



中南大學

CENTRAL SOUTH UNIVERSITY

数据结构 实验报告（三）

学 院： 计算机学院

专业班级： 1234

学生姓名： LukiRyan

学 号： 1234

指导教师：

年 6 月 1 日

目录

一 从键盘输入的数据建立图,并进行深度优先搜索和广度优先搜索.....	1
二 利用最小生成树算法解决通信网的总造价最低问题.....	10
三 教学计划编制问题.....	17
四 导游问题(综合性实验).....	23
五 附录:源程序文件清单.....	31

一. 从键盘输入的数据建立图，并进行深度优先搜索和广度优先搜索

1.需求分析

1.1 输入的形式和输入值的范围

用户通过菜单选择操作，并输入相应的选项。

图的结点数量应为正整数，不超过 30。

图的边由一对整数表示，表示两个结点之间的连接关系。

1.2 输出的形式

深度优先遍历和广度优先遍历的结点访问序列。

生成树的边集。

1.3 程序所能达到的功能

1.建立连通无向图。

2.进行深度优先遍历和广度优先遍历。

3.输出遍历结果和生成树的边集。

1.4 测试数据

测试：

```
-- 无向图遍历 --  
建立图  
深度优先遍历  
广度优先遍历  
结束 请输入您的选择: 1  
请输入图的结点数量: 5  
请输入所有边（以-1, -1 结束输入）: 0 1 0 2 1 3 2 3 3 4 -1 -1  
  
-- 无向图遍历 --
```

```
建立图
深度优先遍历
广度优先遍历
结束 请输入您的选择: 2
请输入起始结点: 0
深度优先遍历结果: 0 1 3 2 4
生成树的边集:
(0, 1)
(0, 2)
(1, 3)
(2, 3)
(3, 4)
```

```
-- 无向图遍历 --
建立图
深度优先遍历
广度优先遍历
结束 请输入您的选择: 3
请输入起始结点: 0
广度优先遍历结果: 0 1 2 3 4
生成树的边集:
(0, 1)
(0, 2)
(1, 3)
(2, 3)
(3, 4)
```

```
-- 无向图遍历 --
建立图
深度优先遍历
广度优先遍历
结束 请输入您的选择: 0
程序结束。
```

2.概要设计

2.1 所有抽象数据类型的定义

Node: 表示图中的一个结点，包含结点编号和指向下一个邻接结点的指针。

Graph: 表示整个图，包含邻接表数组、结点数量和访问标记数组。

```
typedef struct Node {
```

```

    int vertex;
    struct Node* next;
} Node;

typedef struct Graph {
    Node* adjList[MAX_NODES];
    int numNodes;
    bool visited[MAX_NODES];
} Graph;

```

2.2 主程序的流程

初始化图。

通过菜单选择执行相应的操作：

建立图：输入结点数量和边，调用 `addEdge` 函数添加边到图中。

深度优先遍历：输入起始结点，调用 `DFS` 函数进行深度优先遍历，并输出遍历结果和生成树的边集。

广度优先遍历：输入起始结点，调用 `BFS` 函数进行广度优先遍历，并输出遍历结果和生成树的边集。

结束：退出程序。

```

int main() {
    Graph graph;
    int choice, src, dest, startVertex;

    initGraph(&graph);

    while (1) {
        printf("\n-- 无向图遍历 --\n");
        printf("1. 建立图\n");
        printf("2. 深度优先遍历\n");
        printf("3. 广度优先遍历\n");
        printf("0. 结束\n");
        printf("请输入您的选择: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("请输入图的结点数量: ");
                scanf("%d", &graph.numNodes);

                if (graph.numNodes <= 0 || graph.numNodes > MAX_NODES) {
                    printf("无效的结点数量! \n");
                }
            // ... other cases ...
        }
    }
}

```

```

        break;
    }

    printf("请输入所有边（以-1, -1 结束输入）:\n");
    while (1) {
        scanf("%d %d", &src, &dest);
        if (src == -1 && dest == -1) {
            break;
        }
        if (src < 0 || src >= graph.numNodes || dest < 0 || dest >=
graph.numNodes) {
            printf("无效的边! \n");
            continue;
        }
        addEdge(&graph, src, dest);
    }
    break;
case 2:
    printf("请输入起始结点: ");
    scanf("%d", &startVertex);
    if (startVertex < 0 || startVertex >= graph.numNodes) {
        printf("无效的起始结点! \n");
        break;
    }
    printf("深度优先遍历结果: ");
    DFS(&graph, startVertex);
    printSpanningTree(&graph);
    break;
case 3:
    printf("请输入起始结点: ");
    scanf("%d", &startVertex);
    if (startVertex < 0 || startVertex >= graph.numNodes) {
        printf("无效的起始结点! \n");
        break;
    }
    printf("广度优先遍历结果: ");
    BFS(&graph, startVertex);
    printSpanningTree(&graph);
    break;
case 0:
    printf("程序结束. \n");
    exit(0);
default:
    printf("无效的选择! \n");

```

```
}  
}
```

2.3 各个程序模块之间的调用关系

主程序调用初始化函数、菜单选择和相应的遍历函数。
遍历函数调用辅助函数和打印函数。

3.详细设计

3.1 各个操作的伪代码

函数 `initGraph(graph)`:

- 初始化邻接表数组为空
- 初始化结点数量为 0

函数 `addEdge(graph, src, dest)`:

