算法分析与设计第四次作业

黄丛宇 2010212439

October 22, 2010

1 实验环境

- CPU: Intel(R) Core(TM)2 Duo CPU T5870 2.00GHz
- MEM: 1GB
- OS : Debian 5.0 （1GB swap）
- Java: java version "1.6.0_21"

2 Exercise 15.4-4

由算法可知，在算法执行过程中，算法每次循环只使用当前行和前一行的数据。因此，可以使用一个只有两行的二维滚动数组来存储数据。另外，使用一个变量，标记那一行是当前行，则另一行是前一行。由于算法中两个循环谁在外边谁在里面不应想算法的正确性，因此可以min(m,n)放在内循环中。这样，滚动数据只需要2*min(n,m) 的长度，外加一个$O(1)$的标记变量。

可以进一步将两行的二维数组变成一个一维的数组。观察算法可得，内循环的每次运算中，假如当前要计算的元素是c[i,j]，那么计算只使用了c[i-1, j-1]，c[i-1, j]和c[i, j-1]。当使用一维数组的时候，假设当前计算的元素是c'[i]，那么c'[i - 1]存放的就是原来的c[i-1, j]，而当前的c'[i]中存放的是c[i, j-1] 的值，现在只缺少c[i-1, j-1]，这个值恰好就是c[i-1]中存放的前一个值。因此，可以使用一个变量，在计算c[i-1]的时候，将c[i-1]的前一个值保存起来，给c[i]使用。同样，计算c[i]的时候，在覆盖c[i]之前，将其值保存在这个变量中。同理，数组的长度为min(n,m)，因此，算法只需要min(n,m)长度的数组外加一个 $O(1)$的标记变量。

3 Exercise 15.4-6

定义b[k]表示以s[k]结尾的最长递增子序列的长度，则状态转移方程如下：

$$b[k] = max(max(b[j]|s[j] < s[k], j < k) + 1, 1);$$

在a[k]前面找到满足a[j]<a[k]的最大b[j],然后把a[k]接在它的后面，可得到以a[k]结尾的最长递增子序列的长度，或者a[k]前面没有比它小的a[j]，那么这时a[k]自成一序列，长度为1。最后整个数列的最长递增子序列即为max(b[k]|0<=k<=n-1)；

在寻找最大的b[j]的时候，如果使用顺序查找，则算法复杂度为$O(n^2)$，因此使用二分查找降低时间复杂度。

引入一个新的数组c。c中元素满足$c[b[k]] = a[k]$,即当递增子序列的长度为b[k]时子序列的末尾元素为 $c[b[k]] = a[k]$。算法中对c的修改可以保证c是有序的。如果有多个相同长度的递增子列，那么对应的位置存放的是最后出现的那个子列的最后一个元素。c[1]=s[0]，c[0]=0。c[0]作为二分查找的哨兵使用。

核心代码如下：

```java
public static int getLISLen(final int[] s, int[] lis)
{
   if (null == s) {
      return -1;
   }
   c = new int[s.length + 1];
   cindex = new int[s.length + 1];
   pre = new int[s.length];

   //初始化
      cindex[0] = -1;
      for(int i = 0; i < s.length; ++i){
         pre[i] = -1;
         cindex[i + 1] = -1;
      }

      c[0] = 0; //这个元素作为一个哨兵。在二分查找中使用。
      c[1] = s[0];
      cindex[1] = 0;
      len = 1; //此时只有c[1]求出来，最长递增子序列的长度为1.
   int j;
   for(int i = 1; i < s.length; ++i){
      /*
       * 二分查找。返回值表示n在数组a中的位置。如果在数组中有元素等
           于n
       * 那么返回最后一个等于n的元素的下一个位置。
       */
      j = binarySearch(c, len, s[i]);
      c[j] = s[i];
      cindex[j] = i;
      /*
```

```
             * 以s[i]结尾的最长子串的倒数第二个元素是c[j-1]。
             */
            pre[i] = cindex[j - 1];
            if(len < j){
                len = j;
                lastIndex = i;
            }

        }
        getSubsequence(s, lis);
        return len;
    }

    //最长递增子列的长度
    private static int len = 0;
    //最长递增子列最后一个元素的位置。
    private static int lastIndex = -1;
    /*
     * c[i]=a[j, ]表示c[i]中存储的是长度为i的最长递增子列的最后一个元
         素。
     * 并且，c中存放的就是最长递增子列。
     * c从1开始，c[0]最为哨兵在二分搜索中使用
     */
    private static int[] c;
    /*
     * cindex[i]存储c[i]对应的元素在序列中的位置。
     */
    private static int[] cindex;
    /*
     * pre[i]表示s[i]所在的最长递增子列的前一个元素的位置。
     * 注，这个最长子列可能不是s的最长子列，只是包含s[i]中所有
     * 递增子列最长的。
     */
    private static int[] pre;
```

