图的两种遍历算法

20123211 高雨晴

0. 概述

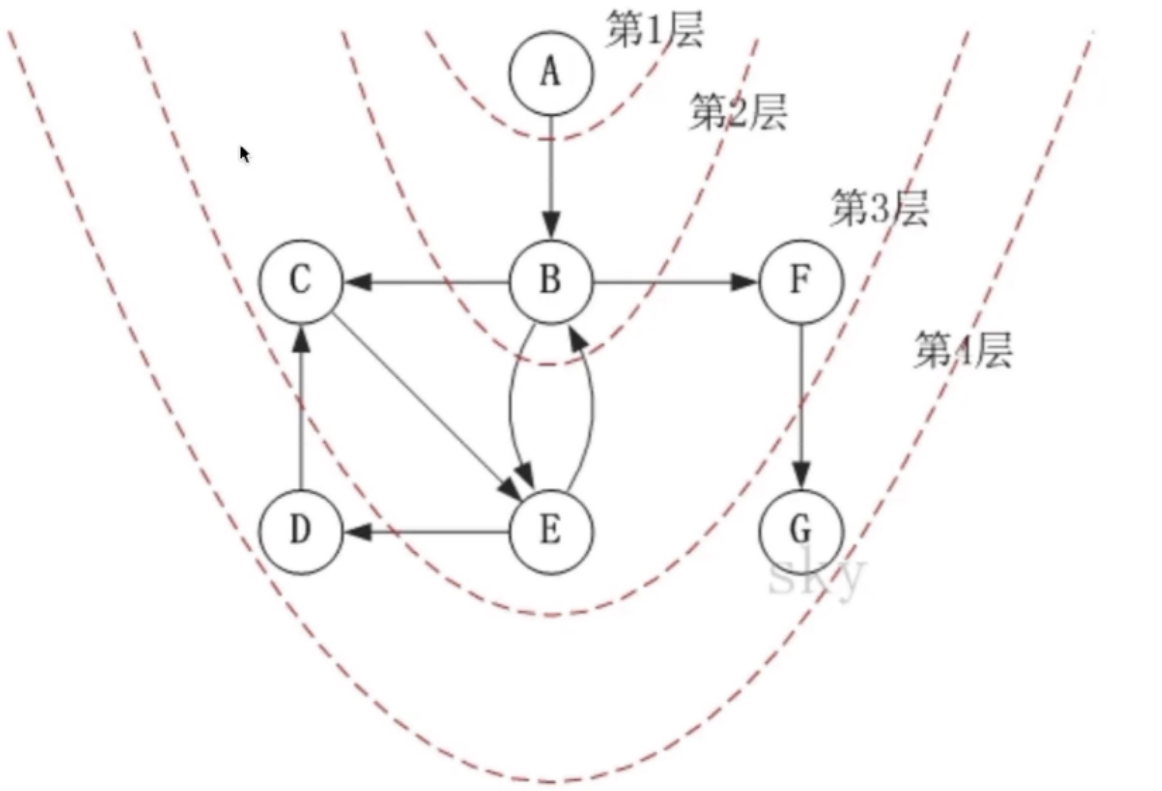
概要说明本实验的内容、主要方法或技术、以及实验结果

- 实验内容：实现图的广度优先遍历和深度优先遍历算法。
- 实验方法：使用C++语言，通过MinGW-w64 C/C++编译器
- 实验结果：输出结果和预期相符。