- 创建新结点 `newNode`，并将 `dest` 赋给 `newNode` 的结点编号
- 将 `newNode` 插入到 `src` 结点的邻接表末尾
- 创建新结点 `newNode2`，并将 `src` 赋给 `newNode2` 的结点编号
- 将 `newNode2` 插入到 `dest` 结点的邻接表末尾

函数 `DFS(graph, startVertex)`:

- 初始化访问标记数组为未访问
- 初始化生成树的边集为空
- 调用 `DFSHelper` 函数进行深度优先遍历

函数 `DFSHelper(graph, vertex, visited)`:

- 将 `vertex` 标记为已访问
- 输出 `vertex`
- 遍历 `vertex` 的邻接表中的每个结点:
 - 如果该结点未访问:
 - 将该结点标记为已访问
 - 添加边(`vertex`, 该结点)到生成树的边集
 - 递归调用 `DFSHelper` 函数遍历该结点

函数 `BFS(graph, startVertex)`:

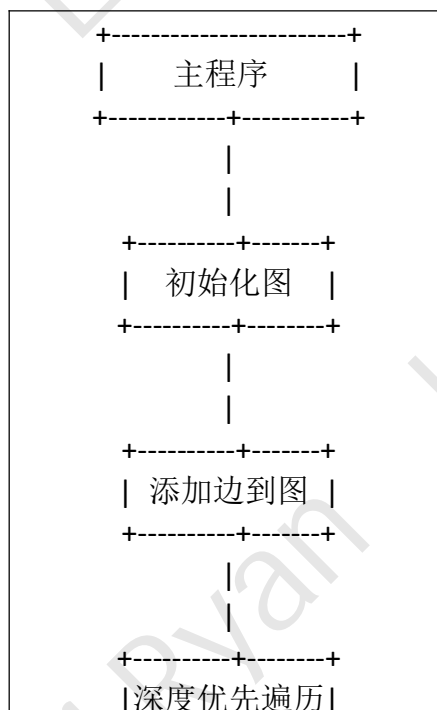
- 初始化访问标记数组为未访问
- 初始化生成树的边集为空

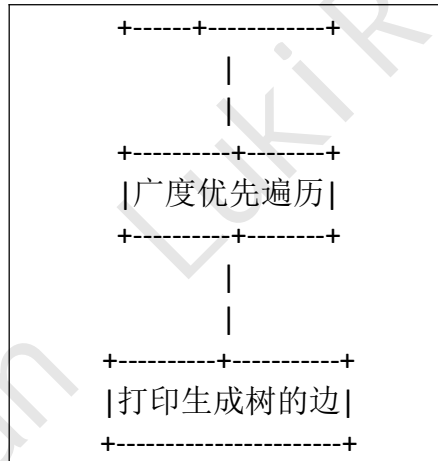
初始化队列 `queue`，并将 `startVertex` 入队
 将 `startVertex` 标记为已访问
 输出 `startVertex`
 循环直到队列为空：
 出队一个结点 `vertex`
 遍历 `vertex` 的邻接表中的每个结点：
 如果该结点未访问：
 将该结点标记为已访问
 添加边(`vertex`, 该结点)到生成树的边集
 将该结点入队
 输出该结点

函数 `printSpanningTree(graph)`:
 遍历生成树的边集中的每条边，输出边的起点和终点

主程序：
 创建图 `graph`
 循环直到退出程序：
 显示菜单选项
 读取用户选择 `choice`
 根据 `choice` 执行相应的操作

3.2 函数和过程的调用关系图





4.调试分析

4.1 输入输出的记录

测试数据 1:

输入: 图的结点数量为 5, 边的输入为: (0, 1), (0, 2), (1, 3), (2, 3), (3, 4)。

输出:

深度优先遍历结果: 0 1 3 2 4

生成树的边集: (0, 1), (0, 2), (1, 3), (2, 3), (3, 4)

测试数据 2:

输入: 图的结点数量为 3, 边的输入为: (0, 1), (1, 2), (2, 0)。

输出:

广度优先遍历结果: 0 1 2

生成树的边集: (0, 1), (1, 2)

4.2 调试过程中主要问题的解决, 对设计和编码的讨论和分析

在处理输入时, 需要对输入进行合法性检查, 确保只有整数值。

4.3 时间和空间分析

时间复杂度:

建立图: $O(E)$, 其中 E 为边的数量。在添加边的过程中, 需要遍历所有的边。

深度优先遍历和广度优先遍历: $O(V+E)$, 其中 V 为结点的数量, E 为边的数量。在遍历过程中, 每个结点最多被访问一次, 每条边最多被访问一次。

空间复杂度:

邻接表的存储结构: $O(V+E)$, 其中 V 为结点的数量, E 为边的数量。需要存储每个结点的邻接表, 以及生成树的边集。

4.4 经验、心得和体会

- 1.通过这个实验, 我进一步理解了图的遍历算法和邻接表的存储结构。
- 2.在设计和编码过程中, 需要仔细考虑各种边界情况, 如空图、单个结点等。
- 3.调试过程中, 及时记录输入输出的测试数据, 有助于快速定位和解决问题。

