后缀数组学习笔记

henix
http://blog.henix.info/

2012 年 10 月

目录

1 定义

有字符串 S = S[1:n] ，定义其后缀 Suffix(i) = S[i:n] ，即从第 i 个字符开始一直到末尾的字符串。

Suffix(i) 的类型为 suffix_t ，suffix_t 根据实现的不同，可以实现成 int 或 char * ：int 即序号，char * 即指向后缀字符串的指针。总之，只保存一个 i 就足以表示该后缀了。

定义数组 suffix_t SA[int] 为将 S 的所有后缀排序后得到的数组。由于历史原因命名成 SA ，但我觉得用 sortedSuffix 会更好。

数组 int rank[suffix_t] 可得到某一后缀的索引（序号）。从这里可以看出，suffix_t 一般会实现成 int ，因为要用做数组的索引。同样，关于命名，我个人觉得用 suffixOrder 可能会更好。

从上面可以看出，sa 和 rank 为互逆关系，即 sa[i] = s $\Leftrightarrow$ rank[s] = i 。

我看到的其他资料一般没有区分 suffix_t 和 int ，一律用 int ，我觉得区分类型更有助于我们的理解和写出正确的程序。

表1是字符串“aabaaaab”的后缀数组和 rank 数组样例。

注：为了跟 C 语言接近，本文所有下标均从 0 开始。

i	SA[i] 表示的后缀	SA[i] 的实际值
0	aaaab	3
1	aaab	4
2	aab	5
3	aabaaaab	0
4	ab	6
5	abaaaab	1
6	b	7
7	baaaab	2

S[i]	a	a	b	a	a	a	a	b
rank[i]	3	5	7	0	1	2	4	6

表 1：“aabaaaab”的后缀数组和 rank 数组

2 串匹配

若有另一串 W ，试问：W 是否是 S 的子串？

应用后缀数组可在 $O(|W| \log |S|)$ 时间内求解。显然，若 W 是 S 的子串，则 W 是 S 的某一后缀的前缀。由于后缀数组的有序性质，在 SA 上进行二分查找即可。

3 最长公共前缀

定义序号为 i 的后缀和序号为 j 的后缀的最长公共前缀（Longest Common Prefix, LCP）：LCP(i,j) = lcp(SA[i], SA[j]) 。

引理 **3.1** *设 $i \le k \le j$ ，若 SA[i] 和 SA[j] 的前 p 个字符相同，则 SA[k] 的前 p 个字符也跟它们相同。*

证明：

根据 SA 的有序性质，SA[k] 的前 p 个字符不可能比 SA[i] 的前 p 个字符小（那样的话它就应该排在 SA[i] 前面），也不可能比 SA[j] 的前 p 个字符

大（否则它应该排在 SA[j] 后面）。因此 SA[k] 的前 p 个字符只能等于 SA[i]/SA[j] 的前 p 个字符。

证毕

引理 **3.2** *若 $i \le k \le j$，则 LCP(i,j) = min{LCP(i,k),LCP(k,j)}* 。

证明：

不妨设 LCP(i,k) < LCP(k,j) 。由最长公共前缀的定义可知，SA[i] 和 SA[k] 的前 LCP(i,k) 个字符是相同的，而 SA[k] 和 SA[j] 的前 LCP(k,j) 个字符是相同的，所以 SA[i] 和 SA[j] 至少有前 min{LCP(i,k),LCP(k,j)} 个字符相同，即 LCP(i,j) ≥ min{LCP(i,k),LCP(k,j)} 。

那么有没有可能 LCP(i,j) 比 min{LCP(i,k),LCP(k,j)} 还大呢？没那种可能。反证如下：

如果 LCP(i,j) > min{LCP(i,k),LCP(k,j)} ，设 p = LCP(i,j) ，根据 LCP 的定义，SA[i] 和 SA[j] 的前 p 个字符是相同的。由引理3.1，SA[k] 的前 p 个字符跟 SA[i] 的前 p 个字符也相同，所以我们找到了一个比 LCP(i,k) 更长的公共前缀，与 LCP(i,k) 的定义矛盾。

证毕

定理 **3.1 (LCP** 定理**)** *设 $i < j$，则 LCP(i,j) = min{LCP(k-1,k) | $i < k \le j$}* 。

由引理3.2，再使用归纳法即可证明。

根据定理3.1，我们只需要先求出相邻的 LCP ，然后任意两个后缀之间的 LCP 都可以通过一个区间最值查询（Range Minimum Query, RMQ）得到。

由定理3.1可得如下推论：

推论 **3.1** *设 $i \le j < k$，则 LCP(j,k) ≥ LCP(i,k)* 。

即如果一个区间包含了另一个区间，那么较大区间的 LCP 较小。这个推论在后面 LCP 的计算中会用到。

定义数组 int[int] height[i] = LCP(i-1, i) （i > 0），并令 height[0] = 0 。我们的目标就是计算出 height 数组。[1]

