2011937 信息安全 姜志凯

基于网络最大流的资源最大化输出

一、问题描述

假设有 m 份资源，每份资源的资源含量可以相同，也可以不同，且一份资源只对拥有它的使用权的使用者开放。

假设有 n 个使用者，每个使用者可以拥有多份资源的使用权，同一份资源也可能被不同的人占有，且一个使用者在取用资源时，可以对其能支配的多份资源进行资源重新分配。

n 个使用者，每人有一定的资源需求量，按照一定次序取用资源，计算最终所有人能取得的资源和的最大值。

直白的描述有些抽象，让我们将它放到真实的场景加深对问题的理解。

假设你在一家银行工作，你的任务是帮助客人取走装在保险柜里的金币，每个保险柜里的金币数量随意。

银行有 m 个保险柜，每个都上了锁，你无法打开，钥匙在客人手里。一个客人可能有多个保险柜的钥匙，一个保险柜的钥匙也可能被多个顾客拥有。某天，你被告知有 n 个顾客会按顺序来到银行取金币，并且你也知道，每位顾客将要取多少枚金币，每位顾客来到银行，会打开所有他能打开的保险柜，然后从中取他需要数量的金币，然后关上离开。若顾客在能打开的所有保险柜里未能得到需求数量的金币，则他会非常生气，并投诉你，你当然不想这种事情发生。

一个补救的办法就是，在顾客取金币时，你可以偷偷地对其打开的保险柜中的金币进行重新分配。比如，一个顾客打开了保险柜 1 和 2，你可以将 1 中的部分金币移动到 2 中，这样说不定下一个拥有保险柜 2 钥匙的顾客就能取到更多的金币了。

总之，你的目的就是尽可能地让顾客取到更多的金币，尽管可能还是无法满足所有人的需求。

理解题目后，首先想到的就是**网络最大流问题**，关键在于**如何建模**，建完模后就可以利用 **Dinic 算法**来计算最大流了。

金币从保险柜中流出，流到每个顾客手中，可以建立虚拟金币源点，往外输送金币，每条边的容量值为各个保险柜里的金币数；建立虚拟汇点，接受金币，每条边容量值为顾客需求的金币数；中间节点可以包含各保险柜和各顾客，也可以只包含顾客，算法复杂度不同，后面会介绍。

大体框架有了，这道题不同之处在于，每位顾客取完金币后，可以对打开的保险柜中的金币数量进行调整。因此，拥有共同保险柜的两个人，后来者可以间接拥有先前的人所拥有的所有保险柜的使用权，即可以从中随意获取金币，也就是可以从先前来的人身上任意获取金币，下面用具体例子解释：

{

顾客 1 先来了，打开了保险柜 1 和保险柜 2

保险柜 1 --> 顾客 1

保险柜 2 --> 顾客 1

此时可以任意调整保险柜 1、2 中的金币数量，关闭保险柜

然后顾客 2 来了，打开了保险柜 2

保险柜 2 --> 顾客 2

由于你事先就知道顾客 2 的需求，所以在之前调整保险柜 1、2 金币数量的时候，你可以按照顾客 2 的需求调整，尽量满足他的需求，所以相当于顾客 2 有了保险柜 1、2 的使用权

保险柜 1、2 --> 顾客 2

（其实这时保险柜 1、2 相当于合并成一个保险柜了）

};

