算法作业3

丁元杰 17231164

2019 年 11 月 24 日

1 行列均衡问题

算法思路

尝试构造贪心算法。

记此矩阵A的第i行j列的元素为a_{ij}，第i行的元素和记为

$$r_i = \sum_j a_{ij}$$

同时第j列的元素和记为

$$c_j = \sum_i a_{ij}$$

显然地有：

$$\sum_i r_i = \sum_j c_j = \sum_{ij} a_{ij}$$

观察可知，限制在此矩阵上的操作只能为元素自增，所以记初始行列和的最大值为m，有

$$m = \max_{i,j} \{r_i, c_j\}$$

现在构造性地给出算法。算法首先按照行连续的方式遍历整个矩阵的所有元素，并对每个元素进行相应的修改。记第k次修改后的矩阵为$A^{(k)}$，那么矩阵元素记为$a_{ij}^{(k)}$，对应的行列和为$c_i^{(k)}, r_j^{(k)}$。

算法进行到$a_{ij}^{(k)}$位置时，考察$c_i^{(k)}$和$r_j^{(k)}$。如果记

$$\delta^{(k)} := \min \{m - c_i^{(k)}, m - r_j^{(k)}\}$$

则第k轮的操作就是对当时的$a_{ij}^{(k)}$加上了$\delta^{(k)}$，即：

$$a_{ij}^{(k+1)} = \begin{cases} a_{ij}^{(k)} + \delta^{(k)} & \text{if } k = in + j \\ a_{ij}^{(k)} & \text{else} \end{cases}$$

可以看出，此算法中的一次操作，相当于题目的δ次操作，所以最后的操作总数即为

$$ans = \sum_k \delta^{(k)} = mn - \sum_{ij} a_{ij}$$

下面证明这个算法的正确性：

证 显然地，任何一种满足条件的构造，$mn - \sum_{ij} a_{ij}$是它的一个下界。

首先，在第k轮的时候，整个矩阵始终能够保持循环不变性：

$$r_i^{(k)} \leq m, 1 \leq i \leq n c_j^{(k)} \leq m, 1 \leq i \leq n$$

设在第k轮操作a_{ij}元素。有：

$$\delta^{(k)} = \min\{m - c_i^{(k-1)}, m - r_j^{(k-1)}\}$$

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} + \delta^{(k)}$$

则

$$\begin{aligned}
r_i^{(k)} &= r_i^{(k-1)} + \delta^{(k)} \\
&= r_i^{(k-1)} + \min\{m - c_i^{(k-1)}, m - r_j^{(k-1)}\} \\
&= \min\{r_i^{(k-1)} + m - c_i^{(k-1)}, r_i^{(k-1)} + m - r_j^{(k-1)}\} \\
&= \min\{r_i^{(k-1)} + m - c_i^{(k-1)}, m\} \\
&\leq m
\end{aligned}$$

同理可证

$$c_j^{(k)} \leq m, 1 \leq i \leq n$$

其次，在所有的操作之后，必然有：

$$r_i^{(n^2)} = m, 1 \leq i \leq n c_j^{(n^2)} = m, 1 \leq i \leq n$$

假设此条件不成立，则必然存在一行i^*，其和

$$r_{i^*}^{(n^2)} < m$$

那么由恒等式，

$$\sum_j c_j = \sum_i r_i < mn$$

可知必然存在一列j^*，其和

$$c_{j^*}^{(n^2)} < m$$

然而，在遍历到(i^*, j^*)位置时，此二者必有一个依据算法变得不再成立，且由于操作的单调属性，不可能因为其他操作而重新成立。所以不可能在整个矩阵中找到相应的一行，即原假设成立。

综上所述，此算法能够保证在结束之后，有

$$r_i^{(n^2)} = m, 1 \leq i \leq n c_j^{(n^2)} = m, 1 \leq i \leq n$$

所以此下界为下确界。

伪代码

参见算法1

算法 1 求行列均衡问题

输入: $A[1..n][1..n]$矩阵

输出: 最小的操作方案

```
function CALC(A[1..n][1..n])
    r[1..n] ← 0
    c[1..n] ← 0
    m ← 0
    for do i ∈ [1, n]
        for do j ∈ [1, n]
            r[i] ← r[i] + A[i][j]
            c[j] ← c[j] + A[i][j]
            m ← MAX (r[i], c[j])
        end for
    end for
    for do i ∈ [1, n]
        for do j ∈ [1, n]
            δ ← MIN (m − r[i], m − c[j])
            A[i][j] ← A[i][j] + δ
            r[i] ← r[i] + δ
            c[j] ← c[j] + δ
        end for
    end for
    return A
end function
```

