《算法设计与分析》上机报告

姓名：	廖洲洲	学号：	PB17081504	日期：	2019.12.19
上机题目：	图搜索 BFS 算法及存储优化				

实验环境：
CPU：Intel Core i7-8550U; 内存:8G ; 操作系统：Win 10 ; 软件平台: JetBrains CLion ;

一、算法设计与分析：

题目一:

实现传统 BFS 以及双向 BFS,并可以找到对应搜索节点对之间的路径信息。
要求利用 CSR（Compressed Sparse Row）方式存储图。

二、核心代码：

题目一:

(一)CSR 存储图思想：

1. CSR 使用三个数组来存储矩阵：数值，列号，以及行偏移数组
2. CSR 不是三元组，而是整体的编码方式。行偏移数组每个元素表示某一行的第一个非 0 元素在数值数组里面的起始偏移位置，即 row[i]表示第 i 行第一个非 0 元素在所有非 0 元素中的偏移量。列数组表示每个非 0 元素在矩阵中的列号，即 colum[row[i]]表示第 i 行第一个非 0 元素的列位置。数值数组从上自下从左自右记录矩阵中非 0 元素数值。
3. 在本题中，由于矩阵中非零数值都为 1，故不设数值数组。

(二)CSR 的构建

```
while(feof(fp)==0){//当文件记录指针未到文件结尾
    fscanf(fp, _Format: "%d%d",&u,&v);
    //printf("%d %d\n",u,v);
    if(v>r)//保存最大列数，即行数
        r=v;
    if((r+2)> sizeof(row)){//所需空间不足，扩充加100
        row=(int *)realloc(row, _NewSize: (r+2+100)* sizeof(int));
    }
    edge++;
    if((edge+1)> sizeof(column)){
        column=(int *)realloc(column, _NewSize: (edge+1+200)*sizeof(int));
    }
    column[edge]=v;
    if(u!=oldu){//若u更新了,更新偏移量，因为是第一次输入u，故该(u,v)对的偏移量edge就是该行第一个非0元素的偏移量
        row[u]=edge;
        for(i=u-1;i>oldu;i--)//如果有行i的元素都是0，则row[i+1]-row[i]=0,即row[i]=row[i+1]
            row[i]=row[i+1];
    }
    oldu=u;
}
row[r+1]=edge+1;//最后在矩阵末尾补上矩阵元素总数
for(i=r;i>u;i--){//最后没有元素的行偏移为矩阵元素总数
    row[i]=row[r+1];
}
adj=(int *)malloc( _Size: (r+1)* sizeof(int));//r+1位存储了其相邻节点个数
```

(三)单项 BFS 算法思想

1. 广度优先搜索为每个顶点着色：白色、灰色或黑色。
2. 算法开始前所有顶点都是白色，随着搜索的进行，各顶点会逐渐变成灰色，然后成为黑色。
3. 在搜索中第一次碰到一顶点时，我们说该顶点被发现，此时该顶点变为非白色顶点。因此，灰色和黑色顶点都已被发现。
4. 若(u,v)∈E 且顶点 u 为黑色，那么顶点 v 要么是灰色，要么是黑色，就是说，所有和黑色顶点邻接的顶点都已被发现。灰色顶点可以与一些白色顶点相邻接，它们代表着已找到和未找到顶点之间的边界。
5. 广度优先搜索过程中建立了一棵广度优先树，起始时只包含根节点，即源顶点 s。在扫描已发现顶点 u 的邻接表的过程中每发现一个白色顶点 v，该顶点 v 及边(u,v)就被添加到树中。

(四)单向 BFS 的实现

```
G=(struct node *)malloc( Size: (r+1)* sizeof(struct node));
for(i=0;i<=r;i++){
    G[i].color=0;
    G[i].d=MAX;
    G[i].pi=-1;
}
G[s].color=1;
G[s].d=0;
G[s].pi=-1;
InitQueue();//构造队列
EnQueue(s);
while(!EmptyQueue()){//当队列不空
    u=DeQueue();
    Adj(u);//找出u的相邻点
    for(i=0;i<adj[r+1];i++){//对每个相邻点进行检测
        v=adj[i];
        if(G[v].color==0){//未被发现
            G[v].color=1;//已发现但未处理
            G[v].d=G[u].d+1;
            G[v].pi=u;
            EnQueue(v);
        }
    }
    G[u].color=2;//处理结束
}
```