所以，问题是如何高效地计算出 height 数组，如果完全按照定义的话就没有利用各个后缀之间的联系。

下面将讨论关于最长公共前缀的一个重要性质。

[1]同样，关于命名，我认为 height 这个名字简直莫名其妙，我建议用 neighborLcp 替代之。

	SA[i]
	aaaab
i	aaab
j,next(i)	aab
	aabaaaab
next(j)	ab
	abaaaab
	b
	baaaab

表 2: 计算 height 数组

见表2，i 和 j 是后缀数组中两个相邻的后缀，i 在 j 的前面，i 和 j 的类型都是 suffix_t 。

定义函数 suffix_t next(suffix_t s) ，返回后缀 s 后面一个后缀，即去掉 s 开头一个字符得到的后缀。next(i) 和 next(j) 都已标在表上的正确位置。

考虑这样一个事实，既然 i 排在 j 前面，那么 next(i) 显然也应该排在 next(j) 前面。

再定义一个数组 int h[suffix_t s] = height[rank[s]] 。h 和 height 的区别仅仅是一个的输入直接是序号，另一个用后缀作为输入。

根据定义，h[j] = lcp(i,j) 。如果 i 和 j 至少有前 1 个字符相同，即 lcp(i,j) $\geq$ 1 ，那么去掉第一个字符后，有：lcp(next(i),next(j)) = lcp(i,j) - 1 。又由于 next(i) 在 next(j) 前面，由推论3.1，h[next(j)] $\geq$ lcp(next(i),next(j)) = lcp(i,j) - 1 = h[j] - 1 。

于是：

定理 3.2 *对于后缀 j ，如果 h[j] $\geq$ 1 ，则 h[next(j)] $\geq$ h[j] - 1* 。

于是我们可以按照 s 的第一个、第二个、……后缀的顺序计算 height（而不是 SA 的顺序）。代码如下：

代码清单 1: 计算 height 数组

```
typedef char *suffix_t;

char *s;
suffix_t sa[n];
int rank[n];
int height[n];

```

```
#define RANK(suf) rank[suf-s] // rank of a suffix, int RANK(suffix_t)

void calcHeight() {
    char *cur = s;
    int skip = 0;
    height[0] = 0;
    while (*cur != '\0') {
        if (RANK(cur) > 0) {
            // calc lcp of cur and sa[RANK(cur) - 1]
            // 1. skip first `skip` characters
            const char *pa = cur + skip;
            const char *pb = sa[RANK(cur) - 1] + skip;
            while (*pa == *pb) {
                pa++;
                pb++;
            }
            // 2. save lcp, update skip
            height[RANK(cur)] = skip = pa - cur;
            if (skip > 0) skip--;
        } else {
            skip = 0;
        }
        cur++;
    }
}
```

上面的代码中用 char * 实现 suffix_t 。我认为这样比 int 更清晰可读，而且一旦误用，编译器马上会报错。

4 后缀数组构造算法：倍增算法

最后终于说到构造了。如果完全按照后缀数组的定义构造后缀数组，将所有后缀进行排序，那么就没有充分利用各个后缀之间的联系，效率低。对 n 个元素排序需要 $O(n \log n)$ 的时间，但考虑到每个元素都是长度为 n 的字符串，故这种方法需要的时间为 $O(n^2 \log n)$ 。

倍增算法（Prefix Doubling）是由后缀数组原始论文 [2] 提出的一个构造算法，时间复杂度 $O(n \log n)$ 。

定义 $S_n(i)$ 为后缀 S(i) 的前 n 个字符。倍增算法的思想是：如果对于所有

的 $i \in [0, n)$ ，$S_n(i)$ 的顺序已经确定，那么可以利用这个顺序确定 $S_{2n}(i)$ 的顺序。按照前 1、2、4、……个字符的顺序排序所有后缀。每次排序利用基数排序的扫描方法可以做到 $O(n)$ ，而总的次数为 $\log n$ 次，故总的时间复杂度为 $O(n \log n)$ 。

具体算法将在后续笔记中讨论。

5 参考资料

- 罗穗骞：《后缀数组——处理字符串的有力工具》[4]
- 许智磊：《后缀数组》[5]
- Manber 和 Myers 的原始论文《Suffix arrays: a new method for on-line string searches》[2]
- 《Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications》[1] ：lcp 计算原始论文。首次将 lcp 的计算和后缀数组的计算分离开来。
- 《Two space saving tricks for linear time LCP computation》[3] ：不需要 rank 数组，计算 lcp

参考文献

[1] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, CPM '01, pages 181--192, London, UK, UK, 2001. Springer-Verlag.

[2] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, SODA '90, pages 319--327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[3] Giovanni Manzini. Two space saving tricks for linear time lcp computation. In *In: Proc. SWAT. Volume 3111 of Lecture Notes in Computer Science*, pages 372--383. Springer, 2004.

[4] 罗穗骞 and 张学东. 后缀数组——处理字符串的有力工具. *IOI2009 国家集训队论文*, 2009.

[5] 许智磊. 后缀数组. *IOI2004 国家集训队论文*, 2004.