# 作业 欧拉路径（图的一笔画问题） 实验报告

姓名：刘彦 学号：2352018 日期：2024 年 11 月 28 日

## 1. 涉及数据结构和相关背景

### 1.1 图的概念

图是一种表示对象及其相互关系的数据结构，由顶点（节点）和边（连接）组成。

- 无向图：边没有方向，表示双向关系。
- 有向图：边有方向，表示单向关系。
- 带权图：边上有权重，用于表示代价、距离等。

### 1.2 图的存储方式

图的存储方式直接影响算法的效率和资源消耗，常见方法包括：

1.2.1 邻接矩阵

用二维数组表示顶点之间的关系。

优点：访问任意两点关系简单高效 $O(1)$。

缺点：对稀疏图浪费大量空间。

空间复杂度：$O(n^2)$

1.2.2. 邻接表

每个顶点用链表存储与其相连的顶点。

优点：对稀疏图节省空间。

缺点：查找特定边的效率较低。

空间复杂度：$O(n+m)$（n 为顶点数，m 为边数）

1.2.3 逆邻接表

与邻接表类似，但存储的是每个顶点的入边（指向该顶点的边）。

常用于反向图操作，例如反向拓扑排序。

### 1.3 图的遍历

1.3.1 广度优先搜索（BFS）

按层次逐层访问图的节点，通常用队列实现。

优点：找到最短路径，适合连通性检测。

时间复杂度：$O(n+m)$

深度限制：通过限制搜索深度，避免无意义遍历。

1.3.2 深度优先搜索（DFS）

采用递归或栈，沿一条路径尽可能深地探索。

优点：适合路径、环检测等问题。

时间复杂度：$O(n+m)$

深度限制：限制递归深度以控制性能或防止爆栈。

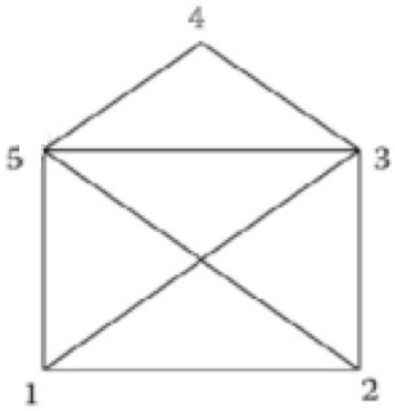
## 2. 实验内容

### 2.1 实验目的

1. 掌握图的存储结构和基本操作；
2. 灵活运用图的遍历方法。

## 2.2 问题描述

请你写一个程序，从下图所示房子的左下角（数字 1）开始，按照节点递增顺序，输出所有可以一笔画完的顶点顺序（欧拉路径），要求所有的边恰好都只画一次。例如，123153452 就是其中的一条路径。




## 2.3 参考信息

2.3.1 图的存储

可以使用邻接矩阵 map[][]存储此图，其中 map 的对角线(map[1][1]， map[2][2]，map[3][3]， map[4][4]， map[5][5])和 map[1][4]， map[4][1]， map[2][4]， map[4][2]为 0，其余元素为 1

2.3.2 深度优先搜索

从顶点 1 开始，沿着一条未走过的边找到一个新的顶点，继续走一条未走过的边；当没有未走过的边时，则退回到上一个顶点，继续试探未走过的边，直到所有的边都被访问。

2.3.3 深度优先搜索(DFS)是个递归的过程，可以参考以下代码

dfs.py

```
def dfs(x, k, s):
    """
    :param x: 所在端点
    :param k: 已经扩展的深度
    :param s: 目前的访问序列
    """

    if k >= 8:
        return s
    for y in [1, 2, 3, 4, 5]:
        if map[x][y] == 1:
            map[x][y] = 0
            map[y][x] = 0

            dfs(y, k+1, s+str(y))

            map[x][y] = 1
            map[y][x] = 1
```

### 2.4 数据结构设计

```
int map[5][5];          // 邻接矩阵
char path[20];          // 用于存储路径
int path_index = 0;     // 路径的当前长度
```

