实验报告电子版

**数据结构实训**

**—— 图的遍历与存储**

姓 名： 江一诺

学 号： 2220220821

指导老师： 韩凤

实验日期： 2024年4月16日

**大连海事大学电子信息工程专业**

**Academic Honesty Violations in Practice of Data Structures (PDS)**

In **PDS**, the MINIMUM penalty recommended for a violation of the Academic Honesty Policy will be a ZERO ON THE ASSIGNMENT, PROJECT or EXAM and a LOWERING OF YOUR FINAL GRADE for below what is otherwise earned. You may NOT withdraw from the course if found guilty. Some examples of academic misconduct in **PDS** include but are not limited to the following actions:

1. Picking up and using or discarding another student's written or computer output;

2. Using the computer account of another student;

3. Representing as one's own the work of another on assignments, quizzes, and projects;

4. Giving another student a copy of one's work on an assignment before the due date.

5. Copying work from online resources (Baidu,Chegg, google forums, etc.)

6. Posting work to online resources where other students can view your work.

All submissions will be checked for similarity. This code will check each submission for similarity between other student submissions, past student submissions, the solution manual, and online resources and postings. If your submission is flagged for a high level of similarity, it will be turned in for an academic honesty violation if deemed appropriate.

NOTE: Changing variable names, adding comments, or spacing will l result in a violation.

1. **实验目的**

（1）深入理解图的基本概念和程序设计方法。

（2）深入理解图的建立方法和深度优先遍历、图的广度优先遍历方法和实现方法。

（3）提高实际动手进行程序设计的能力。

1. **实验内容与要求**

**实验内容**：

1. 定义邻接矩阵存储结构或邻接表存储结构。
2. 按照建立一个带权有向图的操作需要，编写在邻接矩阵或邻接表存储结构下，带权有向图基本操作的实现函数（如初始化图、在图中插入一个结点、在图中插入一条边、在图中寻找序号为v的结点的第一个邻接结点、在图中寻找序号为v1结点的邻接结点v2的下一个邻接结点、图的深度优先遍历、图的广度优先遍历等）。
3. 设计一个测试主函数，首先创建一个图（有n个结点和e条边），然后打印图的n个结点信息和e条边信息，最后分别打印出图的深度优先遍历和广度优先遍历的结点信息序列。

**实验要求：**

1. 遵守工程伦理和学术规范；
2. 解决的问题要来源于实际工程：要对日常生活中的一个具体应用进行抽象建模与求解，鼓励对国家重大工程中的一个具体点进行抽象建模与求解并融入工程管理理念。
3. 深入理解图的基本概念、建立方法和深度优先遍历、图的广度优先遍历方法，要对各种方案进行分析比较。
4. 要对实验结果进行和解释，并通过信息综合得到合理有效地结论。算法描述与流程
5. **算法描述与流程**

## **算法描述：**

### **(1) 基本数据结构**

**图结构 (Graph)**：

* 顶点数组 (VertexNode[])：存储图中的所有顶点
* 顶点数量和边数量：numVertices, numEdges

**顶点结构 (VertexNode)**：

* 整型数据值：data
* 指向第一个邻接点的指针：first

**弧结构 (ArcNode)**：

* 邻接点序号：adjvex
* 权重值：weight
* 下一个邻接点指针：next

### **(2) 初始化图 (initGraph)**

**算法**：initGraph(G)

**输入**：图指针G

**输出**：无

**步骤**：

1. 设置顶点数和边数为0
2. 遍历顶点数组，将每个顶点的data设为-1
3. 将每个顶点的first指针设为NULL

### **(3) 查找顶点 (findVertex)**

**算法**：findVertex(G, data

)**输入**：图指针G，顶点值data

**输出**：顶点在数组中的索引，未找到返回-1

**步骤**：

1. 遍历顶点数组，查找data值
2. 找到则返回索引，未找到返回-1

### **(4) 插入顶点 (insertVertex)**

**算法**：insertVertex(G, data)

**输入**：图指针G，顶点值data

**输出**：无

**步骤**：

1. 检查图是否已满，已满则返回
2. 检查顶点是否已存在，存在则返回
3. 在顶点数组末尾添加新顶点
4. 初始化新顶点的first指针为NULL
5. 顶点数量加1

