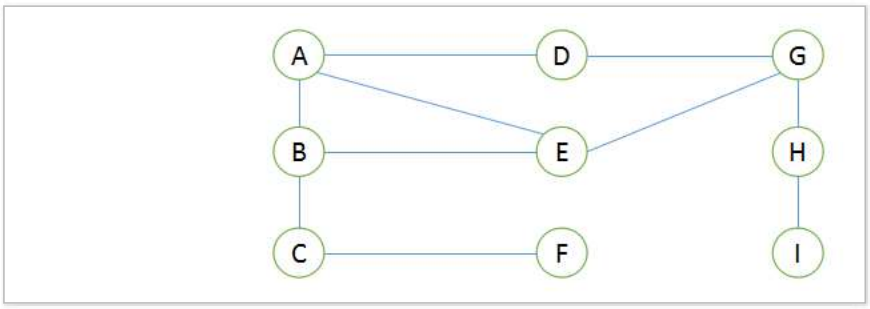
算法导论 -- 图的遍历（DFS 与 BFS） - CSDN 博客

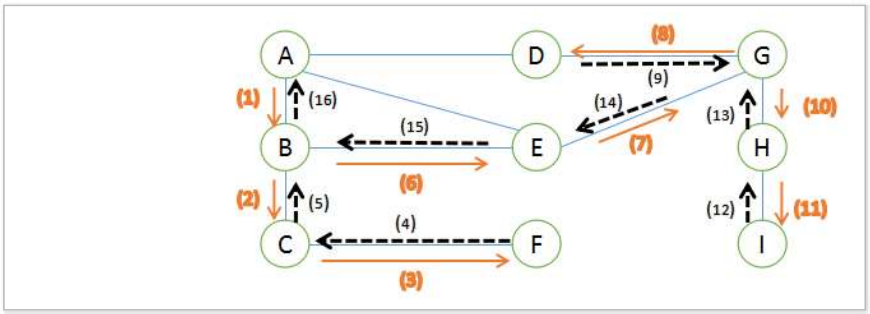
图的遍历就是从图中的某个顶点出发，按某种方法对图中的所有顶点访问且仅访问一次。为了保证图中的顶点在遍历过程中仅访问一次，要为每一个顶点设置一个访问标志。通常有两种方法：深度优先搜索 (DFS) 和广度优先搜索(BFS). 这两种算法对有向图与无向图均适用。
以下面无向图为例：



1. 深度优先搜索 (DFS)

基本步骤：

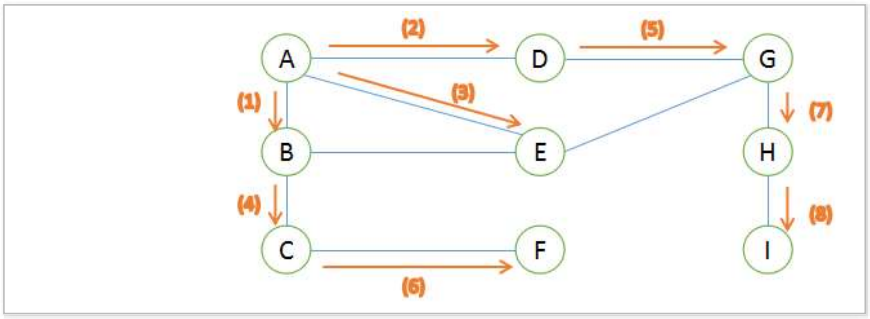
> 1. 从图中某个顶点出发，首先访问;
> 2. 访问结点的第一个邻接点，以这个邻接点作为一个新节点，访问所有邻接点。直到以出发的所有节点都被访问到，回溯到的下一个未被访问过的邻接点，以这个邻结点为新节点，重复上述步骤。直到图中所有与相通的所有节点都被访问到。
> 3. 若此时图中仍有未被访问的结点，则另选图中的一个未被访问的顶点作为起始点。重复深度优先搜索过程，直到图中的所有节点均被访问过。



2. 广度优先搜索 (BFS)

2.1 BFS 类似与树的层次遍历，从源顶点 s 出发，依照层次结构，逐层访问其他结点。即访问到距离顶点 s 为 k 的所有节点之后，才会继续访问距离为 k+1 的其他结点。
基本步骤：

> 1. 从图中某个顶点出发，首先访问;
> 2. 依次访问的各个未被访问的邻接点。
> 3. 依次从上述邻接点出发，访问他们的各个未被访问的邻接点。始终保证一点：**如果在之前被访问，则的邻接点应在的邻接点之前被访问。**重复上述步骤，直到所有顶点都被访问到。
> 4. 如果还有顶点未被访问到，则随机选择一个作为起始点，重复上述过程，直到图中所有顶点都被访问到。



为了按照优先访问顶点的次序，访问其邻接点，所以需要建立一个优先队列（先进先出）。**2.2** 采用此算法还可以很方便计算距离任一顶点的路径长度为 k 的所有顶点；从顶点出发进行广度优先搜索，可以记录到每一步，两步,…,k 步可到达的顶点。采用一个距离队列与访问队列同步，距离队列是访问队列中对应顶点距离的距离。例如距离顶点 B 为 2 的顶点为 D、F、G.