### 2.5 功能说明（函数、类）

深度优先搜索的函数

```
/**
 * @brief           深度优先搜索（DFS）函数，用于寻找欧拉路径
 * @param x         当前顶点
 * @param k         已遍历的边数
 */
void dfs(int x, int k)
{
    if (k >= 8) { // 如果已经遍历了 8 条边
        cnt++;
        path[path_index] = '\0'; // 确保路径是一个合法的字符串
        cout << path << endl;    // 输出路径
        return;
    }
    for (int y = 1; y <= 5; y++) { // 遍历所有可能的下一个顶点
        if (map[x][y] == 1) {      // 如果顶点 x 和 y 之间有边
            map[x][y] = 0;         // 标记边为已访问
            map[y][x] = 0;
            path[path_index++] = '0' + y; // 添加当前顶点到路径中
            dfs(y, k + 1); // 递归处理下一个顶点
            path[--path_index] = '\0'; // 回溯，移除路径中的顶点
            map[x][y] = 1; // 恢复边的状态
            map[y][x] = 1;
        }
    }
}
```

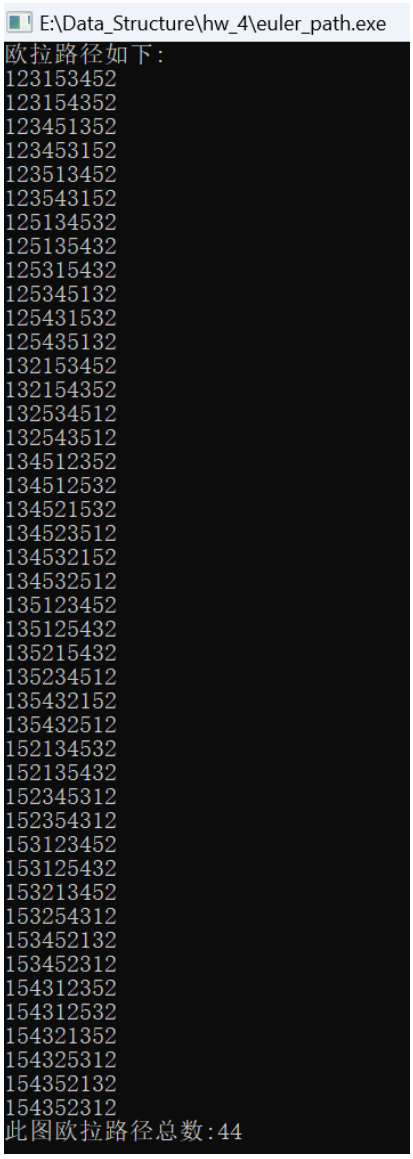
main 函数

```
int main()
{
    memset(map, 0, sizeof(map)); // 初始化邻接矩阵为 0
    for (int i = 1; i <= 5; i++)
        for (int j = i + 1; j <= 5; j++)
            if (i != j)
                map[i][j] =  map[j][i] = 1; // 非对角线元素先设置为 1
    // 两端点没有路径则置 0
    map[1][4] = map[4][1] = 0;
    map[2][4] = map[4][2] = 0;
```

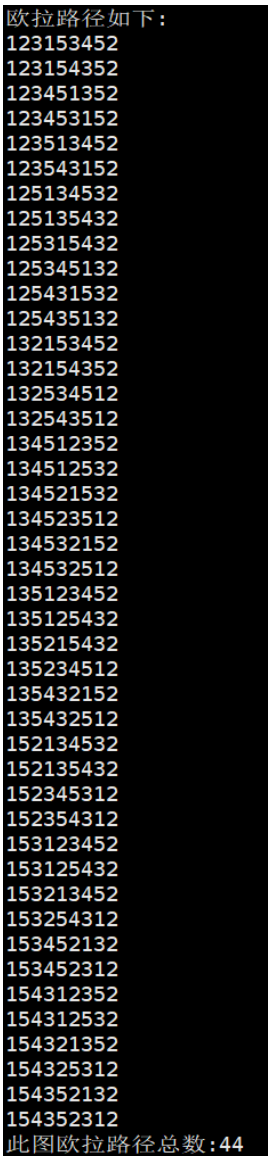
```
    path[path_index++] = '1'; // 初始路径以 1 开头
    cout << "欧拉路径如下:" << endl;
    dfs(1, 0);
    cout << "此图欧拉路径总数:" << cnt << endl;
    return 0;
}
```