### **(5) 插入边 (insertEdge)**

**算法**：insertEdge(G, from, to, weight)

**输入**：图指针G，起点from，终点to，权重weight

**输出**：无

**步骤**：

1. 找到起点和终点的索引
2. 如果索引无效，则返回
3. 创建新的邻接点节点
4. 设置邻接点的adjvex为终点索引，weight为权重
5. 将新节点插入到起点的邻接表头部
6. 边数量加1

### **(6) 获取第一个邻接点 (firstNeighbor)**

**算法**：firstNeighbor(G, v)

**输入**：图指针G，顶点索引v

**输出**：第一个邻接点的索引，无邻接点返回-1

**步骤**：

1. 检查顶点索引是否有效
2. 如果顶点有邻接点，返回第一个邻接点的索引
3. 否则返回-1

### **(7) 获取下一个邻接点 (nextNeighbor)**

**算法**：nextNeighbor(G, v, currentNeighbor)

**输入**：图指针G，顶点索引v，当前邻接点currentNeighbor

**输出**：下一个邻接点的索引，无下一个邻接点返回-1

**步骤**：

1. 检查顶点索引是否有效
2. 从顶点的邻接表中找到当前邻接点
3. 如果找到且其后有邻接点，返回下一个邻接点的索引
4. 否则返回-1

### **(8) 深度优先搜索 (DFS)**

**算法**：DFS(G, v)

**输入**：图指针G，起始顶点索引v

**输出**：无（打印顶点值）

**步骤**：

1. 检查顶点索引是否有效
2. 输出当前顶点的值
3. 标记当前顶点为已访问
4. 遍历当前顶点的所有邻接点：
5. 如果邻接点未被访问，递归调用DFS

### **(9) 广度优先搜索 (BFS)**

**算法**：BFS(G, start)

**输入**：图指针G，起始顶点索引start

**输出**：无（打印顶点值）

**步骤**：

1. 检查顶点索引是否有效
2. 初始化队列和访问标记数组
3. 将起始顶点入队并标记为已访问
4. 输出起始顶点的值
5. 当队列不为空时循环：

* 出队一个顶点v
* 遍历v的所有邻接点：

1. 如果邻接点未被访问，输出其值
2. 标记邻接点为已访问
3. 将邻接点入队

### **(10) 从文件加载图 (loadGraphFromFile)**

**算法**：loadGraphFromFile(G, filename)

**输入**：图指针G，文件名filename

**输出**：无

**步骤**：

1. 打开文件，如果失败返回
2. 循环读取文件每一行：
3. 解析每行为起点、终点和权重三个值
4. 如果解析失败，继续下一行
5. 插入起点和终点（如果不存在）
6. 插入从起点到终点的边
7. 关闭文件

