

算法设计与分析

期末考查作业

姓名：郝裕玮

班级：计科 1 班

学号：18329015

目录

1 题目概述.....	3
2 算法核心知识点介绍	4
2.1 费用流.....	4
2.2 网络最大流 & EK 算法	4
2.3 SPFA	6
2.4 Dijkstra 算法	6
2.5 最小费用最大流.....	7
3 方法 1: EK & SPFA.....	8
4 方法 2: EK & Dijkstra	11
5 结果展示.....	16

1 题目概述

网格上有若干个货物和仓库，可以往水平方向或垂直方向搬运每个货物到相邻的仓库中。每个仓库只能容纳一个货物。至少使用两种思路求让这些货物放到不同仓库所需要的搬运最小路程(单位:格)。

数据输入: $M*N$ ($1 < M, N < 100$) 网格, 地图中的'W'和'G'分别表示仓库和货物的位置, 个数相同, 最多有 100 个仓库, 其它空位置用'.'表示。

结果输出: 所需要的最小路程。

输入样例:

```
. . . W . . . .
. . . W . . . .
. . . W . . . .
G G G W G G G G
. . . W . . . .
. . . W . . . .
. . . W . . . .
```

2 算法核心知识点介绍

2.1 费用流

给定一个网络 $G=(V,E)$ ，每条边有容量限制 $w(u,v)$ ，还有单位流量的费用 $c(u,v)$ 。当 (u,v) 的流量为 $f(u,v)$ 时，需要花费 $f(u,v) * c(u,v)$ 的费用。

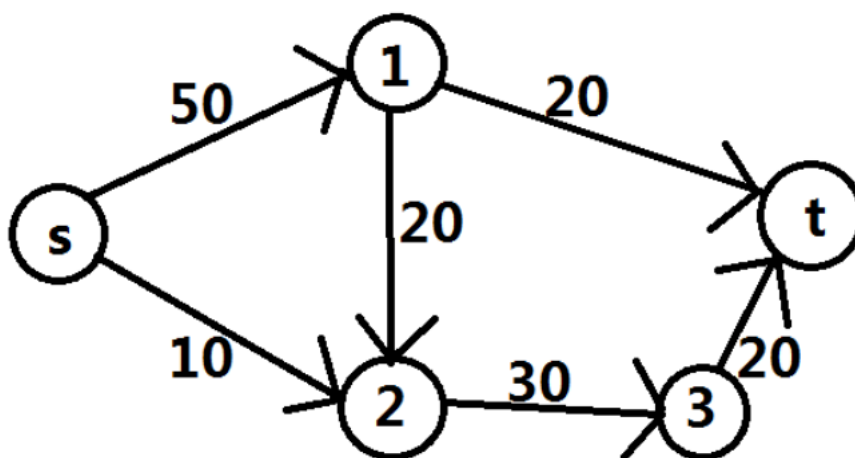
该网络中总花费最小的最大流称为最小费用最大流，总花费最大的最大流称为最大费用最大流，二者合称费用流模型，即在最大流的前提下考虑费用的最值。

一个网络的最大流是唯一的，不同路径有不同的费用。

2.2 网络最大流 & EK 算法

EK 算法通常用于求网络最大流。

网络最大流是指：一个有向图，每条边有一个流量限制，我们需要求出从源点 s 到汇点 t 的最大流量。如下图所示：



上图的最大流为 $20 + 20 = 40$

EK 算法具体思路如下：

1, 通过 BFS（广度优先搜索）找增广路，增广路中的边不为 0 且增广路的最大流量是这条路径上的剩余流量限度最小的边的剩余流量限度（一次 BFS 只能找一条增广路）。

2, 每找到一条增广路，就将这条增广路上的边的剩余流量限度都减去这条增广路能通过的最大流量，并将最大流加上这条增广路的最大流量。

3, 直到找不到增广路，我们得到的最大流就是 s 到 t 的最大流。

补充解释：

(1) 网络是指一个有向图 $G=(V,E)$ ，有两个特殊节点：源点 s 和汇点 t 。每条有向边 (x,y) 都有一个权值 $c(x,y)$ ，称为边的容量。如果 (x,y) 不在图中，那么就有 $c(x,y)=0$ 。

