New LCP

中山纪念中学 高一(24)班 陈启峰

May 31, 2006

目 录

1 基础知识

【字符串后缀】

字符串结尾出现的连续部分。用字符串$S_{i...n}$来表示，其中n表示字符串S的长度。

【LCP】

Longest Common Prefix，也就是两个字符串最长公共前缀。$LCP(S,i,j)$表示$S_{i...n}$和$S_{j...n}$的最长公共前缀的长度。

2 摘要

本文介绍一种本人自创的全新LCP算法，简称为New LCP。

本文首先详细地介绍本算法的基本功能，然后再有侧重地介绍本算法的增加功能。

3 关键字

LCP

平衡系数D

4 引言

对于LCP问题，现在已经有很多复杂度很低的算法能够解决它，比如后缀数组,后缀树等。 然而这些算法都有一定的缺陷：

1. 常数非常大
2. 编程复杂度很高
3. 需要大量的额外空间

针对这些缺陷,我设计了一种新型的算法。这种算法具有以下优点：

1. 编程简单(标准LCP代码不到1Kb)
2. 功能强大(支持高速询问LCP和删除、修改、增加连续一段字符串的功能)
3. 常数很小(运算次数很少)
4. 复杂度低(后面有具体介绍)
5. 需要的额外空间少(仅开一个O(nlogn)的数组)
6. 平衡优化(可以平衡各操作之间的效率)
7. 实现方法多(数据结构和算法都有不同的实现方法)
8. 扩展性强(很多问题可用类似思想解决)

下面一起来揭开这神秘面纱、感受New LCP算法的奇妙吧！

5 经典问题

5.1 问题描述

给出一个长度为$n(n \leq 5 \times 10^5)$的字符串S和$Q(Q \leq 10^6)$个询问，每个询问$LCP(i,j)$都是问$S_{i...n}$与$S_{j...n}$的最长公共前缀的长度是多少？

5.2 问题分析

这既然是经典问题，自然就有经典解法。最朴素的算法莫过于对$S_{i...n}$与$S_{j...n}$从头到尾一一比较。 但是这种算法是很低效的，因为最坏情况下时间复杂度$O(NQ)$。除此之外还有一些高效的经典算法，比如后缀数组[i]、 后缀树[ii]，它们的时间复杂度分别是$O(nlogn+Q)$或者$O(n+Q)$、$O(n \times |\sum| + Q)$,但是这些算法的常数都很大，而且编程复杂度很高。 一种简单高效的算法就应运而生。

5.3 Hash标号法

首先要定义一个状态$X_{i,j}$——

1. $i + 2^j - 1 \leq n$;
2. 表示 $S_{i...i+2^j-1}$ 对应的标号;
3. $\forall$两个状态 $X_{a,b}$、$X_{c,d}$, 如果$S_{a...b+2^b-1} = S_{c...c+2^d-1}$ 则 $X_{a,b}$=$X_{c,d}$, 否则 $X_{a,b} \neq X_{c,d}$ 。

[i]详见IOI2004集训队许智磊的论文
[ii]详见Winter Camp 2006 刘汝佳的讲稿

状态定下以后，就要确定状态转移方程:

1. 对于所有的$X_{i,0}$，我们可以令$X_{i,0} = ord(S_i)$[iii];
2. 对于$X_{i,j}(j > 0)$，$X_{i,j}$完全由前一部分$X_{i,j-1}$和后一部分$X_{i+2^{j-1},j-1}$决定，我们称二元组($X_{i,j-1}$，$X_{i+2^{j-1},j-1}$)为$X_{i,j}$的前趋。 如果之前已经有另一个状态的前趋与$X_{i,j}$的前趋相同，那么就赋予$X_{i,j}$那个状态的值，否则赋予$X_{i,j}$一个从未出现过的值(一般取最大状态值加一)

