了在字符串两端插入删除字符并维护回文树的问题。此外，本文讨论了广义回文树与持久化回文树的拓展，并给出了它在字符串问题上的应用。
关 键 词： 数据结构；字符串算法；回文树

Modification to Both Ends of EERTREE

Li Xiang
Nanjing Foreign Language School, Nanjing, Jiangsu 210022

Abstract: Based on the current algorithm to cope with single-ended modification of EERTREE, this paper independently proposes the double-ended modification algorithm, solving the problem of maintaining the EERTREE while performing insertion and deletion to both ends of a string. Moreover, this paper discusses advanced modifications of EERTREE, such as joint EERTREE and persistent EERTREE, and shows the application of the algorithm to some string problems.
Keywords: data structure; string algorithms; EERTREE

引言

回文问题有着长久的研究历史。Manacher 在1975年提出了 Manacher 算法[1]，统计本质不同回文子串的问题在2010年来受到关注[2,3]，算法竞赛界也逐渐形成了一套研究回文问题的方法论[4]。回文树作为处理回文相关问题的一个强大而便利的工具，自从2015年被提出[5]以来，在算法竞赛中逐渐流行[6]，并延伸出若干拓展[7]。关于在字符串两端插入删除字符并维护回文树的问题，本文基于回文子串的几个性质，提出了一个时空复杂度 $O(n\log|\Sigma|)$ 的在线非均摊算法，给出了算法的实现，并讨论了该算法的拓展和应用。

一、符号约定与回文树的定义

本文用 Σ 表示字符串的字符集。

s 的回文树描述了 s 所有本质不同的回文子串的结构。一个字符串 s 的回文树由两棵树组成，除两棵树的根节点外每个节点表示 s 的一个本质不同的回文子串。树上每条边有一个字符。对于节点 x,y，记它们代表的回文子串分别为 a,b，若 y 是 x 的儿子且 (x,y) 上的字符为 c，则 $b=cac$。特别地，称两棵树的根节点分别为 $even,odd$，则 $even$ 代表着空串，odd 代表着长度为 -1 的串。即，若树上有边 (odd,x,c)，则 x 代表的串为 c。对于每个节点 x，记 $tr_{x,c}$ 表示 x 的出边为 c 的儿子。

树上每个点还有一个 $fail$ 指针指向它代表的串的最长回文真后缀对应的节点。特别地，定义串长为1，即没有非空回文真后缀的点的 $fail$ 值为 $even$，$fail_{even}=odd$。若每个点向它的 $fail$ 连边，则形成一棵以 odd 为根的树，称它为 fail 树。

二、回文树的性质

（一）性质一

往一个串末尾添加一个字符，本质不同的回文子串数量至多增加1。

证明：采用反证法。若增加了超过1个，不妨设增加了 $s_1,s_2(|s_1|>|s_2|)$，则 s_2 是 s_1 的回文后缀。根据回文的基本性质，s_2 也是 s_1 的回文前缀，即 s_2 在之前的串中出现过，矛盾。

（二）性质二

字符串 s 的回文树的节点数至多为 $|s|+2$。

证明：空串的回文树节点个数为 2，从空串开始依次添加字符，由性质一，每添加一个字符节点个数至多增加 1，所以 s 的回

文树的节点数至多为 $|s|+2$ 。

三、回文树的构造算法及其拓展

（一）基础构造算法

我们采用增量法构造回文树，即假设已经构造出了 s 的回文树，接下来在 s 末尾添加字符 c ，并构造 sc 的回文树。

根据性质一，构造回文树的关键是找出 sc 的最长回文后缀。

在增量法的过程中，我们记录当前串最长回文后缀对应的节点 x 。添加一个字符 c 时，我们不断将 x 变为 $fail_x$ ，直到 x 代表的后缀的前一个字符是 c 。注意到 $x=odd$ 时该条件一定成立，所以我们一定找得到这样的 x 。此时将 x 设为 $tr_{x,c}$ ，若没有则新建节点即可。