用 $f(x,y)$ 表示边 (x,y) 上的流量，那么 $c(x,y)-f(x,y)$ 就是边的剩余容量。

通常用 $c(x,y)/f(x,y)$ 的形式标记边上的流量与容量。

(2) 最大流：从源点流向汇点的最大流量

(3) 增广路：一条从源点到汇点的所有边的剩余容量 ≥ 0 的路径

(4) 残留网：由网络中所有结点和剩余容量大于 0 的边构成的子图，这里的边包括有向边和其反向边。

(5) 建图时每条有向边 (x,y) 都构建一条反向边 (y,x) ，初始容量 $c(y,x)=0$ 。构建反向边的目的就是提供一个“退流管道”，一旦前面的增广路堵死可行流，可以通过“退流管道”退流，提供了后悔机制。

2.3 SPFA

SPFA 算法用来解决单源最短路问题，可以求解从某个点开始，到图中其他所有点的最短距离。

算法原理如下所示：

(1) 实现 SPFA 算法需要一个队列 q ，一个标记数组 $vis[N]$ 用来标记某点是否在队列中。数组 $dis[N]$ ，用来存储起点到某个点的最短距离。

(2) 初始化 dis 数组为正无穷。

(3) 从起点开始枚举每个点的所有子节点，设父节点到子节点的距离为 s ，父节点到起点的距离为 $dis[u]$ ，子节点到起点的距离为 $dis[v]$ ，若 $dis[u] + s < dis[v]$ ，则更新 $dis[v] = dis[u] + s$ 。若 v 没有在队列中，则将 v 加入队列。

2.4 Dijkstra 算法

Dijkstra 算法使用了广度优先搜索解决赋权有向图或者无向图单源最短路径问题。

算法原理如下所示：

(1) Dijkstra 算法采用的是一种贪心的策略，声明一个数组 dis 来保存源点到各个顶点的最短距离和一个保存已经找到了最短路径的顶点的集合： T 。

(2) 初始时，源点 s 的路径权重被赋为 0 ($dis[s] = 0$)。若对于顶点 s 存在能直接到达的边 (s, m) ，则把 $dis[m]$ 设为 $w(s, m)$ ，同时把

所有其他(s 不能直接到达的)顶点的路径长度设为无穷大。初始时, 集合 T 只有顶点 s 。

(2) 从 dis 数组选择最小值, 则该值就是源点 s 到该值对应的顶点的最短路径, 并且把该点加入到 T 中。

(3) 观察新加入的顶点是否可以到达其他顶点并且比较通过该顶点到达其他点的路径长度是否比源点直接到达的更短, 如果是, 则替换这些顶点在 dis 中的值。

(4) 从 dis 中找出最小值, 重复上述动作, 直到 T 中包含图的所有顶点。

2.5 最小费用最大流

因为 BFS 只能找增广路而无法找到花费最小的增广路, 所以我们将找增广路这一部分改成利用 SPFA 寻找一条单位费用之和最小的增广路就可以用于在最大流的前提下求解最小费用, 也就是把 $c(u,v)$ 当作边权, 在残留网上求最短路, 这样就能够求出最小费用最大流。

为了退流, 反向边的初始容量为 0, 反向边的容量每次 $+f$

为了退费, 反向边的初始费用为 $-w$, 走反向边的花费 $+f * (-c)$

因为需要退费, 也就是说反向边的费用是负值, 即求解最短路的过程中存在负边权, 所以要用 SPFA 算法。(但后续的第二种方法也会补充如何在存在负边权的情况下使用 Dijkstra 算法进行求解)