复杂度分析

由伪代码可以看出，此算法的复杂度是$O(n^2)$。

2 数据修改问题

3 最大矩阵问题

算法思路

显然，对于一个答案矩阵，一定满足以下两条性质：

1. 第一列全部是1
2. 其余每一列的1的个数多于0的个数

如若不然，就可以分别进行如下调整：

1. 如果第i行第一个元素不为1，那么将这一行取反，答案一定变得更优：
$$2^m > \sum_{0 \le i < m} 2^i$$
2. 如果第j列有a个1，b个0，且有$a < b$，那么将这一列取反，答案将会增加$(b-a)2^{m-j}$。

接下来，尝试将初始矩阵朝着目标矩阵的形式调整，分两步进行：

1. 首先将首元素不为1的行全部取反
2. 对于每一列，如果1的元素多于0的元素，取反

可以观察出，这种操作方式得到的最终答案必定是唯一的（虽然矩阵不一定唯一）。原因是，整个操作序列必定是先行后列，在对列进行调整的时候不会再对行进行操作。因为如果在首行全为1的情况下调整某行，又要保证在调整之后使得性质1始终成立，则必须将剩余行全部取反之后对第一列取反。这种操作加上最初的行操作相当于对除了第一列的所有列取反。换而言之，不影响性质1的行操作必然会转为列操作，同时性质1只能通过行操作进行保证，从而说明了朝着目标形式矩阵的调整必定是先行后列的。

接下来，行调整和列调整的目的都是清楚、且内部互相独立的，因此调整方法是唯一的。

伪代码

参见算法2。其中，ROW_REVERSE是对行进行翻转的函数；COLUMN_REVERSE则是对列进行翻转的函数。BTOI则是将01数组转化为对应的二进制整数的函数。

复杂度分析

由伪代码可以看出，此算法的复杂度是$O(nm)$。

算法 2 求解此问题

输入: $A[1..n][1..n]$

输出: 最优方案的值ans

```
function BestMatrix(A[1..n][1..m])
    for do i ∈ [1..n]
        if then A[i][1] = 0
            ROW_REVERSE(i)
        end if
    end for
    for do j ∈ [1..m]
        a ← 0
        b ← 0
        for do i ∈ [1..n]
            if then A[i][j] = 1
                a ← a + 1
            else
                b ← b + 1
            end if
        end for
        if then a < b
            COLUMN_REVERSE(j)
        end if
    end for
    ans ← 0
    for do i ∈ [1..n]
        ans ← BTOI(A[i])
    end for
    return ans
end function
```

4 环路问题

算法思路

以任意结点为起点进行深度优先搜索（DFS），并且记录当前栈中的结点。

算法思路

有了基本的性质，就可以设计找出最多不用操作的数的算法了。设两个关联数组$l, r : X \to I$其中X是输入序列的元素集合，I是原序列的指标集。其中$l[x]$表示在所有的x中，下标最小的那一个下标；$r[x]$则表示所有序列中的x中，下标最大的那一个下标。

设A是最终求得的，不用操作的数的集合，那么可以看出A满足几条性质：

1. $\forall x \in X : \min A \le x \le \max A \Rightarrow x \in A$
2. $\forall x, y \in A : x < y \Rightarrow r[x] < l[y]$

其中，第1条性质说明了不操作的数必然是“连续”的一段，这里的连续并不是自然数的连续的含义，而是其中不含有任何其他的X的元素的含义，也即在X的范围中连续。

我们再引入一个映射$\sigma : X \to [0..|X|-1]$和它的逆映射$\tau : [0..|X|-1] \to X$，其中$\sigma(x)$表示在$X$中小于$x$的元素数量。有了这个映射，上段所描述的$x, y$在$X$的范围中连续，就可以转化为$\sigma(x), \sigma(y)$在$\mathbb{N}$的范围中连续。