(五)双向 BFS 算法思想

1. 双向 bfs 就是分别从起点和终点做两个 bfs，以此大幅度减少搜索的状态数。
2. 分别维护从起点开始的 bfs 队列 Q1 和从终点开始的 bfs 队列 Q2。
3. 使用白色表示未被发现的点，使用灰 1、黑 1 分别表示被起点队列发现未被处理和被起点队列处理的点，使用灰 2、黑 2 分别表示被终点发现未被处理和被终点队列处理的点。
4. 这样两个队列轮流处理未被发现的点，于是有两个边界分别以源和终点向外扩展，当这两个边界相遇，则说明发现最短路径。

(六)双向 BFS 的实现

```
while(!EmptyQueue(Q1) || !EmptyQueue(Q2)){//当队列不空
    if(!EmptyQueue(Q1)) {
        //printf("@Q1@");
        u = DeQueue(Q1);
        Adj(u);//找出u的相邻点
        for (i = 0; i < adj[r + 1]; i++) {//对每个相邻点进行检测
            v = adj[i];
            if (G[v].color == 0) {//未被发现
                G[v].color = 1;//已发现但未处理
                G[v].d = G[u].d + 1;
                G[v].pi = u;
                EnQueue(Q1, v);
               // printf("Q1En");
            }
            else if(G[v].color==3||G[v].color==4){//v节点被队列2发现过，相交了，该输出
                //由于两个队列间的交界面可能有多个节点（如教材上图的（0->7），故需要在该交界面上取最小的路径
                flag=1;
                if(G[u].d+G[v].d+1<min){
                    min=G[u].d+G[v].d+1;
                    *m=u;*n=v;
                }
            }
        }
        if(flag)
            return min;
        G[u].color = 2;//处理结束
    }
```

```
if(!EmptyQueue(Q2)) {
    //printf("@Q2@");
    u = DeQueue(Q2);
    Adj(u);//找出u的相邻点
   // printf("Q2(%d,%d)",u,adj[r+1]);
    for (i = 0; i < adj[r + 1]; i++) {//对每个相邻点进行检测
        v = adj[i];
        if (G[v].color == 0) {//未被发现
            G[v].color = 3;//已发现但未处理
            G[v].d = G[u].d + 1;
            G[v].pi = u;
            EnQueue(Q2, v);
           // printf("Q1En");
        }
        else if(G[v].color==1||G[v].color==2){//v节点被队列2发现过，相交了，输出
            flag=1;
            if(G[u].d+G[v].d+1<min){
                min=G[u].d+G[v].d+1;
                *m=v;*n=u;
            }
        }
    }
    if(flag)
        return min;
    G[u].color = 4;//处理结束
}
```

三、结果与分析：

题目一:

(一)单项 BFS

输出每个节点和源之间的距离和前驱节点，输出节点数过多，截取部分

```
node559:d=3,pi=348
node560:d=3,pi=348
node561:d=3,pi=348
node562:d=3,pi=348
node563:d=2,pi=107
node564:d=2,pi=198
node565:d=3,pi=348
node566:d=2,pi=107
node567:d=3,pi=348
node568:d=3,pi=348
node569:d=3,pi=348
node570:d=3,pi=348
node571:d=3,pi=348
node572:d=3,pi=348
node573:d=3,pi=414
node574:d=3,pi=414
```

```
node909:d=2,pi=107
node910:d=2,pi=107
node911:d=2,pi=107
node912:d=2,pi=107
node913:d=2,pi=107
node914:d=2,pi=107
node915:d=2,pi=107
node916:d=2,pi=107
node917:d=2,pi=107
node918:d=2,pi=107
node919:d=2,pi=107
node920:d=2,pi=107
node921:d=2,pi=107
node922:d=2,pi=107
node923:d=2,pi=107
node924:d=2,pi=107
node925:d=2,pi=107
```

(二)双向 BFS

```
Inputting from file....
Please input two nodes:4 3020
Distance:4
4->0->107->1505->3020
```

```
Inputting from file....
Please input two nodes:3020 4
Distance:4
3020->1505->107->0->4
```

```
Inputting from file....
Please input two nodes:2000 4000
Distance:5
2000->1912->428->594->3980->4000
```

四、备注

有可能影响结论的因素：

- 在双向 BFS 中，双方边界逐渐向外扩展，当其中一方的点发现被另一方所发现的点时，说明已经到达边界点。这时边界旁会有另一方的黑色、灰色的点，在这些点选择最短的路径点即为结果。

