### 题目功能描述

设计一个交通咨询系统，在系统能让旅客查询从任一个城市顶点到 另一个城市顶点之间的最短路径问题。采用图来构造各个城市之间的联系，图中 顶点表示城市，边表示各个城市之间的交通关系，所带权值包括两个城市间的交 通费用、距离、时间。**（难易程度：中）**

### 问题分析与算法设计思路

该题本质是在考察图的最短路径算法的实现，其中第2问和第3问分别是利用Folyd算法和Dijkstra算法进行求解；同时本问题中的交通网络图采用带权的无向图来进行建模实现，最终实现了从某一城市到其他城市之间的最短路径以及对于两两城市之间最短路径的求解。

### 数据结构定义

struct Graph

{

int n; //本程序中，程序的标号是从一开始

int m;

int Matrix[233][233] = {0};

};

### 函数功能描述

#### 函数

void Great(Graph \*G) 创建城市之间的交通网络

void Dijkstra(Graph \*G) 使用Dijkstra算法求解从一个城市到其他城市的最短路径

void Floyd(Graph \*G) 使用Floyd算法求解每一对城市之间的最短路径

功能描述

**Great函数**通过借用Graph的struct来完成交通网络的创建（包括城市结点的数量，边的数量，边的权重等，这里边的权重可以指里程，花费等，可根据实际情况输入）

**Dijkstra函数**利用迪杰特斯拉算法的思想,将图中结点集合分为两组，第一组为已经求出最短路径的顶点集合，另一组为确定最短路径的顶点集合，其中D[]用于记录初始的城市到其余各个城市的最短路径，p[]集合用于记录最短的路径中顶点i生成的情况，S[]集合用于表示最短的路径的生成的情况。

**Floyd函数**利用弗洛伊德算法来完成对任意两城市之间最短路径的求解，Path[][]为路径的矩阵，而D[i][j]数组则存储了从i城市到j城市的距离。

### 程序流程图

### 程序源代码

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>                                 //有INT\_MAX可以直接调用整形之中最大的数

#include <iostream>

#include <string>

struct Graph

{

    int n;                                         //本程序中，程序的标号是从一开始，而不是从0开始

    int m;

    int Matrix[233][233] = {0};

};

void Great(Graph \*G){

    printf("请输入城市的数量：");

    scanf("%d",&(G->n));

    printf("请输入城市之间连通路线的数量: ");

    scanf("%d", &(G->m));

    for(int i = 1;i<=233;i++){

        for(int j = 1;j<=233;j++){

           G->Matrix[i][j] = 10000;               //该矩阵默认全部是最大的值

        }

    }

    printf("请输入该城市交通网络的邻接矩阵的形式：");

    for(int i = 0;i<G->m;i++){                    //将边给输入好

        int j,k;                                  //j代表横坐标，k代表纵坐标

        int Temp;                               //Temp代表权值的大小

        scanf("%d %d %d",&j,&k,&Temp);

        G->Matrix[j][k] = Temp;

        G->Matrix[k][j] = Temp;

    }

}

void Dijkstra(Graph \*G){                          //从一个城市出发到其他城市的最短的路径

    printf("请输入你选择的城市的序号（从1开始）：");

    int D[233];

    int p[233];                                   //记录最短的路径中顶点i的前驱顶点

    int s[233];                                   //用于标识最短的路径的生成的情况,为1表示从原点到i的最短的路径已经找到

    int v;

    int k;                                        //这个k具体是干啥用得呢

    scanf("%d",&v);

    int min,max = 10000,pre;

    for(int i = 1;i<=G->n;i++){

        D[i] = G->Matrix[v][i];

        if(D[i] != INT\_MAX)                      //如果这里不是空的

            p[i] = v + 1;

        else

            p[i] = 0;

        s[i] = 0;

    }

    s[v] = 1;                                    //将原点送到U

    for(int i = 1;i<G->n;i++){

        min = 10001;                             //min > max 让最大值得那个也能加入U

        for(int j = 1;j<=G->n;j++){

            if((!s[j]) && (D[j] < min)){

                min = D[j];

                k = j;

