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：

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

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

则

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

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

```
1:  function CALC(A[1..n][1..n])
2:      r[1..n] ← 0
3:      c[1..n] ← 0
4:      m ← 0
5:      for i ∈ [1, n] do
6:          for j ∈ [1, n] do
7:              r[i] ← r[i] + A[i][j]
8:              c[j] ← c[j] + A[i][j]
9:              m ← MAX (r[i], c[j])
10:         end for
11:     end for
12:     for i ∈ [1, n] do
13:         for j ∈ [1, n] do
14:             δ ← MIN (m − r[i], m − c[j])
15:             A[i][j] ← A[i][j] + δ
16:             r[i] ← r[i] + δ
17:             c[j] ← c[j] + δ
18:         end for
19:     end for
20:     return A
21: 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

算法思路

可以考虑先将所有的操作预先施加在数组上，再依次考虑删除某一个操作对最后答案带来的影响，在这种意义下，我们可以将操作看作是区间的减法$[l, r, -x]$。这个操作对整个序列的最大值有影响，当且仅当区间$[l, r]$包含了序列中的所有最大值。对于此条件的快速判断，可以通过预先处理最左位置的最大值下标和最右位置的最大值下标解决。

记最大值为m，设最左位置的最大值下标为l_m，最右位置为r_m。

接下来预处理两个数组$\{p_n\}$，以及$\{s_n\}$，分别定义为：

$$p_i = \max_{j \le i} \{a_i\}$$

$$s_i = \max_{j \ge i} \{a_i\}$$

是a_n序列的前缀和后缀最大值数组。

由此，操作$[l_i, r_i, -x_i]$对最后答案的影响分情况：

1. 如果$[l_m, r_m] \nsubseteq [l_i, r_i]$，那么它对答案没有影响
2. 否则，去掉这一个操作之后，新的答案变为$ans_i = \max\{p_{l-1}, s_{r+1}, m - x_i\}$。

对所有的满足情况2的询问计算ans_i，所有的ans_i的最小值即是答案。

伪代码

参见算法2

复杂度分析

由伪代码可以看出，此算法的复杂度是$O(n + Q)$。

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：

算法 2 数据修改问题

输入: $a[1..n]$序列，$l[1..Q]$，$r[1..Q]$，$x[1..Q]$

输出: 答案

```
1: function CALC(A[1..n][1..n], l[1..Q], r[1..Q], x[1..Q])
2:     pre[0..n] ← INT_MIN
3:     suf[1..n + 1] ← INT_MIN
4:     for i ∈ [1, n] do
5:         pre[i] ← MAX(pre[i − 1], a[i])
6:         suf[n + 1 − i] ← MAX(pre[n + 2 − i], a[n + 1 − i])
7:     end for
8:     l_m ← 1
9:     r_m ← 1
10:    for i ∈ [1..n] do
11:        if a[i] > a[l_m] then
12:            l_m ← i
13:            r_m ← i
14:        else if a[i] = a[l_m] then
15:            r_m ← i
16:        end if
17:    end for
18:    ans ← a[l_m]
19:    m ← a[l_m]
20:    for i ∈ [1..n] do
21:        if l[i] ≤ l_m and r[i] ≥ r_m then tmp ← MAX(pre[l[i]−1], suf[r[i]+1], m−x[i]) ans ← MIN(ans, tmp)
22:        end if
23:    end for
24:    return ans
25: end function
```

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3。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$O(nm)$。

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），并且记录每个结点的已访问情况（未访问、已访问），以及使用栈记录当前栈中的结点。

DFS到达一个结点u时，即将u压入栈中，并将其记录为已经访问过，接下来向着与u相连的结点v进行扩展：

1. 如果v还没有被访问过，则立即访问v，并进行递归

2. 如果v已经被访问过，那么v必然在栈中，此时连续弹栈，直至栈中的v被弹出，即可得到一个环

在实现的过程中，应当注意从u向外继续拓展的时候，不能够向栈的次顶（即访问u之前访问的结点）进行拓展。

至于为何第一次访问到已访问结点v，之一定在栈中，可以考虑之前访问到v时的情况。由于v已经完成访问，并且已经弹出了栈，说明从v出发的所有不与$\langle s, v\rangle$相交的结点都已经完成了访问，这其中自然包括当前的元素u。由此可以推出矛盾。

算法 3 求解此问题

输入: $A[1..n][1..n]$

输出: 最优方案的值ans