接下来对于询问$LCP(i,j)$。令$l2(i) = [log_2^i]$，从k=l2(n)开始匹配：

如果$(i + 2^k - 1 \leq n) and (j + 2^k - 1 \leq n) and (X_{i,k} = X_{j_k})$则说明$S_{i...i+2^k-1} = S_{j...j+2^k-1}$,就给i和j加上$2^k$。这时应该给k减1，因为如果用k还能匹配的话则说明在前一步可以用k+1来匹配， 但这是不可能的，因为在上一阶段的匹配后再用k+1来匹配是失败的；否则就要给k减1。

因此询问操作可被写成如下简短的代码：

```
record:=i;
for k:=l2[n] downto 0 do
   if (i+1 shl k-1<=n)and(j+1 shl k-1<=n)and(x[i,k]=x[j,k]) then begin
      inc(i,1 shl k);
      inc(j,1 shl k);
   end;
writeln(i-record);
```

这个算法应该说已经很不错了，因为编程复杂度低，效率还比较高。但是因为用到了Hashtable而导致空间占用太多了！ 如果还有增加、修改、删除等操作，空间复杂度将达到$O(Q * N + NlogN)$[iv]，并且空间复杂度的常数巨大(20～40)！

[iii] ord(ch)表示字符ch的ASCII码

[iv] Q是增修删的次数

5.4 随机标号法：一种更好的算法

这种算法是我强力推荐使用的。它的特点是字符串的值是随机的、确定的。 怎么理解这句话呢？随机性是说字符串的值是随机赋予的，确定性是说对于相同的字符串，它的值是确定的、也就是相同的。

为了保证随机性和确定性，首先要建立一个随机确定数组——Ra数组。$Ra(i)$是一个很大范围内的随机值。 一般地，这范围取-2^{63}～2^{63}-1，在Pascal里可用int64 存下$Ra(i)$，在C/C++里可用long long存下$Ra(i)$。在数据不太大的情况下可以考虑使用longint/long来代替int64/long long来提高速度[v]。

随机标号法的状态转移方程是很简单：

1. 对于$X_{i,0}$，$X_{i,0} = Ra(ord(S_i))$；
2. 对于$X_{i,j}(j > 0)$，$X_{i,j} = X_{i,j-1} \times Ra(j)$ xor[vi] $X_{i+2^{j-1},j-1}$[vii]

写成的简洁代码（使用int64）：

```
for i:=0 to 255 do
   ra[i]:=random(int64(1) shl 32) shl 32+random(int64(1) shl 32);
for i:=1 to n do
   x[i,0]:=ra[ord(s[i])];
for i:=n downto 1 do
   for j:=1 to l2[n+1-i] do
      x[i,j]:=x[i,j-1]*ra[j] xor x[i+1 shl(j-1),j-1];
```

随机极端数据下各种算法的对比[viii]

算法名称	读写数据	预处理	询问	总时间	空间	代码大小
后缀数组	1.20s	7.03s	0.83s	9.06s	102Mb	3Kb多
Hash标号法	1.20s	3.47s	0.59s	5.26s	235Mb	1Kb多
随机标号法(int64)	1.20s	0.96s	0.48s	2.64s	75Mb	不到1Kb
随机标号法(longint)	1.20s	0.26s	0.44s	1.90s	38Mb	不到1Kb

[v]一般可以提高一倍速度
[vi]位异或操作
[vii]必须R-模式下操作
[viii]环境：Freepascal2.0.2 PentiumM1.6GHz 512Mb

这是一种随机算法，就有可能出错。读者可能担心正确率不高。我想这应该是不用担心的。 假设字符串获得的标号是完全随机的。算法出错当且仅当比较的字符串不同但标号相同。那么在一次比较不同的字符串中标号不同的概率是多少呢？ 如果使用int64的话这概率很明显是$\frac{2^{64}-1}{2^{64}}$，整个算法成功的概率是$\left(\frac{2^{64}-1}{2^{64}}\right)^T$，其中T表示比较不同字符串的次数。

