阅读：34,228　　作者：解学武

深度优先搜索（DFS、深搜）和广度优先搜索（BFS、广搜）

上一节　　下一节

前边介绍了有关图的 4 种存储方式，本节介绍如何对存储的图中的顶点进行遍历。常用的遍历方式有两种：深度优先搜索和广度优先搜索。

深度优先搜索（简称“深搜”或DFS）

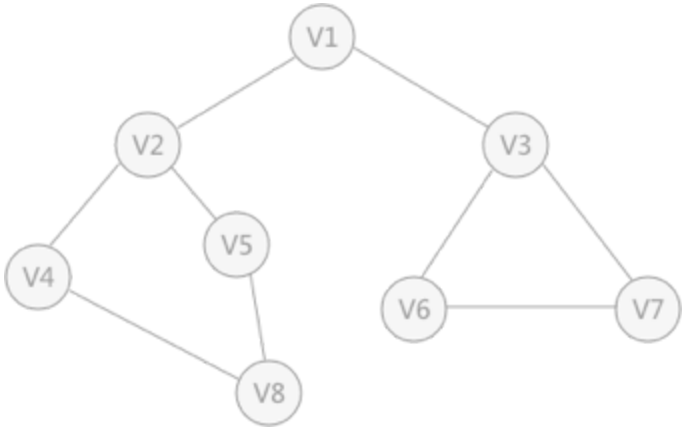

图 1 无向图

深度优先搜索的过程类似于树的先序遍历，首先从例子中体会深度优先搜索。例如图 1 是一个无向图，采用深度优先算法遍历这个图的过程为：

1. 首先任意找一个未被遍历过的顶点，例如从 V1 开始，由于 V1 率先访问过了，所以，需要标记 V1 的状态为访问过；
2. 然后遍历 V1 的邻接点，例如访问 V2 ，并做标记，然后访问 V2 的邻接点，例如 V4 （做标记），然后 V8 ，然后 V5 ；
3. 当继续遍历 V5 的邻接点时，根据之前做的标记显示，所有邻接点都被访问过了。此时，从 V5 回退到 V8 ，看 V8 是否有未被访问过的邻接点，如果没有，继续回退到 V4 ， V2 ， V1 ；
4. 通过查看 V1 ， 找到一个未被访问过的顶点 V3 ， 继续遍历，然后访问 V3 邻接点 V6 ， 然后 V7 ；
5. 由于 V7 没有未被访问的邻接点，所有回退到 V6 ， 继续回退至 V3 ， 最后到达 V1 ， 发现没有未被访问的；
6. 最后一步需要判断是否所有顶点都被访问，如果还有没被访问的，以未被访问的顶点为第一个顶点，继续依照上边的方式进行遍历。

根据上边的过程，可以得到图 1 通过深度优先搜索获得的顶点的遍历次序为：

```
V1 -> V2 -> V4 -> V8 -> V5 -> V3 -> V6 -> V7
```

所谓深度优先搜索，是从图中的一个顶点出发，每次遍历当前访问顶点的临界点，一直到访问的顶点没有未被访问过的临界点为止。然后采用依次回退的方式，查看来的路上每一个顶点是否有其它未被访问的临界点。访问完成后，判断图中的顶点是否已经全部遍历完成，如果没有，以未访问的顶点为起始点，重复上述过程。