```
1:  function BestMatrix(A[1..n][1..m])
2:      for i ∈ [1..n] do
3:          if A[i][1] = 0 then
4:              ROW_REVERSE(i)
5:          end if
6:      end for
7:      for j ∈ [1..m] do
8:          a ← 0
9:          b ← 0
10:         for i ∈ [1..n] do
11:             if A[i][j] = 1 then
12:                 a ← a + 1
13:             else
14:                 b ← b + 1
15:             end if
16:         end for
17:         if a < b then
18:             COLUMN_REVERSE(j)
19:         end if
20:     end for
21:     ans ← 0
22:     for i ∈ [1..n] do
23:         ans ← BTOI(A[i])
24:     end for
25:     return ans
26: end function
```

真代码

```
#include <bits/stdc++.h>

using namespace std;

struct edge {
    int to;
    edge *nxt;
} E[MAXM<<1], *node[MAXN];

int N, M;
bool usd[MAXN];

void add_edge(int f, int t) {
    E[cne].to = f;
    E[cne].nxt = node[f];
    node[f] = E+cne++;
}

void readin() {
    scanf("%d%d", &N, &M);
    int a, b;
    for(int i = 1; i <= M; ++i) {
        scanf("%d%d", &a, &b);
        add_edge(a, b);
        add_edge(b, a);
    }
}

stack<int> stk;

int dfs(int x, int f) {
    usd[x] = true;
    stk.push(x);
    for (edge *i = node[x]; i; i = i->nxt) {
        int now = i->to;
        if (now != f) {
            if (usd[now]) {
                int y;
                do {
```

```
40                      y = stk.top();
41                      stk.pop();
42                      printf("%d\n", y);
43                  } while(y != now);
44                  exit(0);
45              } else {
46                  dfs(now);
47              }
48          }
49      }
50      stk.pop();
51  }
52
53  void process() {
54      dfs(1, 1);
55  }
56
57  int main() {
58
59      readin();
60      process();
61
62      return 0;
63  }
```

复杂度分析

由于每个节点最多被访问一次，每条边最多被尝试两次，所以总体的时间复杂度为$O(V+E)$。

5 最小生成树问题

算法思路和复杂度分析

收到 Kruskal 的启发，优先对边长为1的边进行连接，紧接着时边长为2的。首先只考虑图中边长为1的边，全部的点被这些边划分为了若干互相不连通的连通块。假设这些连通块的个数为n个，那么它们一定被若干边长为2的边连成一个连通图。由于连接n个连通块所需的最小边数为$n-1$，所以最后一共有$n-1$条长度为2的边，以及$(|V|-1)-(n-1)=|V|-n$条长度为1的边。最后的最小生成树总边长$|V|-n+2(n-1)=|V|+n-2$。

真代码

```
#include <bits/stdc++.h>

using namespace std;

struct edge {
    int to, v, id;
    edge *nxt;
} E[MAXM<<1], *node[MAXN];

int N, M;
int usd[MAXN], belong[MAXN], vst[MAXN];

void add_edge(int f, int t, int v, int id) {
    E[cne].to = f;
    E[cne].nxt = node[f];
    E[cne].v = v;
    E[cne].id = id;
    node[f] = E+cne++;
}

void readin() {
    scanf("%d%d", &N, &M);
    int a, b, v;
    for(int i = 1; i <= M; ++i) {
        scanf("%d%d%d", &a, &b, &v);
        add_edge(a, b, v, i);
        add_edge(b, a, v, i);
    }
}

vector<int> ans;

void dfs1(int x, int label) {
    usd[x] = 1
    belong[x] = label;
    for (edge *i = node[x]; i; i = i->nxt) {
        int now = i->to;
        if (i->v == 1 && usd[now] == 0) {
            ans.push_back(i->id);
            dfs1(now, label);
        }
```

```
42      }
43  }
44
45  void dfs2(int x) {
46      usd[x] = 2;
47      vst[belong[x]] = true;
48      for (edge *i = node[x]; i; i = i->nxt) {
49          int now = i->to;
50          if (i->v == 1 && usd[now] == 1) {
51              dfs2(now);
52          } else if (i->v == 2 && !vst[belong[now]]) {
53              dfs2(now);
54              ans.push_back(i->id);
55          }
56      }
57  }
58
59  void process() {
60      for (int i = 1; i <= N; ++i) {
61          if (usd[i] == 0) {
62              dfs1(i, i);
63          }
64      }
65      dfs2(1, 1);
66  }
67
68  int main() {
69
70      readin();
71      process();
72
73      return 0;
74  }
```