在不同的T的情况下的正确率

T	用int64的正确率	用longint的正确率
1	99.99999999999999994578989137572%	99.99999976716935634613037109375%
10	99.99999999999999945789891375725%	99.99999767169356590075859751527%
10^2	99.99999999999999457898913757248%	99.99997671693590295307275859642%
10^3	99.99999999999994578989137572478%	99.99976716938342407753159694435%
10^4	99.99999999999945789891375724793%	99.99767169627369358138476700930%
10^5	99.99999999999457898913757249252%	99.99767172066803420993260732053 4%
10^6	99.99999999994578989137572624766%	99.97671964592698371255780405262 5%
10^7	99.99999999945789891375739471976%	99.76744019662169200075953769214 9%
10^8	99.99999999457898913758717150901%	97.69858947339400678104926190759 0%
10^9	99.99999994578989137719414623472%	79.22877410983713906014434510160 2%
10^{10}	99.99999945789891390418457678838%	9.74606631938765872232116082756 34%
10^{11}	99.99999457898915226615718834853%	极小
10^{12}	99.99994578989284509269027254257%	极小
10^{13}	99.99945789906069401508421983149%	极小
10^{14}	99.99945790038312253116123471084 0%	极小
10^{15}	99.99457913607171121341398473869 4%	极小
10^{16}	99.94580458240031667301532963444%	极小

由表格可知，用int64即使进行一亿亿次不同字符串的比较，正确率也在99.9%以上。 当然，这是建立在字符串的值是完全随机的基础上。实际上是不是这样子呢？我无法证明这点。希望有想法的读者能与我交流。

6 增强功能——增修删字符串

6.1 问题描述

给出一个长度为$N(N \leq 10^5)$的字符串S和$Q(Q \leq 10^5)$个操作，每个操作都是以下的任意一个：

1. LCP(a,b)：询问当前字符串S中$S_{a...n}$[ix]和$S_{b...n}$的最长公共前缀。保证$1 \leq a \leq n$、$1 \leq b \leq n$；
2. ADD(a,S2)：在当前字符串S第a个字符前插入字符串S2,也就是$S \longleftarrow S_{1...a-1} + S2 + S_{a...n}$。如果a=n+1则在S的最后加上S2。保证$1 \leq a \leq n+1$；
3. CHG(a,b,S2)：在当前字符串S中将$S_{a...b}$替换为S2。保证$b-a+1 = |S2|$、$1 \leq a \leq b \leq n$；
4. DEL(a,b)：从当前字符串S中删去第a个字符到第b个字符，也就是$S \longleftarrow S_{1...a-1} + S_{b+1...n}$。保证$1 \leq a \leq b \leq n$。

保证插入的字符串长度总和、修改的字符串长度总和都不超过10^5，ADD、CHG、DEL的总操作数不超过1000个。

6.2 问题分析

刚看完这个题，也许有读者会这样想：在每次增修删字符串后用$O(nlogn)$的时间复杂度去预处理，来方便以后的询问。这种方法固然可行，但未免太慢。实际上，在每次增修删字符串后只要$O(n+klogn)$[x]时间复杂度去进行维护。

[ix] n表示当前字符串S的长度

[x] k表示插入或修改的字符串的长度。如果是删除操作则k=0

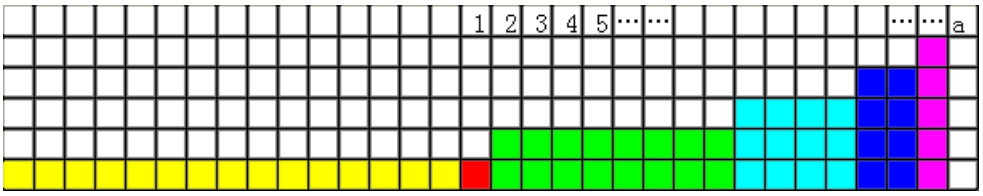
下面我分三种操作就进行讨论：

1. 对于 ADD(a,S2) 操作——在插和S2以后

   (a) 不难发现原来的状态$X_{i,j}(i \geqslant a)$的值不会改变，只是向后移动了k个位置，移动时间复杂度是$O(n)$；

   (b) 对于状态新的状态$X_{i,j}(a+k-1 \geqslant i \geqslant a)$，可以用$O(1)$的时间递推出来， 因此计算这些新状态的时间复杂度是$O(klogn)$；