有了以上前提，就可以进行算法分析设计并进行建模啦！

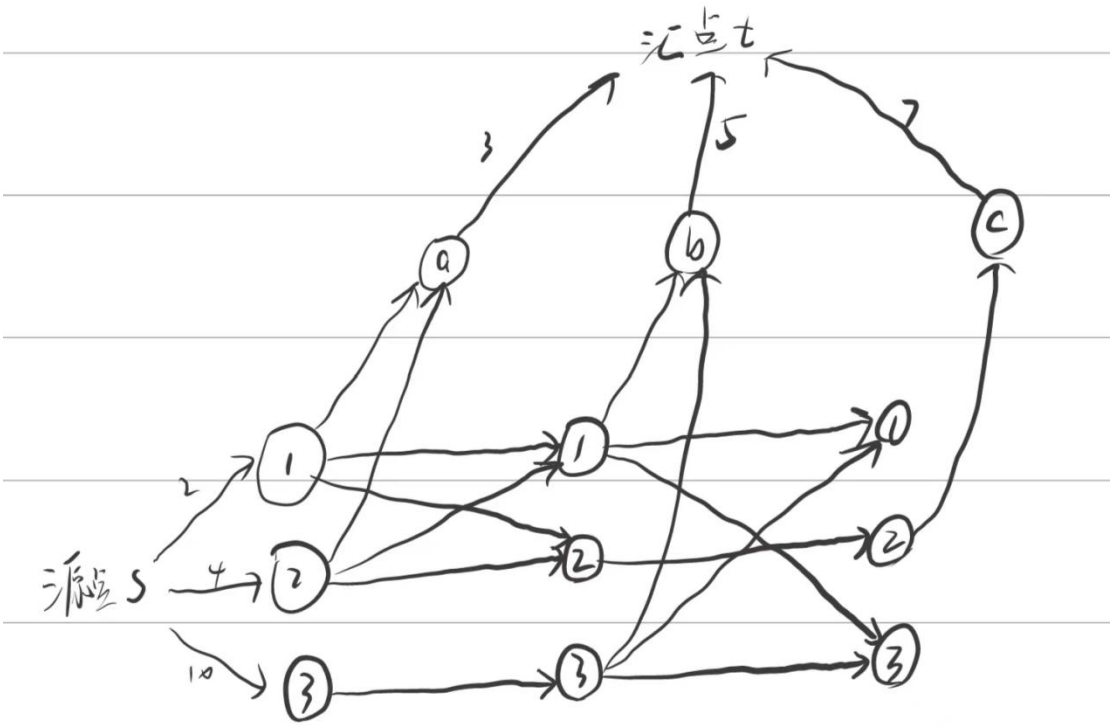
二、算法分析

显然，解决问题需要分为两步，一是根据问题建立网络图，二是根据网络图求最大流，下面分别介绍。

1、建模

首先想到的肯定是为每个保险柜和每位顾客都建立一个节点，再额外建立一个**虚拟源点 s** 和一个**虚拟汇点 t**，用来输出和接受金币。由**源点 s** 向**各个保险柜**建容量为**金币数量的边**，**各保险柜**向拥有钥匙的**顾客**建容量为**无穷的边**，**各顾客**向**汇点 t** 建容量为**金币需求量的边**，每一个顾客取完金币后，**各保险柜向下一轮的自己连一条无穷的边**，且在一轮中**同时打开的各保险柜**，也要向下一轮**互连一条无穷的边**（因为同时打开，金币数量可以互调）（有多少顾客，就有多少层保险柜节点，每一个顾客对应一层保险柜，向该层节点索取流值）。

假设有 3 个保险柜 1、2、3，分别装有金币数 2、4、10，有三个顾客 a、b、c，分别需要 3、5、7 枚硬币，a 持有 1、2 的钥匙，b 持有 1、3 的钥匙，c 持有 2 的钥匙，则根据上述方法，可以建立如下的网络图（所有未标容量的边都是无穷边）：



