|  |  |
| --- | --- |
| 离散数学实验I  实验报告  南京航空航天大学图册_360百科 | |
| 姓名： |  |
| 学号： |  |
| 联系方式： |

目录

[习题2 3](#_Toc152423789)

[习题3 3](#_Toc152423790)

[习题5 欧拉回路 5](#_Toc152423791)

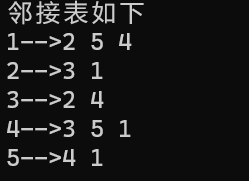
[习题6最小哈密顿回路 10](#_Toc152423792)

## 习题2

使用邻接表表示下图，并打印出邻接表。比较邻接表和矩阵表示哪一种更节省存储空间？



**程序运行结果如下。**



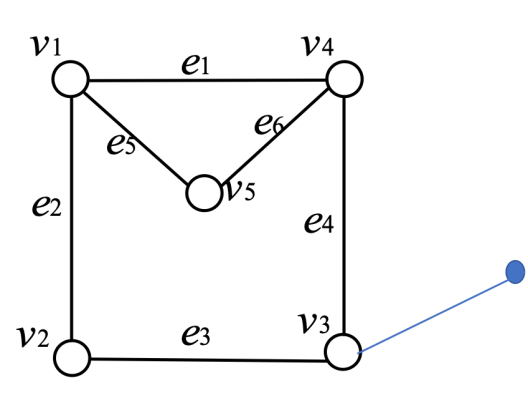
**空间比较：总体来说，邻接表更节省存储空间。**

对于一个具有n个顶点e条边的无向图，它的邻接表表示有n个顶点表结点2e个边表结点；对于一个具有n个顶点e条边的有向图，它的邻接表表示有n个顶点表结点e个边表结点。如果图中边的数目远远小于n2称作稀疏图，这是用邻接表表示比用邻接矩阵表示节省空间;如果图中边的数目接近于n2,对于无向图接近于n\*(n-1)称作稠密图,考虑到邻接表中要附加链域，采用邻接矩阵表示法为宜。

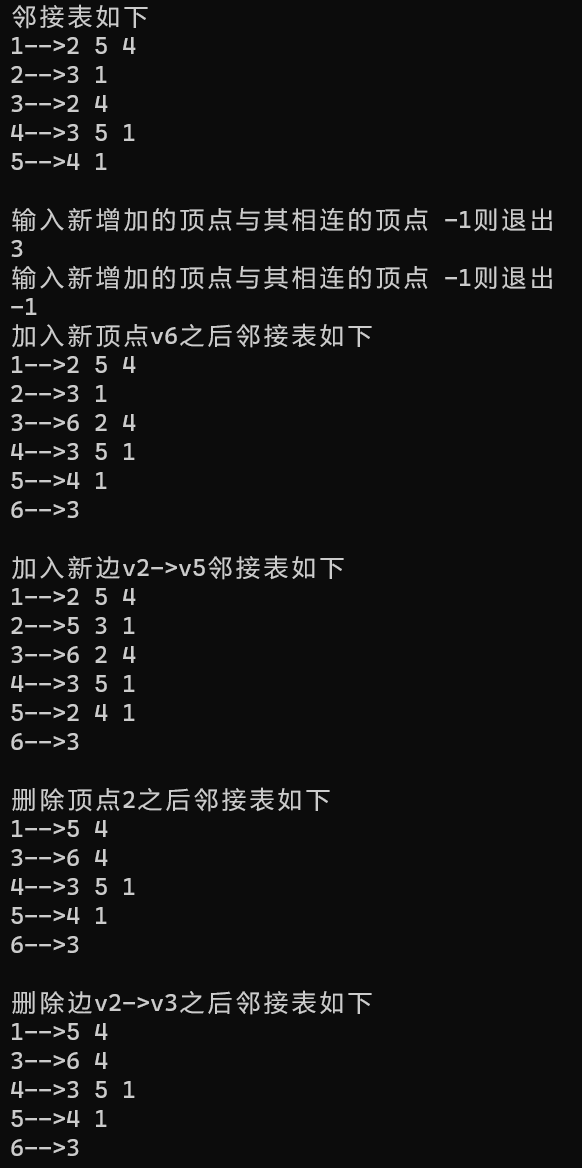
**源代码见附件。**

## 习题3

考虑在已有图上，增加顶点(如下图)、边（v2v5）、删除顶点（v2）、边(e3)如何处理？  
比较矩阵和邻接表两种结构哪种更高效？



程序运行结果如下:



**逐个分析上述操作在邻接矩阵和邻接表中的不同：**

**增加顶点:**

邻接矩阵： 添加一个顶点需要在现有矩阵的基础上增加一行和一列。这需要O(V^2)的时间，其中V是顶点数。

邻接表： 添加一个顶点只需要添加一个新的链表，因此是O(1)的时间。

**增加边:**

邻接矩阵：在矩阵中标记两个顶点之间的边需要O(1)的时间。

邻接表：在两个链表中插入新的边，需要O(V)的时间，其中degree是相关顶点的度数。

**删除顶点:**

邻接矩阵：删除一个顶点需要删除相关的行和列，因此需要O(V^2)的时间。

邻接表：删除一个顶点需要删除相关的链表，以及其他链表中包含对应顶点的边，因此是O(max(degree,V)的时间。其中degree是相关顶点的度数。

**删除边:**

邻接矩阵：在矩阵中取消两个顶点之间的边需要O(1)的时间。

邻接表：在相关链表中删除一条边，需要O(degree)的时间。

总体来说，邻接表在处理稀疏图时通常更有效，因为它可以更好地利用空间。对于增加和删除顶点，邻接表通常更快，因为它们只涉及到相关链表的操作，而不需要修改整个矩阵。然而，在密集图的情况下，邻接矩阵的查询操作可能更快，因为它们直接在矩阵中查找边的存在。因此，选择那种数据结构取决于图的特征和对图操作的需求。

