« 后缀数组（Suffix Array）——理论和思想

数据误删… »

后缀数组（Suffix Array）——实现与应用

2008年03月9日

　　我在前面一篇文章中已经概要地讲了后缀数组的基本理论依据，下面结合一个 ACM/ICPC 竞赛题目来说说后缀数组的简单应用。我们先来实现后缀数组 O(nlogn) 的构造算法。我曾经在老的博客上写过一个比较丑陋的后缀数组构造算法，我在产生写这两篇文章的想法时，有去网上搜了一下，看了别人的一些实现和一些以前留下的论文，现对之前的算法进行优化，使其变得比较美观一些

　　我的构造算法用了O(4n)的空间复杂度，这个和Udi Manber & Gene Myers的论文中提到的O(

2n)的空间复杂度还是有差距的，但是考虑如果按照他们的算法写出来，那么代码必然更长更臭（我之前那个算法就是受了他们思想的很大影响才造就了其丑陋程度），所以还是牺牲一点空间吧。此外，我还看到过几个空间为 O(2n) 而且比一般O(nlogn) 快的算法，但是代码和思想都非常复杂，不利于掌握。

　　定义一种类型：

typedef unsigned char uchar;

　　后缀数组构造算法：

void CreateSuffixArray(uchar\* szText,

int L, int\*\* \_S, int\*\* \_R, int\*\* \_T1, int\*\* \_T2)

{

int i, h, h2, \*T, \*S1, \*S2, \*R, \*B;

S1 = \*\_S; // h阶后缀数组

S2 = \*\_T1; // 2h阶后缀数组

R = \*\_R; // h阶Rank数组

B = \*\_T2; // 某个桶空余空间尾部的索引，兼任2h阶Rank数组

// 花O(n)的时间对h = 1进行计数排序

for(i = 0; i < 256; i++)

B = 0;

for(i = 0; i < L; i++)

B[szText]++;

for(i = 1; i < 256; i++)

B += B[i - 1];

for(i = 0; i < L; i++)

S1[--B[szText]] = i;

// 计算Rank(1)，因为仅仅是1阶的Rank，所有有并列的

for(R[S1[0]] = 0, i = 1; i < L; i++)

{

if(szText[S1] == szText[S1[i - 1]])

R[S1] = R[S1[i - 1]];

else

R[S1] = R[S1[i - 1]] + 1;

}

// log(n)趟O(n)的倍增排序

// SA(h) => Rank(h) => SA(2h) => Rank(2h) => …

for(h = 1; h < L && R[S1[L - 1]] < L - 1; h <<= 1)

{

// 计算Rank(h)相同的后缀形成的h桶尾部的索引

// 即有多少个后缀的h前缀相同，它们被放在一个桶中

for(i = 0; i < L; i++)

B[R[S1]] = i;

// 求SA(2h)

// 在同一个h桶中，所有的后缀的h前缀肯定相同，

// 那么比较他们的2h前缀，只要比较其2h前缀后半的

// 长度为h的串即可，而这个串恰恰是后面某个后缀的

// h前缀，所以我们逆向遍历有序的SA(h)，

// 将S1 – h号前缀放到它所在桶的最后一个空位置，

// 同时，桶尾前进一个位置，这样即形成了2h桶排序

for(i = L – 1; i >= 0; i–)

if(h <= S1)

S2[B[R[S1 - h]]–] = S1 – h;

// 对于长度不超过h的最后几个后缀，由于在h阶段

// 它们每个实际上都已经独立分桶了(长度为h的也是)

// 而且他们的桶中有且仅有一个元素，

// 所以只要直接复制他们h阶段的SA值就可以了

// 同时，由于采用滚动数组，所以S2中“残留”了

// h/2个有效的数据，所以最终我们只需复制h/2个数据

for(i = L – h, h2 = L – (h >> 1); i < h2; i++)

S2[B[R]] = i;

T = S1; S1 = S2; S2 = T;

// 计算Rank(2h)