   (c) 对于前面的状态$X_{i,j}(i < a)$只要改变接触S_a的状态，那需要计算的状态是多少呢？先观察下面的图



图中1至a的彩色部分表示需要改变的状态

   把黄色部分添加上,总的彩色部分小于$2n$,所以需要改变的状态数小于$2n$。所以改变这些状态值的时间复杂度是$O(n)$；

   (d) 所以总的时间复杂度是$O(n+klogn)$。

2. 对于 CHG(a,b,S2) 操作——

   (a) 对于$X_{i,j}(i > b)$,值和位置都不变；

   (b) 对于$X_{i,j}(a \leq i \leq b)$,全部重新计算一次，时间复杂度$O(klogn)$；

   (c) 对于$X_{i,j}(i < a)$，计算接触S_a的状态，时间复杂度$O(n)$；

   (d) 总的时间复杂度是$O(n+klogn)$。

3. 对于 DEL(a,b) 操作——

   (a) 对于原来的$X_{i,j}(i > b)$,值不变，向前移动$j - i + 1$个位置，移动时间复杂度是$O(n)$；

   (b) 对于$X_{i,j}(i < a)$，计算接触S_a的状态，时间复杂度是$O(n)$；

   (c) 总的时间复杂度是$O(n)$。

移动的状态数是O(nlogn)，为什么移动的时间复杂度只是$O(n)$呢？ 其实只要设定n个指针[xi]$W(i)$，$W(i)$指向X_i，移动时只要移动指针即可， 取值时使用$X_{W(i),j}$。 用这种算法已经较好地解决这个问题，在极端数据中各类操作所用的时间如下表(使用longint)：

	预处理	询问操作	移动操作	计算状态值	读数据	总用时
时间	0.15s	0.45s	0.10s	9.34s	0.11s	10.15s

6.3 重要的常数优化

观察上面的表格，询问操作和计算状值的两个核心操作所用的时间相差很远。我当时心想能不能把它们所用的时间平衡一下呢？ 于是我使用了平衡系数D。

平衡系数D是一个自定义的常数，并且还是自然数。它表示的含义是$X_{i,j}$中j的取值范围是0～$max(l2(n - i + 1) - d, 0)$。

平衡系数D的作用是通过减少状态量来加快计算状态值的速度。这样做ADD、DEL的时间效率是(常数小的n[xii]+常数大的n×2^{-d})， CHN的时间效率是(常数大的n×2^{-d})，预处理的时间效率是(常数大的n×$(logn - d)$)，但是询问操作的时间效率却变为$(logn \times 2^d)$。

[xi]建议使用伪指针，也就是用数型代替指针

[xii]移动的指针的时间

通过尝试发现在这题里平衡系数D取4达到最佳效果。应用平衡常数D优化后在极端数据中各类操作所用的时间如下表(使用longint)：

	预处理	询问操作	移动操作	计算状态值	读数据	总用时
时间	0.12s	0.55s	0.10s	1.00s	0.11s	1.88s

应用平衡系数D时有一个地方需要注意的：在ADD(i,S2)操作中，除了要计算跨过S_i的状态值，还要计算新增的状态值。这是因为S的长度增加了，就新增了状态，这些新增的状态不一定都会跨过S_i。

应用平衡系数D优化后程序代码只有2.33Kb,不足80行。

6.4 另类实现方法

有了平衡系数D以后,有时移动操作会成为算法的瓶颈。这时可以用二叉查找树[xiii]来进行高效移动，不过每次取$X_{i,j}$值的效率很低——时间复杂度从$O(1)$变为$O(logn)$。

我不推荐使用这种方法。因为这种方法不但编程复杂度高，而且常数很大。

7 总结

无论是对经典问题还是新型题目，我们都不应该只满足于经典算法和解答， 要敢于对这些问题进行思考、尝试自创的算法，说不定就能发明出一种全新的经典算法。

8 讨论情况

独自思考

[xiii]时间复杂度为O(logn)