1. 实验设计

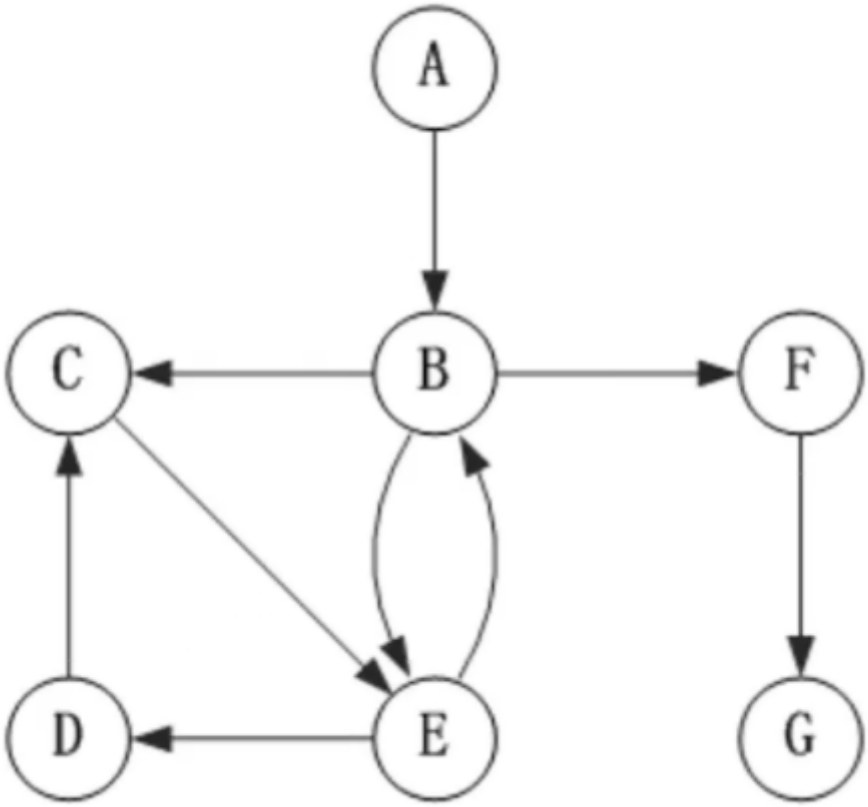
1.1 算法实现原理

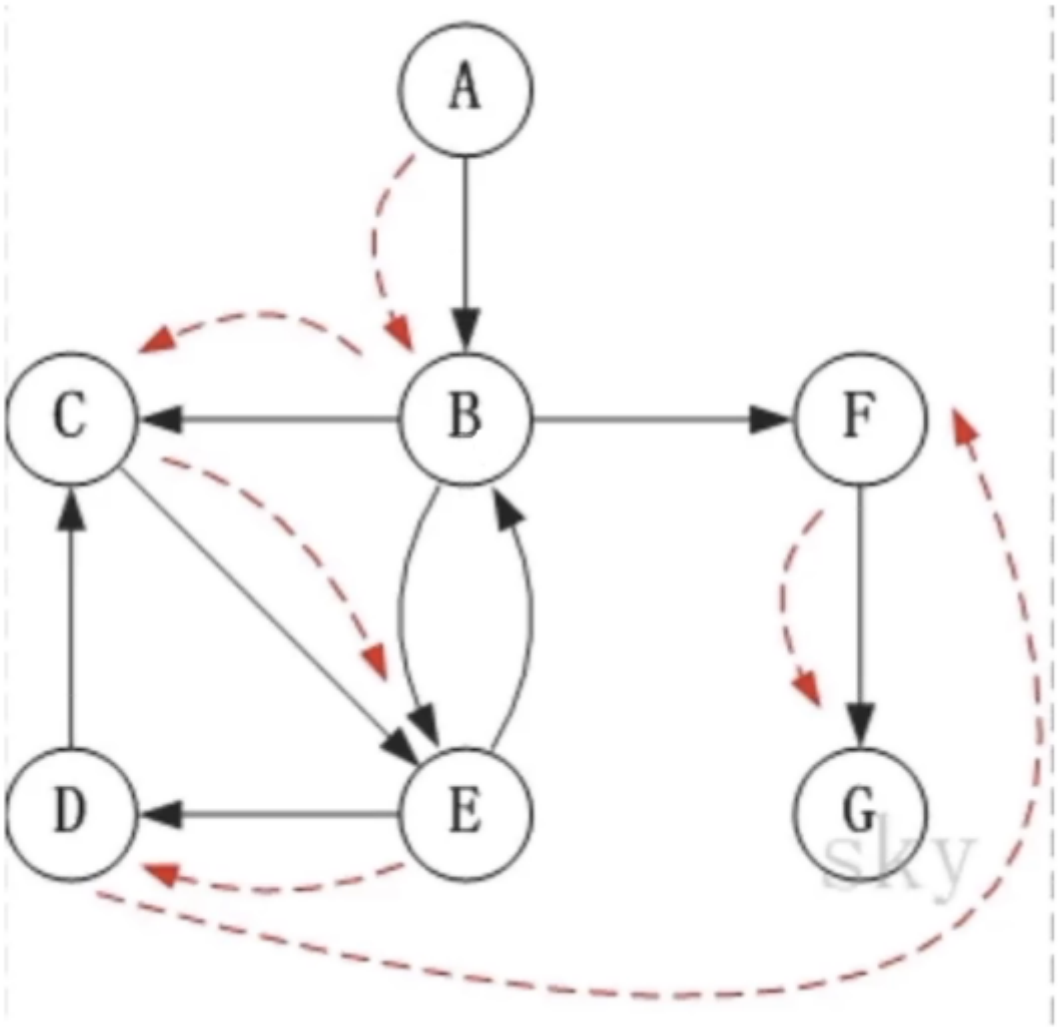
1.1.1 广度优先遍历



- 从图中的某一顶点v出发，在访问v之后依次访问v的各个没有访问到的邻接点。
- 然后，分别从这些邻接点出发，依次访问它们的邻接点，使得先被访问的顶点的邻接点先于后被访问顶点的邻接点被访问，直到图中所有已经被访问的邻接点都被访问到。
- 如果此时途中尚有顶点未被访问，则另选一个未曾访问过的顶点作为新的起始点，重复上述过程直至所有顶点都被访问到。
- 如图所示，广度优先遍历的结果为A-B-C-E-F-D-G

1.1.2 深度优先遍历





- 假设初始状态所有顶点都没有被访问，然后从每一个顶点v出发，先访问该顶点。
- 然后，依次从它的各个未被访问的邻接点出发，深度优先遍历图，直到图中所有和v相通的顶点都被访问到
- 遍历完后，还有其他定点没被访问到，则另选一个未被访问的顶点作为起始点。
- 重复上述过程，直到所有顶点都被访问完为止。
- 如图所示，深度优先遍历的结果为A-B-C-E-D-F-G

1.2 算法设计与分析

1. 定义节点

```
typedef struct eNode {
    int adjVer;                 //该边的邻接点编号
    int weight;                 //该边的的信息，如权值
    struct eNode* nextEdge;     //指向下一条边的指针
}EdgeNode;

typedef struct vNode {
    EdgeNode* firstEdge;        //指向第一个边结点
}VNode;

typedef struct list {
    int n;                      //顶点个数
    int e;                      //边数
```

```
    VNode adjList[MAX_VERTICE]; //邻接表的头结点数组
}ListGraph;
```

2. 定义队列

```
typedef struct quene {                  //定义顺序队
    int front;                          //队头指针
    char data[MAX_VERTICE];             //存放队中元素
    int rear;                           //队尾指针
}SqQueue;
```