> 深度优先搜索是一个不断回溯的过程。

采用深度优先搜索算法遍历图的实现代码为：

```
#include <stdio.h>

#define MAX_VERtEX_NUM 20                       //顶点的最大个数
#define VRType int                              //表示顶点之间的关系的变量类型
#define InfoType char                           //存储弧或者边额外信息的指针变量类型
#define VertexType int                          //图中顶点的数据类型

typedef enum{false,true}bool;                   //定义bool型常量
bool visited[MAX_VERtEX_NUM];                   //设置全局数组，记录标记顶点是否被访问过

typedef struct {
    VRType adj;                                 //对于无权图，用 1 或 0 表示是否相邻；对于带权图，直
    InfoType * info;                            //弧或边额外含有的信息指针
}ArcCell,AdjMatrix[MAX_VERtEX_NUM][MAX_VERtEX_NUM];

typedef struct {
    VertexType vexs[MAX_VERtEX_NUM];            //存储图中顶点数据
    AdjMatrix arcs;                             //二维数组，记录顶点之间的关系
    int vexnum,arcnum;                          //记录图的顶点数和弧（边）数
}MGraph;
//根据顶点本身数据，判断出顶点在二维数组中的位置
int LocateVex(MGraph * G,VertexType v){
    int i=0;
    //遍历一维数组，找到变量v
    for (; i<G->vexnum; i++) {
        if (G->vexs[i]==v) {
            break;
        }
    }
    //如果找不到，输出提示语句，返回-1
    if (i>G->vexnum) {
        printf("no such vertex.\n");
        return -1;
    }
    return i;
}
```

```
//构造无向图
void CreateDN(MGraph *G){
    scanf("%d,%d",&(G->vexnum),&(G->arcnum));
    for (int i=0; i<G->vexnum; i++) {
        scanf("%d",&(G->vexs[i]));
    }
    for (int i=0; i<G->vexnum; i++) {
        for (int j=0; j<G->vexnum; j++) {
            G->arcs[i][j].adj=0;
            G->arcs[i][j].info=NULL;
        }
    }
    for (int i=0; i<G->arcnum; i++) {
        int v1,v2;
        scanf("%d,%d",&v1,&v2);
        int n=LocateVex(G, v1);
        int m=LocateVex(G, v2);
        if (m==-1 ||n==-1) {
            printf("no this vertex\n");
            return;
        }
        G->arcs[n][m].adj=1;
        G->arcs[m][n].adj=1;//无向图的二阶矩阵沿主对角线对称
    }
}

int FirstAdjVex(MGraph G,int v)
{
    //查找与数组下标为v的顶点之间有边的顶点，返回它在数组中的下标
    for(int i = 0; i<G.vexnum; i++){
        if( G.arcs[v][i].adj ){
            return i;
        }
    }
    return -1;
}
int NextAdjVex(MGraph G,int v,int w)
{
    //从前一个访问位置w的下一个位置开始，查找之间有边的顶点
    for(int i = w+1; i<G.vexnum; i++){
        if(G.arcs[v][i].adj){
            return i;
        }
    }
    return -1;
}
void visitVex(MGraph G, int v){
```

```
    printf("%d ",G.vexs[v]);
}
void DFS(MGraph G,int v){
    visited[v] = true;//标记为true
    visitVex( G,  v); //访问第v 个顶点
    //从该顶点的第一个边开始，一直到最后一个边，对处于边另一端的顶点调用DFS函数
    for(int w = FirstAdjVex(G,v); w>=0; w = NextAdjVex(G,v,w)){
        //如果该顶点的标记位false，证明未被访问，调用深度优先搜索函数
        if(!visited[w]){
            DFS(G,w);
        }
    }
}
//深度优先搜索
void DFSTraverse(MGraph G){//
    int v;
    //将用做标记的visit数组初始化为false
    for( v = 0; v < G.vexnum; ++v){
        visited[v] = false;
    }
    //对于每个标记为false的顶点调用深度优先搜索函数
    for( v = 0; v < G.vexnum; v++){
        //如果该顶点的标记位为false，则调用深度优先搜索函数
        if(!visited[v]){
            DFS( G, v);
        }
    }
}

int main() {
    MGraph G;//建立一个图的变量
    CreateDN(&G);//初始化图
    DFSTraverse(G);//深度优先搜索图
    return 0;
}
```

以图 1 为例，运行结果为：

```
8,9
1
2
3
4
5
6
7
```

```
8
1,2
2,4
2,5
4,8
5,8
1,3
3,6
6,7
7,3
1 2 4 8 5 3 6 7
```

广度优先搜索

广度优先搜索类似于树的层次遍历。从图中的某一顶点出发，遍历每一个顶点时，依次遍历其所有的邻接点，然后再从这些邻接点出发，同样依次访问它们的邻接点。按照此过程，直到图中所有被访问过的顶点的邻接点都被访问到。

最后还需要做的操作就是查看图中是否存在尚未被访问的顶点，若有，则以该顶点为起始点，重复上述遍历的过程。

还拿图 1 中的无向图为例，假设 V1 作为起始点，遍历其所有的邻接点 V2 和 V3 ，以 V2 为起始点，访问邻接点 V4 和 V5 ，以 V3 为起始点，访问邻接点 V6 、 V7 ，以 V4 为起始点访问 V8 ，以 V5 为起始点，由于 V5 所有的起始点已经全部被访问，所有直接略过， V6 和 V7 也是如此。
以 V1 为起始点的遍历过程结束后，判断图中是否还有未被访问的点，由于图 1 中没有了，所以整个图遍历结束。遍历顶点的顺序为：

```
V1 -> V2 -> v3 -> V4 -> V5 -> V6 -> V7 -> V8
```

