《离散数学II》实验实验3

课程简介

课程名称：离散数学II

授课对象：计算机科学与技术 本科

教材版本：《离散数学教程》北京大学出版社

学时：52　其中：理论：44　实验：8

授课教师：　　　电子邮箱：

实验三 (必做, 基本实验，4学时)

实验题目：图的最大匹配与中国邮递员问题

实验目的：

1、掌握最大匹配, 交错路径的定义；

2、掌握最大匹配的求解方法；

3、掌握中国邮递员问题与七桥问题的区别与联系；

4、尝试用匹配理论和欧拉图理论给出相应的中国邮递员问题解。

中国邮递员问题: 中国邮递员问题是邮递员在某一地区的信件投递路程问题。邮递员每天从邮局出发，走遍该地区所有街道再返回邮局，问题是他应如何安排送信的路线可以使所走的总路程最短。这个问题由中国学者管梅谷在1960年首先提出，并给出了解法—“奇偶点图上

作业法”，被国际上统称为“中国邮递员问题”。即给定一个无向连通图 G，每边 e 有非负权)，要求一条回路经过每条边至少一次，且满足总权最小。

中国邮递员问题的分析：在一个具有非负权的赋权无向连通图 G 中，找出一条权最小的回路，称为最优环游。若 G 是欧拉图，则 G 的任意欧拉环游都是最优环游。若 G 不是欧拉图，则 G 的任意一个环游必定通过某些边不止一次。将边 e 的两个端点再用一条权为 w(e) 的新边连接时，称边 e 为重复的。此时 CPP 与下述问题等价。

若 G 是给定的有非赋权的赋权连通图，

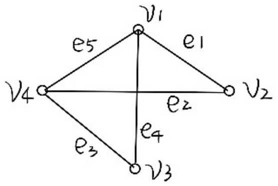
（1）用添加重复边的方法求 G 的一个欧拉赋权图 G^*，使得 G^*的权重和最小；

（2）求 G*的欧拉环游。

实验要求：

输入：无向简单连通图的关联矩阵

（例如：$A=\begin{bmatrix}1&0&0&1&1\\1&1&0&0&0\\0&0&1&1&0\\0&1&1&0&1\end{bmatrix}$）。



输出1：此图的最大匹配

例如:M={e_1,e_3}

输入2： 假设各边的权相同，均为1。将该图作为中国邮递员问题的图，输出相应的最优环游解。

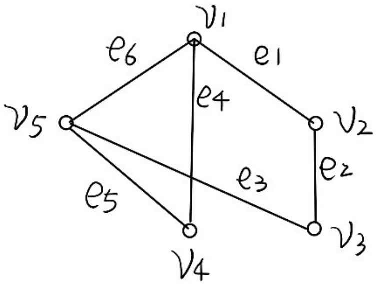
例如：中国邮递员问题的解 $e_1e_5e_3e_4e_5e_2$

*说明：

要求学生设计的程序不仅对给定相邻矩阵得出正确结果，还要对测试数据集得出正确结果。

测试案例：

$$A=\begin{bmatrix}1&0&0&1&0&1\\1&1&0&0&0&0\\0&1&1&0&0&0\\0&0&0&1&1&0\\0&0&1&0&1&1\end{bmatrix}$$



匹配数=2， M={e1,e5},或{e1,e3},或{e2,e4},或{e2,e5},或{e2,e6}

或{e3,e4},或{e4,e5},或{e2,e6}

中国邮递员问题的解 $e_1e_6e_5e_4e_6e_3e_2$

实验内容和实验步骤：（由学生填写）

程序设计的任务：

设计一个程序，能够处理无向简单连通图的关联矩阵输入。
计算并输出该图的最大匹配。
假设各边的权相同，均为1，将该图作为中国邮递员问题的图，计算并输出相应的最优环游解。
输入规定：

输入形式：无向简单连通图的关联矩阵，例如一个二维数组，其中元素 a[i][j]表示顶点 i 与顶点 j 之间是否存在边（1表示存在，0表示不存在）。
输入值的范围：矩阵的大小（即顶点的数量）和矩阵中的元素值（0或1）。
输出规定：