若新建了节点，我们还需计算出新节点的 $fail$ 。可以采用类似的方法，从上面找到的 x 的 $fail$ 出发，继续寻找前一个字符是 c 的后缀。

在串长为 1 时需要特殊处理。

接下来我们分析复杂度。

定义势能函数为当前 x 代表的串的长度，可以发现势能每次最多增加 2 ，而跳 $fail$ 会减少势能。

计算 $fail$ 的部分，可以定义势能函数为当前 $fail_x$ 代表的串的长度，也是一样的。

我们也可以定义势能函数为 x 在 fail 树上的深度。跳 $fail$ 时势能减 1 。记 $tr_{x,c}=y$ ，当 x 变为 y 时，深度最多加 2 ，理由是 y 的每个长度为 $l(l\geq 1)$ 的 border 都意味着 x 有长为 $l-2$ 的 border。计算 $fail$ 的部分同理。

因此，以上算法的时间复杂度是 $O(n)$ 。

（二）在开头添加字符

上述算法支持了在字符串末尾添加字符时维护回文树。下面我们尝试支持在字符串开头添加字符。

可以发现回文串的回文前缀和回文后缀是一样的，反串的回文树和原串是一样的，所以前端插入和后端插入是本质相同的。额外记一个当前串的最长回文前缀对应的节点即可。注意当整个串变为回文串时，后端插入会影响最长回文前缀，反之亦然。

此时的时间复杂度分析只能采用第二种势能函数，即最长回文前缀和最长回文后缀的 fail 树深度和。因为当整个串变成回文串时，不妨设我们是后端插入，那么最长回文前缀的长度可能会增加很多，但 fail 树上的深度只会增加 1。

（三）复杂度非均摊的构造方式

上面的构造方式复杂度是均摊的，无法支持末尾删除字符。

我们尝试在每个点 x 额外开一个数组 $qk_{x,c}$ ，记录从 x 开始跳 $fail$ ，目标字符是 c 时会到达哪个点。

但是 x 在 s 中不同出现位置的前一个字符是不同的，无法仅根据 x 确定 $qk_{x,c}$ 的值。

稍作修改，令 $qk_{x,c}$ 表示从 $fail_x$ 开始跳的结果。因为从 $fail_x$ 开始的每个节点都是 x 的真子串，所以它们的前一个字符都是确定的。

新建节点 x 并计算出 $fail_x$ 后， qk_x 从 qk_{fail_x} 拷贝过来，再加上 $fail_x$ 的转移即可。

时间复杂度 $O(n|\Sigma|)$ ，使用持久化数组 [8] 实现拷贝数组即可做到 $O(n\log|\Sigma|)$ 。

四、同时支持前后插入删除

（一）回文树双端修改算法流程

在两端都有插入删除操作时，插入并不困难，跑非均摊的插入算法即可。删除分为两个部分：求出当前串的最长回文前缀和后缀、判断回文树上的这个节点是否需要删除。下面逐个解决。

定义 s 的子串 $s[l,r]$ 是重要的，当且仅当它是回文的，且不存在 $l'<l$ 满足 $s[l',r]$ 是回文的，且不存在 $r'>r$ 满足 $s[l,r']$ 是回文的。注意到重要子串的性质： s 的所有前缀中，重要子串有且仅有一个，其就是 s 的最长回文前缀；后缀同理。于是，我们考虑维护所有重要子串，令 L_i 和 R_i 分别存储以 i 开头 / 结尾的重要子串（根据定义，若存在则唯一；不存在则设为 0 ），这样就能解决我们的问题。

设当前 s 的最长回文后缀为 $[l,n]$ ，它的最长回文真前缀 / 后缀的长度为 x 。在末尾插入 s_n 时，我们加入了一个重要子串 $[l,n]$ ，并且将 $[l,l+x-1]$ 变为不重要。在末尾删除 s_n 时， $[l,n]$ 变为不重要，并且 $[l,l+x-1]$ 可能从不重要变为重要，依据 R_{l+x-1} 当前的值即可判断出 $[l,l+x-1]$ 是否变为重要。开头的插入删除同理。于是插入删除时造成的修改都是 $O(1)$ 的，我们得以快速找出当前串的最长回文前缀和后缀。