**源代码见附件。**

## 习题5 欧拉回路

**5.1问题描述**

寻找一条通过图中每条边恰好一次的回路。

**5.2算法概述**

Hierholzer（逐步插入回路法）。从一条回路开始，每次任取一条目前回路中的点，将其替换为一条简单回路，以此寻找到一条欧拉回路。如果从路开始的话，就可以寻找到一条欧拉路。

**5.3 数据结构描述**

vector<vector<int>> g;  // 采用邻接矩阵存储图

**5.4 伪码和关键代码描述**

**核心求欧拉回路算法（Hierholzer）伪代码：**

Input.The edges of the graph e,where each element in e **is** (u,v)

Ouput.The vertex of the Euler Road of the input graph

Method.

Function **Hierholzer**(v)

    circle <- Find a Circle in e Begin with v

    if circle = Ø

        return v

    e <- e-circle

    for each v∈circle

        v <- **Hierholzer**(v)

    return circle

Endfunction

return **Hierholzer**(any vertex)

**关键代码描述**

1. **先判断图是否为连通图。**

我采用了DFS（即源码中的DFS1）。若无向图中若存在欧拉回路，则必然是连通的；若有向图中若存在欧拉回路，则必然是强连通的。因此，二者均可采用同种遍历方法：从一点出发向下遍历，标记经过的点（遇到标记过的点，则直接跳过），最后检查是否所有的点均被标记。区别在于，无向图选取任意一点作为起始点，结果相同；但有向图，从某一点开始未能标记全部点，只能说明该图不是强连通图（不能证明其不连通），但可以确定其不存在欧拉回路。

void **DFS1**(int x, int \*p) // 用递归搜寻各点，检验是否连通

{

    p[x] = 1;

    for (int i = 1; i <= n; i++)

    {

        if (p[i] == 1)

        { // 点已被记录过，跳过

            continue;

        }

        if (Matrix[x][i] > 0)

        {

**DFS1**(i, p); // 进入与该点连通的未记录的下一个点

        }

    }

}

bool **Judge\_Connect**() // 判断图是否连通

{

    int t = 1;

    int Point[**maxn**] = {0}; // 判断点是否连通

**DFS1**(1, Point);        // 连通，则所有的点均会被置为1

    for (int i = 1; i <= n; i++)

    { // 检查各点是否均被记录

        if (!Point[i])

        { // 存在未记录的点，说明不连通

            t = 0;

            break;

        }

    }

    if (t)

    {

        cout **<<** "\n连通性：该图连通" **<<** **endl**;

        return true;

    }

    else

    {

        if (g == 1)

        {

            cout **<<** "\n连通性：该无向图不连通" **<<** **endl**;

        }

        else

        {

            cout **<<** "\n连通性：该有向图不强连通" **<<** **endl**;

        }

        return false;

    }

}

1. **判断是否存在欧拉回路。**

我们知道，若**无向图有欧拉回路，则无奇度顶点；若有向图有欧拉回路，则所有顶点的出度等于入度**。使用双层循环遍历邻接矩阵，对无向图求每行的和，并判断是否为偶数；对有向图分别求第n行和第n列的和，并判断二者是否相等。

bool **Judge\_Path**() // 判断图是否有欧拉回路

{

    int t = 1, sum = 0;

    if (g == 1)

    { // 无向图

        int d;

        for (int i = 1; i <= n; i++)

        {

            d = 0;

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

            {

                d += Matrix[i][j]; // 求度数

            }

            if (d % 2 != 0)

            { // 检验度数是否为偶数

                t = 0;

                break;

            }

            sum += d;

        }

        m = sum / 2; // 无向图的边数

    }

    else

    { // 有向图

        int d1, d2;

        for (int i = 1; i <= n; i++)

        {

            d1 = 0;

            d2 = 0;

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

            {

                d1 += Matrix[i][j]; // 求出度

            }

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

            {

                d2 += Matrix[j][i]; // 求入读

            }

            if (d1 != d2)

            { // 检验出度和入读是否相等

                t = 0;

                break;

            }

            sum += d1;

        }

        m = sum; // 有向图的边数

    }

    if (t)

    {

        cout **<<** "\n该图存在欧拉回路" **<<** **endl**;

        return true;

    }

    else

    {

        cout **<<** "\n该图不存在欧拉回路" **<<** **endl**;

        return false;

    }

}

1. **求欧拉回路**

精妙的dfs设计（即源码中的DFS2）。该算法的核心思想是逐个寻找回路，过程中删除经过的路线，并通过回溯，从一个回路末端进入到下一个回溯的入口，并把相应的点加入到路线中。通俗的说，就是不断地寻找图中的回路，并把沿途路线删掉，找到了一个回路，就相当于从图中删除了这个回路，对欧拉图而言，删掉了一个回路，剩下部分的仍然构成欧拉图，然后回溯到上一个非末端点（把末端点加入路线），继续寻找回路。直到找到了所有的回路，这些回路的路线连起来就构成了一条欧拉回路，之后，开始回溯，把该路线记录下来。

void **DFS2**(int x) // 用递归通过各边(插入回路法)