3. 初始化队列，定义出队入队操作

```
// 初始化队列
void initQueue(SqQueue*& q) {
    q = (SqQueue*)malloc(sizeof(SqQueue));  //分配一个空间
    q->front = q->rear = -1;                //置 -1
}

// 判断队列是否为空
bool emptyQueue(SqQueue*& q) {
    if (q->front == q->rear) {              //首指针和尾指针相等，说明为空
        return true;                        //返回真
    }
    else {
        return false;                       //返回假
    }
}

// 进队列
int enQueue(SqQueue*& q, char c) {
    if (q->rear == MAX_VERTICE - 1) {       //判断队列是否已经满了
        return -1;                          //返回 -1
    }
    q->rear++;                              //头指针加 1
    q->data[q->rear] = c;                   //传值
    return c;                               //返回 c
}

// 出队列
int deQueue(SqQueue*& q, char ch) {
    if (q->front == q->rear) {              //判断队列是否已经空了
        return -1;                          //返回假 -1
    }
    q->front++;                             //尾指针加 1
    ch = q->data[q->front];                 //取值
    return ch;                              //返回 ch
}
```

4. 创建邻接表

```

```

```
void createAdjListGraph(ListGraph* &LG, int A[MAX_VERTICE][MAX_VERTICE], int
n, int e) {
    int i, j;
    EdgeNode* p;
    LG = (ListGraph*)malloc(sizeof(ListGraph));
    for (i = 0; i < n; i++) {
        LG->adjList[i].firstEdge = NULL;                          //给邻接表中所
有头结点指针域置初值
    }
    for (i = 0; i < n; i++) {                                     //检查邻接矩阵
中的每个元素
        for (j = n - 1; j >= 0; j--) {
            if (A[i][j] != 0) {                                   //存在一条边
                p = (EdgeNode*)malloc(sizeof(EdgeNode));          //申请一个结点
内存
                p->adjVer = j;                                    //存放邻接点
                p->weight = A[i][j];                              //存放权值
                p->nextEdge = NULL;

                p->nextEdge = LG->adjList[i].firstEdge;           //头插法
                LG->adjList[i].firstEdge = p;
            }
        }
    }
    LG->n = n;
    LG->e = e;
}
```

5. 输出邻接表

```
void displayAdjList(ListGraph* LG) {
    int i;
    EdgeNode* p;
    for (i = 0; i < MAX_VERTICE; i++) {
        p = LG->adjList[i].firstEdge;
        printf("%d:", i);
        while (p != NULL) {
            if (p->weight != INF) {
                printf("%2d[%d]->", p->adjVer, p->weight);
            }
            p = p->nextEdge;
        }
        printf(" NULL\n");
    }
}
```

6. 深度优先遍历

```
int visitedDFS[MAX_VERTICE] = { 0 };                              //全
局数组，记录是否遍历

void DFS(ListGraph* LG, int v) {
    EdgeNode* p;
```

```
    visitedDFS[v] = 1;                                          //记录已访问，
标记为1
    printf("%2d", v);                                           //输出顶点编号
    p = LG->adjList[v].firstEdge;                               //p 指向顶点 v
的第一个邻接点
    while (p != NULL) {
        if (visitedDFS[p->adjVer] == 0 && p->weight != INF) {   //如果 p-
>adjVer 没被访问，递归访问它
            DFS(LG, p->adjVer);
        }
        p = p->nextEdge;                                        //p 指向顶点 v
的下一个邻接点
    }
}
```

7. 广度优先遍历

```
void BFS(ListGraph* LG, int v) {
    int ver;                                                    //定义出队
顶点
    EdgeNode* p;
    SqQueue* sq;                                                //定义指针
    initQueue(sq);                                              //初始化队
列
    int visitedBFS[MAX_VERTICE] = { 0 };
//初始化访问标记数组
    enQueue(sq, v);                                             //初始点进
队
    printf("%2d", v);
    visitedBFS[v] = 1;                                          //打印并标
记要出队顶点
    while (!emptyQueue(sq)) {                                   //队为空结
束循环
        ver = deQueue(sq, v);                                   //出队，并
得到出队信息
        p = LG->adjList[ver].firstEdge;                         //指向出队
的第一个邻接点
        while (p != NULL) {                                     //查找
ver 的所有邻接点
            if (visitedBFS[p->adjVer] == 0 && p->weight != INF) {   //如果没被
访问
                printf("%2d", p->adjVer);                       //打印该顶
点信息
                visitedBFS[p->adjVer] = 1;                      //置已访问
状态
                enQueue(sq, p->adjVer);                         //该顶点进
队
            }
            p = p->nextEdge;                                    //找下一个
邻接点
        }
    }
    printf("\n");
}
```