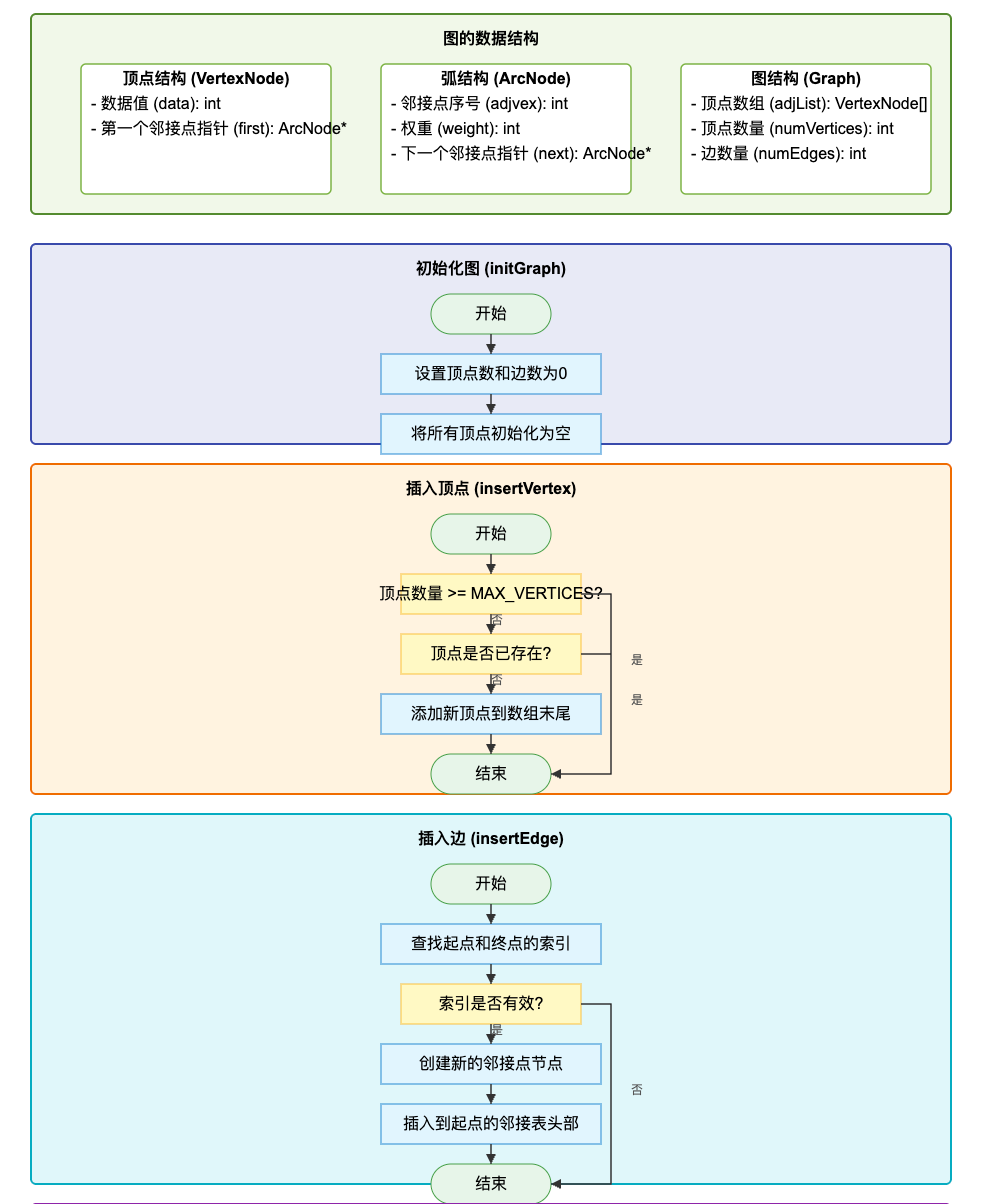
### **(11) 主函数**

**步骤**：

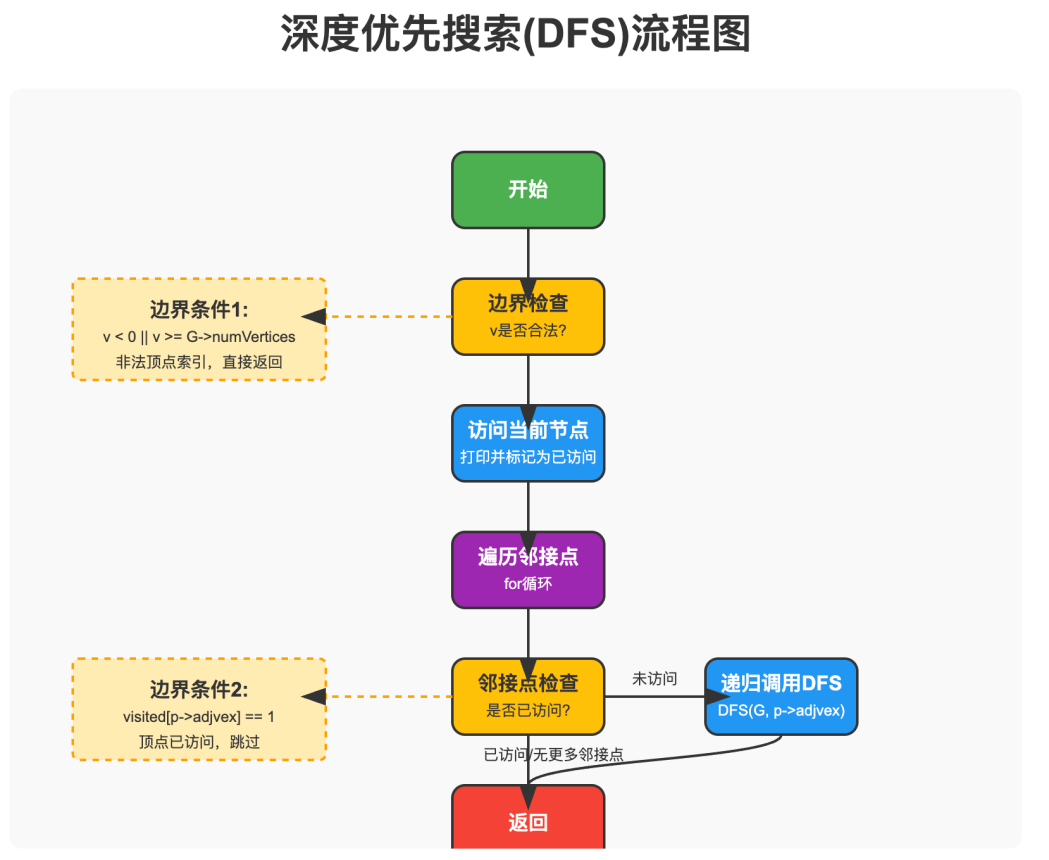
1. 创建并初始化图结构
2. 从data.txt文件加载图数据
3. 打印图的邻接表表示
4. 打印图的邻接矩阵表示
5. 从第5个顶点开始进行深度优先搜索并打印结果
6. 从第5个顶点开始进行广度优先搜索并打印结果
7. 插入两个新顶点(10和11)和一条边(10->11，权重1)
8. 打印修改后的图
9. 再次进行深度优先搜索和广度优先搜索
10. 程序结束