关于判断删除，我们考虑类似只有一端插入删除时的方法，对于回文树上每个节点，记录它成为了多少次前缀的最长回文后缀，或后缀的最长回文前缀，记为 tg 。

我们声称：对于任意字符串 s ，无论采用何种顺序插入，得到的 tg 值都是一样的。

证明：我们只需要证明，对于任意字符串 s ，将其中的字符正着插入一遍和倒着插入一遍，得到的 tg 值一样即可。如果该命题成立，则运用归纳法，设长度 $<k$ 的两个插入序列得到的 tg 值相同，若第 k 次插入在同侧，则显然成立；若在异侧，则根据归纳假设，可以将两个操作序列等价地看做是正着插入和倒着插入的，此时得到的 tg 相同。

该命题等价于， s 的所有前缀的最长回文后缀组成的可重集（记为 T ），和 s 反串的这个集合相等。记 s 的所有回文子串的字符串本身组成的可重集为 S ，我们尝试证明 S 唯一决定了 T 。考虑字符串长度从大到小地枚举 S 中的元素，设当前枚举到 t ，我们在 S 中删去 t 的所有回文真后缀恰好一个。整个过程结束后 $S=T$ ，于是原命题得证。

这样，我们说明了 tg 值是良定义的，并且执行插入删除时直接用单侧插入删除的方法维护 tg 值是正确的。

使用持久化数组即可做到时空复杂度 $O(n\log|\Sigma|)$ ，无均摊。

（二）代码实现

以下是该算法的 C++ 代码实现。为避免过于冗长，数组拷贝

仅使用了朴素枚举的方法。

```
const int MN=1e6,MS=26; // 操作次数上界和字符集大小
int a[MN*2],L=MN,R=MN-1,cnt=2,len[MN],fail[MN],tr[MN][MS],fa[MN],tg[MN],qk[MN][MS],lm[MN*2],rm[MN*2];
void init() // 初始化
{
	memset(a,-1,sizeof a);
	len[1]=-1,fail[1]=fail[2]=1;
	for(int i=0;i<26;i++)
		qk[1][i]=qk[2][i]=1;
}
void push_front(int c) // 开头插入
{
	int xl,xr;
	if(L>R)
		xl=xr=1;
	else
		xl=lm[L],xr=rm[R];
	a[--L]=c;
	if(a[L+len[xl]+1]!=c)
		xl=qk[xl][c];
	if(tr[xl][c])
		xl=tr[xl][c];
	else
	{
		tr[xl][c]=++cnt;
		fa[cnt]=xl;
		len[cnt]=len[xl]+2;
		if(len[cnt]==1)
			fail[cnt]=2;
		else
			fail[cnt]=tr[qk[xl][c]][c];
		xl=cnt;
		for(int i=0;i<26;i++)
			qk[xl][i]=qk[fail[xl]][i];
		qk[xl][a[L+len[fail[xl]]]]=fail[xl];
	}
	int r=L+len[xl]-1,t=fail[xl],l=r-len[t]+1;
	lm[L]=rm[r]=xl;
	if(len[xl]>1&&lm[l]==t)
		lm[l]=0;
	tg[xl]++;
}
void push_back(int c) // 末尾插入
{
	int xl,xr;
	if(L>R)
		xl=xr=1;
	else
		xl=lm[L],xr=rm[R];
	a[++R]=c;
	if(a[R-len[xr]-1]!=c)
		xr=qk[xr][c];
	if(tr[xr][c])
		xr=tr[xr][c];
	else
	{
		tr[xr][c]=++cnt;
		fa[cnt]=xr;
		len[cnt]=len[xr]+2;
		if(len[cnt]==1)
			fail[cnt]=2;
		else
			fail[cnt]=tr[qk[xr][c]][c];
		xr=cnt;
		for(int i=0;i<26;i++)
			qk[xr][i]=qk[fail[xr]][i];
		qk[xr][a[R-len[fail[xr]]]]=fail[xr];
	}
	int l=R-len[xr]+1,t=fail[xr],r=l+len[t]-1;
	rm[R]=lm[l]=xr;
	if(len[xr]>1&&rm[r]==t)
		rm[r]=0;
	tg[xr]++;
}
void pop_front() // 开头删除
{
	int xl,xr;
	if(L>R)
		xl=xr=1;
	else
		xl=lm[L],xr=rm[R];
	int r=L+len[xl]-1,t=fail[xl],l=r-len[t]+1;
	if(len[xl]>1&&len[lm[l]]<=len[t])
		lm[l]=rm[r]=t;
	else
		rm[r]=0;
	lm[L]=rm[L]=0;
	if(!(--tg[xl]))
		tr[fa[xl]][a[L]]=0;
	a[L++]=-1;
}
void pop_back() // 末尾删除
{
```



```
    int xl,xr;
    if(L>R)
        xl=xr=1;
    else
        xl=lm[L],xr=rm[R];
    int l=R-len[xr]+1,t=fail[xr],r=l+len[t]-1;
    if(len[xr]>1&&len[rm[r]]<=len[t])
        lm[l]=rm[r]=t;
    else
        lm[l]=0;
    rm[R]=lm[R]=0;
    if(!(--tg[xr]))
        tr[fa[xr]][a[R]]=0;
    a[R--]=-1;
}
```

