第五次作业 NF

姓名：梁付槐　　学号：2018Z8013261003

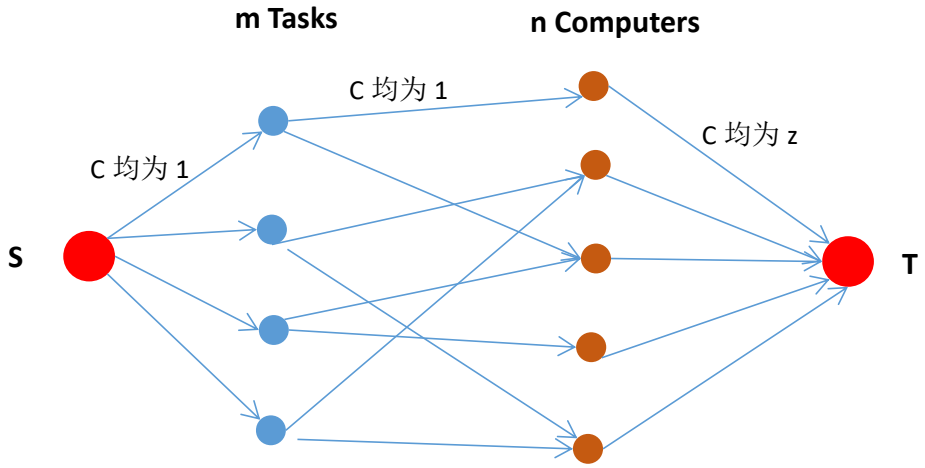
题目一：Load balance

基本思路：

由题意知有 m 个任务和 n 台计算机，每个任务 a 可以由 2 台计算机来做。我们可以把每个任务和其对应的两台电脑建立连边（容量为 1），然后建立一个超级源点 S 指向 m 个任务（这就有 m 条有向边，然后每条边的容量为 1），最后把 n 台计算机连接向一个超级汇点 T，设它们的容量都为 z。

这里的 z 为 n 台中负载最大的计算机，而题目要求我们要最小化负载最大的值，我们可以二分搜索来“猜测”最小可行的 z，而显然 z 的范围为[1,m]，这样可以运行 log(m)次最大流就可以得到答案。

图示：



伪代码：

```
def min_load():
    for each task:
      add edge with capacity 1 to the two PC which can solve them.
    add a node s and t
    add edge from s to each task with capacity 1
    L = 1, R = m + 1 # m is the task
    add edge from each PC to t with capacity (L+R) / 2
    while L < R:
        mid = (L+R) / 2
        change capacity (all edge from PC to t) to mid
        if max_flow(s,t) == m:
            R = mid
        else:
            L = mid + 1
    return L
```

算法的正确性：

首先，z 为 PC 到 T 的容量上限，这个值就限制了我们的最大负载是多少，换句话说，PC 的最大负载不能超过 z。而我们用二分搜索来猜测 z 的值，然后运行最大流。若 T 能收到的流为 m，说明我们这个 z 是偏大的，我们可以继续缩小它，若不能收到 m，说明可以增加负载，直至找到最优的 z。因此，我们的算法是正确的。

算法的复杂度：

本算法需要 log(m)次最大流的时间，而最大流的时间取决于所采用的最大流算法。如果使用 Dinitz 算法的话，总时间复杂度就是 $O(MN^2 \log m)$，其中 M=m+2m+n 为边数，N=m+n+2 为节点的数量。

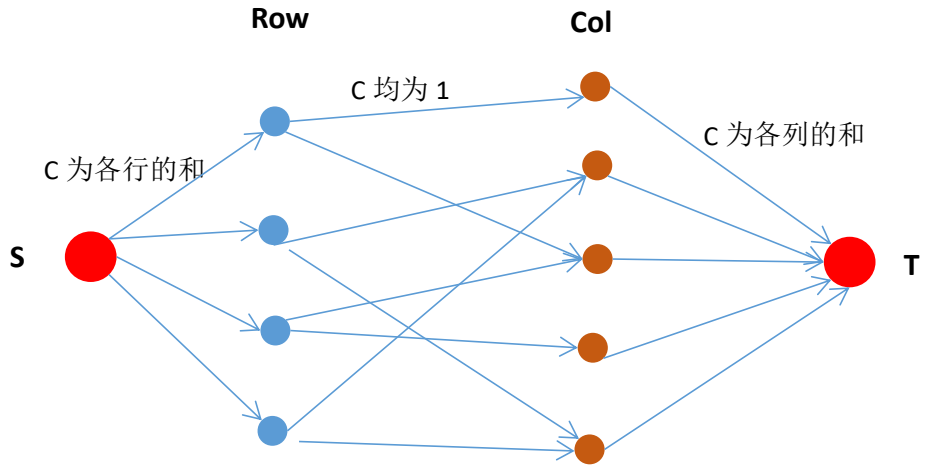
题目二：Matrix

基本思路：

每一行作为结点，这样有 m 个行结点，然后每一列也作为结点，这样有 n 个列结点。建图，把 m 个行结点和 n 个列结点相连（每个都要，一共 m*n 条），容量为 1。然后建立一个超级源点 S 指向 m 个行结点，容量为对应行的和；最后把 n 个列结点连接到汇点 T，而容量为对应的列的和。