## **流程图**

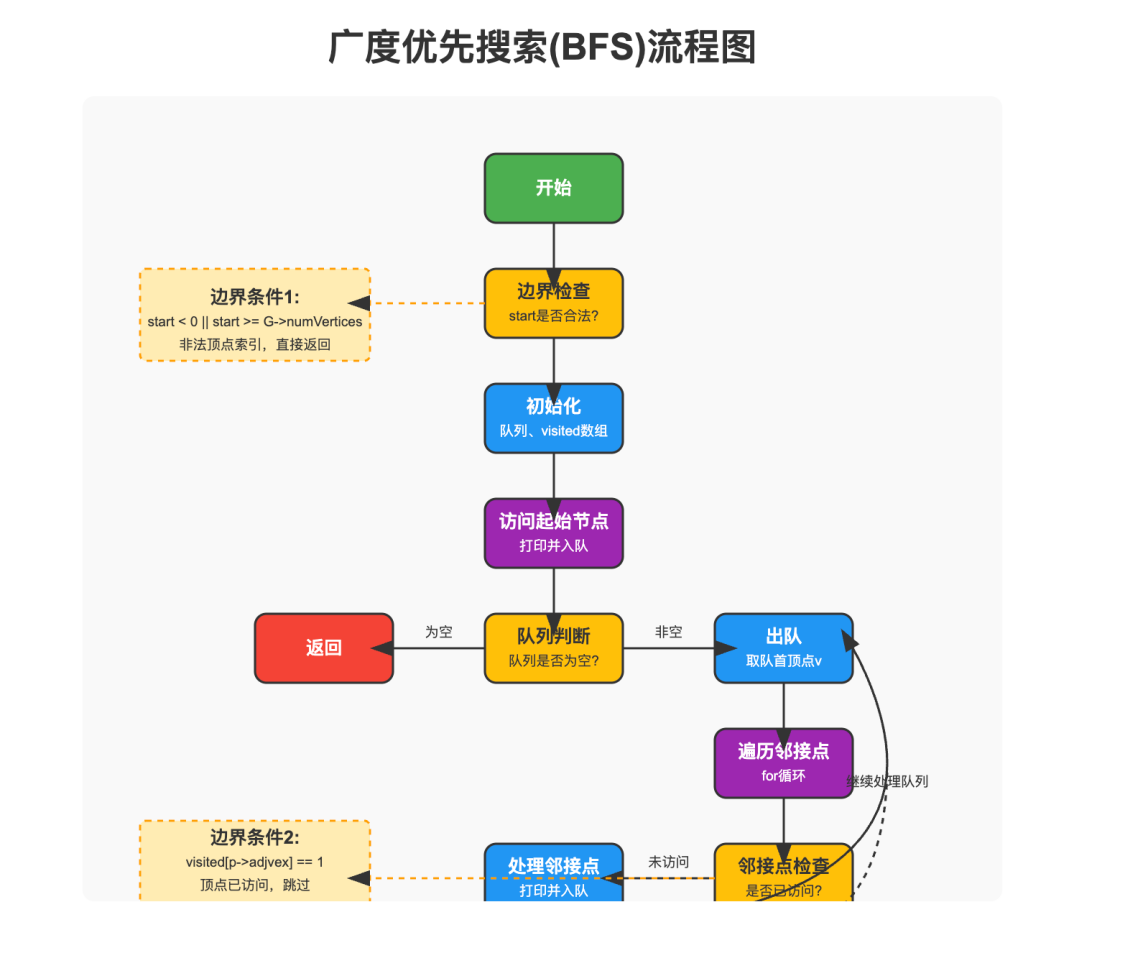
数据结构：



DFS：



BFS：



1. **算法实现**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_VERTICES 100

typedef struct ArcNode {

    int adjvex; *// 邻接点序号*

    int weight; *// 权重*

    struct ArcNode \*next;

} ArcNode;

typedef struct VertexNode {

    int data; *// 顶点值*

    ArcNode \*first;

} VertexNode;

typedef struct {

    VertexNode adjList[MAX\_VERTICES];

    int numVertices, numEdges;

} Graph;

*// 初始化图*

void initGraph(Graph \*G) {

    G->numVertices = 0;

    G->numEdges = 0;

    for (int i = 0; i < MAX\_VERTICES; i++) {

        G->adjList[i].data = -1;

        G->adjList[i].first = NULL;

    }

}

*// 查找顶点在图中的位置（返回索引）*

int findVertex(Graph \*G, int data) {

    for (int i = 0; i < G->numVertices; i++) {

        if (G->adjList[i].data == data)

            return i;

    }

    return -1; *// 边界：未找到对应顶点，返回 -1*

}

*// 插入一个顶点（如果不存在）*

void insertVertex(Graph \*G, int data) {

    if (G->numVertices >= MAX\_VERTICES) return; *// 边界：图已满*

    if (findVertex(G, data) == -1) {

        G->adjList[G->numVertices].data = data;

        G->adjList[G->numVertices].first = NULL;

        G->numVertices++;

    }

}

*// 插入一条边（有向 + 权重）*

void insertEdge(Graph \*G, int from, int to, int weight) {

    int fromIdx = findVertex(G, from);

    int toIdx = findVertex(G, to);

    if (fromIdx == -1 || toIdx == -1) return; *// 边界：from 或 to 顶点不存在*

    ArcNode \*newArc = (ArcNode \*)malloc(sizeof(ArcNode));

    newArc->adjvex = toIdx;

    newArc->weight = weight;