{

    for (int i = 1; i <= n; i++)

    {

        if (Matrix[x][i] > 0)

        {                   // 点x存在到点i的通路

            Matrix[x][i]--; // 消去走过的路

            if (g == 1)

            {

                Matrix[i][x]--;

            }

**DFS2**(i); // 继续向下遍历

        }

    }

    total++;

    Path[total] = x; // 回溯到无法继续走的点，则加入回路

}

void **Find\_Path**() // 从图中任找一条欧拉回路

{

**DFS2**(1);

    cout **<<** "其中一条欧拉回路为：";

    int i;

    for (i = m + 1; i > 1; i--)

    { // 倒序输出

        cout **<<** "v" **<<** Path[i] **<<** "->";

    }

    cout **<<** "v" **<<** Path[1];

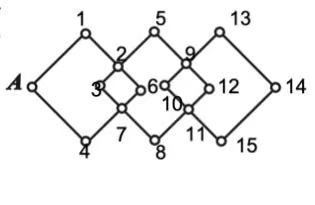
    cout **<<** **endl**;

}

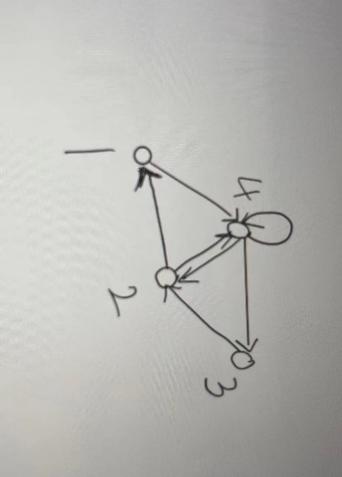
**5.5 实验设置（自定义一个/数个带权图）**

**图中（A为16号顶点）**

G1:

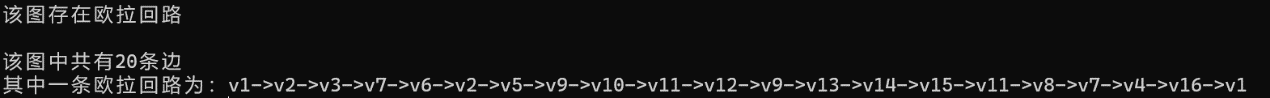


G2：



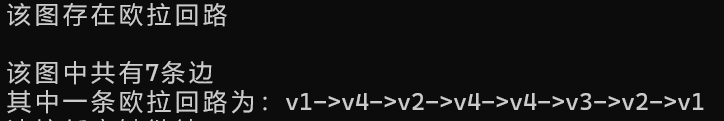
**5.6 实验结果分析**

1.G1运行结果：



如上图所示即为G1的欧拉回路。

2.G2运行结果：



**5.7 结论**

时间复杂度：矩阵的输入、遍历操作的时间复杂度都为O(n2)，检查矩阵是否连通和找出一条欧拉回路操作的时间复杂度为O(n2)。所以，程序总体时间复杂度约O(n2)。

空间复杂度：一维数组Path的空间复杂度为O(n)，二维数组Matrix的的空间复杂度为O(n^2)；另外，程序中还涉及递归，两个递归操作所占的空间复杂度都为O(n)。所以，程序总体的空间复杂度约为O(n^2)。

**源代码见附件。**

## 习题6最小哈密顿回路

**6.1问题描述**

旅行商问题，即TSP问题(Travelling Salesman Problem)又译为旅行推销员问题、货郎担问题，是数学领域中著名问题之一。假设有一个旅行商人要拜访n个城市，他必须选择所要走的路径，路径的限制是每个城市只能拜访一次，而且最后要回到原来出发的城市。路径的选择目标是要求得的路径路程为所有路径之中的最小值。将下来将给出

**6.2算法概述**

**6.2.1最邻近法**

设G=<V,E,W>为无向带权图,各边所带权均为正数.求从某顶点出发的哈密顿回路作为最短哈密顿回路的近似解的算法的大体步骤如下:  
v设作为始点.  
（1）先访问v1,形成初始路径P1=v1.  
（2）若已访问完了第k(k≤n-1)个顶点,形成了路径Pk=v1v2v3…vk，下一步访问的顶点vk+1应该是V-{v1,v2,…,vk}中离vk最近的顶点.  
（3）当访问完G中所有顶点后,形成路径Pn=v1v2…vn,得回路C=v1v2…vkv1即为G中一条哈密顿回路，它作为货郎担问题的近似解.

**6.2.2最小生成树**

（1）利用最小生成树算法构造无向图 G 的TSP问题的最小生成树；

（2）然后从最小生成树开始构造闭合回路(N个城市不重复排列序列)；

（3）最后确定从不同顶点生成的最小生成树开始构造的闭合回路中距离最小的一个 ,即最短城市序列.

