医院血库题解

整体思路

设顶点 $a_1, a_2, \ldots, a_n$ 分别表示血库中的 n 种血型（表示为下图左边一列圆形）

设顶点 $b_1, b_2, \ldots, b_m$ 分别表示 m 种血液需求（表示为下图右边一列方形）

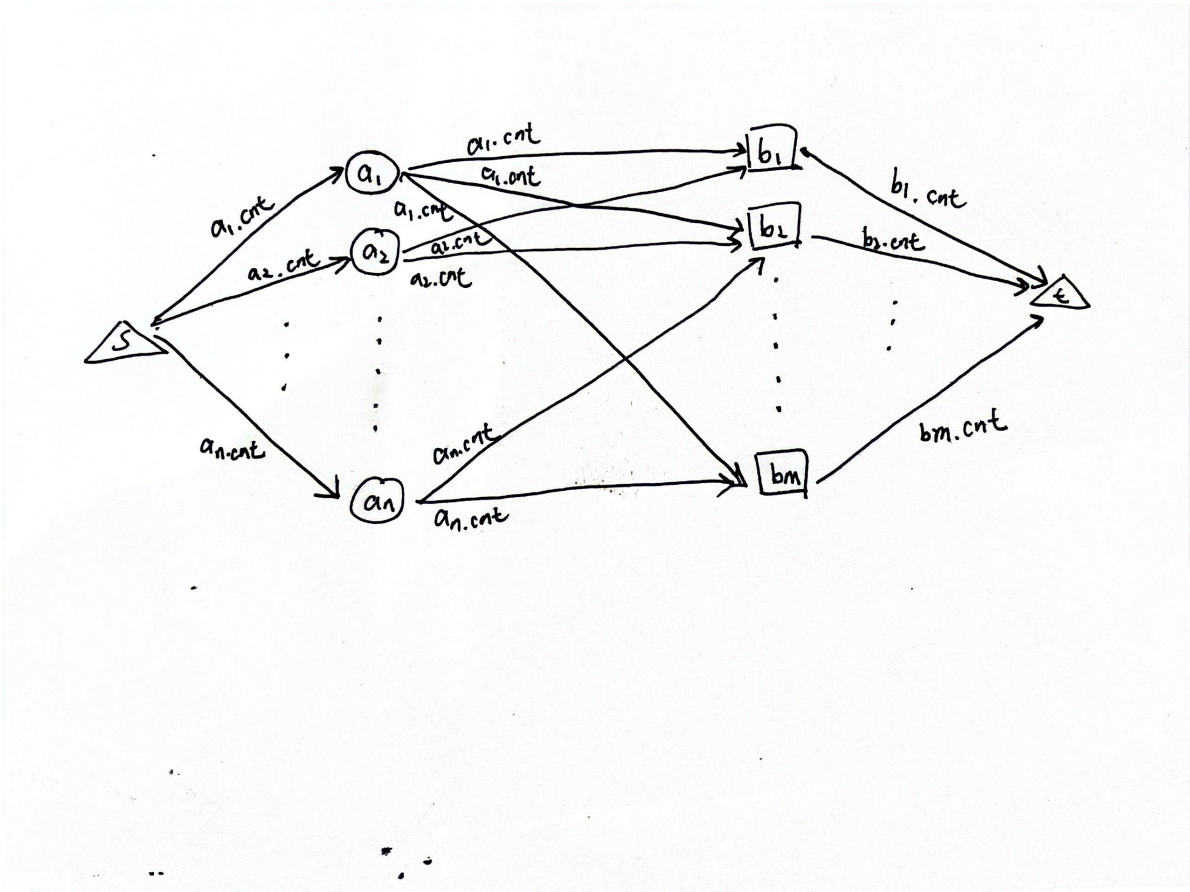
添加源点 s 和汇点 t （表示为下图最左最右的两个三角形）

对以上 $n+m+2$ 个顶点如下编号：

$a_1, a_2, \ldots, a_n$ 分别标记为 $1, 2, \ldots, n$ 号顶点

$b_1, b_2, \ldots, b_m$ 分别标记为 $n+1, n+2, \ldots, n+m$ 号顶点

源点 s 标记为 0 号顶点，汇点 t 标记为 $n+m+1$ 号顶点



按如下规则建边：

如果血库中的血型 a_i 可以用于救助血液需求 b_j ，则建立一条从 a_i （顶点序号 i) 到 b_j （顶点序号 $n+j$ ），流量为 " 血库中血型 a_i 的剩余数量 " 的边，和，一条从 b_j （顶点序号 $n+j$) 到 a_i （顶点序号 i ），流量为 0 的边。