            }

        }

        s[k] = 1;                                //将找到得顶点k送入U

        for(int j = 1;j<=G->n;j++){

            if((!s[j]) && (D[j] > D[k] + G->Matrix[k][j]))

            {

                D[j] = D[k] + G->Matrix[k][j];

                p[j] = k+1;                      //k是j的前驱

            }

        }                                        //所有的顶点已经扩充到了U中

        }

        for(int k = 1;k<=G->n;k++){

            if(k == v && k <  10){

                printf(" 0 %10d",k);

            }else if(k == v && k >= 10){

                printf(" 0 %11d",k);

            }else if(D[k] == 10000 && k < 10){

                printf(" ∞ %10d",k);

            }else if(D[k] == 10000 && k >= 10){

                printf(" ∞ %11d",k);

            }else{

                printf(" %-10d %d",D[k],k);

            }

            pre= p[k];

            if(k != v){

            while((pre != 0) && (pre != v+1)){

                printf("<\_%d",pre-1);

                pre = p[pre-1];

            }

            printf("<\_%d",v);}

            printf("\n");

    }

}

void Floyd(Graph \*G){

    printf("接下来输出每一对城市之间的最短路径：\n");

    int Path[233][233];                             //路径矩阵

    int D[233][233];

    int i,j,k,pre;

    int w,max = 10000;

    for(i = 0;i<=G->n;i++){

        for(j = 0;j<=G->n;j++){

            if(G->Matrix[i][j] != max){

                Path[i][j] = i+1;                   //i是j得前驱

            }else{

                Path[i][j] = 0;

            }

            D[i][j] = G->Matrix[i][j];

        }

    }

    for(k = 1;k<=G->n;k++){

        for(i = 1;i<=G->n;i++){

            for(j = 1;j<=G->n;j++){

                if(D[i][j] > (D[i][k] + D[k][j])){

                    D[i][j] = D[i][k] + D[k][j];    //修改路径的长度

                    Path[i][j] = Path[k][j];        //修改路径

                }

            }

        }

    }

    for(i = 1;i<=G->n;i++){

        printf("第%d组：\n",i);

        for(j = 1;j<=G->n;j++){

            if(i == j && j < 10){

                printf(" 0 %10d",j);

            }else if(i == j && j >= 10){

                printf(" 0 %11d",j);

            }else if(D[i][j] == 10000 && j < 10){

                printf(" ∞ %10d",j);

            }else if(D[i][j] == 10000 && j >= 10){

                printf(" ∞ %11d",j);

            }else{

            printf(" %-10d %d",D[i][j],j);

            }

            pre = Path[i][j];

            if(i!=j){

            while((pre != 0) && (pre != i+1)){

                printf("<\_%d",pre-1);

                pre = Path[i][pre-1];

            }

            }

            printf("<\_%d\n",i);

        }

    }

}

int main(void){

    Graph G;

    Great(&G);

    Dijkstra(&G);

    Floyd(&G);

    return 0;