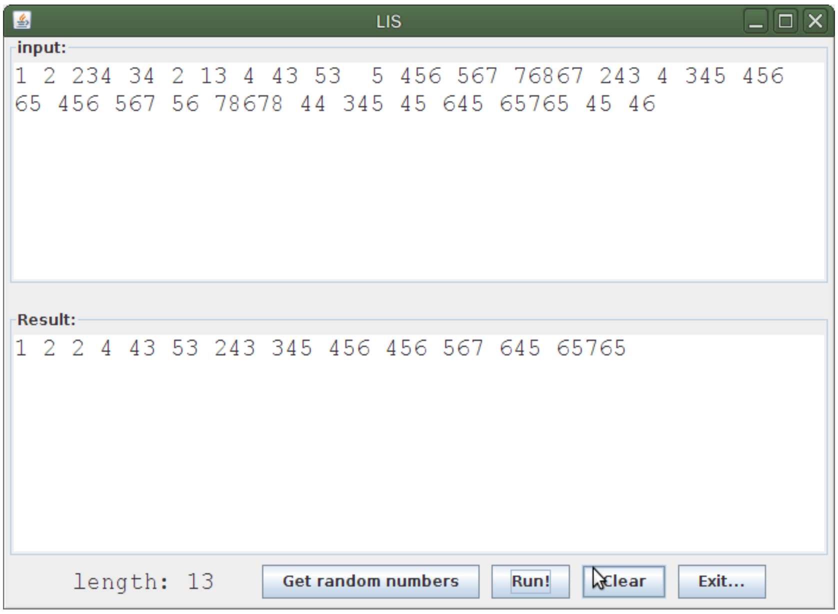
使用一个pre数组保存每个元素所在的最长递增子列的前一个元素的位置。pre[i]表示s[i]所在的最长递增子列中，s[i]前一个元素的下标。使用pre数组，可以在$O(n)$的时间内构造出一个最长的递增子列。

程序执行的命令为：java -jar lis.jar。运行结果如图1。

4 Problem 15-1 bitonic tours

1. 将所有的点按照x坐标从小到大排序。时间复杂度为$O(nlgn)$。排许后用序列 $p_1, P_2, p_3 ... p_n$表示。

Figure 1 运行结果



2. 定义从p_i到p_j之间的双调路径$d(i,j)$为：从p_i开始，严格的向左走，直到p_1，然后严格的向右走直到p_j。途中经过p_1到$p_{max(i,j)}$ 之间的所有点一次其只一次。由于对称性，只考虑$i \geq j$的情况。定义 $\Delta(i,j)$表示p_i和p_j之间的距离。

3. 计算$d(i,j), i \geq j$，最基本的情况是$d(2,1) = \Delta(2,1)$。

   (a) 如果$j < i-1$，由于是严格的向左或向右走，从p_j出发向左走只能经过位置小于j的点。对于大于j小于i的点，由于要保持严格向右，因此只有一条路径。而(p_i, p_{i-1})一定在路径中。因此边(p_i, p_{i-1})一定在从p_i到p_j之间的双调路径上。那么$d(i,j) = d(i-1,j) + \Delta(i,i-1)$。

   (b) 如果$j = i-1$，考虑在最短路径中最先连接的两个点，可以是p_i 到p_1, p_i到$p_2, \ldots p_i$到p_{-2}。所以 $d(i,j) = min(d(1,i-1) + \Delta(i,1), d(2,i-1) + \Delta(i,2), ..., d(i-2,i-1) + \Delta(i,i-2))$

4. 对于d[n][n]， $d(n,n) = min(d(1,n)+\Delta(n,1), d(2,n)+\Delta(n,2), ..., d(n-1,n) + \Delta(n,n-1))$ 对于$i < n$的d[i][i]，由于在整个计算中没有使用，因此不予计算。当然，如果计算 d[i][i]，那么程序将更简单。

算法如下：

```
d[n][n]
p[n] //存储所有的点。
pre[n][n] = {-1}//保存计算信息，用于构造路径。
```

```
按x坐标对p中的点进行排序。
d[2][1] = distance(p[2], p[1])
for(i从1到n)
    //j < i-1
    for(j从1到i-2)
        d[i][j]=d[i-1][j]+distance(p[i], p[i-1])
        pre[i][j]=i-1
    //j = i-1
    d[i][i-1] = d[1][i-1] + distance(p[1], p[i])
    pre[i][i-1]=1
    for(k从2到i-2)
        tmp = d[k][i-1] + distance(p[k], p[i])
        if(tmp > d[i][i-1])
            d[i][i-1] = tmp
            pre[i][i-1]=k
d[n][n] = d[1][n] + distance(p[1], p[n])
for(k从2到i-2)
    tmp = d[k][n] + distance(p[k], p[n])
    if(tmp > d[n][n])
        d[n][n] = tmp
        pre[n][n]=k
d[n][n]中存储最终结果。

path //存储最短路径的一半
i = n, pre_i = n
j = n, pre_j = n
while(pre[i][j] != -1)
    if(j == i - 1)
        i = pre_i - 1
    将边(p[pre[i][j]], p[j])加入path中
    pre_i = i, pre_j =j
    i=pre[i][j]
```