总结：

- 通过此次实验，了解学习了一种新的高效的稀疏矩阵存储格式 CSR。
- 实现了图的单项 BFS 算法和双向 BFS 算法。
- 对图搜索算法有了更加深入的了解。
- 广度优先搜索算法的总运行时间为 O(V+E)

附录（源代码）	算法源代码（C/C++/JAVA 描述）
	● 单向 BFS 算法
	#include <stdio.h>
	#include <stdlib.h>
	int MAX=999999999;
	int *row;//稀疏矩阵的行偏移数组，row[i]表示第 i 行第一个元素在非 0 元素中的偏移量，则 row[i+1]-row[i]为该行中的非 0 数

```
int *column;//对矩阵中的非 0 元素从左往右从上往下记录其列位置，colum[row[i]]即第 i 行第一个非 0 元素的列位置
// 我们的偏移量都是从 0 开始计算的
//由于我们只有两个点之间是否有边的信息，故矩阵非 0 即 1，则不用存储值数组了
int *adj;//存储一个节点的相邻节点
int r=-1;//记录矩阵的最大行列号(从 0 计)，即 row 矩阵的大小减 2
int edge=-1;//记录矩阵的最大边号(从 0 计)，即 column 矩阵的大小减 1
struct node{
    int color;//白色 0，灰色 1，黑色 2
    int d;
    int pi;
};
struct QNode{
    int data;
    struct QNode *next;
};//队列节点
struct QueueLink{
    struct QNode *front;
    struct QNode *rear;
};
struct QueueLink Q;//队列
struct node *G;//节点数组，存储每个节点的相关信息
void creat_CSR(){//处理上三角矩阵
    row=(int *)malloc(100* sizeof(int));//初始化 100，以后动态增加
    column=(int *)malloc(200* sizeof(int));//初始化 200，以后动态增加
    int u,v,oldu=0;int i;
    row[0]=0;
    FILE *fp;
    if((fp=fopen("E:\\alg project\\project5\\facebook_combined.txt","r"))==NULL){
        printf("Cannot open this file!");
        exit(0);
    }
    while(feof(fp)==0){//当文件记录指针未到文件结尾
        fscanf(fp,"%d%d",&u,&v);
        //printf("%d %d\n",u,v);
        if(v>r)//保存最大列数，即行数
            r=v;
        if((r+2)> sizeof(row)){//所需空间不足，扩充加 100
```

```
                row=(int *)realloc(row,(r+2+100)* sizeof(int));
            }
            edge++;
            if((edge+1)> sizeof(column)){
                column=(int
*)realloc(column,(edge+1+200)*sizeof(int));
            }
            column[edge]=v;
            if(u!=oldu){//若 u 更新了,更新偏移量，因为是第一次
输入 u，故该(u,v)对的偏移量 edge 就是该行第一个非 0 元素的
偏移量
                row[u]=edge;
                for(i=u-1;i>oldu;i--)//如果有行 i 的元素都是 0，则
row[i+1]-row[i]=0,即 row[i]=row[i+1]
                    row[i]=row[i+1];
            }
            oldu=u;
        }
        row[r+1]=edge+1;//最后在矩阵末尾补上矩阵元素总数
        for(i=r;i>u;i--){//最后没有元素的行偏移为矩阵元素总数
            row[i]=row[r+1];
        }
        adj=(int  *)malloc((r+1)*  sizeof(int));//r+1 位存储了其相邻
节点个数
      /* printf("row offsets:");
        for(i=0;i<=r+1;i++){
            printf("%d ",row[i]);
        }
        printf("\ncolumn indices:");
        for(i=0;i<=edge;i++){
            printf("%d ",column[i]);
        }*/

}

int search_CSR(int u,int v){//在压缩矩阵中查找(u,v),若存在边，
返回 1，不存在返回 0
        if(u==v)
            return 0;
        int tmp,i;
        if(u>v){
            tmp=u;
            u=v;
            v=tmp;
```

```
        }//令 u 始终大于 v
        if(v>column[row[u+1]-1])
            return 0;//如果比该行最后一个非 0 元素列号都大，说明不存在，减少比较次数
        for(i=row[u];i<row[u+1];i++){//在该行非 0 元素间查询，若有相同的列索引，说明存在该非零矩阵元素
            if(column[i]==v)
                return 1;