    newArc->next = G->adjList[fromIdx].first;

    G->adjList[fromIdx].first = newArc;

    G->numEdges++;

}

*// 寻找结点 v 的第一个邻接结点*

int firstNeighbor(Graph \*G, int v) {

    if (v < 0 || v >= G->numVertices) return -1; *// 边界：非法索引*

    if (G->adjList[v].first != NULL)

        return G->adjList[v].first->adjvex;

    return -1; *// 边界：无邻接点*

}

*// 寻找序号为v1结点的邻接结点v2的下一个邻接结点*

int nextNeighbor(Graph \*G, int v, int currentNeighbor) {

    if (v < 0 || v >= G->numVertices) return -1; *// 边界：非法索引*

    ArcNode \*p = G->adjList[v].first;

    while (p && p->adjvex != currentNeighbor) p = p->next;

    if (p && p->next)

        return p->next->adjvex;

    return -1; *// 边界：未找到当前邻接点或其后无邻接点*

}

*// 打印图*

void PrintGraph(Graph \*G) {

    for (int i = 0; i < G->numVertices; i++) {

        printf("%d:", G->adjList[i].data);

        ArcNode \*p = G->adjList[i].first;

        while (p != NULL) {

            printf(" -> %d(权重:%d)", G->adjList[p->adjvex].data, p->weight);

            p = p->next;

        }

        printf("\n");

    }

}

*// 打印邻接矩阵*

void PrintAdjMatrix(Graph \*G) {

    int matrix[MAX\_VERTICES][MAX\_VERTICES];

*// 初始化矩阵*

    for (int i = 0; i < G->numVertices; i++) {

        for (int j = 0; j < G->numVertices; j++) {

            matrix[i][j] = 0; *// 或者用 -1 表示无连接*

        }

    }

*// 填充矩阵*

    for (int i = 0; i < G->numVertices; i++) {

        ArcNode \*p = G->adjList[i].first;

        while (p) {

            matrix[i][p->adjvex] = p->weight;

            p = p->next;