**6.2.3动态规划法**

在无向完全图G中，对于任意两个顶点vi和vj，我们可以在多项式时间内找到vi和vj这两个顶点之间的所有路径，选择其中路程最短的一条，令S[i,j]表示vi和vj这两个顶点之间最短距离的那条路径。搜索路径S[i,j]，找到vi到达的在S[i,j]上的第一个顶点，记该顶点为vk，将其记录在数组中R[][]，递归查找vi到vk和vk到vj的最短路径及其相应权值，最后将数组D[]中的顶点和权值之和打印出来即为所求。

**6.2.4全排列暴力求解**

通过遍历出所有满足条件的路径情况，并保持更新最优解，直到所有情况都遍历完，得到全局最优解。但是，使用蛮力法需要遍历的城市个数高达n的阶乘，当n=12的时候，需遍历479001600种情况，程序所需时间以小时为单位。所以该算法适合于数据规模较小的情况。

**6.3 数据结构描述（习题5中若有重复可以说明情况并省略）**

**6.3.1最邻近法**

vector<vector<int>> g;  // 采用邻接矩阵存储图

vector<int> ans;        // 存储该算法下最优答案路径

vector<int> **flag**(N, 0); // 标记访问过顶点的数组

vector<int> temp;       // 临时记录路径

**6.3.2最小生成树**

vector<int> **minstdu** // 最小生成树每个顶点的度

vector<vector<int>> **minst** // 最小生成树

vector<int> **Connected** // 标记连通分量

vector<vector<int>> **graph** // 保存输入的带权无向图

vector<int> **sum**      // 保存各回路的距离值

**6.3.3动态规划法**

vector<vector<int>> G; // 图的邻接矩阵

vector<vector<int>> D; // 表示顶点到s顶点集合的距离

vector<vector<int>> R; // 表示i到集合s路径的第一个顶点是R[i][s]

unsigned int s;        // 用于表示顶点集合，第i位为1，表示第(i+1)个顶点在集合s中(i>0)

**6.3.4全排列暴力求解**

vector<int> ans;     // 最优路线

vector<int> x;       // 城市编号数组,初值赋为0

vector<int> path;    // 路线,初值赋为0

int bestw = **INT\_MAX**; // 最优的费用

vector<vector<int>> **g**//邻接矩阵存储无向权图

**6.4 伪码和关键代码描述**

**6.4.1最邻近法**

**最临近法伪代码:**

输入: 无向带权图G=(V,E),顶点w

输出: 回路长度TSPLength

1. 初始化,P={};TSPLength=0;

2. u=w;V=V-{w};

3. 循环直到集合P中包含n-1条边

   3.1 查找与顶点u邻接的最小代价边{u,v},并且v属于集合V;

   3.2 P=P+{(u,v)];V=V-{v};TSPLength=TSPLength+Cuv;

   3.3 输出经过的路径u->v,u=v,转步骤3继续求解;

4. 输出TSPLength+Cuv.

**关键代码描述：对固定起点x采用最邻近法，描述见代码注释。**

void **TSP**(int x)

{

    vector<int> **flag**(N, 0);

    vector<int> temp;

    temp.**push\_back**(x);//记录以x起点的路径

    int sum = 0;

    int u, v, min;//v表示找出当前距离城市u最小的城市

    u = x; // u是当前城市的位置 ，现在是设置的x城市

    flag**[**u**]** = 1;

    for (int i = 0; i < N - 1; i++) // 走N步回到起点

    {

        min = **INT\_MAX**;

        for (int j = 0; j < N; j++) // 比较当前城市距离其他每个城市的距离

        {

            if (flag**[**j**]** != 1 && g**[**u**][**j**]** < min && g**[**u**][**j**]** != 0) // flag查看城市是否已经走过，利用min依次比较

            {

                min = g**[**u**][**j**]**;

                v = j;

            }

        }

        sum += min;

        flag**[**v**]** = 1;

        temp.**push\_back**(v);

        u = v;

    }

    sum += g**[**v**][**x**]**;

    temp.**push\_back**(x);

    if (sum < bestx)//更新最优解

    {

        bestx = sum;

        ans **=** temp;

    }

}

**6.4.2最小生成树法**

**伪代码：**

输入: 无向带权图G=(V,E)

输出: 近似最短回路

1. 在图中对于每一个顶点v;

2. 采用Prim算法生成以顶点v为根节点的最小生成树T;

3. 对生成树T从顶点v出发进行深度优先遍历,得到遍历序列L;

4. 根据L得到图G的哈密顿回路

5. 得到代价最小的路径。

**关键代码描述：**

**(1)求最小生成树 prim算法**

由于采用的Prim算法默认从顶点进行构造，这里我们用graph数组来表示我们的带权无向图，通过交换顶点1与我们需要开始构造最小生成树的顶点v到其他各个顶点之间的距离即可从顶点v开始构造最小生成树。

void **prim**(int n) // prim算法求最小生成树

{

    vector<int> **lowcost**(n + 1);

    // lowcost存放顶点1可达点的路径长度

    vector<int> **mst**(n + 1);

    int a, min, minid;

    for (int i = 2; i <= n; i++)

    {

        lowcost**[**i**]** = graph**[**1**][**i**]**;

        mst**[**i**]** = 1; // 初始化以1位起始点

    }

    mst**[**1**]** = 0;

    for (int i = 2; i <= n; i++)

    {

        min = **MAXCOST**;

        minid = 0;

        for (int j = 2; j <= n; j++)

        {

            if (lowcost**[**j**]** < min && lowcost**[**j**]** != 0)