输出1：此图的最大匹配，例如 M={e1,e3}，表示边1和边3构成最大匹配。

输出2：中国邮递员问题的最优环游解，例如 e1e5e3e4e5e2，表示邮递员的最优路线。

程序功能：

计算无向简单连通图的最大匹配。

解决中国邮递员问题，输出最优环游解。

2. 概要设计

数据结构定义：

Graph：用于存储无向简单连通图的关联矩阵。

Match：用于存储最大匹配中的边。

Tour：用于存储中国邮递员问题的最优环游解。

主程序流程：

读取输入的无向简单连通图关联矩阵。

调用最大匹配算法计算最大匹配，并输出结果。

将图转换为欧拉图（若非欧拉图，则添加重复边），调用欧拉环游算法计算最优环游解，并输出结果。

程序模块调用关系：

主程序调用最大匹配算法模块。

主程序调用欧拉环游算法模块（在此之前可能需要调用将图转换为欧拉图的模块）。

4. 调试分析

调试过程中所遇到的问题及解决方法：

问题1：在最大匹配算法中，匈牙利算法的实现较为复杂，容易出现逻辑错误。

解决方法：仔细研究匈牙利算法的原理，逐步调试算法中的每一步，确保逻辑正确。

问题2：在将非欧拉图转换为欧拉图的过程中，如何高效地添加重复边是一个挑战。

解决方法：采用贪心策略，优先添加权重较小的重复边，同时确保图的连通性和欧拉性质。

问题3：欧拉环游算法中，深度优先搜索可能会陷入死循环。

解决方法：在搜索过程中记录已访问的边，避免重复访问，确保算法能够正确终止。

算法的时空分析：

最大匹配算法：匈牙利算法的时间复杂度为 O(V2)。在实际应用中，可以通过优化算法来降低时间复杂度。

将非欧拉图转换为欧拉图的算法：该算法的时间复杂度取决于图中奇度顶点的数量，空间复杂度为 O(E)（其中 E 为边数）。

欧拉环游算法：深度优先搜索的时间复杂度为 O(E+V)，空间复杂度为 O(E)（用于存储搜索路径）。

实验测试数据、代码及相关结果分析：（由学生填写）

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <queue>
#include <sstream>

using namespace std;

// Dijkstra 算法实现
void Dijkstra(int start, vector<int>& distance, vector<int>& precursor,
vector<vector<int>> adjacencyMatrix)
{
    // 定义无穷大的距离值
    const int INF = INT_MAX;
    int n = adjacencyMatrix.size();
    distance.resize(n, INF); // 初始化距离数组为无穷大
    distance[start] = 0; // 起点到自身的距离为 0
    precursor.resize(n, -1); // 初始化前驱节点数组，表示还未访问的节点前驱为-1

    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq; //
小顶堆存储未访问节点
    pq.push(make_pair(0, start)); // 将起点加入小顶堆

    while (!pq.empty()) {
        int dist = pq.top().first;
        int currNode = pq.top().second;
        pq.pop();

        // 如果当前节点已经被访问过，跳过
        if (dist > distance[currNode]) {
            continue;
        }

        // 遍历当前节点的相邻节点
        for (int i = 0; i < n; ++i)
        {
```

```
                if (adjacencyMatrix[currNode][i] != INF)
                { // 判断当前节点与相邻节点是否存在连接
                    int newDistance = dist + adjacencyMatrix[currNode][i];
                    if (newDistance < distance[i])
                    { // 如果新的距离更小
                        distance[i] = newDistance;
                        precursor[i] = currNode; // 更新前驱节点
                        pq.push(make_pair(newDistance, i));
                    }
                }
            }
        }
}

void dfs(int u, int n, int m, int grid[][1000], bool edgest[], bool pointst[],
vector<vector<int>>& ans, vector<int>& setsize)
{
    if (u > m)
    {
        vector<int>path;
        for (int i = 0; i < m; i++)
        {
            if (edgest[i])path.push_back(i);
        }
        setsize.push_back(path.size());
        ans.push_back(path);
        return;
    }
    int temp = 0;//临时用来计算此边关联的点有没有被搜索过
    int edge[2] = { 0 };//用来看是哪两个点
    for (int i = 0; i < n; i++)
    {
        if (grid[i][u] && !pointst[i])
        {
            temp++;
            if (temp == 1)edge[0] = i;
            else edge[1] = i;
        }
    }
    if (temp == 2)//说明这条边能用
    {
        edgest[u] = true;
        pointst[edge[0]] = true;
        pointst[edge[1]] = true;
```