2. 实验结果说明与分析

2.1 实验设置

首先写出我们例子中的图的邻接表。

	A(0)	B(1)	C(2)	D(3)	E(4)	F(5)	G(6)
A(0)	0	1					
B(1)		0	1		1	1	
C(2)			0		1		
D(3)			1	0			
E(4)		1		1	0		
F(5)						0	1
G(6)							0

在主函数中调用两种遍历算法。

```
int main() {
    ListGraph* LG;
    int array[MAX_VERTICE][MAX_VERTICE] = {
        {  0,  1,INF,INF,INF,INF,INF},
        {INF,  0,  1,INF,  1,  1,INF},
        {INF,INF,  0,INF,  1,INF,INF},
        {INF,INF,  1,  0,INF,INF,INF},
        {INF,  1,INF,  1,  0,INF,INF},
        {INF,INF,INF,INF,INF,  0,  1},
        {INF,INF,INF,INF,INF,INF,  0}
    };

    int e = 12;
    createAdjListGraph(LG, array, MAX_VERTICE, e);

    printf("adjacency list: \n");
    displayAdjList(LG);                          // 打印邻接表
    printf("\n");

    printf("Depth-First:");                      // 深度优先遍历结果
    DFS(LG,0);                                   // 从A（标号为0）出发
    printf("\n");

    printf("Breadth-First:");                    // 广度优先遍历结果
    BFS(LG, 0);
    printf("\n");

    system("pause");
    return 0;
```

```
}
```

2.2 实验结果以及相应的分析

输出结果：

```
adjacency list:
0: 1[1]-> NULL
1: 2[1]-> 4[1]-> 5[1]-> NULL
2: 4[1]-> NULL
3: 2[1]-> NULL
4: 1[1]-> 3[1]-> NULL
5: 6[1]-> NULL
6: NULL

Depth-First: 0 1 2 4 3 5 6
Breadth-First: 0 1 2 4 5 3 6
```

深度优先遍历的结果是 A-B-C-E-D-F-G

广度优先遍历的结果是 A-B-C-E-F-D-G

与我们自行计算的结果相符。

3. 总结

在遍历图时，对每个顶点进行访问，若某顶点被标记成为已访问，则不再从它出发进行搜索。因此遍历图的过程实质上是查找每个顶点和其邻接点的过程，耗费的时间取决于采用的储存结构。

在邻接表中，查找每个顶点所需的时间为O(|V|)，查找每个邻接点的时间为O(|E|)，其中V为点的集合，E为边的集合。故两种算法的时间复杂度为O（|V|+|E|）。