外循环要n次，两个内循环都是i-2次，因此上面的算法的时间复杂度是$O(n^2)$。通过一个pre 数组记录每次构造d[i][j]的值所使用的d[k][j]的k。通过这个数组可以构造出整个路径的一半，也就是从n到1的路径，另一半可以根据双调性很容易的构造出来。构造路径的时间复杂度是$O(n)$。

参考原文如图2

5 Problem 15-6 checker

用一个n×n的二维数组c表示checkerboader。那么，对于格子c[i][j]，checker只可能从c[i][j-1]，c[i-1][j-1]和c[i+1][j-1]三个格子移动过

Figure 2 双调旅行商问题

15-1 Bitonic Euclidean Traveling Salesman Problem

1. Sort the points using an O(nlogn) sort from the smallest to largest x-coordinate. Denote them as $P_1, P_2, \ldots P_n$.

2. Define a bitonic walk from P_i to P_j as a walk that starts from P_i, goes strictly from right to left to P_1, then goes strictly left to right to P_j, and passes every points between P_1 and $P_{\max(i,j)}$ exactly once. Denote $d(i,j)$ as the shortest bitonic walk from P_i to P_j. Due to symmetry, we only consider the situation that $i \ge j$.
Also use $\Delta(i,j)$ to denote the straight-line distance from P_i to P_j.

3. We need to compute $d(n,n)$. Look at the last two connected points in the shortest bitonic walk from P_n to P_n. It can go from P_1 to P_n, or P_2 to P_n, …, or P_{n-1} to P_n. Then

$$d(n,n) = \min\{d(n,1)+\Delta(n,1), d(n,2)+\Delta(n,2), \ldots, d(n,n-1)+\Delta(n,n-1)\}$$

4. To compute $d(i,j)$ for $i \ge j$, the base case is $d(2,1) = \Delta(2,1)$.
i) if $j < i-1$, the edge (P_i, P_{i-1}) must be in the shortest bitonic walk from P_i to P_j.
Then $d(n,n) = d(i-1,j) + \Delta(i,i-1)$
ii) if $j = i-1$. The similar idea in 3 applies on this case, but we look at he first two connected points in the shortest walk. It can be from P_i to P_1, or P_i to P_2, …, or P_i to P_{i-2}.
Then $d(i,i-1) = \min\{d(1,i-1)+\Delta(i,1), d(2,i-1)+\Delta(i,2), \ldots, d(i-2,i-1)+\Delta(i,i-2)\}$
Note that $d(1,i-1) = d(i-1,1)$ …
In order to track the shortest bitonic walk from P_n to P_n. We use $\pi(i)$ to record how we get the value $d(i,i-1)$

来。用一个二维数组b表示到达每个格子所能获得的最大钱数。b[i][j]表示checker到达c[i][j]时所能获得的最大钱数。对每一个格子进行编号，用i*n+j表示，编号之后便于查找p(x,y)的值。递推方程如下：

$$\begin{aligned} b[i][j] = max(&b[i-1][j] + p((i-1)*n+j, i*n+j) \\ &, b[i-1][j-1] + p((i-1)*n+j-1, i*n+j) \\ &, b[i-1][j+1] + p((i+1)*n+j+1, i*n+j)) \end{aligned}$$

算法如下：

```
maxValue = -1
b存储checker到达每个格子所得到的最大值。
for(i=1; i < n; ++i)
    for(j=0; j < n; ++j)
        //对于超出边界的点，不与考虑。
        //但是，也可以在checkerboader的周围设置一个“围墙”。
        //对与围墙上的点,p(x,y)总是负无穷大，这样在下面的代码
        //中就不需要判断格子是否超出边界。
        b[i][j] = max(b[i-1][j]+p((i-1)*n+j, i*n+j)
                     ,b[i-1][j-1]+p((i-1)*n+j-1, i*n+j)
                     ,b[i-1][j+1]+p((i+1)*n+j+1, i*n+j));
```

```
for(i=0; i < n; ++i)
    if(maxValue < b[n-1][i])
    maxValue = b[n-1][i]
```

外层循环和内层循环分别是n-1次和n次，因此算法的复杂度是$O(n^2)$。