## 2.6 实验结果

2.6.1 windows 系统




2.6.2 linux 系统




## 2.7 调试分析（遇到的问题和解决方法）

2.7.1 避免无向图边的重复计数

为了避免无向图边的重复计数，可以确保在邻接矩阵中每条边只统计一次。这可以通过只统计矩阵上半部分（即 i<j）的边来实现。原实现如下：

```
for (int i = 1; i <= 5; i++)
        for (int j = 1; j <= 5; j++)
```

现改为：

```
for (int i = 1; i <= 5; i++)
        for (int j = i + 1; j <= 5; j++)
            if (i != j)
                map[i][j] =  map[j][i] = 1; // 非对角线元素先设置为 1
```

2.7.2 回溯逻辑错误

在深度优先搜索（DFS）中，未正确回溯可能导致错误路径或重复计数。我未恢复邻接矩阵的边状态。路径未正确移除，导致路径输出错误。

后面进行修改，在递归结束后，恢复邻接矩阵的边状态。确保路径索引 path_index 的递增和递减正确匹配。

```
path[--path_index] = '\0'; // 回溯，移除路径中的顶点
            map[x][y] = 1; // 恢复边的状态
            map[y][x] = 1;
```

## 2.8 总结和体会

2.8.1 时间复杂度分析

代码中使用了深度优先搜索（DFS）来遍历图。分析时间复杂度需要考虑以下几个方面：

- 递归的深度：递归的深度取决于图中边的数量。对于一个具有 E 条边的图，最深的递归深度是 E。
- 每次递归的循环：每次递归函数调用都要遍历所有的顶点，因此每次递归的时间复杂度为 O(V)，其中 V 是图中的顶点数量。
- 边的遍历和回溯：对于每次递归，在遍历所有邻接点后，会进行回溯，恢复边的状态。这个过程在 O(1)时间内完成。

结合以上几点，我们可以推导出代码的时间复杂度：

最坏情况：考虑到每次递归会遍历 V 个邻接点，递归深度为 E。因此，时间复杂度为 O(E*V)。

在本例中，E=8 和 V=5，所以最坏时间复杂度是 O(8*5)=O(40)。这种复杂度对于小型图来说是可以接受的。

2.8.2 空间复杂度分析

空间复杂度主要包括：

- 递归栈空间：递归调用的深度是 O(E)，每次递归调用占用 O(1)的空间。因此，递归栈的空间复杂度是 O(E)。
- 路径存储：用于存储路径的 path 数组的空间复杂度是 O(V)，因为路径中最多可能包含所有的顶点。
- 邻接矩阵：邻接矩阵的空间复杂度是 $O(V^2)$。

总体空间复杂度是 $O(V^2+E)$，对于本例来说 O(25+8)=O(33)，这在实际运行时是可以接受的。

2.8.3 算法改进

使用 Hierholzer 算法实现：

- 遍历与标记：在栈不为空的情况下，取栈顶元素 current，遍历其邻接的顶点，查找未访问的边。如果找到，标记这条边为已访问，将新的顶点入栈，并将 hasEdge 置为 true。
- 路径记录：如果当前顶点 current 没有未访问的边，则将该顶点记录到 eulerPath 中，并出栈。

- 输出路径：由于欧拉路径是从终点回溯到起点得到的，最终输出时需要将 eulerPath 逆序输出。

```
void hierholzer(int start) {
    stack[++top] = start; // 初始顶点入栈
    while (top >= 0) {    // 栈非空时循环
        int current = stack[top];
        bool hasEdge = false;
        // 查找当前顶点是否有未访问的边
        for (int i = 1; i <= 5; i++) {
            if (map[current][i] == 1) { // 如果有边
                map[current][i] = 0;   // 标记边已访问
                map[i][current] = 0;   // 无向图双向删除
                stack[++top] = i;      // 新顶点入栈
                hasEdge = true;
                break;
            }
        }
        if (!hasEdge) { // 如果当前顶点没有未访问的边
            eulerPath[pathIndex++] = '0' + current; // 记录到欧拉路径
            top--;                                  // 出栈
        }
    }
}
```