3 方法 1: EK & SPFA

EK 和 SPFA 的算法原理和具体步骤已在上一大节中进行阐述，代码如下所示（代码中已包含必要的补充解释）：

```
#include<bits/stdc++.h>
using namespace std;
const int maxn = 1e5+5;
const int inf = 0x3f3f3f3f;//无穷大

//链式前向星，用于存储图结构信息
//关键字 inline: 内联函数，同宏函数一样将在被调用处进行代码展开，省去了参数压
//栈、栈帧开辟与回收，结果返回等，从而提高程序运行速度
struct graph{
    //head: 第一个以 i 为起点的边的位置，同时 head 初始化为-1
    //next: 与第 i 条边同一个起点的下一条边
    //to: 第 i 条边的终点
    //wei: 容量
    //cost: 流量
    int head[maxn],next[maxn],to[maxn],wei[maxn],cost[maxn],cnt;
    inline graph():cnt(1){}
    inline void add(int u,int v,int w,int c){
        next[++cnt] = head[u];
        to[cnt] = v;
        wei[cnt] = w;
        head[u] = cnt;
        cost[cnt] = c;
    }
}gr;

//存放货物和仓库的坐标
struct node{
    int x,y;
    inline node(){}
    inline node(int x,int y):x(x),y(y){}
};

int n,m;//地图的行数和列数
node a[maxn],b[maxn]; //a 代表仓库，b 代表货物
int a_len,b_len,s,t;//仓库数量，货物数量，费用流起点，费用流终点

//计算两点间曼哈顿距离
```



```

inline int getdis(const node& p,const node& q){
    return abs(p.x-q.x)+abs(p.y-q.y);
}

int dis[maxn],pre[maxn][2];//dis[i]表示当前从起点走到点 i 的最少费用,pre[i][0]表示最短路中的节点 i 的前驱节点, pre[i][1]则表示节点 i 的前驱边
bool inq[maxn];//inq[i]表示点 i 是否在队列中

//使用 SPFA 求最短路径
inline bool spfa(){
    queue<int> que;
    for(int i = 1;i <= t;i++){
        dis[i] = inf;//首先设置起点到所有点的距离为无穷大
    }

    que.push(s);//放入起点
    inq[s] = true;//起点在队列中
    dis[s] = 0;//起点到起点的距离为 0

    //SPFA 算法求最短路
    while(!que.empty()){
        int u = que.front();
        que.pop();
        inq[u] = false;
        for(int i = gr.head[u];i;i = gr.next[i]){
            int v = gr.to[i],w = gr.wei[i],c = gr.cost[i];
            if(w == 0){
                continue;//如果边的容量不为 0
            }
            if(dis[v] > dis[u] + c){//松弛操作, 更新两点间最短距离
                dis[v] = dis[u] + c;
                pre[v][0] = u;//设置前驱点
                pre[v][1] = i;//设置前驱边
                if(!inq[v]){//如果没在队列中, 就将 v 点加入队列
                    inq[v] = true;
                    que.push(v);
                }
            }
        }
    }

    return dis[t] != inf;//判断起点到终点是否还有流量
}

//求解最小费用最大流

```

```

inline int solve(){
    int ans = 0;
    while(spfa()){
        ans += dis[t];
        int u = t; //从终点往前走
        while(pre[u][0]){
            gr.wei[pre[u][1]]--; //正向边的容量-1
            gr.wei[pre[u][1]^1]++; //反向边的容量+1
            u = pre[u][0];
        }
    }
    return ans;
}

//主函数
int main(){
    //输入需要求解的地图
    cin >> n >> m;
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= m; j++){
            char v;
            cin >> v;
            if(v == 'W'){
                a[++a_len] = node(i, j); //仓库
            } else if(v == 'G'){
                b[++b_len] = node(i, j); //货物
            }
        }
    }

    //货物向仓库连边
    for(int i = 1; i <= b_len; i++){
        int u = i;
        for(int j = 1; j <= a_len; j++){
            int v = b_len + j;
            int d = getdis(b[i], a[j]);
            gr.add(u, v, 1, d);
            gr.add(v, u, 0, -d);
        }
    }

    s = a_len + b_len + 1;
    t = s + 1;

    //起点向货物连边