从源点 s （顶点序号 0 ）向顶点 $a_1, a_2, \ldots, a_n$ 分别建立一条从 s 到顶点 $a_1, a_2, \ldots, a_n$ 的，流量为 " 血库中血型 a_i 的剩余数量 " 的边，和，n 条分别从顶点 $a_1, a_2, \ldots, a_n$ 指向源点 s ，流量为 0 的边。

从顶点 $b_1, b_2, \ldots, b_m$ 向汇点 t （顶点序号 $n+m+1$ ） 分别建立一条从顶点 $b_1, b_2, \ldots, b_m$ 到汇点 t 的，流量为 " 第b_j 种血液需求的数量 " 的边，和，m 条分别从汇点 t 指向顶点 $b_1, b_2, \ldots, b_m$ ，流量为 0 的边。

通过 dinic 算法计算上图的最大流，即为答案。

dinic算法不断重复以下步骤，直到在残留网络中无法到达汇点 t：

1. 在残量网络上 BFS 建立分层图
2. 在分层图上 DFS 遍历增广路，在回溯时同时更新边容量，初始给源点 s 的可增加流量上限为无穷大

   时间复杂度 $O(nm^2)$，实际中远远达不到该上界，可以解决本题 10^4 级别的数据

详解可以参考以下博客和下文代码的注释

https://blog.csdn.net/EQUINOX1/article/details/135793883

https://www.cnblogs.com/LUO77/p/6115057.html

完整代码解释

```
#include <bits/stdc++.h>
using namespace std;
#define int long long
int n,m,s,t;
const int MAXN = 5e6+10;
struct EDGE{
    int to,next,flow;
}edge[MAXN];
int head[MAXN], len = 0;
void add(int x,int y,int z){          //数组方式邻接表存图
    edge[len].to = y;
    edge[len].flow = z;
    edge[len].next = head[x];
    head[x] = len;
```

```
    len ++ ;
}
void add_edge(int x,int y,int z){    //每次加入一条边同时加入它的反向边
    add(x,y,z);
    add(y,x,0);                      //反向边流量设为0
}
struct node{
    int type, cnt;
}a[1010], b[1010];
//辅助函数 cal () : 输入血型字符串，输出 int 类型表示（方便 ok 函数判断）
//int的低 $i$ 位表示血型除了 ‘O’ 的 25 个抗原是否存在，输入 "O" 时 return 0
int cal(string s){
    if(s == "O") return 0;
    int ret = 0;
    for(int i=0;i<s.size();i++){//否则转化为对应的01bit值
        int idx = s[i]-'A';
        ret |= (1LL<<idx);
    }
    return ret;
}
//辅助函数 ok (typeA, typeB) == 1 表示血型 A 可以用于供应对血型 B 的需求。
int ok(int xx,int yy){
    for(int i=0;i<26;i++){
        if(xx & (1LL<<i)){
            if(yy & (1LL<<i));
            else return 0;
        }
```

```
    }
    return 1;
}
int deep[MAXN], tail, be, q[MAXN];
//bfs找分层图
int bfs(){
    memset(deep,0,sizeof(deep));
    deep[s] = 1;
    be = 0, tail = 1;
    q[1] = s;        //源点s入队
    while(be!=tail){
        int u = q[++be];//取出当前队首
        for(int i=head[u];~i;i=edge[i].next){//遍历队首u的下一个顶点中还没有遍历到的
            if(!deep[edge[i].to] && edge[i].flow){
                deep[edge[i].to] = deep[u] + 1;//记录层次
                q[++tail] = edge[i].to;
            }
        }
    }
    return deep[t];//返回0说明汇点t无法到达
}
//dfs遍历增广路，在回溯时同时更新边容量
int dfs(int now,int fa1){//两个参数分别是当前顶点和可增加流量
    if(now==t) return fa1;//到达汇点t，返回增加流量值
    int fa = 0;
```

```
    for(int i=head[now];~i && fa1;i=edge[i].next){//遍历当前顶点now的相邻顶点中层次编号高一层的
        if(deep[edge[i].to] == deep[now]+1 && edge[i].flow){
            int d = dfs(edge[i].to, min(fa1,edge[i].flow));//在下一层顶点edge[i].to处可增加的流量
            if(d>0){
                edge[i].flow -= d;//i是从当前顶点now向下一层顶点edge[i].to的有向边，更新其边最大流量
                edge[i^1].flow += d;//i^1是从下一层顶点edge[i].to向当前顶点now的有向边，更新其边最大流量
                fa1 -= d;
                fa += d;//在当前顶点now处的可增加流量fa增加d
            }
        }
    }
    if(!fa) deep[now] = -1;
    return fa;
}
//一直重复以上两个过程，直到汇点t无法到达
int dinic(){
    int res = 0;
    while(bfs()){
        res += dfs(s,1e18);
    }
    return res;
}
signed main(void)
```

```
{
//  ios::sync_with_stdio(false);
//  cin.tie(0), cout.tie(0);
//  freopen("special/sp_18.in","r",stdin);
//
freopen("special/sp_18.out","w",stdout);
    cin >> n >> m;
    for(int i=1;i<=n;i++){
        string s;
        cin >> s >> a[i].cnt;
        a[i].type = cal(s);
    }
    for(int i=1;i<=m;i++){
        string s;
        cin >> s >> b[i].cnt;
        b[i].type = cal(s);
    }
    memset(head,-1,sizeof(head));
    s = 0, t = n+m+1;
    for(int i=1;i<=n;i++){
        add_edge(0,i,a[i].cnt);
    }
    for(int i=1;i<=m;i++){
        add_edge(n+i,t,b[i].cnt);
    }
    for(int i=1;i<=n;i++){
        for(int j=1;j<=m;j++){
            if(ok(a[i].type,b[j].type)){
                add_edge(i,n+j,a[i].cnt);
            }
        }
    }
    int res = dinic();
```

```
122         cout << res << '\n';
123         return 0;
124 }
```