3. 完整代码

```
/*************************************************************************
CSDN 勿在浮沙筑高台 http://blog.csdn.net/luoshixian099算法导论--图的遍历2016年7月13日
*************************************************************************/
#include <iostream>
#include <vector>
#include <queue>
using namespace std;
char vextex[] = { 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I' };
typedef struct VertexNode   //链表表头结点
{
    char data;
    struct ArcNode * firstarc;
}VertexNode;
typedef struct ArcNode   //弧结点
{
    char data;
    struct ArcNode * nextarc;
}ArcNode;
ArcNode * InSertArcNode(char name)
{
    ArcNode * p = new ArcNode;
    p->data = name;
    p->nextarc = NULL;
    return p;
}
VertexNode * AdjList()//邻接链表表示法
{
    ArcNode * p=NULL;
    VertexNode * List_head = new VertexNode[9];
    int count = 0;
    List_head[count].data = 'A';
```

```
    p = List_head[count].firstarc = InSertArcNode('B');
    p = p->nextarc = InSertArcNode('D');
    p = p->nextarc = InSertArcNode('E');
    count++;
    List_head[count].data = 'B';
    p = List_head[count].firstarc = InSertArcNode('A');
    p = p->nextarc = InSertArcNode('C');
    p = p->nextarc = InSertArcNode('E');
    count++;
    List_head[count].data = 'C';
    p = List_head[count].firstarc = InSertArcNode('B');
    p = p->nextarc = InSertArcNode('F');
    count++;
    List_head[count].data = 'D';
    p = List_head[count].firstarc = InSertArcNode('A');
    p = p->nextarc = InSertArcNode('G');
    count++;
    List_head[count].data = 'E';
    p = List_head[count].firstarc = InSertArcNode('A');
    p = p->nextarc = InSertArcNode('B');
    p = p->nextarc = InSertArcNode('G');
    count++;
    List_head[count].data = 'F';
    p = List_head[count].firstarc = InSertArcNode('C');
    count++;
    List_head[count].data = 'G';
    p = List_head[count].firstarc = InSertArcNode('D');
    p = p->nextarc = InSertArcNode('E');
    p = p->nextarc = InSertArcNode('H');
    count++;
    List_head[count].data = 'H';
    p = List_head[count].firstarc = InSertArcNode('G');
    p = p->nextarc = InSertArcNode('I');
    count++;
    List_head[count].data = 'I';
    p = List_head[count].firstarc = InSertArcNode('H');

    return List_head;
}

void AdjMatrix(char arc[][9])
{
    for (int i = 0; i < 9; i++)   //初始化邻接矩阵
        for (int j = 0; j < 9; j++)
        {
            arc[i][j] = 0;
        }
    arc[0][1] = arc[0][3] = arc[0][4] = 1;
    arc[1][0] = arc[1][2] = arc[1][4] = 1;
    arc[2][1] = arc[2][5] = 1;
    arc[3][0] = arc[3][6] = 1;
    arc[4][0] = arc[4][1] = arc[4][6] = 1;
    arc[5][2] = 1;
    arc[6][3] = arc[6][4] = arc[6][7] = 1;
    arc[7][6] = arc[7][8] = 1;
    arc[8][7] = 1;
}
void DFS_matrix(char G[][9],int i,bool *visited)  //深度优先搜索与结点i相通的所有节点
{
    visited[i] = true;  //顶点i被访问，标志置为true
    for (int j = 0; j < 9; j++)
    {
        if (!visited[j] && G[i][j]==1)
        {
            cout << vextex[j] << ",";
            DFS_matrix(G, j, visited); //递归
        }
    }
}
void DFS_AdjMatrix(char G[][9])  //深度优先搜索_邻近矩阵存储
{
```

```
    bool visited[9] = { 0 };  //初始化访问标志数组
    for (int i = 0; i < 9; i++) //检测是否所有节点都被访问过
    {
        if (!visited[i])//顶点i未被访问过，结点i进行深度优先搜索