```
            dfs(u + 1, n, m, grid, edgest, pointst, ans, setsize);
            edgest[u] = false;
            pointst[edge[0]] = false;
            pointst[edge[1]] = false;
            dfs(u + 1, n, m, grid, edgest, pointst, ans, setsize);
        }
        else
        {   //这边不能用也能搜，但没有上面那么多，因为不用改变状态
            dfs(u + 1, n, m, grid, edgest, pointst, ans, setsize);
        }
}

void copyGraph(int src[][100], int dest[][100], int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            dest[i][j] = src[i][j];
        }
    }
}

void SearchPath(int x, int graph[][100], int n, vector<int>& ans)
{
    for (int y = 0; y < n; ++y)
    {
        if (graph[x][y] > 0)
        {
            graph[x][y]--;
            graph[y][x]--;
            SearchPath(y, graph, n, ans);
        }
    }
    ans.push_back(x);
}



// 构造最短路径
void shortestPath(const vector<int>& precursor, int start, int end, vector<int>& path)
{
    path.clear();
    int curr = end;
    while (curr != start)
    { // 从终点回溯到起点
        path.push_back(curr);
```

```
        curr = precursor[curr];
    }
    path.push_back(start);

    reverse(path.begin(), path.end()); // 反转路径，使其从起点到终点
}



void circuit(int n, int m, int incMartix[][1000])
{
    bool flag1 = false, flag2 = false, flag3 = false;//与上面情况对应
    vector<int> path;//存放每个点度数
    for (int i = 0; i < n; i++)
    {
        int sum = 0;
        for (int j = 0; j < m; j++)
        {
            if (incMartix[i][j])
                sum++;
        }
        path.push_back(sum);
    }
    int odd = 0;//记录有多少个奇数顶点,同时存下奇数度顶点
    vector<int> oddnum;
    for (int i = 0; i < n; i++)
        if (path[i] % 2 == 1)
        {
            odd++;
            oddnum.push_back(i);
        }
    vector<int>result;//存放结果
    //若 odd 为 0 则代表其为欧拉图,所有的度数均为偶数
    //将关联矩阵转化为邻接矩阵和边矩阵
    int graph[50][100] = { 0 }, bian[50][100] = { 0 }, graph2[50][100] = { 0 };
    for (int j = 0; j < m; j++)
    {
        vector<int>arr;
        for (int i = 0; i < n; i++)
        {
            if (incMartix[i][j])arr.push_back(i);
        }
        int L = arr[0], r = arr[1];
        graph[L][r] = 1;
```

```
        graph[r][L] = 1;
        bian[L][r] = j;
        bian[r][L] = j;
    }

    if (odd != 0)
    {
        int dfsgridst[100] = { 0 };
        for (int i = 0; i < path.size(); i++)dfsgridst[path[i]] = i;
        int oddgrid[100][100] = { 0 };//存奇数顶点的对应的最短路的矩阵
        vector<vector<vector<int>>>oddpath(odd, vector<vector<int>>(odd));
        // 定义无穷大的距离值
        const int INF = INT_MAX;
        vector<vector<int>> adjacencyMatrix(n, vector<int>(n, INF));
        vector<int> ans;
        //用 adjacencyMatrix 存邻接矩阵
        for (int j = 0; j < m; j++)
        {
            vector<int>arr;
            for (int i = 0; i < n; i++)
            {
                //若不能直接到达那么距离是无穷
                if (incMartix[i][j])arr.push_back(i);
            }
            int L = arr[0], r = arr[1];
            adjacencyMatrix[L][r] = 1;
            adjacencyMatrix[r][L] = 1;
        }
        for (int i = 0; i < n; i++)adjacencyMatrix[i][i] = 0;

        for (int i = 0; i < odd; i++)
        {
            int start = oddnum[i]; // 起点
            vector<int> distance; // 保存最短距离的数组
            vector<int> precursor; // 保存前驱节点的数组

            Dijkstra(start, distance, precursor, adjacencyMatrix);
            for (int k = 0; k < distance.size(); k++)//distance 里面存起点到第 k 个点的
最短距离
            {
                for (int z = 0; z < odd; z++)
                {
                    if (oddnum[z] == k)
                    {
```

```
                    oddgrid[i][z] = distance[k];
                }
            }
        }
        for (int j = 0; j < odd; j++)
        {
            if (i == j)
            {
                oddpath[i][j].push_back(0);
                continue;
            }
            vector<int> path; // 保存最短路径的数组
            int end = oddnum[j]; // 终点
            shortestPath(precursor, start, end, path);
            for (auto x : path)//存好路径
            {
                oddpath[i][j].push_back(x);
            }
        }
    }

    vector<vector<int>>ansdfs;
    vector<int>setsize;
    bool edgest[100] = { false }, pointst[100] = { false };
    //将 oddnum 转化为关联矩阵,有 n*（n-1）/2 条边
    int dfsgrid[20][1000] = { 0 };
    int eddfs[100][100] = { 0 };
    for (int i = 0; i < odd; i++)
        for (int j = 0; j < odd; j++)eddfs[i][j] = -1;
    int idx = 0;
    for (int p = 0; p < odd; p++)
    {
        for (int z = p + 1; z < odd; z++)
        {
            if (oddgrid[p][z])//若有边就将其存入关联矩阵当中
            {
                dfsgrid[p][idx] = 1;
                dfsgrid[z][idx] = 1;
                eddfs[p][z] = idx;
                eddfs[z][p] = idx;
                idx++;
            }
        }
    }
```