        }

    }

*// 打印矩阵*

    printf("邻接矩阵:\n   ");

    for (int j = 0; j < G->numVertices; j++) {

        printf("%3d", G->adjList[j].data);

    }

    printf("\n");

    for (int i = 0; i < G->numVertices; i++) {

        printf("%3d", G->adjList[i].data);

        for (int j = 0; j < G->numVertices; j++) {

            printf("%3d", matrix[i][j]);

        }

        printf("\n");

    }

}

*// DFS遍历,深度优先搜索*

*//v：起始顶点的索引*

int visited[MAX\_VERTICES];

void DFS(Graph \*G, int v) {

    if (v < 0 || v >= G->numVertices) return; *// 边界：非法起始点*

    printf("%d ", G->adjList[v].data);

    visited[v] = 1;

*//从当前顶点v出发，访问所有与v相邻的顶点，但不访问已访问过的顶点*

    for (ArcNode \*p = G->adjList[v].first; p != NULL; p = p->next) {

        if (!visited[p->adjvex])

            DFS(G, p->adjvex);

    }

}

*// BFS遍历，广度优先搜索*

void BFS(Graph \*G, int start) {

    if (start < 0 || start >= G->numVertices) return; *// 边界：起始点为非法顶点*

    int queue[MAX\_VERTICES], front = 0, rear = 0;

    memset(visited, 0, sizeof(visited));

    printf("%d ", G->adjList[start].data);

    visited[start] = 1;

    queue[rear++] = start;

*// 广度优先搜索*

*//从起始顶点开始，依次访问其邻接点，然后将邻接点入队，直到队列为空。*

    while (front < rear) {

        int v = queue[front++];

*// 遍历该顶点的邻接点*

        for (ArcNode \*p = G->adjList[v].first; p != NULL; p = p->next) {

*// 未访问过的邻接点入队*

            if (!visited[p->adjvex]) {

                printf("%d ", G->adjList[p->adjvex].data);

                visited[p->adjvex] = 1;

                queue[rear++] = p->adjvex;

            }

        }

    }

}

*// 从文件读取图（顶点1,顶点2,权重）*

void loadGraphFromFile(Graph \*G, const char \*filename) {

    FILE \*fp = fopen(filename, "r");

    if (!fp) {

        perror("文件读取失败");

        return;

    }

    char line[128];

    while (fgets(line, sizeof(line), fp)) {

        int v1, v2, w;

        if (sscanf(line, "%d,%d,%d", &v1, &v2, &w) != 3) continue; *// 边界：格式不正确*

        insertVertex(G, v1);

        insertVertex(G, v2);

        insertEdge(G, v1, v2, w);

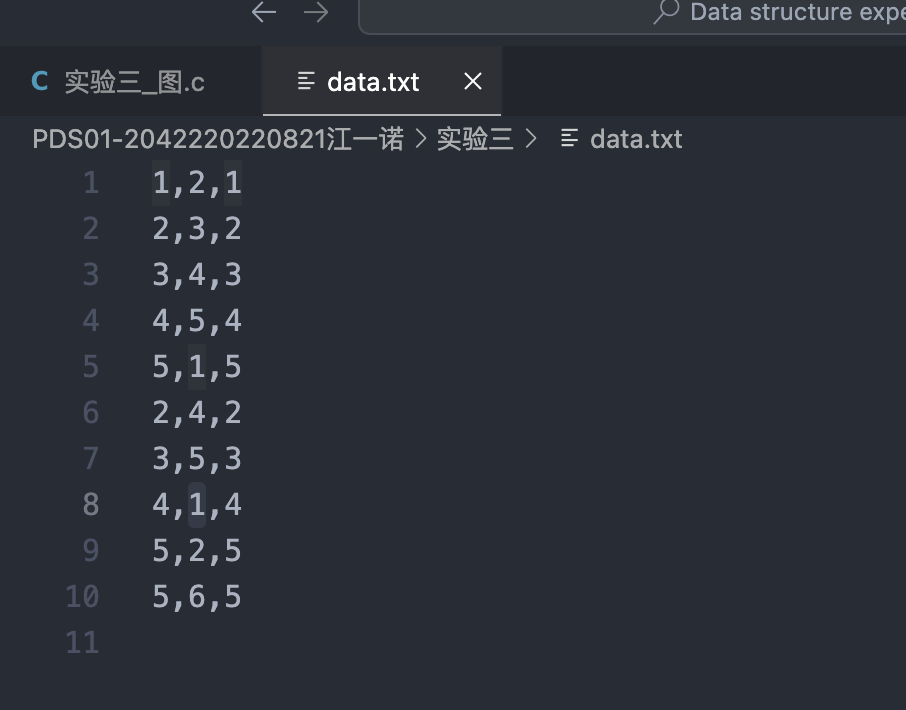
    }

    fclose(fp);