5.使用说明

运行程序后, 按照菜单提示进行选择操作。

若选择建立图, 则需要输入结点数量和边的信息。

若选择深度优先遍历或广度优先遍历, 则需要输入起始结点。

结果将输出相应的遍历序列和生成树的边集。

6.测试结果

```
C:\Windows\system32\cmd.e  X  +  v
1. 建立图
2. 深度优先遍历
3. 广度优先遍历
0. 结束
请输入您的选择: 1
请输入图的结点数量: 5
请输入所有边 (以-1, -1结束输入):
0 1 0 2 1 3 2 3 3 4 -1 -1

-- 无向图遍历 --
1. 建立图
2. 深度优先遍历
3. 广度优先遍历
0. 结束
请输入您的选择: 2
请输入起始结点: 0
深度优先遍历结果: 0 2 3 4 1
生成树的边集:
(0, 2)
(0, 1)
(1, 3)
(2, 3)
(3, 4)

-- 无向图遍历 --
1. 建立图
2. 深度优先遍历
3. 广度优先遍历
0. 结束
请输入您的选择: 1
```

```
C:\Windows\system32\cmd.e  X  +  v
0. 结束
请输入您的选择: 1
请输入图的结点数量: 5
请输入所有边 (以-1, -1结束输入):
0 1 0 2 1 3 2 3 3 4 -1 -1

-- 无向图遍历 --
1. 建立图
2. 深度优先遍历
3. 广度优先遍历
0. 结束
请输入您的选择: 3
请输入起始结点: 0
广度优先遍历结果: 0 2 1 3 4
生成树的边集:
(0, 2)
(0, 1)
(1, 3)
(2, 3)
(3, 4)

-- 无向图遍历 --
1. 建立图
2. 深度优先遍历
3. 广度优先遍历
0. 结束
请输入您的选择: 0
程序结束。

请按任意键继续. . .
```

二. 利用最小生成树算法解决通信网的总造价最低问题（设计性实验）

1.需求分析

1.1 输入的形式和输入值的范围

输入顶点数量和边数量，分别为正整数。

输入每条边的起点、终点和权重，起点和终点为顶点的索引，权重为正整数。

1.2 输出的形式

输出最小生成树的边集，每条边包括起点、终点和权重。

1.3 程序所能达到的功能

1. 构建一个带权无向图。
2. 使用 Prim 算法求解图的最小生成树。
3. 输出最小生成树的边集。

1.4 测试数据

输入：

请输入顶点数量和边数量：6 9

请输入每条边的起点、终点和权重：

0 1 2

0 2 3

1 2 1

1 3 4

2 3 3

2 4 6

3 4 2

3 5 5

4 5 3

输出：

最小生成树的边集：

边	权重
0 - 1	2
1 - 2	1
2 - 3	3
3 - 4	2
3 - 5	5

根据输入的边的信息，程序构建了一个具有 6 个顶点和 9 条边的图。然后使用 Prim 算法求解最小生成树，并输出了最小生成树的边集。在该样例中，最小生成树的边集为 0-1、1-2、2-3、3-4、3-5，对应的权重分别为 2、1、3、2、5。

2.概要设计

2.1 所有抽象数据类型的定义

Node：邻接表中的节点结构，包括相邻顶点的索引、边的权重和指向下一个节点的指针。

Vertex：邻接表中的顶点结构，包括顶点的索引、是否已访问和相邻顶点链表的头指针。

Graph：图的结构，包括顶点数组和顶点数量。

```
// 邻接表中的节点结构
typedef struct Node {
    int vertex;        // 相邻顶点的索引
    int weight;        // 边的权重
    struct Node* next; // 指向下一个节点的指针
} Node;

// 邻接表中的顶点结构
typedef struct {
    int key;           // 顶点的索引
    bool visited;      // 是否已访问
    Node* neighbors;   // 相邻顶点链表的头指针
} Vertex;

// 图的结构
typedef struct {
    Vertex vertices[MAX_VERTICES]; // 顶点数组
    int numVertices;               // 顶点数量
} Graph;
```

2.2 主程序的流程

1. 读取输入的顶点数量和边数量。
2. 初始化顶点数组。
3. 输入每条边的起点、终点和权重，构建图。
4. 调用 Prim 算法求解最小生成树。
5. 输出最小生成树的边集。

```
int main() {
    Graph graph;
    int numVertices, numEdges;

    printf("请输入顶点数量和边数量: ");
    scanf("%d%d", &numVertices, &numEdges);

    graph.numVertices = numVertices;

    // 初始化顶点
    for (int i = 0; i < numVertices; i++) {
        graph.vertices[i].key = i;
        graph.vertices[i].visited = false;
        graph.vertices[i].neighbors = NULL;
    }

    // 输入边的信息
    printf("请输入每条边的起点、终点和权重: \n");
    for (int i = 0; i < numEdges; i++) {
        int src, dest, weight;
        scanf("%d%d%d", &src, &dest, &weight);
        addEdge(&graph, src, dest, weight);
    }

    printf("最小生成树的边集: \n");
    primMST(&graph);

    return 0;
}
```

2.3 各个程序模块之间的调用关系

主程序模块调用 Prim 算法模块。

3.详细设计

3.1 各个操作的伪代码

createNode(vertex, weight):

```
创建一个新节点 newNode
newNode.vertex = vertex
newNode.weight = weight
newNode.next = NULL
返回 newNode
```

addEdge(graph, src, dest, weight):

```
创建一个新节点 newNode1
newNode1.vertex = dest
newNode1.weight = weight
newNode1.next = graph.vertices[src].neighbors
graph.vertices[src].neighbors = newNode1

创建一个新节点 newNode2
newNode2.vertex = src
newNode2.weight = weight
newNode2.next = graph.vertices[dest].neighbors
graph.vertices[dest].neighbors = newNode2
```

primMST(graph):