        {
            cout << vextex[i]<<",";
            DFS_matrix(G, i, visited);//深度优先搜索顶点i
        }
    }
}
void DFS_list(VertexNode * GRAPH, int i, bool *visited)
{
    visited[i] = true;  //顶点i被访问，标志置为true
    cout << vextex[i] << ",";
    ArcNode * p = GRAPH[i].firstarc; //找到第一个邻接链表结点
    while (p!=NULL)
    {
        int temp = p->data - 'A'; //计算节点的位置
        if (!visited[temp]) //检测邻接顶点是否被访问过
            DFS_list(GRAPH, temp, visited); //深度优先搜索结点temp
        p = p->nextarc;//回溯到下一个邻接顶点
    }
}
void DFS_AdjList(VertexNode * GRAPH)  //深度优先搜索--邻接链表存储
{
    bool visited[9] = { 0 }; //初始化访问标志数组
    for (int i = 0; i < 9; i++)//检测是否所有节点都被访问过
    {
        if (!visited[i])
        {
            DFS_list(GRAPH, i, visited);//深度优先搜索顶点i
        }
    }
}
void BFS_list(VertexNode *GRAPH, int i, bool *visited, queue<char> &Q)
{
    cout << Q.front() << ",";
    Q.pop(); //出队列
    /*访问到顶点i的所有邻接点*/
    ArcNode *p = GRAPH[i].firstarc; //第一个邻结点
    while ( p!=NULL )  //依次访问顶点i的邻接点
    {
        /*(p->data - 'A')代表顶点的序号*/
        if (*(visited + (p->data - 'A')) == 0)//检测邻接点是否被访问过
        {
            *(visited + (p->data - 'A')) = true;//访问标志置1
            Q.push(p->data); //邻接点加入优先队列
        }
        p = p->nextarc;
    }
    if (!Q.empty()) //递归遍历队列里的顶点
    {
        BFS_list(GRAPH, Q.front() - 'A', visited, Q);
    }
}
void BFS_AdjList(VertexNode *GRAPH)//广度优先搜索顶点i--邻接表存储
{
    bool visited[9] = { 0 }; //访问标志初始化
    queue<char> Q; //优先队列
    for (int i = 0; i < 9; i++)
    {
        if (!visited[i])
        {
            visited[i] = true;  //访问标志置1
            Q.push(vextex[i]);   //进入顶点队列
            BFS_list(GRAPH, i, visited, Q); //广度优先搜索顶点i
        }
    }
}
void BFS_KLevel(VertexNode * GRAPH, int i,int k) //计算距离顶点i为k的所有顶点
{
```

```
    if (k==0) //如果k=0，输出此顶点
    {
        cout << GRAPH[i].data << endl;
        return;
    }
    queue<char> Q1; //已访问顶点
    queue<unsigned int> Q2; //已访问顶点与顶点i的距离
    bool visited[9] = { 0 };//访问标志
    visited[i] = true;   //顶点i置1
    Q1.push(vextex[i]); //进入队列
    Q2.push(0);  //距离队列

    while (!Q1.empty())
    {
        int index = Q1.front() - 'A';  //顶点的序号

        ArcNode *p = GRAPH[index].firstarc;//第一个邻接点
        int level = Q2.front();
        while (p!=NULL)
        {
            if (*(visited+(p->data-'A')) == 0)  //结点没有被访问过
            {
                *(visited + (p->data - 'A')) =true;//访问标志置1
                Q1.push(p->data);
                Q2.push(level + 1);  //距离+1
                if (level + 1 == k)  //判断距离
                {
                    cout << p->data << ",";
                }
            }
            p = p->nextarc;
        }
        Q1.pop();
        Q2.pop();
    }

}

int main()
{

    VertexNode * GRAPH = AdjList();  //邻接链表
    char G[9][9] = { 0 };
    AdjMatrix(G);   //邻接矩阵
    DFS_AdjMatrix(G); //DFS--邻接矩阵
    cout <<"  DFS--邻接矩阵"<< endl;
    DFS_AdjList(GRAPH); //DFS--邻接链表
    cout << "  DFS--邻接链表" << endl;
    BFS_AdjList(GRAPH); //BFS--邻接链表
    cout << "  BFS--邻接链表" << endl;
    cout << "------------" << endl;
    BFS_KLevel(GRAPH,1,2);//计算距离顶点B为2的顶点
    cout << " 距离顶点B为2的顶点" << endl;
    return 0;
}
```

Reference :

数据结构--耿国华

算法导论 - 第三版

全文完