9 程序代码

9.1 标准LCP(使用int64)

```
program CQF_LCP;
var x:array[1..500000,0..18] of int64;
    l2:array[0..500000] of longint;
    ra:array[0..255] of int64;
    a,b,rec,q,i,j,n:longint;
    ch:char;
begin
   for i:=0 to 255 do
      ra[i]:=random(int64(1) shl 32)shl 32+random(int64(1) shl 32);
   readln(n);
   l2[0]:=-1;
   for i:=1 to n do
      l2[i]:=l2[i-1]+ord(i and(i xor(i-1))=i);
   for i:=1 to n do begin
      read(ch);
      x[i,0]:=ra[ord(ch)];
   end;
   for i:=n downto 1 do
      for j:=1 to l2[n+1-i] do
         x[i,j]:=x[i,j-1]*ra[j] xor x[i+1 shl(j-1),j-1];
   readln(q);
   for q:=1 to q do begin
      readln(a,b);
      rec:=a;
      for j:=l2[n] downto 0 do
         if (a+1 shl j-1<=n)and(b+1 shl j-1<=n)and(x[a,j]=x[b,j]) then begin
            inc(a,1 shl j);
            inc(b,1 shl j);
         end;
     writeln(a-rec);
   end;
end.
```

9.2 有增加功能的LCP(使用longint)

```
program CQF_PLUS;
const d=4;
var x:array[1..200000,0..17-d] of longint;
    ra:array[0..255] of longint;
    l2,w:array[0..200000] of longint;
    i,j,k,n,q,rec,tt:longint;
    ch:char;
    s:ansistring;
procedure cal(j,k:longint);
begin
   for j:=j to k do
      x[w[i],j]:=x[w[i],j-1]*ra[j]xor x[w[i+1 shl(j-1)],j-1];
end;
begin
   readln(n,q);
   for i:=0 to 255 do
      ra[i]:=random(int64(1)shl 32);
   for i:=1 to n do begin
      read(ch);
      x[i,0]:=ra[ord(ch)];
      w[i]:=i;
   end;
   readln;
   l2[0]:=-1;
   for i:=1 to 200000 do
      l2[i]:=l2[i-1]+ord(i and(i xor(i-1))=i);
   for i:=n downto 1 do
      cal(1,l2[n-i+1]-d);
   tt:=n;
   for q:=1 to q do begin
      read(ch);
      case ch of
         'L':begin
                readln(i,j);
                rec:=i;
                for k:=l2[n] downto 0 do
```

```
            while ((k=0)or(k<=l2[n-i+1]-d)and(k<=l2[n-j+1]-d))and
                  (i<=n)and(j<=n)and(x[w[i],k]=x[w[j],k]) do begin
               inc(i,1 shl k);
               inc(j,1 shl k);
            end;
         writeln(i-rec);
      end;
   'A':begin
         readln(k,ch,s);
         move(w[k],w[k+length(s)],(n-k+1)*sizeof(w[k]));
         inc(n,length(s));
         for i:=k+length(s)-1 downto k do begin
            inc(tt);
            w[i]:=tt;
            x[w[i],0]:=ra[ord(s[i-k+1])];
            cal(1,l2[n-i+1]-d);
         end;
         for i:=k-1 downto 1 do
            if l2[k-i]<l2[n-length(s)-i+1]-d then
               cal(l2[k-i]+1,l2[n-i+1]-d)
            else
               cal(l2[n-length(s)-i+1]-d+1,l2[n-i+1]-d);
      end;
   'C':begin
         readln(j,k,ch,s);
         for i:=k downto j do begin
            x[w[i],0]:=ra[ord(s[i-j+1])];
            cal(1,l2[n-i+1]-d);
         end;
         for i:=j-1 downto 1 do
            cal(l2[j-i]+1,l2[n-i+1]-d);
      end;
   'D':begin
         readln(j,k);
         move(w[k+1],w[j],(n-k)*sizeof(w[k]));
         dec(n,k-j+1);
         for i:=j-1 downto 1 do
            cal(l2[j-i]+1,l2[n-i+1]-d);
```

```
                end;
        end;
    end;
end.
```