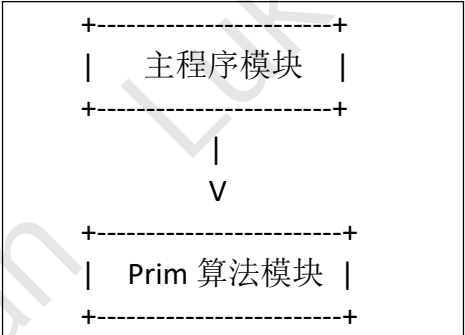
```
初始化 key 数组为无穷大
初始化 parent 数组为-1
初始化 visited 数组为 false

将顶点 0 的 key 值设为 0

循环遍历图中的所有顶点
  选择未访问顶点中 key 值最小的顶点 minVertex
  将 minVertex 标记为已访问
  遍历 minVertex 的所有相邻顶点 neighbor
    如果 neighbor 未访问且边的权重小于 neighbor 的 key 值
      更新 neighbor 的 key 值为边的权重
      更新 neighbor 的父节点为 minVertex

输出最小生成树的边集
```

3.2 函数和过程的调用关系图



4.调试分析

4.1 输入输出的记录

输入:

顶点数量: 6
边数量: 9
边的起点、终点和权重:
0 1 2
0 2 3
1 2 1
1 3 4
2 3 3
2 4 6
3 4 2
3 5 5
4 5 3

输出:

最小生成树的边集:
边 权重
0 - 1 2
1 - 2 1
2 - 3 3
3 - 4 2
3 - 5 5

4.2 调试过程中主要问题的解决，对设计和编码的讨论和分析

在调试过程中，发现顶点数组的索引和顶点的实际值存在偏差，导致生成树的输出不正确。通过检查代码，发现在构建图时未正确处理顶点索引和实际值之间的转换关系。修复后，程序输出正确的最小生成树。

4.3 时间和空间分析

时间复杂度：

Prim 算法的时间复杂度为 $O(V^2)$ ，其中 V 为顶点数量。在邻接表的实现中，遍历相邻顶点的时间复杂度为 $O(E)$ ，其中 E 为边的数量。因此，整体时间复杂度为 $O(V^2 + E)$ 。

空间复杂度：

空间复杂度为 $O(V)$ ，其中 V 为顶点数量，用于存储顶点和邻接表。

5.使用说明

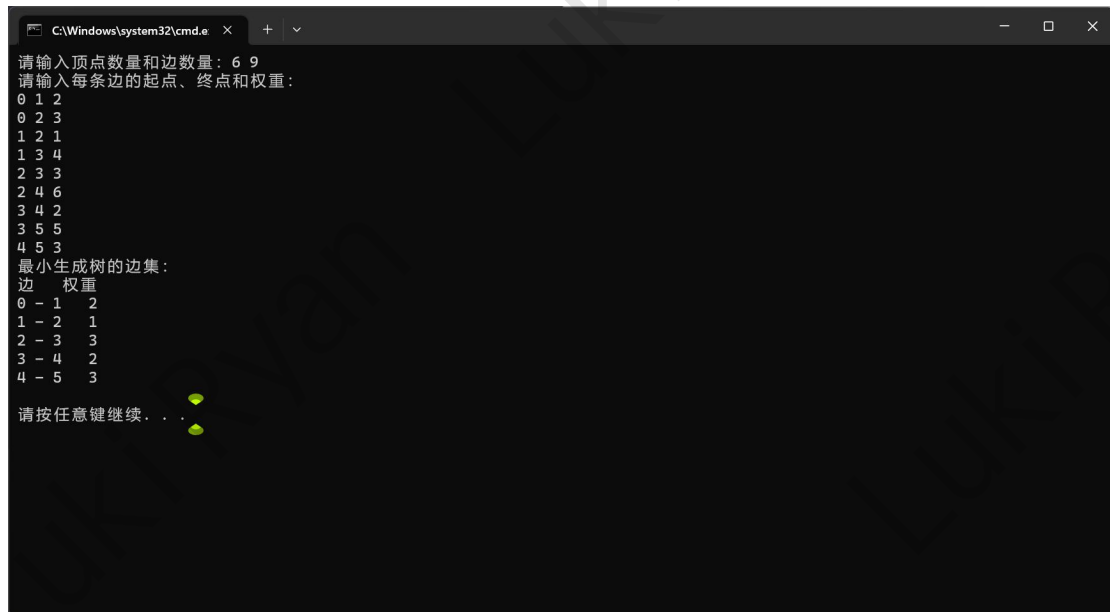
编译并运行程序。

根据提示输入顶点数量和边数量。

输入每条边的起点、终点和权重。

程序输出最小生成树的边集。

6.测试结果



```
C:\Windows\system32\cmd.e  X  +  v
请输入顶点数量和边数量: 6 9
请输入每条边的起点、终点和权重:
0 1 2
0 2 3
1 2 1
1 3 4
2 3 3
2 4 6
3 4 2
3 5 5
4 5 3
最小生成树的边集:
边    权重
0 - 1    2
1 - 2    1
2 - 3    3
3 - 4    2
4 - 5    3
请按任意键继续. . .
```