```

```

for(int i = 1;i <= b_len;i++){
    int u = i;
    gr.add(s,u,1,0);
    gr.add(u,s,0,0);
}

//仓库向终点连边
for(int i = 1;i <= a_len;i++){
    int v = b_len + i;
    gr.add(v,t,1,0);
    gr.add(t,v,0,0);
}

//输出答案
cout << solve() << endl;
return 0;
}

```

4 方法 2: EK & Dijkstra

在方法 1 中，由于有反向边权为负值的情况，所以导致无法使用 Dijkstra 算法，但如果我们将所有的边权都拉回到正数的话，就可以正常使用该算法。我们可以考虑给每个节点一个势，然后将我们的最短路转移 $dis_v = dis_u + w$ 改为 $dis_v = dis_u + w + h_u - h_v$ (h_i 是势)。保证 $w + h_u - h_v \geq 0$ 。

假设有一条 $p_1-p_2-p_3.....p_n$ 这样一条路径，其路径长度为 $(w_1 + w_2 + w_3... + w_{n-1}) + (h_1 - h_2) + (h_2 - h_3) + (h_3 - h_4) +(h_{n-1} - h_n)$ ，那我们显然可得 $(h_1 - h_2) + (h_2 - h_3) + (h_3 - h_4) +(h_{n-1} - h_n) = h_1 - h_n$ ，所以我们在算出加上势以后的（无论路线是什么样的）最短路后，再减去 $h_{begin} - h_{end}$ 就可以解决负边权的问题，从而在运算过程中使用 Dijkstra 算法。

代码如下所示（代码中已包含必要的补充解释）：

```
#include<bits/stdc++.h>
using namespace std;
const int maxn = 1e5+5;
const int inf = 0x3f3f3f3f;//无穷大

//链式前向星，用于存储图结构信息
//关键字 inline: 内联函数，同宏函数一样将在被调用处进行代码展开，省去了参数压
//栈、栈帧开辟与回收，结果返回等，从而提高程序运行速度
struct graph{
    //head: 第一个以 i 为起点的边的位置，同时 head 初始化为-1
    //next: 与第 i 条边同一个起点的下一条边
    //to: 第 i 条边的终点
    //wei: 容量
    //cost: 流量
    int head[maxn],next[maxn],to[maxn],wei[maxn],cost[maxn],cnt;
    inline graph():cnt(1){}
    inline void add(int u,int v,int w,int c){
        next[++cnt] = head[u];
        to[cnt] = v;
        wei[cnt] = w;
        head[u] = cnt;
        cost[cnt] = c;
    }
}gr;

//存放货物和仓库的坐标
struct node{
    int x,y;
    inline node(){}
    inline node(int x,int y):x(x),y(y){}
};

int n,m;//地图的行数和列数
node a[maxn],b[maxn];//a 代表仓库，b 代表货物
int a_len,b_len,s,t;//仓库数量，货物数量，费用流起点，费用流终点

//计算两点间曼哈顿距离
inline int getdis(const node& p,const node& q){
    return abs(p.x-q.x)+abs(p.y-q.y);
}

struct qnode{
```

```

int p,d;
inline qnode(int p,int d):p(p),d(d){}
inline bool operator < (const qnode& o)const{
    return d>o.d;
}
};

int dis[maxn],pre[maxn][2];//dis[i]表示当前从起点走到点 i 的最少费用,pre[i][0]表示最短路中的节点 i 的前驱节点, pre[i][1]则表示节点 i 的前驱边
int h[maxn];//h[i]表示点 i 的势能
bool vis[maxn];

//使用 Dijkstra 算法求最短路径
inline bool dijkstra(){
    for(int i = 1;i <= t;i++){
        h[i] = dis[i];//当前点的势能为上一次求增广路的最短路径
    }

    priority_queue<qnode> pq;
    for(int i = 1;i <= t;i++){
        dis[i] = inf;
        vis[i] = false;
    }

    dis[s] = 0;
    pq.push(qnode(s,0));

    while(!pq.empty()){
        qnode top = pq.top();
        pq.pop();
        int u = top.p;
        if(vis[u] == true){
            continue;
        }
        vis[u] = true;

        for(int i = gr.head[u];i;i = gr.next[i]){
            int v = gr.to[i];
            int w = gr.wei[i];
            int c = gr.cost[i] + h[u] - h[v];//改变边权为 cost[i] + h[u]
            - h[v]
            if(w == 0){
                continue;
            }
        }
    }
}