}

1. **实验结果与分析**

**5.1测试用例**



**5.2测试程序**

int main() {

    Graph G;

    initGraph(&G);

    loadGraphFromFile(&G, "data.txt");

    printf("Graph:\n");

    PrintGraph(&G);

    PrintAdjMatrix(&G);

*//DFS和BFS：*

*// 起始顶点*

    int i = 5;

    printf("DFS:\n");

    memset(visited, 0, sizeof(visited));

*//从顶点i开始深度*

    DFS(&G, i-1);

    printf("\n");

    printf("BFS:\n");

    BFS(&G, i-1); *// 同上*

    printf("\n");

*// 插入新顶点和边*

    int newV1 = 10, newV2 = 11, newW = 1;

    insertVertex(&G, newV1);

    insertVertex(&G, newV2);

    insertEdge(&G, newV1, newV2, newW);

    printf("插入后:\n");

    PrintGraph(&G);

    PrintAdjMatrix(&G);

    printf("DFS:\n");

    memset(visited, 0, sizeof(visited));

    DFS(&G, i-1);

    printf("\n");

    printf("BFS:\n");

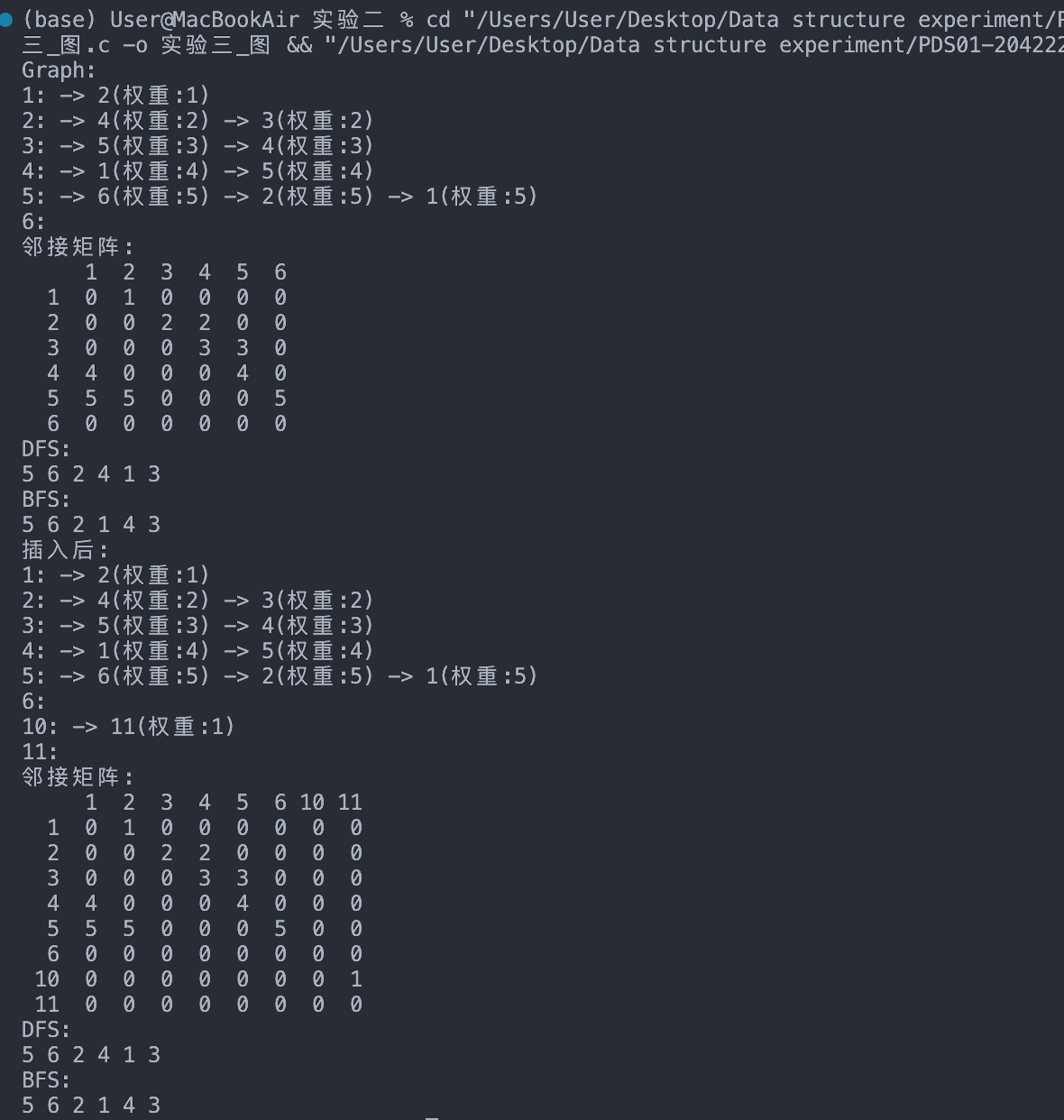
    BFS(&G, i-1);

