int a[1005][1005]; 迪杰斯特拉

int dis[1005]; bool vis[1005]; const int INF=0x3f3f3f3f; int n;

void Dij()

{

for(int i=0;i<=n;i++)

{

dis[i]=a[0][i];

vis[i]=false;

} //第一次输入dis[]

vis[0]=true;

dis[0]=0;

for(int i=0;i<=n;i++)

{

int minn=INF;

int p;

for(int j=0;j<=n;j++)

{

if(!vis[j]&&dis[j]<minn)

{

minn=dis[p=j];

}

}

vis[p]=true;

if(minn==INF) break;

for(int j=0;j<=n;j++)

{

if(!vis[j]&&dis[j]>dis[p]+a[p][j])

dis[j]=dis[p]+a[p][j];

} } }

int main()

{

int t,s,d;

int x,y,z;

while(scanf("%d%d%d",&t,&s,&d)!=EOF)

{

n=0;

memset(a,INF,sizeof(a));

while(t--)

{

scanf("%d%d%d",&x,&y,&z);

if(z<a[x][y]) //构造一个二维数组

a[x][y]=a[y][x]=z; //两个城市之间取短的路

n=max(n,max(x,y));

}

while(s--)

{

scanf("%d",&x);

a[0][x]=0;

}

Dij();

int minn=INF;

while(d--)

{

scanf("%d",&x);

minn=min(minn,dis[x]); //多个终点取路近的

}

printf("%d\n",minn);

}

return 0;

}

弗洛伊德

for(int k=1;k<=n;k++)

{

for(int i=1;i<=n;i++)

{

if(s[i][k]==INF) //在这里要优化一下，否则会超时～～

continue;

for(int j=1;j<=n;j++)

{

if(map[i][j]>map[i][k]+map[k][j]) map[i][j]=map[i][k]+map[k][j]; //考虑第三方节点k的优化

}

}

}

DFS:

/\*

该DFS 框架以2D 坐标范围为例，来体现DFS 算法的实现思想。

\*/

#include<cstdio>

#include<cstring>

#include<cstdlib>

using namespace std;

const int maxn=100;

bool vst[maxn][maxn]; // 访问标记

int map[maxn][maxn]; // 坐标范围

int dir[4][2]={0,1,0,-1,1,0,-1,0}; // 方向向量，(x,y)周围的四个方向

bool CheckEdge(int x,int y) // 边界条件和约束条件的判断

{

if(!vst[x][y] && ...) // 满足条件

return 1;

else // 与约束条件冲突

return 0;

}

void dfs(int x,int y)

{

vst[x][y]=1; // 标记该节点被访问过

if(map[x][y]==G) // 出现目标态G

{

...... // 做相应处理

return;

}

for(int i=0;i<4;i++)

{

if(CheckEdge(x+dir[i][0],y+dir[i][1])) // 按照规则生成下一个节点

dfs(x+dir[i][0],y+dir[i][1]);

}

return; // 没有下层搜索节点，回溯

}

int main()

{

......

return 0;

}

#include<cstdio>

#include<cstring>

#include<queue>

#include<algorithm>

using namespace std;

const int maxn=100;

bool vst[maxn][maxn]; // 访问标记

int dir[4][2]={0,1,0,-1,1,0,-1,0}; // 方向向量

struct State // BFS 队列中的状态数据结构

{

int x,y; // 坐标位置

int Step\_Counter; // 搜索步数统计器

};

State a[maxn];

boolCheckState(State s) // 约束条件检验

{

if(!vst[s.x][s.y] && ...) // 满足条件

return 1;

else // 约束条件冲突

return 0;

}

void bfs(State st)

{

queue <State> q; // BFS 队列

State now,next; // 定义2 个状态，当前和下一个

st.Step\_Counter=0; // 计数器清零

q.push(st); // 入队

vst[st.x][st.y]=1; // 访问标记

while(!q.empty())

{

now=q.front(); // 取队首元素进行扩展

if(now==G) // 出现目标态，此时为Step\_Counter 的最小值，可以退出即可

{

...... // 做相关处理

return;

}

for(int i=0;i<4;i++)

{

next.x=now.x+dir[i][0]; // 按照规则生成 下一个状态

next.y=now.y+dir[i][1];

next.Step\_Counter=now.Step\_Counter+1; // 计数器加1

if(CheckState(next)) // 如果状态满足约束条件则入队

{

q.push(next);

vst[next.x][next.y]=1; //访问标记

}

}

q.pop(); // 队首元素出队

}

return;

}

int main()

{

......

return 0;

}

这个优化主要是用来在O（1）时间内求出一个序列a中,a[i]+a[i+1]+……+a[j]的和。

具体原理十分简单：用sum[i]表示（a[1]+a[2]+……+a[i])，其中sum[0]=0，则（a[i]+a[i+1]+……+a[j]）即等于sum[j]-sum[i-1]。

二维前缀和

同理，有一维就有二维。对于一个矩阵a，我们也能在O（1）时间内求出子矩阵[x1~x2][y1~y2]的和。

设sum[i][j]为子矩阵[1~i][1~j]的和。则由容斥原理得：

sum[0][j]=sum[i][0]=0

a[x1~x2][y1~y2]=sum[x2][y2]-sum[x1-1][y2]-sum[x2][y1-1]+sum[x1-1][y1-1]

两点式：k≠0　y-y1=(y2-y1)(x-x1)/(x2-x1).