然后跑最大流算法。当最大流算法结束后，若原问题有解，所得的最大流必等于之前各行的和等于之前各列的和。在跑完最大流算法后，设行结点 i 到达列结点 j 的边的容量为 x,那么矩阵 a[i][j] 的值就是 1- x。

图示：



算法的正确性：

将行的点和列组成的点建立连边（一共 m*n 条），容量为 1，就是该矩阵最大的取值，就是限制了从一行流向对某列的值最多为 1。这样建立连边也相当于若原矩阵 a[i][j] = 1,那么第 i 行会向第 j 列贡献 1。然后源点向各行建立连边，容量为对应的行的和 row[j]，这样限制了行的最大值。同理各列向汇点建立连边，容量为对应的列的和 col[j]，这样限制了列的最大值。

因此，若最后都是流量等于之前各行以及各列和，原问题有解。而且这些边一定是满流的，这样才能使得总和相等，且 a[i][j]的值就是从第 i 行流向第 j 列的流量（也等于 1-该边的容量）。

伪代码：

```
def test(m, n, row, col):
    s = 0
    t = m + n + 1
    for i in range(1, m + 1):  # link source s to [1,m]
        add_edge(0, i, row[i - 1])
        for j in range(m + 1, m + n + 1):  # row link to column
            add_edge(i, j, 1)

    for i in range(m + 1, m + n + 1):
        add_edge(i, t, col[i - m - 1]) # link column to sink t

    max_flow(m, n, s, t)

    matrix = [[0 for _ in range(n)] for _ in range(m)]
    for i in range(1, m + 1):
        for edge in g[i]:
            if edge.to != 0:
                matrix[i - 1][edge.to - m - 1] = 1 - edge.cap
    return matrix
```

算法的复杂度：

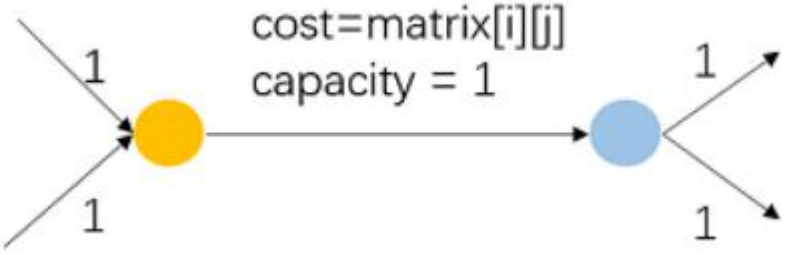
本题时间复杂度就是其最大流的时间。

如用 Dinitz 的话，最大流的时间为$O(MN^2)$，其中 N=m+n+2, M = m+mn+n.

题目三：Problem Reduction

基本思路：

矩阵已经给定了所需要的花费，因此我们只需要建图。建图方法为：首先把矩阵中每个点和其右方、下方的两个点建立连边（容量为 1，费用为 0），然后对于每个元素，拆分成两个结点，以 matrix[i][j]为例,拆分成两个结点 a 和 b，他们之间的连边为容量 1（左上角和右下角的两个拆分容量为 2），费用就是 matrix[i][j]。如下图而题目从左上角到右下角，然后从右下角再到左上角的最小花费等价于从左上角到右下角走两次的最小花费。因此，相当于流为 2 的最小费用流。



伪代码：

```
build_graph_from_matrix(matrix) # 根据上面的方法建图
solve_min_cost_flow(2) #求最流为2的最小费用流
```

算法的正确性：

首先建图的时候，只向右边、下方的两个点建立容量为 1 的边，保证了方向是向右或者向下的，且容量为 1 保证只走一次。接着把点进行拆分为两个，由于这两个点之间的边的容量为 1，因此，它们不会被重复走。而对于左上角和右下角的两个点，我们拆分的两个点的容量为 2，因为要出发和到达 2 次。拆分时两个结点边的花费就是对应点的权重，代表了走这条路的费用。然后从左上角走到右下角再返回左上角，其实和再次从左上角到右下角是一样的。因此，我们求从左上角 S 到右下角 T 流为 2 的最小费用流算法即可。

算法的复杂度：

前面分析出本题求流为 2 的最小费用流，因此若用 Bellman_Ford 求最短路，时间复杂度为 O(2VE)也就是 O(VE)；如果采用二叉堆的 Dijkstra，时间复杂度为 O(2ElogV)也就是 O(ElogV)。