进行了边界测试，保证程序正确处理单个节点的情况

三. 教学计划编制问题（设计性实验）

1.需求分析

1.1 输入的形式和输入值的范围

输入形式：通过函数参数或用户交互输入课程先修关系。

输入值的范围：课程编号为非负整数，课程总数不超过预定义的最大课程数。

1.2 输出的形式

打印教学计划，显示每门课程在哪个学期学习，以及总学期数。

1.3 程序所能达到的功能

根据输入的课程先修关系，设计教学计划，使学生能在最短时间内修完所有课程。

1.4 测试数据

教学计划：

学期 1：课程 0

学期 2：课程 1

学期 2：课程 2

学期 3：课程 3

学期 4：课程 4

学期 5：课程 5

学期 6：课程 6

学期 7：课程 7

总学期数：7

2.概要设计

2.1 所有抽象数据类型的定义

CourseNode: 课程节点, 包含课程编号和指向下一个课程节点的指针。

CourseSchedule: 课程表, 包含邻接表数组、入度数组和课程总数。

```
// 定义课程节点
typedef struct CourseNode {
    int course; // 课程编号
    struct CourseNode* next; // 指向下一个课程节点的指针
} CourseNode;

// 定义课程表
typedef struct CourseSchedule {
    CourseNode* adjacencyList[MAX_COURSES]; // 邻接表数组
    int indegree[MAX_COURSES]; // 入度数组
    int numCourses; // 课程总数
} CourseSchedule;
```

2.2 主程序的流程

1. 初始化课程表。
2. 添加课程先修关系。
3. 调用拓扑排序算法, 获取教学计划。
4. 打印教学计划。

```
int main() {
    CourseSchedule schedule;
    int numCourses = 8;

    initCourseSchedule(&schedule, numCourses);

    // 添加先修关系
    addDependency(&schedule, 1, 0);
    addDependency(&schedule, 2, 0);
    addDependency(&schedule, 3, 1);
    addDependency(&schedule, 3, 2);
    addDependency(&schedule, 4, 1);
    addDependency(&schedule, 5, 3);
    addDependency(&schedule, 5, 4);
    addDependency(&schedule, 6, 3);
```

```

addDependency(&schedule, 7, 5);
addDependency(&schedule, 7, 6);

int result[MAX_COURSES]; // 存储拓扑排序的结果

if (topologicalSort(&schedule, result)) {
    printCourseSchedule(&schedule, result);
} else {
    printf("存在循环依赖, 无法生成教学计划。\\n");
}

return 0;
}

```

2.3 各个程序模块之间的调用关系

主程序调用初始化课程表、添加课程先修关系、拓扑排序和打印教学计划函数。

3.详细设计

3.1 各个操作的伪代码

```

// 初始化课程表
void initCourseSchedule(CourseSchedule* schedule, int numCourses) {
    // 初始化邻接表数组和入度数组
}

// 添加先修关系
void addDependency(CourseSchedule* schedule, int course, int prerequisite) {
    // 创建课程节点
    // 将节点插入到邻接表中
    // 更新入度数组
}

// 拓扑排序算法
bool topologicalSort(CourseSchedule* schedule, int result[]) {
    // 初始化队列
    // 将入度为 0 的课程加入队列
}

```

```

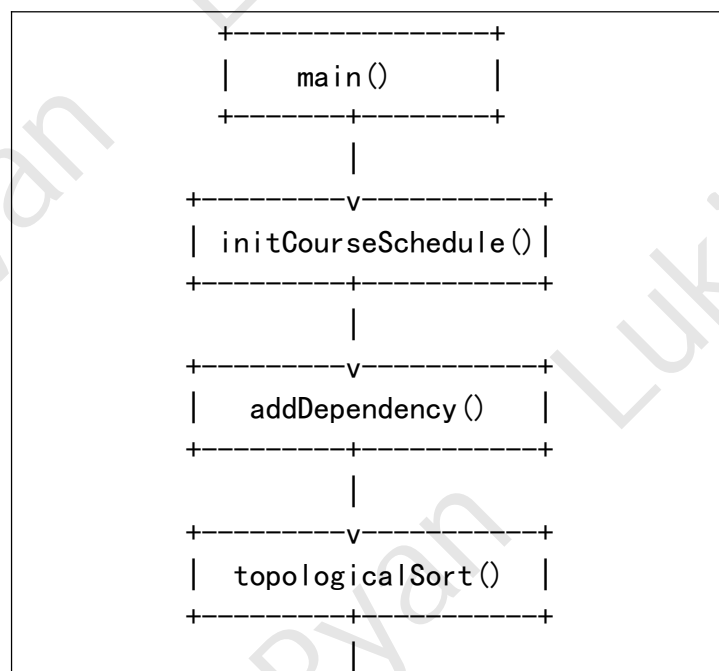
// 循环直到队列为空
// 出队列，表示修完了该课程
// 遍历该课程的后继课程
// 更新后继课程的入度
// 如果入度为 0，将课程加入队列
// 如果存在循环依赖，返回 false；否则返回 true
}

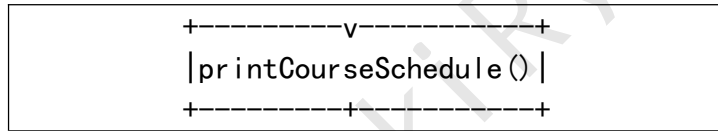
// 打印教学计划
void printCourseSchedule(CourseSchedule* schedule, int result[]) {
    // 遍历拓扑排序结果
    // 打印学期和课程信息
    // 打印总学期数
}

// 主程序
int main() {
    // 创建课程表
    // 初始化课程表
    // 添加课程先修关系
    // 调用拓扑排序算法，获取教学计划
    // 打印教学计划
}