五、回文树构造算法的拓展

（一）广义回文树

上述算法可以拓展到如下问题：我们有若干个字符串，每次操作在其中一个字符串的开头或末尾插入或删除字符，实时维护所有串的回文树的并。我们对于每个串分别维护 L,R 数组，并把一个节点的 tg 值定义为它在所有串的回文树中的 tg 值之和，即可解决该问题。

（二）持久化回文树

上述算法可以拓展到持久化回文树。

数组拷贝已经用持久化数组实现，我们可以将每个节点记录 qk 数组改为只记录一个版本号，即应有的 qk_x 数组在持久化数组里的版本号。对 qk 的拷贝和修改正常在持久化数组里进行，此外算法中每次操作只有 $O(1)$ 次对数组值的修改（包括对记录的版本号的修改），所以可以简单地将回文树的所有数组改为持久化数组来实现持久化回文树。时空复杂度 $O(n(\log n+\log|\Sigma|))$ 。

六、两端插入删除算法的应用

上述算法不仅可以使很多经典问题支持在字符串的两端插入删除，还能对于某些题目给出更直接的解法。

例如《基因》一题[9]，给定长为 n 的字符串 s ， q 次查询 s 的一个子串中本质不同的回文子串数量。以往的解法需要用到扫描线等算法，思维难度较高。我们可以用莫队算法[10]将问题转化为字符串两端插入删除并维护回文树的问题，再用本文中阐述的算法解决，可做到时间复杂度 $O(n\sqrt{n}\log|\Sigma|)$ ，足以通过此题。该算法的优点是较为直接，容易想到。

七、结束语

本文参考了回文树单侧修改的构造算法，设计出了重要子串和 tg 值的概念，成功解决了求出最长回文前后缀和判断是否需要删除两个困难，提出了回文树双端修改算法，解决了在字符串两端插入删除字符并维护回文树的问题。此外，本文讨论了广义回文树，并用持久化数组解决了持久化回文树的问题。本文还引用了《基因》一题，体现了回文树双端修改算法在字符串问题上的强大应用。

参考文献

[1]Manacher, G.: A new linear-time on-line algorithm finding the smallest initial palindrome of a string. J. ACM 22(3), 346 - 351 (1975).

[2]Groult, R., Prieur, E., Richomme, G. Counting distinct palindromes in a word in linear time. Inform. Process. Lett. 110, 908 - 912 (2010).

[3]Kosolobov, D., Rubinchik, M., Shur, A.M.: Finding distinct subpalindromes online. In: Proc. Prague Stringology Conference. PSC 2013. pp. 63 - 69. Czech Technical University in Prague (2013).

[4] 徐毅 . 浅谈回文子串问题 . 北京：2014年信息学奥林匹克中国国家队候选队员论文集，2014.

[5]Mikhail Rubinchik, Arseny M. Shur. EERTREE: An Efficient Data Structure for Processing Palindromes in Strings. arXiv:1506.04862 (2015).

[6] 翁文涛 . 回文树及其应用 . 北京：IOI2017中国国家候选队论文集，2017.

[7] 徐安矣 . 浅谈回文串问题的相关算法及其应用 . 北京：2023年信息学奥林匹克中国国家集训队论文，2017.

[8]OI-Wiki. 可持久化线段树 .https://oi-wiki.org/ds/persistent-seg/.

[9]2017山东一轮集训 Day4. 基因 . https://loj.ac/p/6070.

[10]OI-Wiki. 普通莫队算法 . https://oi-wiki.org/misc/mo-algo/.