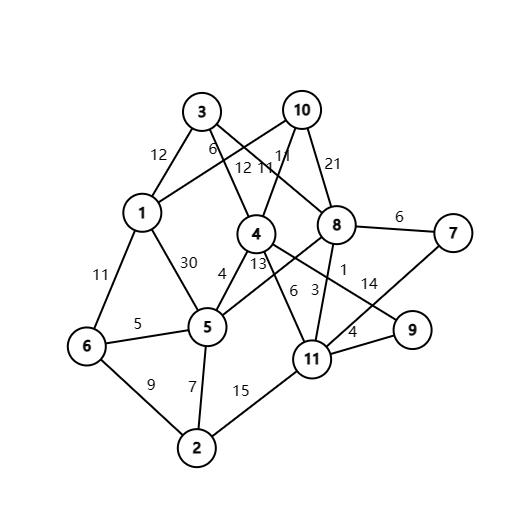
}

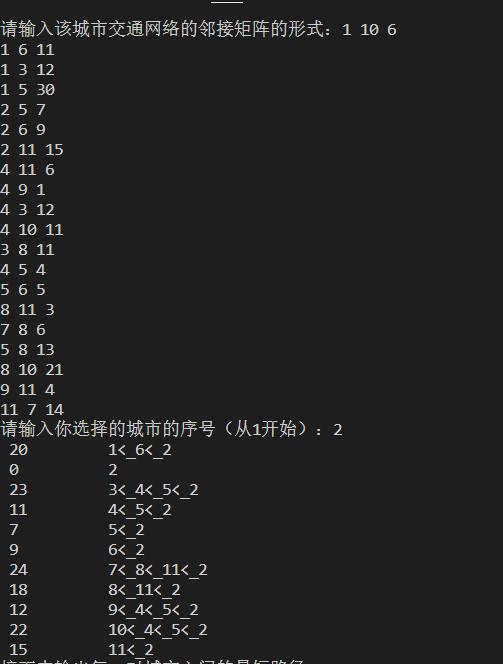
### 数据输入/输出截图

**测试样例1：**

Graph1

该图由11个结点和20条边，属于交通网络中较复杂的情况

、

输入输出如下：

由于求任意两城市最短路径的输出太长，故以文本形式给出  
**接下来输出每一对城市之间的最短路径：**

**第1组：**

**0 1<\_1**

**20 2<\_6<\_1**

**12 3<\_1**

**17 4<\_10<\_1**

**16 5<\_6<\_1**

**11 6<\_1**

**29 7<\_8<\_3<\_1**

**23 8<\_3<\_1**

**18 9<\_4<\_10<\_1**

**6 10<\_1**

**22 11<\_9<\_4<\_10<\_1**

**第2组：**

**20 1<\_6<\_2**

**0 2<\_2**

**23 3<\_4<\_5<\_2**

**11 4<\_5<\_2**

**7 5<\_2**

**9 6<\_2**

**24 7<\_8<\_11<\_2**

**18 8<\_11<\_2**

**12 9<\_4<\_5<\_2**

**22 10<\_4<\_5<\_2**

**15 11<\_2**

**第3组：**

**12 1<\_3**

**23 2<\_5<\_4<\_3**

**0 3<\_3**

**12 4<\_3**

**16 5<\_4<\_3**

**21 6<\_5<\_4<\_3**

**17 7<\_8<\_3**

**11 8<\_3**

**13 9<\_4<\_3**

**18 10<\_1<\_3**

**14 11<\_8<\_3**

**第4组：**

**17 1<\_10<\_4**

**11 2<\_5<\_4**

**12 3<\_4**

**0 4<\_4**

**4 5<\_4**

**9 6<\_5<\_4**

**14 7<\_8<\_11<\_9<\_4**

**8 8<\_11<\_9<\_4**

**1 9<\_4**

**11 10<\_4**

**5 11<\_9<\_4**

**第5组：**

**16 1<\_6<\_5**

**7 2<\_5**

**16 3<\_4<\_5**

**4 4<\_5**

**0 5<\_5**

**5 6<\_5**

**18 7<\_8<\_11<\_9<\_4<\_5**

**12 8<\_11<\_9<\_4<\_5**

**5 9<\_4<\_5**

**15 10<\_4<\_5**

**9 11<\_9<\_4<\_5**

**第6组：**

**11 1<\_6**

**9 2<\_6**

**21 3<\_4<\_5<\_6**

**9 4<\_5<\_6**

**5 5<\_6**

**0 6<\_6**

**23 7<\_8<\_11<\_9<\_4<\_5<\_6**

**17 8<\_11<\_9<\_4<\_5<\_6**

**10 9<\_4<\_5<\_6**

**17 10<\_1<\_6**

**14 11<\_9<\_4<\_5<\_6**

**第7组：**

**29 1<\_3<\_8<\_7**

**24 2<\_11<\_8<\_7**

**17 3<\_8<\_7**

**14 4<\_9<\_11<\_8<\_7**