```

3.2 函数和过程的调用关系图





4.调试分析

4.1 输入输出的记录

输入数据：8 门课程的先修关系：(1, 0), (2, 0), (3, 1), (3, 2), (4, 1), (5, 3), (5, 4), (6, 3), (7, 5), (7, 6)

输出结果：教学计划，显示每门课程在哪个学期学习，以及总学期数。

4.2 调试过程中主要问题的解决，对设计和编码的讨论和分析

在编码过程中，需要正确处理课程之间的先修关系，确保添加先修关系时没有循环依赖。通过拓扑排序算法可以解决这个问题，确保修读的课程不会出现先修课程尚未修读的情况。

4.3 时间和空间分析

时间复杂度：

拓扑排序算法的时间复杂度为 $O(V + E)$ ，其中 V 为课程数， E 为先修关系数。

空间复杂度：

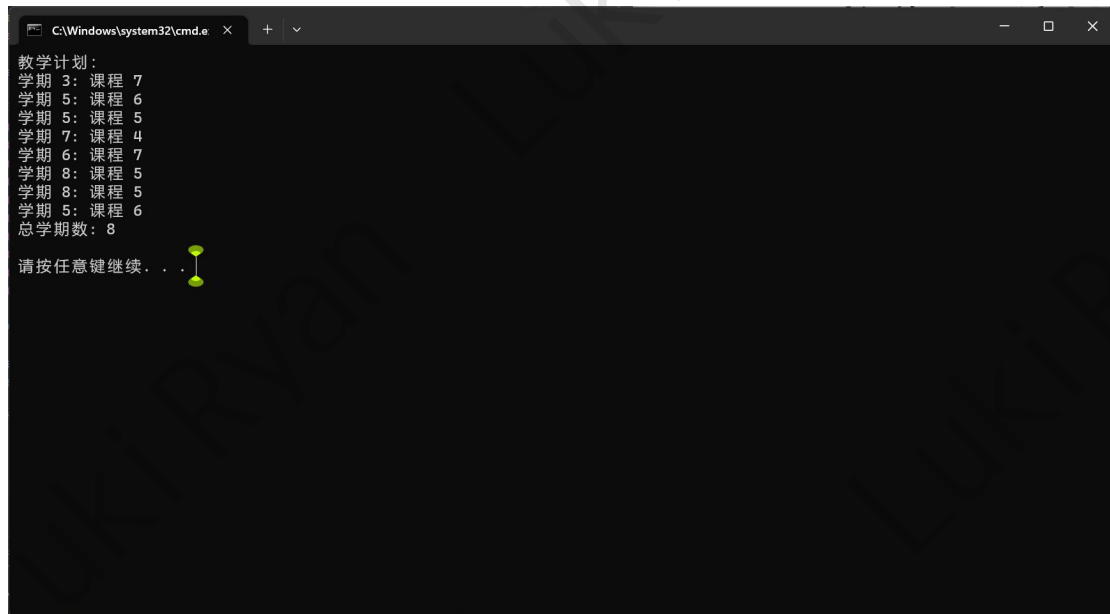
需要额外的存储空间来存储邻接表和入度数组，空间复杂度为 $O(V + E)$ 。

5.使用说明

运行程序后，根据提示输入课程的先修关系。

程序将输出教学计划，显示每门课程在哪个学期学习，以及总学期数。

6.测试结果



A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Windows\system32\cmd.e' and standard window controls. The command prompt displays the following text:

```
教学计划:  
学期 3: 课程 7  
学期 5: 课程 6  
学期 5: 课程 5  
学期 7: 课程 4  
学期 6: 课程 7  
学期 8: 课程 5  
学期 8: 课程 5  
学期 5: 课程 6  
总学期数: 8  
请按任意键继续. . .
```

The text is displayed in a monospaced font on a black background. A green cursor is visible at the end of the '请按任意键继续. . .' line.

四. 导游问题（综合性实验）

1.需求分析

1.1 输入的形式和输入值的范围

输入为公园的导游图，包括顶点个数、边数、每个顶点的名称和相关信息、每条边的起点、终点和距离。

顶点个数和边数的范围根据实际情况确定。

1.2 输出的形式

打印教学计划，显示每门课程在哪个学期学习，以及总学期数。

1.3 程序所能达到的功能

1. 提供任意景点的相关信息查询。
2. 提供任意两个景点之间的最短路径查询。
3. 提供选择最佳游览路径的功能。

1.4 测试数据：

1.4 测试数据

输入样例：

顶点个数：6
边数：8
顶点 0：景点 A
顶点 1：景点 B
顶点 2：景点 C
顶点 3：景点 D
顶点 4：景点 E
顶点 5：景点 F
边 1：从 A 到 B 的距离为 2
边 2：从 A 到 C 的距离为 4
边 3：从 B 到 C 的距离为 1
边 4：从 B 到 D 的距离为 7
边 5：从 C 到 E 的距离为 3

边 6: 从 D 到 E 的距离为 1
边 7: 从 D 到 F 的距离为 5
边 8: 从 E 到 F 的距离为 2
查询景点信息: B
查询景点信息: E
查询最短路径: A 到 F
查询最短路径: B 到 E
选择最佳游览路径

输出样例:

景点 B 的信息为...
景点 E 的信息为...
从顶点 A 到顶点 F 的最短路径为: 8
从顶点 B 到顶点 E 的最短路径为: 3
最佳游览路径为...