有了这些记号，我们就能够方便地描述我们的工作。最终所期望的结果，是找出$[0..|X|-1] =: M$的一个最大的连续子集A^*，使得下面的性质成立：

$$\forall x, y \in A^* : r_{\tau(x)} < l_{\tau(y)}$$

现在使用动态规划的思想解决问题，设$f[i]$表示以i结尾的M的子区间中，满足上述条件最长子区间长度。由此可以得到一个简略的状态转移函数：

$$f[i] = \begin{cases} 1 & , i = 0 \\ 1 & , r_{\tau(i-1)} > l_{\tau(i)} \\ f[i-1]+1 & , r_{\tau(i-1)} < l_{\tau(i)} \end{cases}$$

伪代码

参见算法3

复杂度分析

可以从伪代码中看出，在使用 Map 作为有序关联数组时，可以使用基于平衡二叉树实现的关联数组，从而做到单次$O(\log n)$的访问和修改复杂度。上面有n次访问，因此因为关联数组带来的复杂度有$O(n \log n)$。

综上所述，算法的总体复杂度为$O(n \log n)$

算法 3 求解此问题

输入: $A[1..n]$，

输出: 最优方案的值ans

```
function MinOperation(A[1..n])
    f[0..n − 1]
    l ← Map⟨int, int⟩
    r ← Map⟨int, int⟩
    for i = 1 → n do
        l[A[i]] ← min {l[A[i]], i}
        r[A[i]] ← max {l[A[i]], i}
    end for
    m ← l.size()
    ll[0..m]
    rr[0..m]
    i ← 0
    for x ∈ l do
        ll[i] ← x
        i ← i + 1
    end for
    i ← 0
    for x ∈ r do
        rr[i] ← x
        i ← i + 1
    end for
    f[0] ← 1
    ans ← 0
    for i = 1 → m − 1 do
        if rr[i − 1] < ll[i] then
            f[i] ← f[i − 1] + 1
        else
            f[i] = 1
        end if
        ans ← max {ans, f[i]}
    end for
    return ans
end function
```

5 能耗降低问题

算法思路和复杂度分析

在k小于所有字符串中1的个数时，我们无需将任何0变成1，因此期望每一次翻转都对答案有正向的贡献；反之，如果k太大，那么答案可以直接记为$k-\#1$。所以我们现在只考虑消除1的情形。

考虑到每一个串在消除掉一定量k时，其产生的能耗是一定的。我们可以预处理出每一个串在消除j个1之后节省的能耗。如果第i个串有s_i个1，那么可以把这个串拆成s_i件物品，其中第j个物品表示第i个串去掉j个1之后，节省的能耗。这样，以去掉的1的个数为体积，k为容量，原问题被转化为了若干个类型的分组背包。

由于分组背包有着完整的、已解决的实现，我们这里着重关注如何将每一个字符串拆成若干个类型相同的物品。

对于第i个字符串，可以抽取一个单调的指标序列A_i，并且有$s_i = |A_i|$。其中，A_i储存了S_i字符串的所有1出现位置的下标。现在用w_{ij}表示第i类的第j个物品的价值，含义是字符串S_i去掉j个1之后节省的能耗，其体积为j。根据定义

$$w_{ij} = w_{i0} - \max_{0\le k\le j} A_i[k+s_i-j] - A_i[k+1] + 1$$

不难看出，为了计算w_{ij}需要花费$O(m^2)$的时间。因此，为了初始化得到所有的物品，需要花费$O(nm^2)$的时间。

最后，我们得到了n组物品，其中每组物品最多m件，总容量为k，由分组背包的时间复杂度可以知道，用于动态规划的时间是$O(nmk)$。

综上，总体的时间复杂度

$$T(n,m,k) = O(nmk + nm^2)$$

伪代码

参见算法4

算法 4 求解此问题

输入: $S[1..n][1..m], k$,

输出: 最优方案的值ans

```
function MinCost(S[1..n][1..m], k)
    w[1..n][1..m]
    p[1..n]
    for i = 1 → n do
        A ← []
        for j = 1 → m do
            if S[i][j] = '1' then
                A.append(j)
            end if
        end for
        s ← A.size()
        p[i] ← s
        w[i][0] ← max A − min A
        for j = 1 → s do
            w[i][j] ← 0
            for l = 0 → j do
                w[i][j] ← max {w[i][j], w[i][0] − (A[k + s − j] − A[k + 1] + 1)}
            end for
        end for
    end for
    return MultiPack(w, p)
end function
```