视频发送题解

给出一个性质：只要将所有视频全部发送出去之后满足速率限制，那么一定存在可行的发送顺序，否则一定不存在满足限制的发送顺序。根据题意可以很容易想到，在遍历一遍视频列表判断是否有可行的发送顺序后，将所有的视频按照 $\frac{\text{大小}}{\text{发送时长}}$ 从小到大排序后逐一发送就可以按照要求发送完成，时间复杂度 $\mathrm{O(nlogn)}$ 。也可以在排序之后遍历视频列表检查是否符合题意，若是则输出排序后的列表。

实际上，我们可以先发送所有 $\frac{\text{大小}}{\text{发送时长}} \leq$ 速率 的视频（未超出原始速率限制的视频，在任何位置发送这些视频一定不会在发送前未超出速率限制时发送完该视频后超出速率限制，同时发送该视频可能可以放宽后续发送视频的限制），再发送所有其它未被发送的视频（超出原始链路速率限制的视频），也一定能满足题中要求，时间复杂度 $\mathrm{O(n)}$ 。

代码以及代码解释：

```
1 #include<bits/stdc++.h>
2 using namespace std;
3 //由于发送的总时长和总大小一定超出 int 类型范围，所以采用 long long 类型
4 #define int long long
5
6 void sov()
7 {
```

```cpp
    int i,n;
    int speed,sumsize=0,sumtime=0;

    //读入视频总数和平均速率限制
    cin>>n>>speed;
    //使用动态数组存储视频的大小和时长，读入的同时计算总大小和总时长
    vector<int> vidsize(n+1),vidtime(n+1);
    for(i=1;i<=n;++i)
    {
        cin>>vidsize[i];
        sumsize+=vidsize[i];
    }
    for(i=1;i<=n;++i)
    {
        cin>>vidtime[i];
        sumtime+=vidtime[i];
    }

    //根据上述性质，如果 视频总时长 * 速率 < 总大小，那么一定不存在满足限制的发送顺序。
    //由于 总时长 * 速率 有可能会超出 long long 类型的范围，所以不能简单地对该条件进行判断。
    //可以采用的方法是使用 __int128 数据类型后比较，或者将 总时长 * 速率 转换成浮点型，如果其大于视频总大小判断一定存在。
    //这里使用了固定值 10^15 ，因为所有视频的总大小不会超过 10^15 ，一定可以按要求发送完。
    if(sumsize>sumtime*speed&&(sumtime*(double)speed)<pow(10,15))
    {
        //如果上述两个条件都不满足那么一定没有可行方案
```

```
33          cout<<"No";return;
34      }
35      cout<<"Yes\n";
36
37      //首先输出所有 大小 <= 时长 * 限速 的视频，再输出所有剩下没有输出的视频，时间复杂度 O(n) 。
38      //这里有过尝试将排序（O(nlogn)）的做法卡掉，但是最后发现两种做法的耗时相差不大。
39      for(i=1;i<=n;++i)
40      {
41          if(vidsize[i]<=vidtime[i]*speed)
42          {
43              cout<<i<<' ';
44          }
45      }
46      for(i=1;i<=n;++i)
47      {
48          if(vidsize[i]>vidtime[i]*speed)
49          {
50              cout<<i<<' ';
51          }
52      }
53  }
54
55  signed main()
56  {
57      ios::sync_with_stdio(false);
58      sov();
59      return 0;
60  }
61
```