            {

                min = lowcost**[**j**]**; // 找出权值最短的路径长度

                minid = j;        // 找出最小的ID

            }

        }

**printf**("V%d-V%d=%d\n", mst**[**minid**]**, minid, min);

        a = mst**[**minid**]**;

        minst**[**a**][**minid**]** = min;

        minst**[**minid**][**a**]** = min;

        minstdu**[**a**]**++;

        minstdu**[**minid**]**++;

        mstcostnum++;

        lowcost**[**minid**]** = 0; // 该处最短路径置为0

        for (int j = 2; j <= n; j++)

        {

            if (graph**[**minid**][**j**]** < lowcost**[**j**]**) // 对这一点直达的顶点进行路径更新

            {

                lowcost**[**j**]** = graph**[**minid**][**j**]**;

                mst**[**j**]** = minid;

            }

        }

    }

}

（2）当生成最小生成树之后，由于闭合回路中每个节点的度都为2 ,因此在构造闭合回路时需要处理最小生成树中度不等于2的节点。处理时,第一步是通过删除边的方法降低最小生成树中度大于2的节点的度 ,保证每个节点的度都不大2。删除边时,首先选择与待处理节点(度大于2的节点)相连接的节点中度最大的节点,如果被选择节点的度大于2 ,则删除这两节点之间的边,降低这两节点的度。否则,选择与待处理节点相连接的节点中权值大的节点,删除这两节点之间的边 ,降低这两节点的度。

 for (int k = 1; k <= m; k++) // 第一步,降低最小生成树中度大于2的节点的度 ,保证每个节点的度都不大2。

        {

            if (minstdu**[**k**]** > 2)

            {

                maxdu = **mstMAXdu**(k, m);

                maxp = **mstMAXdupt**(k, maxdu, m);

                if (maxdu > 2)

                {

                    minst**[**k**][**maxp**]** = 0;

                    minst**[**maxp**][**k**]** = 0;

                    minstdu**[**k**]**--;

                    minstdu**[**maxp**]**--;

                    mstcostnum--;

                }

                else

                {

                    maxmc = **mstMAXcost**(k, m);

                    maxmstcostpoint = **mstMAXcostpt**(k, maxmc, m);

                    minst**[**k**][**maxmstcostpoint**]** = 0;

                    minst**[**maxmstcostpoint**][**k**]** = 0;

                    minstdu**[**k**]**--;

                    minstdu**[**maxmstcostpoint**]**--;

                    mstcostnum--;

                }

            }

        }

第二步是通过连接的方法,连接最小生成树中度小于2的节点,构成闭合回路连接时为了保证所有节点在同一个连通分量中,首先标记各连通分量,然后选择不同连通分量中度小于 2 的节点并且两点之间权值小的点进行连接,从而构成一个大的连通分量,最后连接同一个连通分量中仅有的两个度为1的节点,从而构成一个闭合回路。

for (; mstcostnum < m;) // 第二步是通过连接的方法 , 连接最小生成树中度小于2 的节点

        {

            for (int i = 1; i <= m; i++) // 标记连通分量

            {

                if (minstdu**[**i**]** == 0)

                {

                    Connected**[**i**]** = i;

                }

                else

                {

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

                    {

                        if (i != j && minst**[**i**][**j**]** != 0)

                        {

                            if (Connected**[**i**]** == 0)

                            {

                                Connected**[**i**]** = i;

                            }

                            Connected**[**j**]** = Connected**[**i**]**;

                        }

                    }

                }

            }

            for (int i = 1; i <= m; i++) // 连接最小生成树中度小于2 的节点

            {

                if (minstdu**[**i**]** < 2)

                {

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

                    {

                        if (i != j && Connected**[**j**]** != Connected**[**i**]** && minstdu**[**j**]** < 2)

                        {

                            minst**[**i**][**j**]** = graph**[**i**][**j**]**;

                            minst**[**j**][**i**]** = graph**[**i**][**j**]**;

                            minstdu**[**i**]**++;

                            minstdu**[**j**]**++;

                            mstcostnum++;

                            x = Connected**[**i**]**;

                            y = Connected**[**j**]**;

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

                            {

                                if (Connected**[**k**]** == y)

                                {

                                    Connected**[**k**]** = x;

                                }

                            }

                        }

                    }

                    if (minstdu**[**i**]** < 2)

                    {

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

                        {

                            if (i != j && Connected**[**j**]** == Connected**[**i**]** && minstdu**[**j**]** < 2)

                            {

                                minst**[**i**][**j**]** = graph**[**i**][**j**]**;

                                minst**[**j**][**i**]** = graph**[**i**][**j**]**;

                                minstdu**[**i**]**++;

                                minstdu**[**j**]**++;

                                mstcostnum++;

                            }

                        }

                    }

                }

            }

        }

最后通过比较各个顶点的各回路的距离值 ,从而确定最佳闭合回路(城市序列)。

**6.4.3 动态规划法**

**伪代码：**

输入: 图的代价矩阵G[n][n]

输出: 从顶点0出发经过所有顶点一次且一次再回到顶点0的最短路径长度

1. 初始化第0列:

   for(i=1; i<n; i++)

       D[i][0]=G[i][0];

2. 依次处理每一个子集数组V[2^(n-1)]

   for(i=1; i<n; i++)

       if(子集s中不包含i)

        对s中的每个元素k,计算d[i][j]=min{G[i][k]+D[k][j-1]};

3. 输出最短路径长度d[0][2^(n-1)-1].