    printf("\n");

    return 0;

}

**5.3实验结果**



**5.4实验分析**

(1) 时间和空间复杂度分析

时间复杂度

初始化图(initGraph): O(n)，需要遍历所有顶点并初始化。

查找顶点(findVertex): O(n)，最坏情况需要遍历整个顶点数组。

插入顶点(insertVertex): O(n)，需要先查找顶点是否存在。

插入边(insertEdge): O(n)，需要查找两个顶点的位置，然后插入边的操作为O(1)。

深度优先搜索(DFS): O(n + e)，n为顶点数，e为边数，需要访问所有顶点和边。

广度优先搜索(BFS): O(n + e)，同样需要遍历所有顶点和边。

打印邻接矩阵(PrintAdjMatrix): O(n²)，需要创建并填充n×n的矩阵。

空间复杂度

基本数据结构: O(n + e)，顶点数组占用O(n)空间，邻接表占用O(e)空间。

DFS递归调用: O(n)，最坏情况下需要递归n层。

BFS队列: O(n)，最坏情况下所有顶点都在队列中。

邻接矩阵: O(n²)，需要创建n×n的矩阵来表示图。

(2) 边界条件分析

顶点操作:

检查图是否已满(顶点数量达到MAX\_VERTICES)。

插入前验证顶点是否已存在，避免重复。

访问顶点前验证索引是否有效，防止越界访问。

边操作:

插入边前确保起点和终点都存在。

查找邻接点时验证顶点索引有效性。

图遍历:

DFS和BFS均使用visited数组防止重复访问顶点。

检查起始点的有效性，无效则直接返回。

文件读取:

验证文件是否成功打开，处理打开失败情况。

解析文件内容时处理格式不正确的行。

(3) 实验中出现的问题与改进方案

内存管理:

问题: 邻接表中的ArcNode节点通过malloc分配内存，但程序结束前未释放。

改进: 实现一个destroyGraph函数，遍历释放所有动态分配的ArcNode节点。

数据结构选择:

问题: 邻接表适合稀疏图，但对于稠密图效率不高。

改进: 可以根据图的特性动态选择邻接表或邻接矩阵表示。

算法效率:

问题: 查找顶点和查找路径的操作效率较低。

改进: 可以使用哈希表提高顶点查找效率，使用优先队列优化最短路径算法。

功能完善:

问题: 当前实现仅支持有向图，功能相对简单。

改进: 增加无向图支持，实现更多图算法如最短路径、拓扑排序等。

(4) 实验结论

数据结构选择:

邻接表表示法适合稀疏图，便于查找顶点的所有邻接点。

顶点和边的分离表示提高了灵活性，方便进行各类图操作。

算法实现:

DFS和BFS实现简洁明了，能够有效遍历图的所有顶点。

插入顶点和边的操作高效，适合动态变化的图结构。

局限性:

查找特定顶点的时间复杂度为O(n)，对于大规模图可能影响性能。

固定大小的顶点数组限制了图的规模扩展。

优化方向:

引入更高效的顶点索引结构，如哈希表。

实现图的压缩存储，减少内存占用。

添加更多实用算法，如最小生成树、最短路径等。

增加对大规模图处理的支持，如分块存储和并行算法。