```

```

        if(dis[v] > dis[u] + c){
            pre[v][0] = u;
            pre[v][1] = i;
            dis[v] = dis[u] + c;
            pq.push(qnode(v,dis[v]));
        }
    }
}
return dis[t] != inf;
}

//求解最小费用最大流
inline int solve(){
    int ans = 0;
    while(dijkistra()){
        //加上点 t 处的势能
        ans += dis[t] + h[t];
        //从终点往前走
        int u = t;
        while(pre[u][0]){
            gr.wei[pre[u][1]]--; //正向边的容量-1
            gr.wei[pre[u][1]^1]++; //反向边的容量+1
            u = pre[u][0];
        }
    }
    return ans;
}

//主函数
int main(){
    //输入要求解的地图
    cin >> n >> m;
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= m; j++){
            char v;
            cin >> v;
            if(v == 'W'){
                a[++a_len] = node(i,j); //仓库
            } else if(v == 'G'){
                b[++b_len] = node(i,j); //货物
            }
        }
    }
}

```

```

//货物向仓库连边
for(int i = 1;i <= b_len;i++){
    int u = i;
    for(int j = 1;j <= a_len;j++){
        int v = b_len + j;
        int d = getdis(b[i],a[j]);
        gr.add(u,v,1,d);
        gr.add(v,u,0,-d);
    }
}
s = a_len + b_len + 1;
t = s + 1;

//起点向货物连边
for(int i = 1;i <= b_len;i++){
    int u = i;
    gr.add(s,u,1,0);
    gr.add(u,s,0,0);
}

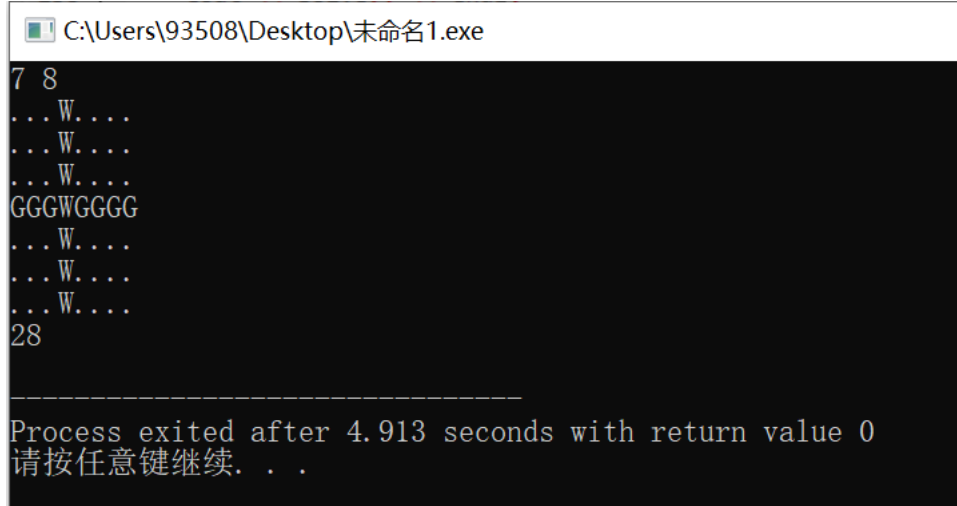
//仓库向终点连边
for(int i = 1;i <= a_len;i++){
    int v = b_len+i;
    gr.add(v,t,1,0);
    gr.add(t,v,0,0);
}

//输出答案
cout << solve() << endl;
return 0;
}