**关键代码描述：**在程序中用该函数来记录添加到集合s中的顶点，本函数采用自底向上来填写。

void **solve**() // 采用自底向上填表

{

    int i, k;

    unsigned int max, s2;

    max = **Exp**(n - 1) - 1;

    for (s = 1; s <= max; s++)

    {

        for (i = 1; i <= n; i++)

            if ((s & 1 << (i - 2)) == 0) // 如果第i个顶点不属于集合s

            {

                D**[**i**][**s**]** = **MAXNUM**;

                for (k = 2; k <= n; k++)

                {

                    if ((s & 1 << (k - 2)) != 0)

                    {

                        s2 = s & ~(1 << (k - 2));

                        if (G**[**i**][**k**]** + D**[**k**][**s2**]** < D**[**i**][**s**]**)

                        {

                            D**[**i**][**s**]** = G**[**i**][**k**]** + D**[**k**][**s2**]**;

                            R**[**i**][**s**]** = k;

                        }

                    }

                }

            }

    }

}

**6.4.4全排列暴力求解**

**伪代码：利用dfs出所有路线的可能，并更新最小代价以及路线。**

**关键代码描述:套用dfs模板即可**

void **Tsp**(vector<vector<int>> &g, int n, int s)

{

    if (s == n)

    {

        int w = 0;

        for (int i = 0; i < n; i++)

        {

            path**[**i**]** = x**[**i**]**;

            if (i < n - 1)

                w += g**[**x**[**i**]][**x**[**i + 1**]]**;

            else

                w += g**[**x**[**i**]][**x**[**0**]]**; // 回到起点的费用

        }

        if (bestw > w)

        {

            ans **=** path;

            bestw = w;

        }

    }

    else

    {

        for (int i = s; i < n; i++)

        {

            // dfs模板

            int t = x**[**i**]**;

            x**[**i**]** = x**[**s**]**;

            x**[**s**]** = t;

**Tsp**(g, n, s + 1);

            t = x**[**i**]**;

            x**[**i**]** = x**[**s**]**;

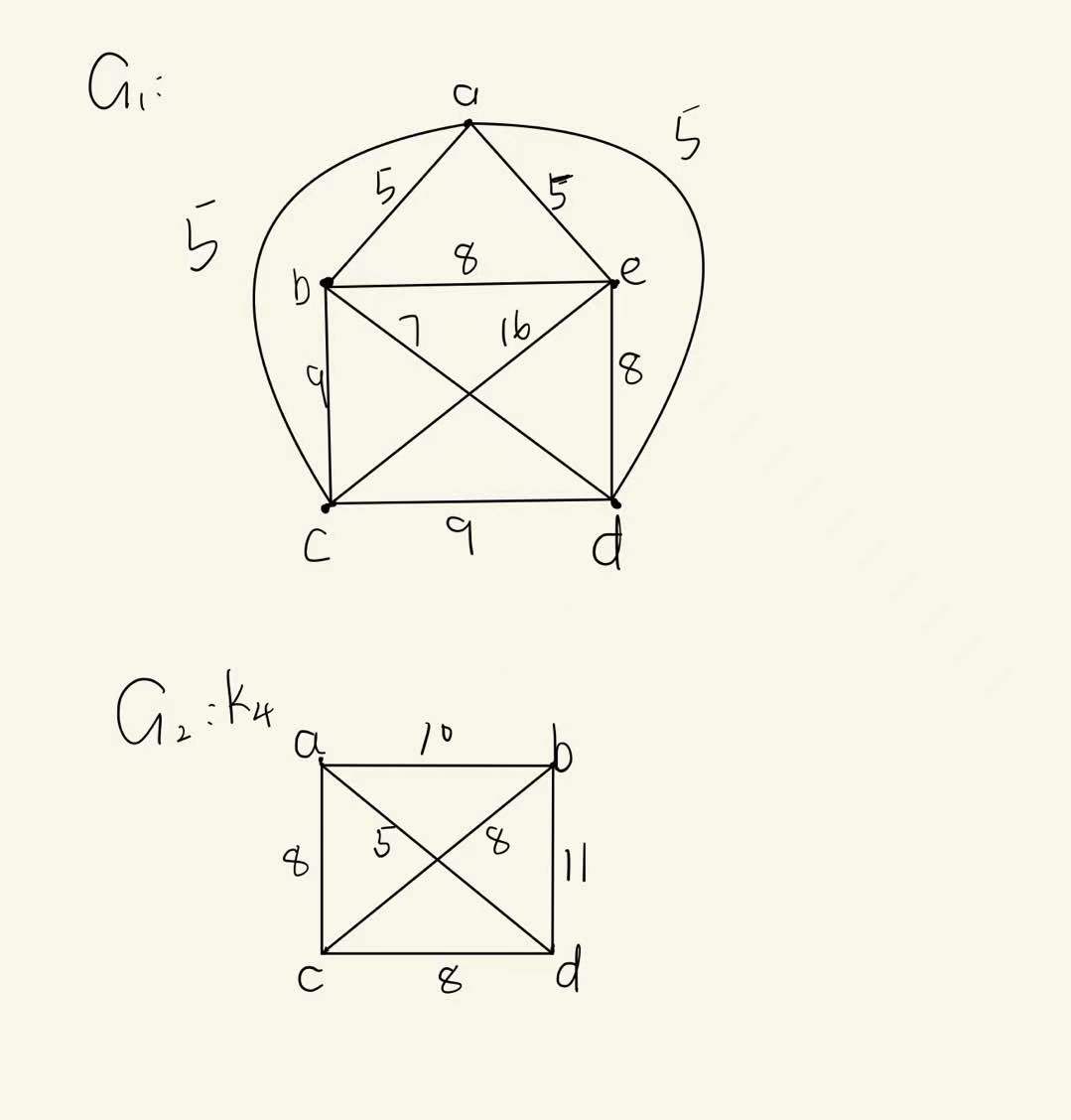
            x**[**s**]** = t;