            if(column[i]>v)
                return 0;//由于对于该行非 0 元素列索引来说，其是递增的
            // 当当前的索引号都比 v 大，则后面的也比 v 大，故无该元素
        }
        return 0;
}
void Adj(int s){//s 点求与其相邻的点
        int i,j;
        adj[r+1]=row[s+1]-row[s];
        for(i=0;i<adj[r+1];i++)
            adj[i]=column[row[s]+i];//这只能求出上三角的相邻边
        for(j=0;j<=s;j++){//对下三角
            if(search_CSR(s,j)){
                adj[r+1]=adj[r+1]+1;
                adj[i]=j;
                //printf("*%d*",j);
                i++;
            }
        }
}
int InitQueue(){
        Q.front=Q.rear=(struct QNode*)malloc(sizeof(struct QNode));
        if(!Q.front)
            exit(0);
        Q.front->next=NULL;
        return 1;
}
int EnQueue(int e){
        struct QNode *p;
        p=(struct QNode*)malloc(sizeof(struct QNode));
        p->data=e;
        p->next=NULL;
        Q.rear->next=p;
```

```
        Q.rear=p;
        return 1;
    }
    int DeQueue(){//从列表从出队
        if(Q.rear==Q.front)
            exit(0);
        struct QNode *p;
        p=Q.front->next;
        int e=p->data;
        Q.front->next=p->next;
        if(Q.rear==p)
            Q.rear=Q.front;
        free(p);
        return e;
    }
    int EmptyQueue(){
        if(Q.front==Q.rear)
            return 1;
        return 0;
    }
    void BFS(int s){//以 s 为源进行广度优先搜索
        int i,u,v;
        G=(struct node *)malloc((r+1)* sizeof(struct node));
        for(i=0;i<=r;i++){
            G[i].color=0;
            G[i].d=MAX;
            G[i].pi=-1;
        }
        G[s].color=1;
        G[s].d=0;
        G[s].pi=-1;
        InitQueue();//构造队列
        EnQueue(s);
        while(!EmptyQueue()){//当队列不空
            u=DeQueue();
            Adj(u);//找出 u 的相邻点
            for(i=0;i<adj[r+1];i++){//对每个相邻点进行检测
                v=adj[i];
                if(G[v].color==0){//未被发现
                    G[v].color=1;//已发现但未处理
                    G[v].d=G[u].d+1;
                    G[v].pi=u;
                    EnQueue(v);
                }
```

```
            }
            G[u].color=2;//处理结束
        }
    }
    int main() {
        int i,j,s;
        printf("Inputting from file.... \n");
        creat_CSR();
     /*     for(i=0;i<=r;i++){
            printf("\n");
            for(j=0;j<=r;j++)
                printf("%d ",search_CSR(i,j));
        }
           for(j=0;j<=r;j++) {
                printf("\n");
                Adj(j);//找出 u 的相邻点
                for (i = 0; i < adj[r + 1]; i++) {//对每个相邻点进行
    检测
                    printf("%d ", adj[i]);
                }
           }*/
        printf("\nPlease input a node: ");
        scanf("%d",&s);
        BFS(s);
        printf("\n");
        for(i=0;i<=r;i++){
            printf("node%d:d=%d,pi=%d\n",i,G[i].d,G[i].pi);
        }

    }
```