// 2h阶段是否要分桶只需看相邻两个2h前缀前后两半

// h前缀是否全部h阶相等

for(B[S1[0]] = 0, i = 1; i < L; i++)

{

// 这里不用考虑S1 + h会越界

// 如果i达到了S1 + h越界的数值，

// 那么前面一个条件显然不会满足了

// 因为此时i前缀肯定已经独立分桶了

if(R[S1] != R[S1[i - 1]] ||

R[S1 + h] != R[S1[i - 1] + h])

{

B[S1] = B[S1[i - 1]] + 1;

}

else

B[S1] = B[S1[i - 1]];

}

T = B; B = R; R = T;

}

if(\*\_S != S1)

\*\_S = S1, \*\_T1 = S2;

if(\*\_R != R)

\*\_R = R, \*\_T2 = B;

}

　　介绍一个重要概念：LCP！LCP是Longest Common Prefix的缩写，即最长公共前缀，表示某个串从第一个字符开始对应位置字符相同的连续的位置数。比如，后缀abcda和后缀abcca的LCP就是3。我们将后缀数组中连续的两个后缀Ai-1和Ai的LCP称为Ai的Height，即Height(i) = LCP(j , j – 1)，并规定ASA[0]的Height为0。那么很显然，后缀数组某个区间的两个区间边界元素所表示的后缀的LCP就是区间内所有元素所代表的后缀的Height的最小值。我们要求这个LCP，就相当于一个RMQ（Range Minimum Query）问题，当Height已知的时候，只要常数时间就可以求出RMQ，即所求的LCP。所以，关键是如何降低求Height数组的复杂度。不过人们发现Height数组有一个令人兴奋的性质。令 h(x) = Height(Rank(x))，即x号前缀的Height值，那么，

　　当 x > 0 且 Rank(x) > 0 时， h(i) ≥ h(i – 1) – 1

　　这个在这里就不证明了，反正证明过程相当巧妙 利用这个性质，有了下面的这个线性的求Height的算法：

void CalculateHeight(uchar\* szText,

int L, int\* S, int\* R, int\* H, int\* T)

{

int i, j, k;

for(k = 0, i = 0; i < L; i++)

{

if(R == 0)

H = 0;

else

{

for(j = S[R - 1]; szText[i + k] == szText[j + k]; k++);

H[R] = k;

if(k > 0)

k–;

}

}

}

　　初一看，这个不是 O(n2) 的吗？其实根据上面说的性质，可以证明，它是线性的，证明也略了

　　下面是一个具体的ACM/ICPC竞赛题目的解法，原题你可以在这里找到：http://acm.pku.edu.cn/JudgeOnline/problem?id=2774

char C[200002];

int D[4][200001];

int main()

{

int i, l1, l2, b;

int \*S, \*R, \*H, \*T;

gets(C);

l1 = (int)strlen(C);

C[l1] = '$';

gets((char\*)C + l1 + 1);

l2 = l1 + 1 + (int)strlen(C + l1 + 1);

S = D[0]; R = D[1];

H = D[2]; T = D[3];

CreateSuffixArray((uchar\*)C, l2, &S, &R, &H, &T);

CalculateHeight((uchar\*)C, l2, S, R, H, T);

// 求两个串的最长公共子串，只要让两个串s1、s2

// 连接在一起形成一个新串，求出新串的SA、Rank和Height

// 很显然，最长公共子串肯定出现在后缀数组某相邻两项之中

// 根据Height的定义，扫描一遍Height数组，找相邻两个分别开始于

// s1和s2串某个位置的后缀，求出所有满足这个条件的最大Height即可

for(b = 0, i = 1; i < l2; i++)

{

if(S < l1 && S[i - 1] > l1 ||

S > l1 && S[i - 1] < l1)

{

if(H > b)

b = H;

}

}

printf(“%d\n”, b);

return 0;

}

　　后缀数组的用处很大，除了上面的求两个串的最长公共字串之串之外，多模式匹配、最长回文串、全文检索等等都它的拿手好戏，可以说后缀数组是后缀树良好的替代品。