        }

    }

}

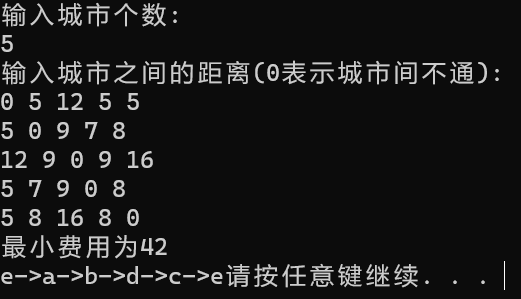
**6.5 实验设置（自定义一个/数个带权图）**

****

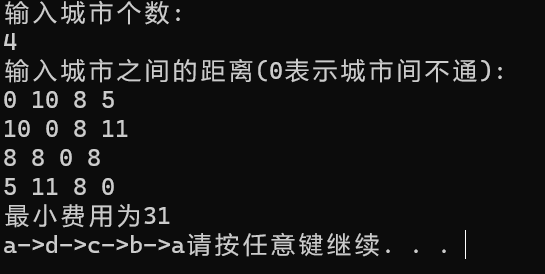
**6.6 实验结果分析**

**6.6.1最邻近法（得到近似解）**

**G1:**

****

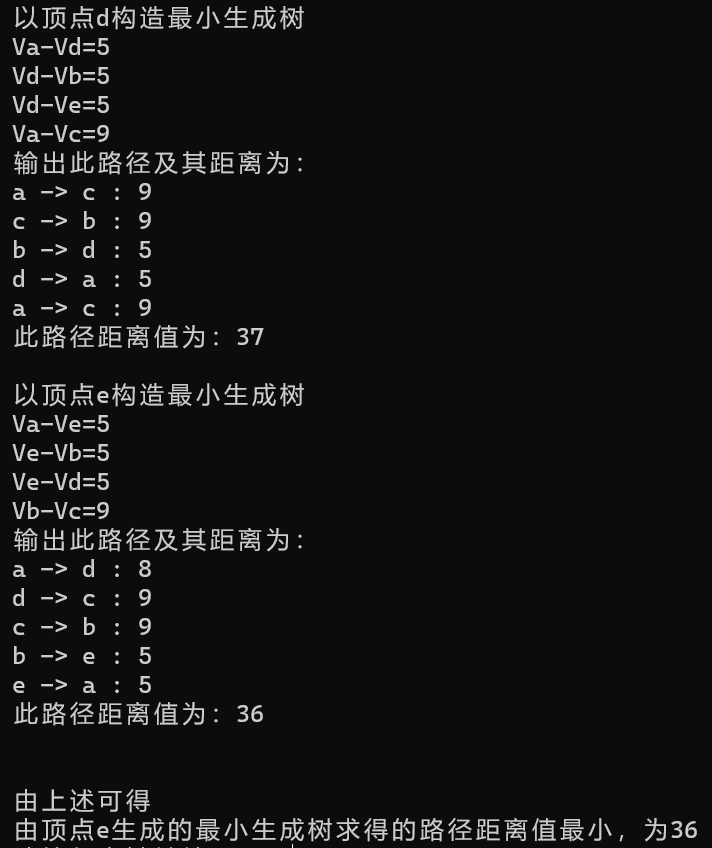
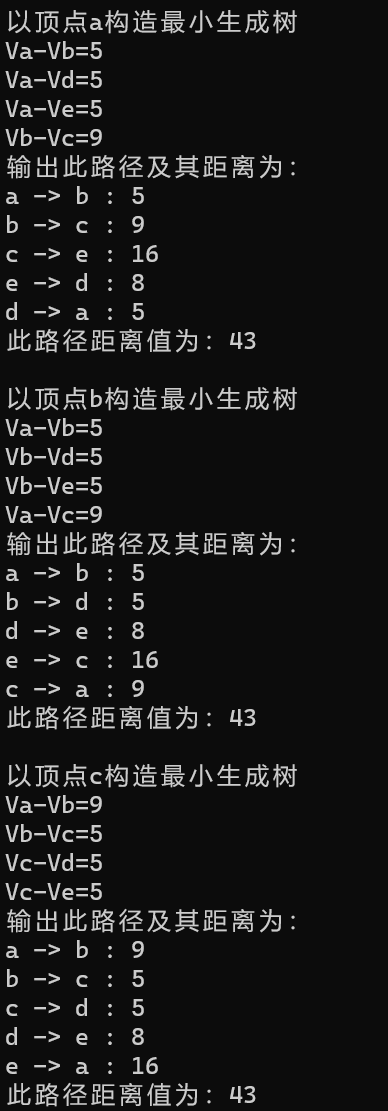
**G2:K4**