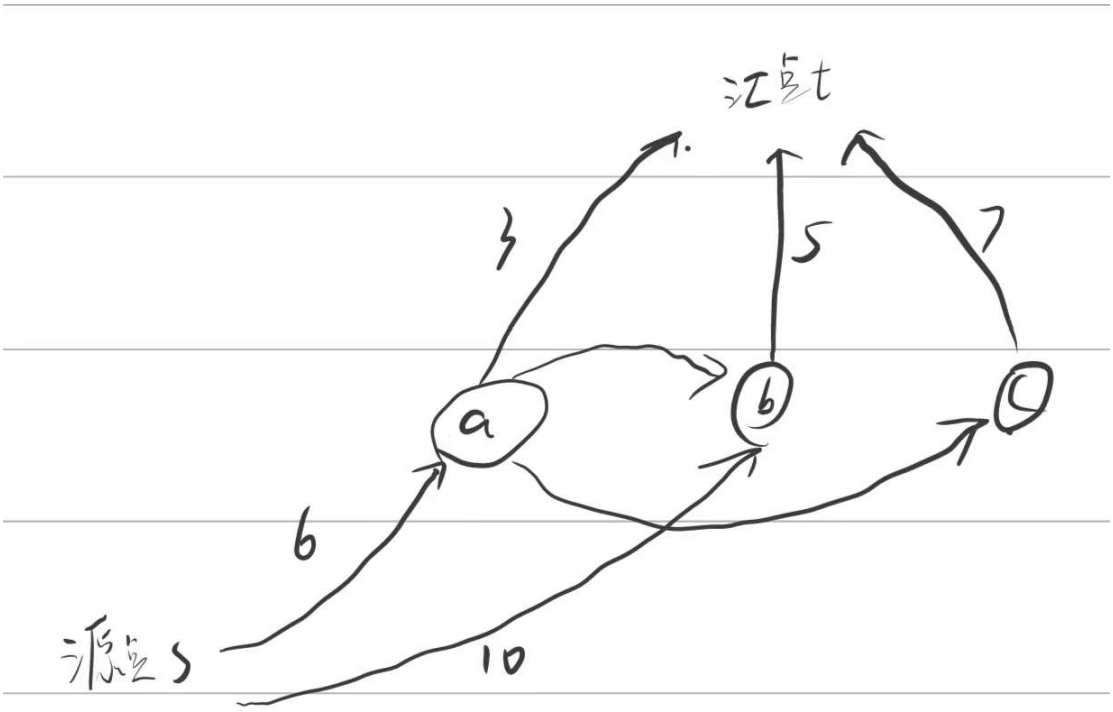
这样当然能解决问题，但显然不是最优的，因为**冗余节点**太多，而网络图最大流算法 **Dinic** 的时间复杂度为 **O(mn^2)**，与节点数密切相关（后边会讲到复杂度，m 为边的数量，n 为节点数），若有 5000 名顾客，500 个保险柜，则节点数 n=2+N+M×N=2505002，非常大，对效率的影响不言而喻，这时可以考虑**将冗余的点去掉**，来减少节点数，提高效率。

可以参考以下规律：

①两个节点之间由无穷边相连，且接收节点无其他流来源，则这两个节点相当于一个；

②若几个节点的流值来源或者去向完全相同，则可以将它们合并。

根据以上规则，反复利用，重新建模，如下图（未标容量即为无穷）：



这个图可以这样理解，我们可以将**每个顾客作为一个节点**。对于打开每个保险柜的**第一个**顾客，从源点向他连一条边，容量就是该保险柜里的初始金币数。如果从源点到一名顾客有**多条边**，则可以把它们**合并成一条**，容量相加。每个保险柜只与**第一个打开它的顾客相连**，之后打开该保险柜的顾客**向之前打开的顾客索取**，即同一个保险柜有多个顾客打开的话，将顾客**按顺序**用**容量无穷的边**连上。

这个模型中节点个数仅为 n=2+N=5002，大大简化了 Dinic 算法的计算量，提高了效率。

伪代码：

```
for(int i=1,x,y;i<=n;i++)//循环 m 个顾客
{
    scanf("%d",&x);//第 i 个顾客有 x 个钥匙
    for(int j=1;j<=x;j++)
    {
        scanf("%d",&y);//y 即钥匙编号
        p[y].push_back(i);//顾客 i 有保险柜 y 的使用权，不定长数组存储
    }
    scanf("%d",&x);//顾客的需求量
    add(i,t,x),add(t,i,0);//顾客点到汇点建边
}
for(int i=1;i<=m;i++)
{
    int l=p[i].size();
    for(int j=0;j<l;j++)
    {
        if(j==0)//买第 i 个猪圈的第一个用户
        {
            num[p[i][j]]+=pig[i];//容量累加
```

```
            continue;
        }
        add(p[i][j-1],p[i][j],Inf);//开同一个保险箱的客户连边
        add(p[i][j],p[i][j-1],0);
    }
}
for(int i=1;i<=n;i++)
    if(num[i]!=0)//以累加好的结果为容量给开某个保险箱的第一个顾客建边
        add(s,i,num[i]),add(i,s,0);
```