**18 5<\_4<\_9<\_11<\_8<\_7**

**23 6<\_5<\_4<\_9<\_11<\_8<\_7**

**0 7<\_7**

**6 8<\_7**

**13 9<\_11<\_8<\_7**

**25 10<\_4<\_9<\_11<\_8<\_7**

**9 11<\_8<\_7**

**第8组：**

**23 1<\_3<\_8**

**18 2<\_11<\_8**

**11 3<\_8**

**8 4<\_9<\_11<\_8**

**12 5<\_4<\_9<\_11<\_8**

**17 6<\_5<\_4<\_9<\_11<\_8**

**6 7<\_8**

**0 8<\_8**

**7 9<\_11<\_8**

**19 10<\_4<\_9<\_11<\_8**

**3 11<\_8**

**第9组：**

**18 1<\_10<\_4<\_9**

**12 2<\_5<\_4<\_9**

**13 3<\_4<\_9**

**1 4<\_9**

**5 5<\_4<\_9**

**10 6<\_5<\_4<\_9**

**13 7<\_8<\_11<\_9**

**7 8<\_11<\_9**

**0 9<\_9**

**12 10<\_4<\_9**

**4 11<\_9**

**第10组：**

**6 1<\_10**

**22 2<\_5<\_4<\_10**

**18 3<\_1<\_10**

**11 4<\_10**

**15 5<\_4<\_10**

**17 6<\_1<\_10**

**25 7<\_8<\_11<\_9<\_4<\_10**

**19 8<\_11<\_9<\_4<\_10**

**12 9<\_4<\_10**

**0 10<\_10**

**16 11<\_9<\_4<\_10**

**第11组：**

**22 1<\_10<\_4<\_9<\_11**

**15 2<\_11**

**14 3<\_8<\_11**

**5 4<\_9<\_11**

**9 5<\_4<\_9<\_11**

**14 6<\_5<\_4<\_9<\_11**

**9 7<\_8<\_11**

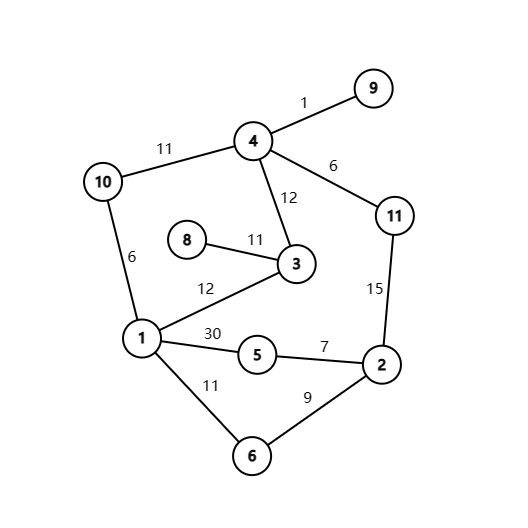
**3 8<\_11**

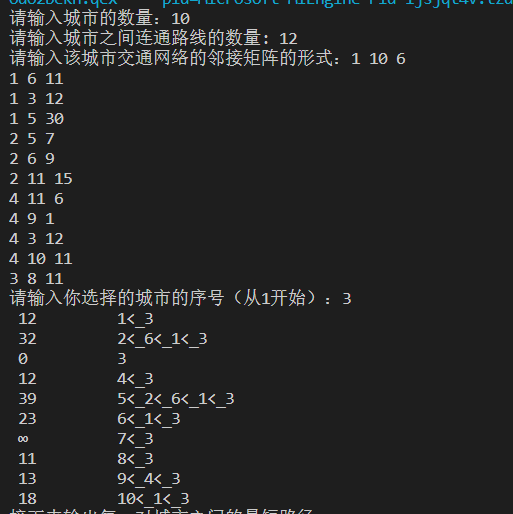
**4 9<\_11**

**16 10<\_4<\_9<\_11**

1. **11<\_11**

**测试样例二：**

该图有10个结点，以及12条边

输入输出如下：

**接下来输出每一对城市之间的最短路径：**

**第1组：**

**0 1<\_1**

**20 2<\_6<\_1**

**12 3<\_1**

**17 4<\_10<\_1**

**27 5<\_2<\_6<\_1**

**11 6<\_1**

**∞ 7<\_1**

**23 8<\_3<\_1**

**18 9<\_4<\_10<\_1**

**6 10<\_1**

**第2组：**

**20 1<\_6<\_2**

**0 2<\_2**

**32 3<\_1<\_6<\_2**

**37 4<\_10<\_1<\_6<\_2**

**7 5<\_2**

**9 6<\_2**

**∞ 7<\_2**

**43 8<\_3<\_1<\_6<\_2**

**38 9<\_4<\_10<\_1<\_6<\_2**

**26 