- 双向 BFS 算法

```
#include <stdio.h>
#include <stdlib.h>
int MAX=999999999;
int  *row;//稀疏矩阵的行偏移数组，row[i]表示第 i 行第一个元素在非 0 元素中的偏移量，则 row[i+1]-row[i]为该行中的非 0 数
int *column;//对矩阵中的非 0 元素从左往右从上往下记录其列位置，colum[row[i]]即第 i 行第一个非 0 元素的列位置
// 我们的偏移量都是从 0 开始计算的
//由于我们只有两个点之间是否有边的信息，故矩阵非 0 即 1，则不用存储值数组了
int *adj;//存储一个节点的相邻节点
```

```
int r=-1;//记录矩阵的最大行列号(从 0 计），即 row 矩阵的大小减 2
int edge=-1;//记录矩阵的最大边号(从 0 计），即 column 矩阵的大小减 1
struct node{
    int color;//白色 0，灰色 1 3，黑色 2 4
    int d;
    int pi;
};
struct QNode{
    int data;
    struct QNode *next;
};//队列节点
struct QueueLink{
    struct QNode *front;
    struct QNode *rear;
};
struct QueueLink *Q1,*Q2;//队列
struct node *G;//节点数组，存储每个节点的相关信息
void creat_CSR(){//处理上三角矩阵
    row=(int *)malloc(100* sizeof(int));//初始化 100，以后动态增加
    column=(int  *)malloc(500*  sizeof(int));//初始化 500，以后动态增加
    int u,v,oldu=0;int i;
    row[0]=0;
    FILE *fp;
    if((fp=fopen("E:\\alg project\\project5\\facebook_combined.txt","r"))==NULL){
        printf("Cannot open this file!");
        exit(0);
    }
    while(feof(fp)==0){//当文件记录指针未到文件结尾
        fscanf(fp,"%d%d",&u,&v);
        //printf("%d %d\n",u,v);
        if(v>r)//保存最大列数，即行数
            r=v;
        if((r+2)> sizeof(row)){//所需空间不足，扩充加 100
            row=(int *)realloc(row,(r+2+100)* sizeof(int));
        }
        edge++;
        //printf("@%d@",edge);
        if((edge+1)> sizeof(column)){
            column=(int
```

```
*)realloc(column,(edge+1+200)*sizeof(int));
            }
            column[edge]=v;
            if(u!=oldu){//若 u 更新了,更新偏移量，因为是第一次输入 u，故该(u,v)对的偏移量 edge 就是该行第一个非 0 元素的偏移量
                row[u]=edge;
                for(i=u-1;i>oldu;i--)//如果有行 i 的元素都是 0，则 row[i+1]-row[i]=0,即 row[i]=row[i+1]
                    row[i]=row[i+1];
            }
            oldu=u;
        }
        row[r+1]=edge+1;//最后在矩阵末尾补上矩阵元素总数
        for(i=r;i>u;i--){//最后没有元素的行偏移为矩阵元素总数
            row[i]=row[r+1];
        }
        adj=(int  *)malloc((r+1)*  sizeof(int));//r+1 位存储了其相邻节点个数
      // printf("row offsets:");
        /*for(i=0;i<=r+1;i++){
            printf("%d ",row[i]);
        }
        printf("\ncolumn indices:");
        for(i=0;i<=edge;i++){
            printf("%d ",column[i]);
        }*/

}
int search_CSR(int u,int v){//在压缩矩阵中查找(u,v),若存在边，返回 1，不存在返回 0
        if(u==v)
            return 0;
        int tmp,i;
        if(u>v){
            tmp=u;
            u=v;
            v=tmp;
        }//令 u 始终大于 v
        if(v>column[row[u+1]-1])
            return 0;//如果比该行最后一个非 0 元素列号都大，说明不存在，减少比较次数
        for(i=row[u];i<row[u+1];i++){//在该行非 0 元素间查询，若有相同的列索引，说明存在该非零矩阵元素
```

```
            if(column[i]==v)
                return 1;
            if(column[i]>v)
                return 0;//由于对于该行非 0 元素列索引来说，其是递增的
            // 当当前的索引号都比 v 大，则后面的也比 v 大，故无该元素
        }
        return 0;
}

void Adj(int s){//s 点求与其相邻的点
        int i,j;
        adj[r+1]=row[s+1]-row[s];
        for(i=0;i<adj[r+1];i++)
            adj[i]=column[row[s]+i];//这只能求出上三角的相邻边
        for(j=0;j<=s;j++){//对下三角
            if(search_CSR(s,j)){
                adj[r+1]=adj[r+1]+1;
                adj[i]=j;
                //printf("*%d*",j);
                i++;
            }
        }
}

int InitQueue(struct QueueLink *Q){
        Q->front=Q->rear=(struct QNode*)malloc(sizeof(struct QNode));
        if(!Q->front)
            exit(0);
        Q->front->next=NULL;
        return 1;
}
int EnQueue(struct QueueLink *Q,int e){
        struct QNode *p;
        p=(struct QNode*)malloc(sizeof(struct QNode));
        p->data=e;
        p->next=NULL;
        Q->rear->next=p;
        Q->rear=p;
        return 1;
}
int DeQueue(struct QueueLink *Q){//从列表从出队
```

```
        if(Q->rear==Q->front)
            exit(0);
        struct QNode *p;
        p=Q->front->next;
        int e=p->data;
        Q->front->next=p->next;
        if(Q->rear==p)
            Q->rear=Q->front;
        free(p);
        return e;
}
int EmptyQueue(struct QueueLink *Q){
        if(Q->front==Q->rear)
            return 1;
        return 0;
}