```
dfs(0, odd, idx, dfsgrid, edgest, pointst, ansdfs, setsize);
int len = ansdfs.size();
int max = 0;//记录最大匹配的位置;
for (int i = 0; i < len; i++)
{
    if (setsize[i] > max)
        max = i;
}
int reslong = ansdfs[max].size();//记录找到的数组的长度
//将所有的最大匹配集输出
int mindfs = INF;
int flagdfs = 0;
for (int k = 0; k < len; k++)
{
    if (setsize[k] == reslong)//若是最大匹配集的长度就将对应的路径加到矩阵当中
    {
        int sum = 0;
        for (int i = 0; i < reslong; i++)
        {
            for (int edi = 0; edi < odd; edi++)
            {
                for (int edj = edi + 1; edj < odd; edj++)
                {
                    if (ansdfs[k][i] == eddfs[edi][edj])
                    {
                        sum += oddpath[edi][edj].size() - 1;
                    }
                }
            }
        }
        if (sum < mindfs)
        {
            mindfs = sum;
            flagdfs = k;
        }
    }
}
vector<vector<int>>finalans;
for (int i = 0; i < reslong; i++)
{
    for (int edi = 0; edi < odd; edi++)
    {
        for (int edj = edi + 1; edj < odd; edj++)
        {
```

```
                        if (ansdfs[flagdfs][i] == eddfs[edi][edj])
                        {
                            finalans.push_back(oddpath[edi][edj]);
                        }
                    }
                }
            }
            for (int i = 0; i < finalans.size(); i++)
            {
                for (int j = 0; j < finalans[i].size() - 1; j++)
                {
                    int Le = finalans[i][j], ri = finalans[i][j + 1];
                    graph[Le][ri] += 1;
                    graph[ri][Le] += 1;
                }
            }
        }
        int copiedGraph[100][100];
        copyGraph(graph, copiedGraph, n);
        vector<int> first;
        SearchPath(1, graph, n, first);
        int needlen = first.size();
        vector<int> turned;
        printf("\n第一条{");
        for (int i = 0; i < needlen - 1; i++)
        {
            printf("e%d", bian[first[i]][first[i + 1]] + 1);
        }
        printf("}");
        vector<int> second;
        SearchPath(3, copiedGraph, n, second);
        int len2 = second.size();
        printf("\n第二条{");
        for (int i = 0; i < len2 - 1; i++)
        {
            printf("e%d", bian[second[i]][second[i + 1]] + 1);
        }
        printf("}");

    }

    int main()
    {
        int temp1[100][1000] = { 0 };
```

```
    int n, m;

    vector<vector<int>>martix1;
    vector<int>temp2;
    bool edgest[100] = { false }, pointst[100] = { false };
    cout << "请分别输入顶点数和边数：" << endl;
    cin >> n >> m;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
            cin >> temp1[i][j];
    }
    dfs(0, n, m, temp1, edgest, pointst, martix1, temp2);
    int len = martix1.size();

    int max = 0;//记录最大匹配的位置;
    for (int i = 0; i < len; i++)
    {
        if (temp2[i] > temp2[max])
            max = i;
    }
    int reslong = martix1[max].size();//记录找到的数组的长度
    //将所有的最大匹配集输出
    cout << "最大匹配数= " << temp2[max] << endl<<"最大匹配："<<endl;
    for (int k = 0; k < len; k++)
    {
        if (temp2[k] == reslong)//若是最大匹配集的长度就输出
        {
            printf("{");
            for (int i = 0; i < reslong; i++)
            {
                printf("e%d,", martix1[k][i] + 1);
            }
            printf("\b}\n");
        }