广度优先搜索的实现需要借助队列这一特殊数据结构，实现代码为：

```
01.  #include <stdio.h>
02.  #include <stdlib.h>
03.  #define MAX_VERtEX_NUM 20                    //顶点的最大个数
04.  #define VRType int                           //表示顶点之间的关系的变量类型
05.  #define InfoType char                        //存储弧或者边额外信息的指针变量类型
06.  #define VertexType int                       //图中顶点的数据类型
07.  typedef enum{false,true}bool;                 //定义bool型常量
08.  bool visited[MAX_VERtEX_NUM];                 //设置全局数组，记录标记顶点是否被访问过
09.  typedef struct Queue{
10.      VertexType data;
11.      struct Queue * next;
```

```
}Queue;
typedef struct {
    VRType adj;                                 //对于无权图，用 1 或 0 表示是否相邻；对于带权图，直
    InfoType * info;                            //弧或边额外含有的信息指针
}ArcCell,AdjMatrix[MAX_VERtEX_NUM][MAX_VERtEX_NUM];

typedef struct {
    VertexType vexs[MAX_VERtEX_NUM];            //存储图中顶点数据
    AdjMatrix arcs;                             //二维数组，记录顶点之间的关系
    int vexnum,arcnum;                          //记录图的顶点数和弧（边）数
}MGraph;
//根据顶点本身数据，判断出顶点在二维数组中的位置
int LocateVex(MGraph * G,VertexType v){
    int i=0;
    //遍历一维数组，找到变量v
    for (; i<G->vexnum; i++) {
        if (G->vexs[i]==v) {
            break;
        }
    }
    //如果找不到，输出提示语句，返回-1
    if (i>G->vexnum) {
        printf("no such vertex.\n");
        return -1;
    }
    return i;
}
//构造无向图
void CreateDN(MGraph *G){
    scanf("%d,%d",&(G->vexnum),&(G->arcnum));
    for (int i=0; i<G->vexnum; i++) {
        scanf("%d",&(G->vexs[i]));
    }
    for (int i=0; i<G->vexnum; i++) {
        for (int j=0; j<G->vexnum; j++) {
            G->arcs[i][j].adj=0;
            G->arcs[i][j].info=NULL;
        }
    }
    for (int i=0; i<G->arcnum; i++) {
        int v1,v2;
        scanf("%d,%d",&v1,&v2);
        int n=LocateVex(G, v1);
        int m=LocateVex(G, v2);
        if (m==-1 ||n==-1) {
            printf("no this vertex\n");
            return;
```

```
            }
            G->arcs[n][m].adj=1;
            G->arcs[m][n].adj=1;//无向图的二阶矩阵沿主对角线对称
        }
    }

    int FirstAdjVex(MGraph G,int v)
    {
        //查找与数组下标为v的顶点之间有边的顶点，返回它在数组中的下标
        for(int i = 0; i<G.vexnum; i++){
            if( G.arcs[v][i].adj ){
                return i;
            }
        }
        return -1;
    }
    int NextAdjVex(MGraph G,int v,int w)
    {
        //从前一个访问位置w的下一个位置开始，查找之间有边的顶点
        for(int i = w+1; i<G.vexnum; i++){
            if(G.arcs[v][i].adj){
                return i;
            }
        }
        return -1;
    }
    //操作顶点的函数
    void visitVex(MGraph G, int v){
        printf("%d ",G.vexs[v]);
    }
    //初始化队列
    void InitQueue(Queue ** Q){
        (*Q)=(Queue*)malloc(sizeof(Queue));
        (*Q)->next=NULL;
    }
    //顶点元素v进队列
    void EnQueue(Queue **Q,VertexType v){
        Queue * element=(Queue*)malloc(sizeof(Queue));
        element->data=v;
        Queue * temp=(*Q);
        while (temp->next!=NULL) {
            temp=temp->next;
        }
        temp->next=element;
    }
    //队头元素出队列
    void DeQueue(Queue **Q,int *u){
```

```
    (*u)=(*Q)->next->data;
    (*Q)->next=(*Q)->next->next;
}
//判断队列是否为空
bool QueueEmpty(Queue *Q){
    if (Q->next==NULL) {
        return true;
    }
    return false;
}
//广度优先搜索
void BFSTraverse(MGraph G){//
    int v;
    //将用做标记的visit数组初始化为false
    for( v = 0; v < G.vexnum; ++v){
        visited[v] = false;
    }
    //对于每个标记为false的顶点调用深度优先搜索函数
    Queue * Q;
    InitQueue(&Q);
    for( v = 0; v < G.vexnum; v++){
        if(!visited[v]){
            visited[v]=true;
            visitVex(G, v);
            EnQueue(&Q, G.vexs[v]);
            while (!QueueEmpty(Q)) {
                int u;
                DeQueue(&Q, &u);
                u=LocateVex(&G, u);
                for (int w=FirstAdjVex(G, u); w>=0; w=NextAdjVex(G, u, w)) {
                    if (!visited[w]) {
                        visited[w]=true;
                        visitVex(G, w);
                        EnQueue(&Q, G.vexs[w]);
                    }
                }
            }
        }
    }
}
int main() {
    MGraph G;//建立一个图的变量
    CreateDN(&G);//初始化图
    BFSTraverse(G);//广度优先搜索图
    return 0;
}
```

例如，使用上述程序代码遍历图 1 中的无向图，运行结果为：

```
8,9
1
2
3
4
5
6
7
8
1,2
2,4
2,5
4,8
5,8
1,3
3,6
6,7
7,3
1 2 3 4 5 6 7 8
```

总结

本节介绍了两种遍历图的方式：深度优先搜索算法和广度优先搜索算法。深度优先搜索算法的实现运用的主要是回溯法，类似于树的先序遍历算法。广度优先搜索算法借助队列的先进先出的特点，类似于树的层次遍历。

推荐阅读

图的深度优先搜索和广度优先搜索	图文并茂的介绍了两种搜索算法的具体实现过程
深度优先搜索（DNS）和广度优先搜索（BFS）	详细介绍了两种搜索算法的实现过程
图的遍历算法（深度优先算法DFS和广度优先算法BFS）	详细介绍了两种算法的实现过程，并配备的 C++ 的实现代码
广度优先搜索（BFS）和深度优先搜索（DFS）的应用实例	从实例出发介绍两种搜索算法

上一节 下一节