4. 附录

```
#include<stdio.h>
#include<malloc.h>

#define MAX_VERTICE 7
#define INF 32767

typedef struct eNode {
    int adjVer;
    int weight;
    struct eNode* nextEdge;
}EdgeNode;

typedef struct vNode {
    EdgeNode* firstEdge;
}VNode;

typedef struct list {
    int n;
    int e;
    VNode adjList[MAX_VERTICE];
```

```
}ListGraph;

typedef struct quene {
    int front;
    char data[MAX_VERTICE];
    int rear;
}SqQueue;

//初始化队列
void initQueue(SqQueue*& q) {
    q = (SqQueue*)malloc(sizeof(SqQueue));
    q->front = q->rear = -1;
}

//判断队列是否为空
bool emptyQueue(SqQueue*& q) {
    if (q->front == q->rear) {
        return true;
    }
    else {
        return false;
    }
}

//进队列
int enQueue(SqQueue*& q, char c) {
    if (q->rear == MAX_VERTICE - 1) {
        return -1;
    }
    q->rear++;
    q->data[q->rear] = c;
    return c;
}

//出队列
int deQueue(SqQueue*& q, char ch) {
    if (q->front == q->rear) {
        return -1;
    }
    q->front++;
    ch = q->data[q->front];
    return ch;
}

//创建图的邻接表
void createAdjListGraph(ListGraph* &LG, int A[MAX_VERTICE][MAX_VERTICE], int n,
int e) {
    int i, j;
    EdgeNode* p;
    LG = (ListGraph*)malloc(sizeof(ListGraph));
    for (i = 0; i < n; i++) {
        LG->adjList[i].firstEdge = NULL;
    }
    for (i = 0; i < n; i++) {
        for (j = n - 1; j >= 0; j--) {
```

```
            if (A[i][j] != 0) {
                p = (EdgeNode*)malloc(sizeof(EdgeNode));
                p->adjVer = j;
                p->weight = A[i][j];
                p->nextEdge = NULL;

                p->nextEdge = LG->adjList[i].firstEdge;
                LG->adjList[i].firstEdge = p;
            }
        }
    }
    LG->n = n;
    LG->e = e;
}

//输出邻接表
void displayAdjList(ListGraph* LG) {
    int i;
    EdgeNode* p;
    for (i = 0; i < MAX_VERTICE; i++) {
        p = LG->adjList[i].firstEdge;
        printf("%d:", i);
        while (p != NULL) {
            if (p->weight != INF) {
                printf("%2d[%d]->", p->adjVer, p->weight);
            }
            p = p->nextEdge;
        }
        printf(" NULL\n");
    }
}

//深度优先遍历
int visitedDFS[MAX_VERTICE] = { 0 };
void DFS(ListGraph* LG, int v) {
    EdgeNode* p;
    visitedDFS[v] = 1;
    printf("%2d", v);
    p = LG->adjList[v].firstEdge;
    while (p != NULL) {
        if (visitedDFS[p->adjVer] == 0 && p->weight != INF) {
            DFS(LG, p->adjVer);
        }
        p = p->nextEdge;
    }
}

//广度优先遍历
void BFS(ListGraph* LG, int v) {
    int ver;
    EdgeNode* p;
    SqQueue* sq;
    initQueue(sq);
    int visitedBFS[MAX_VERTICE] = { 0 };
    enQueue(sq, v);
```

```
    printf("%2d", v);
    visitedBFS[v] = 1;
    while (!emptyQueue(sq)) {
        ver = deQueue(sq, v);
        p = LG->adjList[ver].firstEdge;
        while (p != NULL) {
            if (visitedBFS[p->adjVer] == 0 && p->weight != INF) {
                printf("%2d", p->adjVer);
                visitedBFS[p->adjVer] = 1;
                enQueue(sq, p->adjVer);
            }
            p = p->nextEdge;
        }
    }
    printf("\n");
}

int main() {
    ListGraph* LG;
    int array[MAX_VERTICE][MAX_VERTICE] = {
        {  0,  1,INF,INF,INF,INF,INF},
        {INF,  0,  1,INF,  1,  1,INF},
        {INF,INF,  0,INF,  1,INF,INF},
        {INF,INF,  1,  0,INF,INF,INF},
        {INF,  1,INF,  1,  0,INF,INF},
        {INF,INF,INF,INF,INF,  0,  1},
        {INF,INF,INF,INF,INF,INF,  0}
    };

    int e = 12;
    createAdjListGraph(LG, array, MAX_VERTICE, e);

    printf("adjacency list: \n");
    displayAdjList(LG);
    printf("\n");

    printf("Depth-First:");
    DFS(LG,0);
    printf("\n");

    printf("Breadth-First:");
    BFS(LG, 0);
    printf("\n");

    system("pause");
    return 0;
}
```