10<\_1<\_6<\_2**

**第3组：**

**12 1<\_3**

**32 2<\_6<\_1<\_3**

**0 3<\_3**

**12 4<\_3**

**39 5<\_2<\_6<\_1<\_3**

**23 6<\_1<\_3**

**∞ 7<\_3**

**11 8<\_3**

**13 9<\_4<\_3**

**18 10<\_1<\_3**

**第4组：**

**17 1<\_10<\_4**

**37 2<\_6<\_1<\_10<\_4**

**12 3<\_4**

**0 4<\_4**

**44 5<\_2<\_6<\_1<\_10<\_4**

**28 6<\_1<\_10<\_4**

**∞ 7<\_4**

**23 8<\_3<\_4**

**1 9<\_4**

**11 10<\_4**

**第5组：**

**27 1<\_6<\_2<\_5**

**7 2<\_5**

**39 3<\_1<\_6<\_2<\_5**

**44 4<\_10<\_1<\_6<\_2<\_5**

**0 5<\_5**

**16 6<\_2<\_5**

**∞ 7<\_5**

**50 8<\_3<\_1<\_6<\_2<\_5**

**45 9<\_4<\_10<\_1<\_6<\_2<\_5**

**33 10<\_1<\_6<\_2<\_5**

**第6组：**

**11 1<\_6**

**9 2<\_6**

**23 3<\_1<\_6**

**28 4<\_10<\_1<\_6**

**16 5<\_2<\_6**

**0 6<\_6**

**∞ 7<\_6**

**34 8<\_3<\_1<\_6**

**29 9<\_4<\_10<\_1<\_6**

**17 10<\_1<\_6**

**第7组：**

**∞ 1<\_7**

**∞ 2<\_7**

**∞ 3<\_7**

**∞ 4<\_7**

**∞ 5<\_7**

**∞ 6<\_7**

**0 7<\_7**

**∞ 8<\_7**

**∞ 9<\_7**

**∞ 10<\_7**

**第8组：**

**23 1<\_3<\_8**

**43 2<\_6<\_1<\_3<\_8**

**11 3<\_8**

**23 4<\_3<\_8**

**50 5<\_2<\_6<\_1<\_3<\_8**

**34 6<\_1<\_3<\_8**

**∞ 7<\_8**

**0 8<\_8**

**24 9<\_4<\_3<\_8**

**29 10<\_1<\_3<\_8**

**第9组：**

**18 1<\_10<\_4<\_9**

**38 2<\_6<\_1<\_10<\_4<\_9**

**13 3<\_4<\_9**

**1 4<\_9**

**45 5<\_2<\_6<\_1<\_10<\_4<\_9**

**29 6<\_1<\_10<\_4<\_9**

**∞ 7<\_9**

**24 8<\_3<\_4<\_9**

**0 9<\_9**

**12 10<\_4<\_9**

**第10组：**

**6 1<\_10**

**26 2<\_6<\_1<\_10**

**18 3<\_1<\_10**

**11 4<\_10**

**33 5<\_2<\_6<\_1<\_10**

**17 6<\_1<\_10**

**∞ 7<\_10**

**29 8<\_3<\_1<\_10**

**12 9<\_4<\_10**

**0 10<\_10**

### 算法核心思想

#### 迪杰斯特拉(Dijkstra)算法

Dijkstra(迪杰斯特拉)算法是典型的单源最短路径算法，用于计算一个节点到其他所有节点的最短路径。主要特点是以起始点为中心向外层层扩展，直到扩展到终点为止。

