树/图类算法

> 数和图也是一种代码中的数据结构，用来描述一些特定场景下的特定逻辑，其中数的使用更为广泛，数的经典算法也更为多样，面试中经常会考察对于树形数据结构的算法

树形数据结构

树的定义

- 根(root): 树中的一个特定元素称为根
- 子树(subtree): 除了根之外，树中的其他元素都称为该树根的子树
- 结点(node):树中的元素
- 双亲(parent): 一个结点有子树，子树的根为该结点的孩子
- 度(degree): 结点拥有的子树的个数
- 叶子(leaf): 度为0的结点
- 层次(level): 根结点的层次为1，其余结点的层次等于结点双亲的层次+1

二叉树和森林

> 二叉树是每个节点最多有两个子树的树结构。通常子树被称作“左子树”和“右子树”。

> 一棵深度为k，且有2^k-1个节点的二叉树，称为满二叉树。这种树的特点是每一层上的结点数都是最大节点数。而在一棵二叉树中，除最后一层外，若其余层都是满的，并且或者最后一层是满的，或者是在右边缺少连续若干节点，则此二叉树被称为完全二叉树，具有n个节点的完全二叉树的深度为floor(log2n)+1。深度为k的完全二叉树，至少有2k-1个叶子子节点，至多有2k-1个节点。

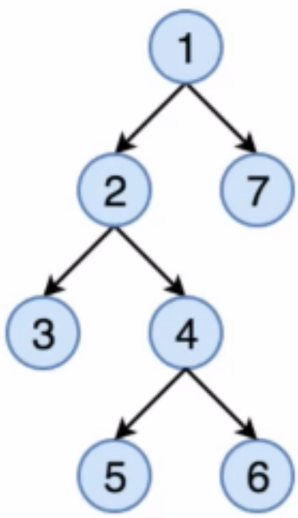
二叉树的性质

- 在二叉树的第i层上至多有2^(i-1)个结点（i>=1）.
- 深度为k的二叉树最多有2^k-1个结点（k>=1）
- 一棵二叉树的叶子结点数为nO, 度为2的结点数为n2，则nO = n2+1。
- 具有n个结点的完全二叉树的深度为Math.floor(log2n)+1。
- 如果对一棵有n个结点的完全二叉树(其深度为floor(log2n)+1) 的结点按层序编号，则对任一结点i(1<=i<=n)有：
    - 如果i = 1,则结点i是二叉树的根，无双亲；如果i>=1,则其双亲parent(i)是结点floor(i/2)。
    - 如果2i > n,则结点i没有左子树；否则其左孩子lchild(i)是结点2i。
    - 如果2i+1>n, 则结点i无右孩子；否则其右孩子rchild(i)是结点2i+1；

二叉树的相关常见算法

在js中，我们可以通过这种结构定义一个数结点：

```
function TreeNode(val) {
    this.val = val;
    this.left = this.right = null;
}
```



深度优先遍历DFS

> 在深度遍历中，每个结点都需要遍历三次，那么我们可以在不同的阶段，执行不同的操作，根据我们处理结点位置的不同，我们分为前序，中序，后序遍历。

前序遍历

在DFS第一次遇到节点时即进行处理。

```
var preorderTraversal = (node, result = []) => {
    if(node) {
        // 先根节点
        result.push(node.val)
        // 然后遍历左子树
        preorderTravereal(node.left,result)
        // 然后遍历右子树
        preorderTravereal(node.right,result)
    }
    return result
}
```

中序遍历

在DFS第二次遇到节点时进行处理。

```
var inorderTraversal = (node,result = []) => {
    if(node) {
        // 遍历左子树
        inorderTraversal(node.left,result)
        // 根节点
        result.push(node.val)
        // 再遍历右子树
        inorderTraversal(node.right,result)
    }
}
```

后序遍历

在DFS第三次遇到节点时进行处理。

```
var inorderTraversal = (node,result = []) => {
    if(node) {
        // 遍历左子树
        inorderTraversal(node.left,result)
        // 遍历右子树
        inorderTraversal(node.right,result)
        // 根节点
        result.push(node.val)
    }
}
```

广度优先遍历BFS

广度优先就简单一些，水平进行树的遍历，每一层横向处理，最终遍历到结尾。 图中的二叉树，处理的结果即为1，2，7，3，4，5，6

```
var levelOrder = function(root) {
    const ret = [];
    if(!root) return ret;

    const q = [];
    q.push(root)
    while(q.length !== 0) {
        const currentLevelSize = q.length;
        ret.push([]);
        for(let i = 1;i<= currentLevelSize;++i){
            const node = q.shift();
            ret[ret.length-1].push(node.val);
            if(node.left) q.push(node.left)
            if(node.right) q.push(node.right)
        }
    }
    return ret;
}
```

计算数的高度

```
var maxDepth = function(root) {
    if(!root) return 0;
    if(!root.left && !root.right) return 1;
    return Math.max(maxDepth(root.left), maxDepth(root.right)) + 1;
}
```

左子叶之和

```
var sumOfLeftLeaves = function(root) {
    var val = 0;
    if(!root) return 0;
    if(root.left && !root.left.left && !root.left.right) {
        val = root.left.val;
    }
    return val + sumOfLeftLeaves(root.left) +
sumOfLeftLeaves(root.right)
}
```

反转二叉树

```
var inverTree = function(root) {
    if(!root || (!root.left && !root.right)) return root;
    root.left = inverTree(root.left);
    root.right = inverTree(root.right);
    return exchangeChildNode(root)
}

var exchangeChildNode = function(node) {
    var temp = new TreeNode();
    temp = node.left;
    node.left = node.right
    node.right = temp;
    return node;
}
```