```

5 结果展示

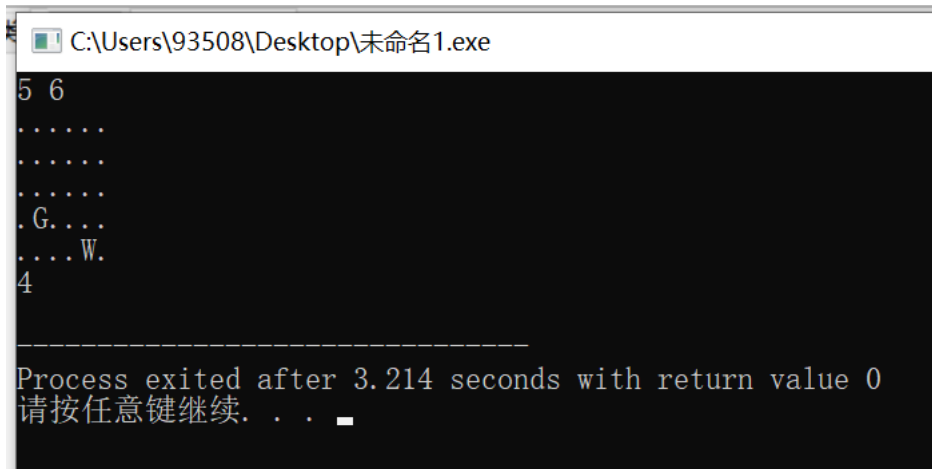
样例 1:



```
C:\Users\93508\Desktop\未命名1.exe
7 8
...W....
...W....
...W....
GGGWGGGG
...W....
...W....
...W....
28

-----
Process exited after 4.913 seconds with return value 0
请按任意键继续. . .
```

样例 2:



```
C:\Users\93508\Desktop\未命名1.exe
5 6
.....
.....
.....
.G....
....W.
4

-----
Process exited after 3.214 seconds with return value 0
请按任意键继续. . .
```

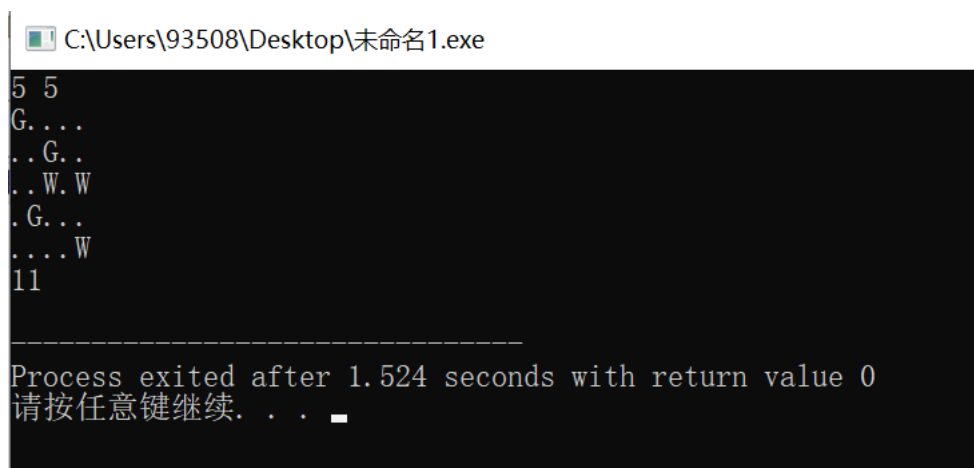

样例 3:



```
C:\Users\93508\Desktop\未命名1.exe
3 3
. . G
WWW
GG.
3

-----
Process exited after 1.422 seconds with return value 0
请按任意键继续. . .
```

样例 4:



```
C:\Users\93508\Desktop\未命名1.exe
5 5
G...
..G..
..W.W
.G...
....W
11

-----
Process exited after 1.524 seconds with return value 0
请按任意键继续. . .
```