2、Dinic 算法

建完网络图了，直接跑 Dinic 即可得出最大流，下面简单介绍一下 Dinic 算法。

与其他网络最大流算法一样，Dinic 算法也是不断**寻找增广路径**并不断更新（如书中的**Ford-Fulkerson 算法**）。

伪代码：

```
Max-Flow
对 G 中的所有的 e 初始化 f(e)=0
While 在增广图 Gf 中存在一条 s-t 路径
  令 P 是 Gf 中的一条简单路径
  f'=augment(f,P)
  把 f 更新为 f'
  把增广图 Gf 更新为 Gf'
Endwhile
Return f
```

但 dinic 算法新加入了一个设置分层图的方法，通过分层避免了增广路径因为反向边的添加后 dfs 时的盲目性导致增广效率低下的问题。

（1）分层思想

Dinic 算法在每次增广前，先用 BFS 来将图分层。设源点的层数为 0，那么一个点的层数便是它离源点的最近距离。

通过分层达到以下目的：

- 如果不存在到汇点的增广路（即汇点的层数不存在），即可停止增广。
- 确保我们找到的增广路是最短的（每次找增广路的时候，都只找比当前点层数多 1 的点进行增广）。

（2）优化

- Dinic 算法使用 DFS 找增广路，这给了我们使用一次 DFS 找到多条增广路并增广的机会（见代码），大大提高了算法的效率。

• 由于上述多路增广操作，对于任意节点，我们每增广它的一条前向弧，意味着这条弧后所有边都被我们多路增广过了，那么当我们再次处理该节点时，就可以不用考虑这条弧。所以，我们下一次进行增广的时候，就可以不必再走那些已经被增广过的边。

伪代码：

```
inline bool bfs()
{
   memset(d,0,sizeof(d));
   queue<int> q;
   d[s]=1,q.push(s);
   while(!q.empty())
   {
    int x=q.front();q.pop();
    for(int i=head[x];i;i=Next[i])
    {
        int y=ver[i];
        if(edge[i]==0 || d[y]!=0)
            continue;
        d[y]=d[x]+1;
        q.push(y);
    }
   }
   if(d[t]==0) return false;
   return true;
}
int dfs(int x,int incf)
{
   if(x==t)
        return incf;
   for(int &i=cur[x];i;i=Next[i])
   {
        int y=ver[i],z=edge[i];
        if(z==0 || d[y]!=d[x]+1)
            continue;
        int dist=dfs(y,min(incf,z));
        if(dist>0)
        {
            edge[i]-=dist;
            edge[i^1]+=dist;
            return dist;
        }
   }
   return 0;
}
int main()
```

```
{
    while(bfs())//Dinic 求解
    {
        for(int i=1;i<=t;i++)
            cur[i]=head[i];
        while(int di=dfs(s,Inf))
            maxflow+=di;
    }
}
```

三、算法有效性论证

· 建模

这个建模就是在保证所有顾客都可以从自己持有钥匙的保险柜中取金币的情况下进行的优化，所以所有顾客都可以支配自己拥有的保险柜，所以无疑是有效的。

· 网络最大流算法

首先介绍 s-t 割的概念：把一个图的节点划分为两部分 A 和 B，使得源点 s 属于 A，汇点 t 属于 B，一个割的容量记为 C(A,B),是从 A 中流出的所有边的容量之和。

容易得出，$v(f) = f^{out}(A) - f^{in}(A)$

而$f^{out}(A) = C(A,B)$，所以 $v(f) <= f^{out}(A) = C(A,B)$

所以当 B 到 A 没有流值，即$f^{in}(A) = 0$ 时，$v(f) = C(A,B)$，f 有 G 中任何流的最大值。

而 Ford-Fulkerson 算法停止的条件为剩余图中不存在 s-t 路径，下证：若 f 使得剩余图中不存在 s-t 路径，则在 G 中定存在一个 s-t 割，使得 v(f) = C(A,B)。

首先我们证明(A^*,B^*)的确是s-t割. 显然它是V的划分. 源点s属于A^*,因为总存在一条从s到s的路径. 此外,$t\notin A^*$,这是由于在这个剩余图中不存在s-t路径的假设;因此$t\in B^*$正如所需.

接着,假设$e=(u,v)$是G中一条边使得$u\in A^*$且$v\in B^*$,如图7.5所示. 我们断言$f(e)=c_e$. 因为如果不是,e将是剩余图G_f中一条前向边,由于$u\in A^*$,在G_f中存在一条s-u路径;把e接到这条路径上,我们将得到G_f中一条s-v路径,与我们假设$v\in B^*$矛盾.