    }
    circuit(n, m, temp1);
    return 0;
}
```

```
0 0 0 1 1 0 0 0 0 0 0 0
0 0 0 0 1 1 0 0 0 0 0 1
0 0 0 0 0 1 1 0 0 0 0 0
0 0 0 0 0 0 1 1 0 1 0 0
0 0 0 0 0 0 0 0 1 1 1 1
最大匹配数= 4
最大匹配:
{e1,e3,e5,e7}
{e1,e3,e5,e10}
{e1,e3,e6,e10}
{e1,e3,e7,e12}
{e1,e4,e6,e10}
{e1,e4,e7,e12}
{e1,e5,e7,e11}
{e2,e4,e6,e8}
{e2,e4,e6,e10}
{e2,e4,e7,e12}
{e2,e4,e8,e12}
{e2,e5,e7,e11}
{e2,e5,e8,e11}
{e2,e6,e8,e11}
{e3,e5,e7,e9}
{e3,e5,e8,e9}
{e3,e6,e8,e9}
{e4,e6,e8,e9}

以v2为起点{e9e10e7e6e12e11e3e2e2e3e4e5e6e7e8e1}
以v4为起点{e11e10e7e6e12e9e2e3e4e5e6e7e8e1e2e3}
C:\Users\HP\Desktop\24 VS C++\中国邮递员问题\x64\Debug\中国邮递员问题.exe (进程 34524)已退出，代码为 0
按任意键关闭此窗口. . .
```

实验总结：（由学生填写）

感悟和体会

理论与实践的紧密结合：

在解决中国邮递员问题的过程中，我深刻体会到了理论知识与实践操作之间的紧密联系。通过理解欧拉图、匹配理论等基础知识，我能够更有效地设计算法并解决实际问题。

算法设计的挑战与乐趣：

设计算法解决中国邮递员问题是一个充满挑战的过程。从最初的思路模糊到逐步明确算法步骤，再到最终成功实现并优化算法，我感受到了解决问题的乐趣和成就感。

问题解决方法的多样性：

在解决中国邮递员问题的过程中，我意识到一个问题可能有多种解决方法。例如，在将非欧拉图转换为欧拉图时，可以采用不同的策略来添加重复边。这种多样性促使我思考如何选择最优的解决方案。

编程技能的提升：

通过实现算法并调试程序，我的编程技能得到了显著提升。我学会了如何更有效地使用数据结构、如何优化算法性能以及如何处理程序中的错误。

经验和教训

深入理解问题背景：

在解决中国邮递员问题之前，我花了大量时间了解问题的背景和相关知识。这使我能够更好地理解问题的本质，并设计出更有效的算法。因此，我认识到在解决问题之前深入了解问题背景的重要性。

注重算法的优化：

在实现算法的过程中，我意识到算法的性能对于解决实际问题至关重要。因此，

我注重算法的优化，包括减少不必要的计算、优化数据结构等。这些优化措施显著提高了算法的效率。

善于借鉴他人的经验：

在解决问题的过程中，我积极借鉴了他人的经验和解决方案。通过查阅相关文献、参加讨论等方式，我获得了更多的灵感和启示。这使我能够更快地找到解决问题的思路和方法。

耐心调试和测试：

在调试程序的过程中，我遇到了许多困难和挑战。但是，我始终保持耐心和细心，逐步排查问题并修复错误。同时，我还进行了充分的测试，以确保算法的正确性和稳定性。这些经验教会了我在面对复杂问题时保持冷静和耐心的重要性。

反思与总结：

在完成实验后，我进行了深入的反思和总结。我分析了自己在解决问题过程中的优点和不足，并提出了改进的方法和建议。这种反思和总结有助于我更好地掌握所学知识，并为未来的学习和工作打下坚实的基础。

附录：实验报告的要求

实验报告是反映学生实验效果的最主要的依据，也是学生正确地表达问题、综合问题和发现问题的能力的基本培养手段，因而是非常重要的内容，本课程的实验报告中要包括以下几项内容：

（一） 实验题目；

（二） 实验目的；

（三） 实验要求；

（四） 实验内容和实验步骤；

1．需求分析：陈述程序设计的任务，强调程序要做什么，明确规定：

（1）输入的形式和输入值的范围；

（2）输出的形式；

（3）程序所能实现的功能；

2．概要设计：说明用到的数据结构定义、主程序的流程及各程序模块之间的调用关系。

3．详细设计：提交带注释的源程序或者用伪代码写出每个操作所涉及的算法。

4．调试分析：

（1）调试过程中所遇到的问题及解决方法；

（2）算法的时空分析；

（五） 实验结果：列出对于给定的输入所产生的输出结果。若可能，

测试随输入规模的增长所用算法的实际运行时间的变化。

（六） 实验总结：有关实验过程中的感悟和体会、经验和教训等。