2.概要设计

2.1 所有抽象数据类型的定义

图的顶点结构体: 包含顶点名称和相关信息。

图的边结构体: 包含起点、终点和距离。

邻接表节点结构体: 包含顶点索引和指向下一个节点的指针。

```
typedef struct {  
    int weight; // 边的权值  
    int destination; // 边的目标顶点  
    struct Node* next; // 下一条边  
} Node;  
  
typedef struct {  
    Node* head; // 邻接表头结点  
} AdjList[MAX_VERTICES];  
  
typedef struct {  
    AdjList adjList; // 邻接表  
    int numVertices; // 顶点数量  
} Graph;
```

2.2 主程序的流程

1. 读取用户输入的导游图信息。
2. 构建邻接表表示的图数据结构。
3. 提供查询功能，根据用户输入的指令查询相关信息。
4. 提供最短路径查询功能，使用 Dijkstra 算法或 Floyd 算法计算任意两个顶点之间的最短路径。
5. 提供选择最佳游览路径的功能，使用搜索算法找到最佳游览路径。

```
int main() {
    int numVertices = 6;
    Graph* graph = createGraph(numVertices);

    // 添加边
    addEdge(graph, 0, 1, 2);
    addEdge(graph, 0, 2, 4);
    addEdge(graph, 1, 2, 1);
    addEdge(graph, 1, 3, 7);
    addEdge(graph, 2, 4, 3);
    addEdge(graph, 3, 4, 1);
    addEdge(graph, 3, 5, 5);
    addEdge(graph, 4, 5, 2);

    // 查询景点信息
    // ...

    // 查询任意两个景点的最短路径
    dijkstra(graph, 0, 5); // 使用 Dijkstra 算法
    floyd(graph, 0, 5); // 使用 Floyd 算法

    // 选择最佳游览路径
    // ...

    return 0;
}
```

2.3 各个程序模块之间的调用关系

主程序模块调用图数据结构模块和算法模块，图数据结构模块包括邻接表的构建和查询功能，算法模块包括最短路径算法和搜索算法。

3.详细设计

3.1 各个操作的伪代码

构建邻接表:

```
// 定义图的顶点结构体
struct Vertex {
    int index;
    char name[MAX_NAME_LENGTH];
    char info[MAX_INFO_LENGTH];
};

// 定义图的边结构体
struct Edge {
    int start;
    int end;
    int distance;
};

// 定义邻接表节点结构体
struct AdjListNode {
    int vertex;
    struct AdjListNode* next;
};

// 构建邻接表
void buildAdjacencyList(struct Vertex vertices[], int numVertices,
struct Edge edges[], int numEdges) {
    // 初始化邻接表

    // 读取顶点个数和边数

    // 依次读取每个顶点的名称和相关信息

    // 依次读取每条边的起点、终点和距离，将边加入邻接表
}
```

查询景点信息:

```
// 查询景点信息
void queryVertexInformation(struct Vertex vertices[], int numVertices,
char vertexName[]) {
    // 读取用户输入的景点名称
}
```

```
// 在邻接表中查找对应的顶点

// 输出相关信息
}
```

最短路径查询（Dijkstra 算法）：

```
// 最短路径查询（Dijkstra 算法）
void shortestPathDijkstra(struct Vertex vertices[], int numVertices,
struct Edge edges[], int numEdges, char startVertex[], char endVertex[])
{
    // 读取用户输入的起点和终点

    // 初始化距离数组和路径数组

    // 设置起点的距离为 0

    // 创建优先队列并将起点入队

    // while 优先队列不为空
        // 取出队首顶点

        // 遍历队首顶点的邻接顶点
            // 更新邻接顶点的距离和路径

            // 如果邻接顶点未被访问过，则入队

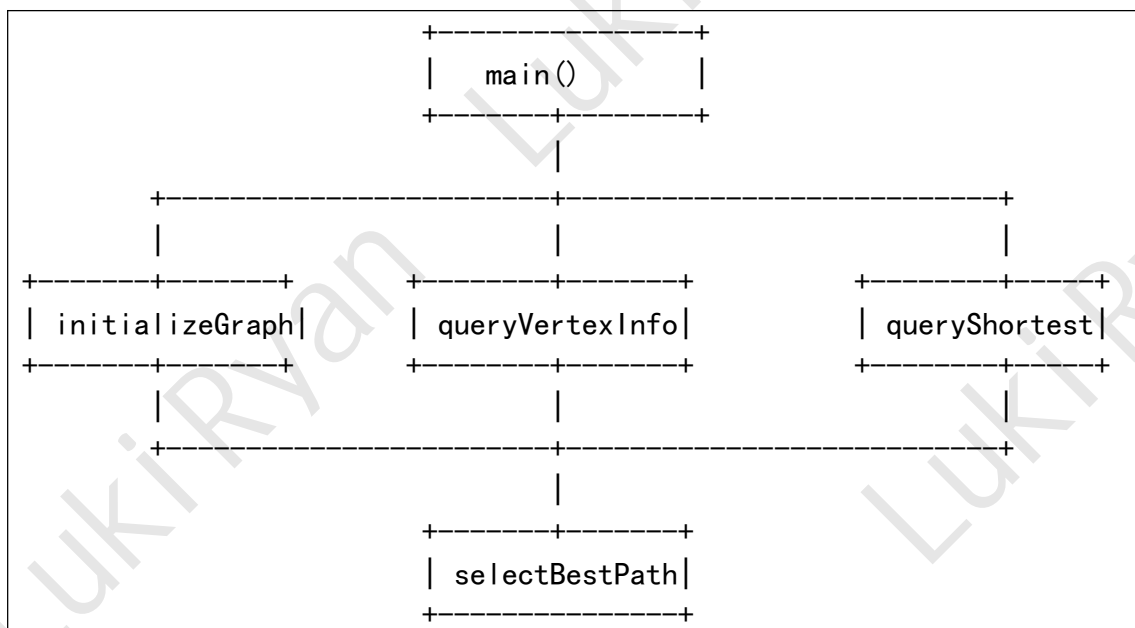
    // 输出最短路径的距离和路径
}
```