int BFS2(int s,int f,int *m,int *n){//以 s 为源，f 为终点进行双向广度优先搜索,返回两节点距离，和交界点节点
        int i,u,v,min=MAX,flag=0;
        G=(struct node *)malloc((r+1)* sizeof(struct node));
        Q1=(struct QueueLink*)malloc(sizeof(struct QueueLink));
        Q2=(struct QueueLink*)malloc(sizeof(struct QueueLink));
        for(i=0;i<=r;i++){
            G[i].color=0;//0 为白色，1、3 分别为队列 1、2 的灰色，2、4 分别为队列 1、2 的黑色
            G[i].d=MAX;
            G[i].pi=-1;
        }
        G[s].color=1;G[f].color=3;
        G[s].d=0;G[f].d=0;
        G[s].pi=-1;G[f].pi=-1;
        InitQueue(Q1);//构造队列
        InitQueue(Q2);
        EnQueue(Q1,s);
        EnQueue(Q2,f);
        while(!EmptyQueue(Q1) || !EmptyQueue(Q2)){//当队列不空
            if(!EmptyQueue(Q1)) {
                //printf("@Q1@");
                u = DeQueue(Q1);
                Adj(u);//找出 u 的相邻点
                for (i = 0; i < adj[r + 1]; i++) {//对每个相邻点进行检测
```

```
                    v = adj[i];
                    if (G[v].color == 0) {//未被发现
                        G[v].color = 1;//已发现但未处理
                        G[v].d = G[u].d + 1;
                        G[v].pi = u;
                        EnQueue(Q1, v);
                       // printf("Q1En");
                    }
                    else  if(G[v].color==3||G[v].color==4){//v 节点被队列 2 发现过，相交了，该输出
                        //由于两个队列间的交界面可能有多个节点（如教材上图的（0->7），故需要在该交界面上取最小的路径
                        flag=1;
                        if(G[u].d+G[v].d+1<min){
                            min=G[u].d+G[v].d+1;
                            *m=u;*n=v;
                        }
                    }
                }
                if(flag)
                    return min;
                G[u].color = 2;//处理结束
            }
            if(!EmptyQueue(Q2)) {
                //printf("@Q2@");
                u = DeQueue(Q2);
                Adj(u);//找出 u 的相邻点
               // printf("Q2(%d,%d)",u,adj[r+1]);
                for (i = 0; i < adj[r + 1]; i++) {//对每个相邻点进行检测
                    v = adj[i];
                    if (G[v].color == 0) {//未被发现
                        G[v].color = 3;//已发现但未处理
                        G[v].d = G[u].d + 1;
                        G[v].pi = u;
                        EnQueue(Q2, v);
                       // printf("Q1En");
                    }
                    else  if(G[v].color==1||G[v].color==2){//v 节点被队列 2 发现过，相交了，输出
                        flag=1;
                        if(G[u].d+G[v].d+1<min){
                            min=G[u].d+G[v].d+1;
```

```
                                        *m=v;*n=u;
                                    }
                                }
                            }
                            if(flag)
                                return min;
                            G[u].color = 4;//处理结束
                        }
                    }
                    return 0;
                }
                void print(int m){
                    if(m==-1)
                        return;
                    print(G[m].pi);
                    printf("%d->",m);
                }
                int main() {
                    int i,j,s,f,m,n,d;
                    printf("Inputting from file.... \n");
                    creat_CSR();
                    /*for(i=0;i<=r;i++){
                        printf("\n");
                        for(j=0;j<=r;j++)
                            printf("%d ",search_CSR(i,j));
                    }*/
                    printf("Please input two nodes:");
                    scanf("%d %d",&s,&f);
                    d=BFS2(s,f,&m,&n);
                    //for(i=0;i<=r;i++){
                    //      printf("node%d:d=%d,pi=%d\n",i,G[i].d,G[i].pi);
                    //}
                    printf("Distance:%d\n",d);
                    print(m);
                    do{
                        if(n==f)
                            printf("%d",n);
                        else
                            printf("%d->",n);
                        n=G[n].pi;
                    }while (n!=-1);
                }
```