时间复杂度分析：

遍历每个顶点和边：每个顶点和边最多被访问一次，因此时间复杂度为 O(V+E)，其中 V 是顶点数，E 是边数。

在本例中：V=5 和 E=8，所以时间复杂度为 O(5+8)=O(13)，这是可以接受的。

空间复杂度分析：

- 栈空间：O(V)，因为栈最多存储所有的顶点。
- 路径数组：O(V)，存储欧拉路径。
- 邻接矩阵：$O(V^2)$，用于表示图的边。

总体空间复杂度：$O(V^2)$，对于本例中的图，即 O(25)，是可行的。

经修改后，实现结果一致，时间复杂度和空间复杂度均有降低。

## 3. 实验总结

### 3.1 一笔画问题的本质

一笔画问题的本质在于确定一个图是否可以通过一条路径或回路遍历其所有边且每条边仅经过一次。这与欧拉路径和欧拉回路的定义密切相关。

### 3.2 欧拉路径和欧拉回路

欧拉路径(Euler Path):一个路径包含图中的每条边且每条边只出现一次。特殊情况：如果欧拉路径是一个回路（起点与终点相同），则称为欧拉回路

### 3.3 欧拉路径和回路的判定条件

3.3.1 无向图

存在欧拉路径的条件：图中度数为奇数的顶点数为 0 个或 2 个。

- 0 个奇数度顶点：说明图是欧拉图，可构成欧拉回路。
- 2 个奇数度顶点：起点和终点分别是两个奇数度顶点，构成欧拉路径。

存在欧拉回路的条件：图中所有顶点的度数均为偶数。

3.3.2 有向图

存在欧拉路径的条件：除了两个特殊顶点外，其余所有顶点的 出度等于入度。

特殊顶点：一个顶点的出度比入度多 1（起点），另一个顶点的入度比出度多 1（终点）。

存在欧拉回路的条件：图中每个顶点的出度等于入度。

## 3.4 DFS 主要逻辑

3.4.1 邻接矩阵表示图

- map[6][6]用于存储图的边关系。
- 双向边的表示：map[i][j]=map[j][i]=1。

3.4.2 DFS 实现欧拉路径搜索

递归核心：

- 遍历所有与当前顶点相连的未访问边。
- 标记边为已访问后继续递归。

回溯处理：恢复已访问的边以支持其他路径的探索。

3.4.3 路径记录与计数

- 使用 path 数组存储当前路径。
- 当遍历边数达到图的总边数时，记录路径并计数。

## 3.5 Hierholzer 算法主要逻辑

3.5.1 Hierholzer 算法的核心思想

a) 从一个顶点出发，沿着未访问的边遍历，直到回到起点或没有未访问的边。

b) 如果遍历结束时图中仍有未遍历的边，选择一个包含这些边的顶点，重复步骤 1。

c) 合并两部分路径，形成一条完整的欧拉路径或回路。

步骤：

- 从任意度数为奇数的顶点或任意顶点开始。
- 沿着一条未访问的边移动到下一个顶点，直到无法再移动。
- 如果存在未访问的边，选择一个当前路径中包含的顶点作为起点，继续遍历。
- 将新的路径插入到原路径的适当位置，确保路径仍然是欧拉路径或回路。
- 重复此过程，直到所有边都被访问。

3.5.2 实验过程

图的构造：使用邻接矩阵表示无向图，初始时构造一个完全图，并通过移除特定的边调整图的结构。

算法实现：

- 利用 Hierholzer 算法，从某一顶点出发，通过深度优先遍历逐步构造欧拉路径。
- 通过栈实现路径记录和回溯，最后输出完整路径。

运行结果：成功输出了一条欧拉路径，验证了 Hierholzer 算法在满足条件的图中可以正确运行。输出的路径是从终点回溯得到的，因此需要逆序打印。

## 3.6 实验收获

通过本次实验，我熟悉了 Hierholzer 算法的实现和原理，理解了利用 DFS 遍历构造欧拉路径的核心思想。

并深刻认识到图论中图结构（如顶点度数）对算法执行结果的重要性。掌握了通过邻接矩阵和栈实现图算法的方法，并体会到逆序输出处理的技巧。