算法步骤：

* 将所有的顶点分为两部分：已知最短路程的顶点集合P和未知最短路径的顶点集合Q。最开始，已知最短路径的顶点集合P中只有源点一个顶点。我们这里用一个S[ i ]数组来记录哪些点在集合P中。例如对于某个顶点i，如果S[ i ]为1则表示这个顶点在集合P中，如果S[ i ]为0则表示这个顶点在集合Q中。（注：此处P和Q均未出现在程序中，其出现是为了更好的阐述算法思想）
* 设置源点A到自己的最短路径为0即D[i]=0（i为A的编号）。若存在源点有能直接到达的顶点i，则把D[ i ]设为G->Matrix[ s ][ i ]。同时把所有其它（源点不能直接到达的）顶点的最短路径为设为。
* 在集合Q的所有顶点中选择一个离源点s最近的顶点u（即dis[u]最小）加入到集合P。并考察所有以点u为起点的边，对每一条边进行松弛操作。例如存在一条从u到v的边，那么可以通过将边u->v添加到尾部来拓展一条从s到v的路径，这条路径的长度是D[u]+ G->Matrix [u][v]。如果这个值比目前已知的D[v]的值要小，我们可以用新值来替代当前dis[v]中的值。
* 重复第3步，如果集合Q为空，算法结束。最终dis数组中的值就是源点到所有顶点的最短路径。

#### 弗洛伊德(Floyd)算法

Floyd算法是解决任意两点间的最短路径的一种算法，可以正确处理有向图或负权的最短路径问题，同时也被用于计算有向图的传递闭包。Floyd算法的时间复杂度为。

算法步骤：

* 从任意一条单边路径开始。所有两点之间的距离是边的权，如果两点之间没有边相连，则权为无穷大。
* 对于每一对顶点 u 和 v，看看是否存在一个顶点 w 使得从 u 到 w 再到 v 比己知的路径更短。如果是更新它。

#### 关于第三问的算法实现分析

第二问中使用的迪杰斯特拉(Dijkstra)算法其实也可以用于第三问中求任意两对顶点之间的最短路径问题，但是一般多源的最短路径还是由弗洛伊德(Floyd)算法去求解

本文在此分析了下原因：

Dijkstra时间复杂度为空间复杂度为  
Flyod时间复杂度为空间复杂度为

故在求取单源问题时时候一般倾向于使用Dijkstra，若是将Dijkstra中的存取最短路径D[]数组拓展为2维情况时候，其时间复杂度和空间复杂度都会去乘以，最终导致时间复杂度与Floyd相同的情况下，空间复杂度不如Floyd。

综上，故针对任意两队结点之间的最短路径的求取，Floyd仍是最主流的算法。

### 图存储结构（邻接矩阵）

#### 优缺点

容易获得每个边的权值，取得第i条边的权值的时间复杂度仅为。

同时能较方便地实现本程序中迪杰斯特拉算法与弗洛伊德算法。

缺点为对于边数相对顶点较少的图，浪费了极大的存储空间，是典型地以空间换时间的存储形式。

#### 采用邻接矩阵的图存储结构的理由

存取速度高效,通过下标来直接存储。

无需为表示结点间的逻辑关系而增加额外的存储空间。

比起其他的存储结构，能更方便地实现解决问题所需的算法。

### 实验心得体会

深化了我对于求取最短路径的Floyd和Dijkstra两大算法的理解；

在程序实现的过程中，强化了记忆，提高了自己的编程能力；

此外，在探索第三问其他的求解算法的过程中，我对于两种算法的复杂度的讨论有了更为深刻的理解和认识；

综上，很有幸能参于这次数据结构大作业，感谢这次大作业为我带来的能力上的提升。