****

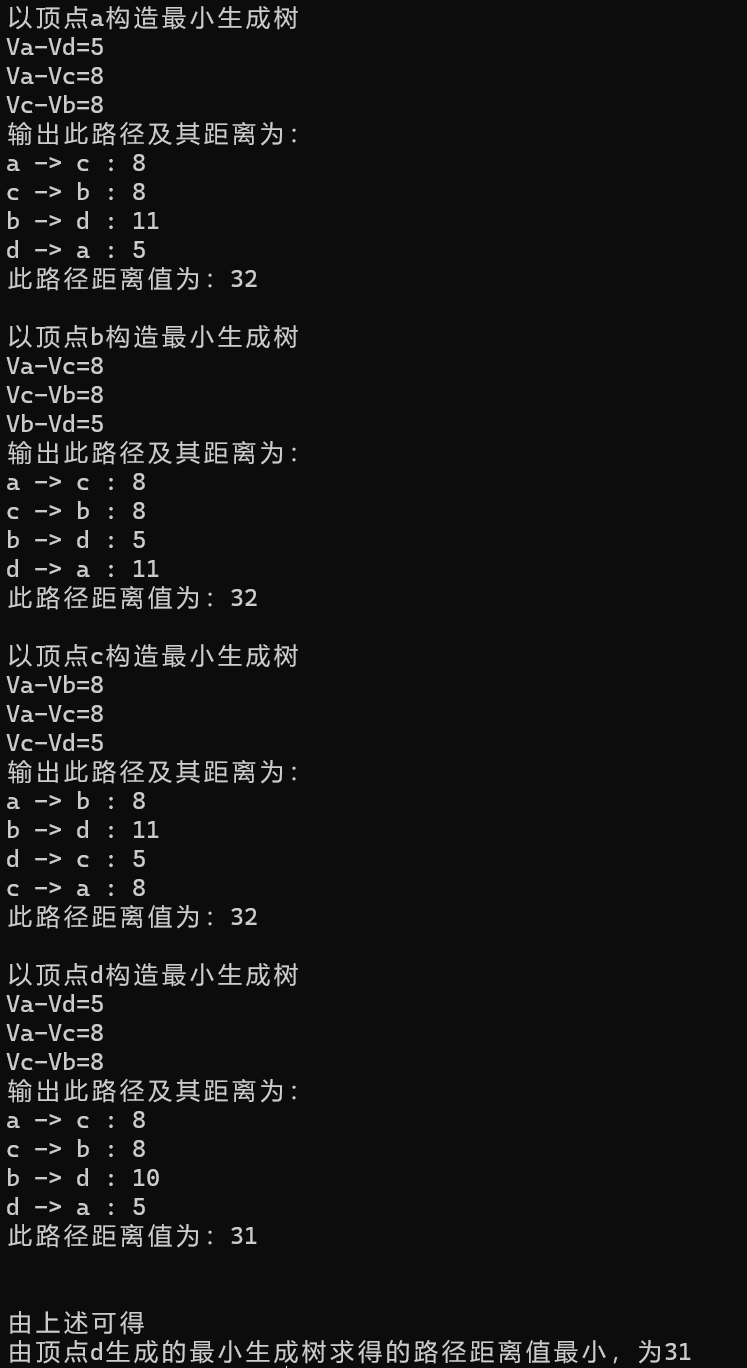
**6.6.2最小生成树法**

**展示以各顶点为起点生成的最小生成树后得到的哈密顿回路。**

**G1:**

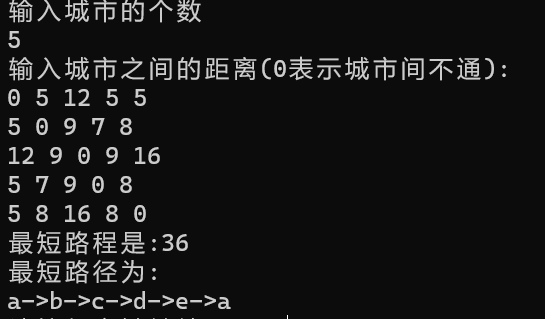
****

**G2:K4**

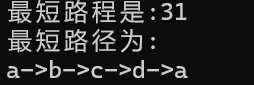
****

**6.6.3动态规划法**

**G1:**

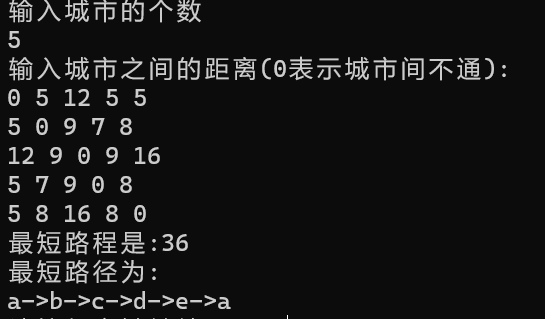
****

**G2:K4**

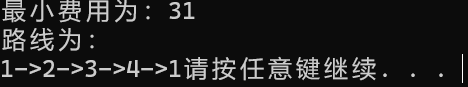
****

**6.6.4全排列暴力求解**

**G1:**

****

**G2:K4**

****

**6.7 结论**

**最邻近法时间复杂度为O（n2），最小生成树法的时间复杂度为O（n2），动态规划法的时间复杂度为O（n\*2n）,全排列暴力解法的时间复杂度为O（n！）。其中n为顶点的个数。但方法一方法二仅仅只能得到该问题的近似解，并不能得到完全精确解。动态规划法能得到精确解，但其只能固定起始点进行求解。全排列暴力法也能够得到精确值，但显然一旦数据过大，程序将永不停止地跑下去。**