现在假设$e'=(u',v')$是G中的一条边使得$u'\in B^*$且$v'\in A^*$. 我们断言$f(e')=0$. 因为如果不是,e'将产生剩余图G_f中一条后向边$e''=(v',u')$,且由于$v'\in A^*$,在G_f中存在一条s-v'路径;把e''接到这条路径上,我们将得到G_f中一条s-u'路径,与我们假设$u'\in B^*$矛盾.

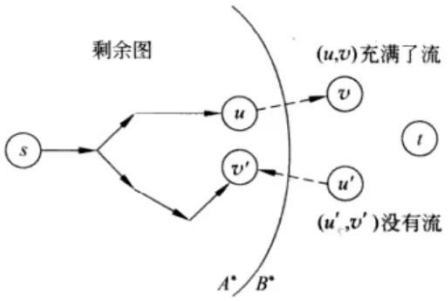


图7.5 在定理7.9证明中的(A^*,B^*)割

因此所有从A^*出来的边完全充满了流,而所有进入A^*的边则完全没有用过. 我们现在可以使用定理7.6来得到所需要的结果:

$$v(f)=f^{\text{out}}(A^*)-f^{\text{in}}(A^*)=\sum_{e\text{从}A^*\text{出来}}f(e)-\sum_{e\text{进入}A^*}f(e)$$

$$=\sum_{e\text{从}A^*\text{出来}}c_e-0=c(A^*,B^*)\ \blacksquare$$

来自教材

对于改进后的 Dinic 算法，dinic 算法总是寻找最短的增广路并沿着它增广，增广路的长度不会在增广过程中改变，则当无法增广时，说明分层图上没有可以增广的路线了，这有两种情况，第一，已经求出了最大流，第二，可能存在长一些的增广路可以继续增广，因此，继续 bfs 构造分层网络。每次完成后最短增广路长度+1,由于最短路<n,则最对重复 n-1 次 bfs 就可完成了。

四、算法复杂度分析

1、时间复杂度

• 建图过程

```
for(int i=1,x,y;i<=n;i++)//循环 m 个顾客
{
    scanf("%d",&x);//第 i 个顾客有 x 个钥匙
    for(int j=1;j<=x;j++)
for(int i=1;i<=m;i++)
{
    int l=p[i].size();
    for(int j=0;j<l;j++)
```

时间复杂度为 O(mn)
• Dinic 算法
最多仅需 n-1 轮增广即可求得最大流[bfs()最多 n-1 轮],单轮增广[dfs()]的最坏复杂度是 O(mn)，所以总的时间复杂度为 $O(mn^2)$。

综上，时间复杂度为 $O(mn^2)$。

2、空间复杂度

• 建图过程
创建顾客和保险柜变量并建立顾客拥有钥匙这样的关系和网络图
```
p[y].push_back(i);//顾客 i 有保险柜 y 的使用权，不定长数组存储
```
最坏的情况每人都有所有钥匙，所以空间复杂度为 O(mn)
• Dinic 算法
主要用建图的变量进行计算，空间复杂度为常数 O(1)

综上，空间复杂度为 O(mn)。

五、程序输入和输出

• 输入

第一行有两个正整数 m 和 n，分别表示保险柜数量和顾客数量；
第二行有 m 个非负整数，表示各保险柜中的初始金币数量；
接下来有 n 行，按照顾客到银行的次序，一行表示一个顾客：
每行第一个数 k 为该顾客拥有的钥匙数量，接下来 k 个数表示该顾客拥有的 k 个钥匙，最后一个数 c 为该顾客的金币需求量；

• 输出

输出一个整数，表示所有顾客可以取走的金币总数的最大值。

六、程序演示

假设有 3 个保险柜 1、2、3，金币含量分别为 3、1、10；
假设有 3 位顾客，按顺序取金币：
第一位持有 1、2 保险柜的钥匙，需要 2 枚金币；
第二位持有 1、3 保险柜的钥匙，需要 3 枚金币；
第三位持有 2 保险柜的钥匙，需要 6 枚金币。


c:\Users\PC\Desktop\算法课件\project.exe
3 3
3 1 10
2 1 2 2
2 1 3 3
1 2 6
7请按任意键继续. . .