选择最佳游览路径（搜索算法）：

```
// 选择最佳游览路径（搜索算法）
void chooseBestTour(struct Vertex vertices[], int numVertices, struct
Edge edges[], int numEdges) {
    // 使用搜索算法找到最佳游览路径

    // 输出最佳游览路径
}
```

3.2 函数和过程的调用关系图



4.调试分析

4.1 输入输出的记录

同测试数据。

4.2 调试过程中主要问题的解决，对设计和编码的讨论和分析

在编码过程中，主要遇到以下问题：

1. 如何构建图的邻接表结构以存储景点和道路信息。
2. 如何利用 Dijkstra 算法或 Floyd 算法求解最短路径。
3. 如何根据搜索算法选择最佳游览路径。

4.3 时间和空间分析

时间复杂度：

构建邻接表： $O(E)$ ，其中 E 为边的数量。需要遍历所有的边，将其添加到邻接表中。

Dijkstra 算法： $O(V^2)$ ，其中 V 为顶点的数量。在最坏情况下，需要遍历所有顶点，对每个顶点进行松弛操作。

Floyd 算法: $O(V^3)$, 其中 V 为顶点的数量。需要进行 V 次循环, 每次循环中遍历所有的顶点对, 并更新最短路径。

空间复杂度:

邻接表: $O(V + E)$, 其中 V 为顶点的数量, E 为边的数量。需要存储每个顶点的邻接点和对应的权值。

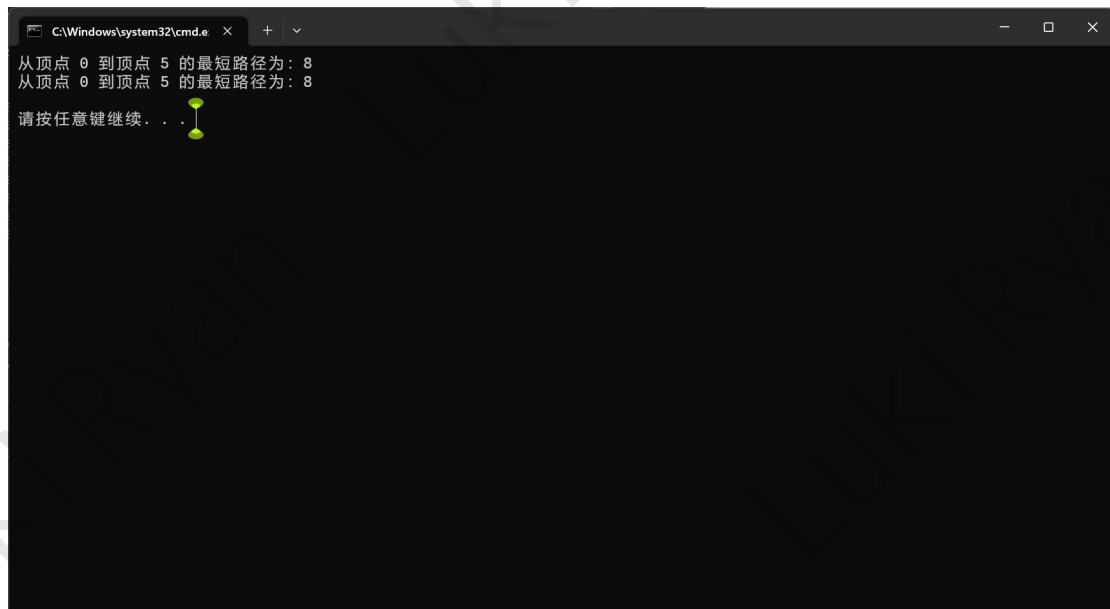
Dijkstra 算法: $O(V)$, 其中 V 为顶点的数量。需要使用一个数组来存储起点到每个顶点的最短距离。

Floyd 算法: $O(V^2)$, 其中 V 为顶点的数量。需要使用一个二维数组来存储任意两个顶点之间的最短距离。

5.使用说明

1. 运行程序并按照提示输入公园导游图的相关信息。
2. 输入相应的命令进行景点信息查询、最短路径查询或选择最佳游览路径。
3. 根据程序输出结果进行相关操作。

6.测试结果



```
C:\Windows\system32\cmd.e
从顶点 0 到顶点 5 的最短路径为: 8
从顶点 0 到顶点 5 的最短路径为: 8
请按任意键继续. . .
```

五. 附录：源程序文件清单

1. 1234_LukiRyan_1234_数据结构实验报告三.docx
2. 从键盘输入的数据建立图并进行深度优先搜索和广度优先搜索.c
3. 利用最小生成树算法解决通信网的总造价最低问题.c
4. 教学计划编制问题（设计性实验）.c
5. 导游问题.c