图类型数据结构

图的定义

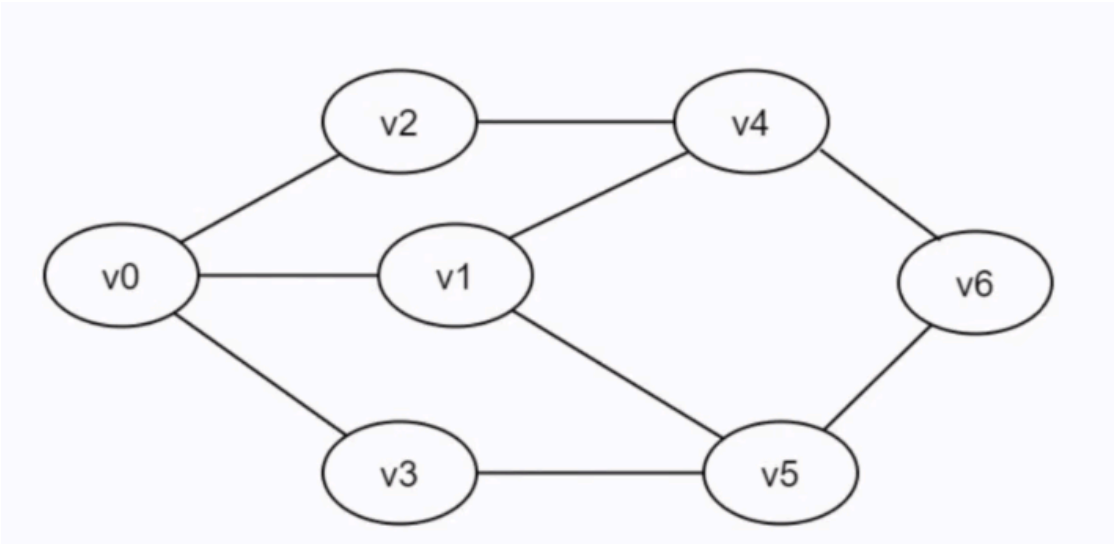
图的遍历算法DFS与BFS

BFS和DFS算法解析图的遍历的定义：从图的某个定点触发访问遍图中所有的定点，且每个顶点仅被访问一次

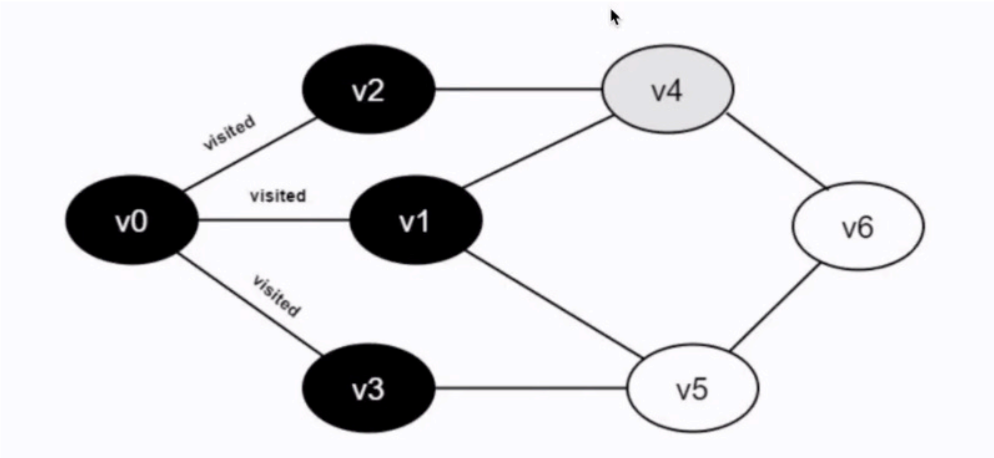
BFS

广度优先搜索类似于树的层次遍历过程。它需要借助一个队列来实现。要想遍历从v0到v6的每一个定点，我们可以设v0为第一层，v1、v2、v3为第三层，v4、v5为第三层，v6为第四层，再逐个遍历每一层的每个定点

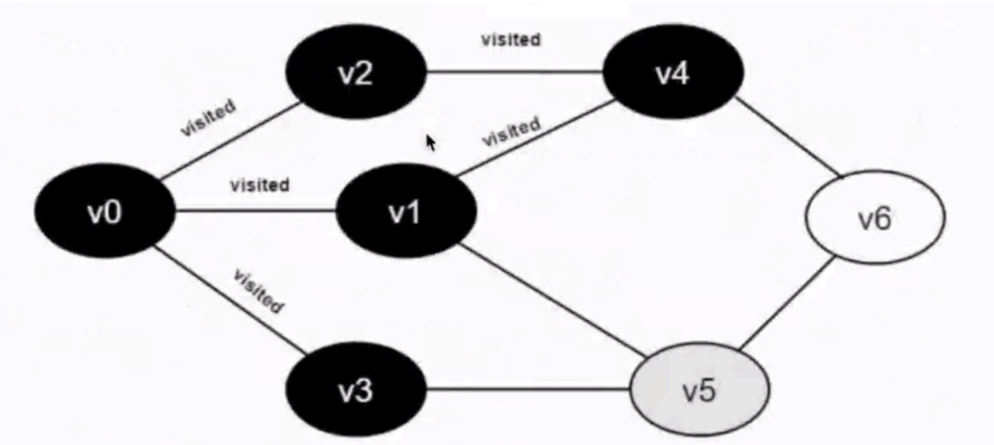
具体步骤



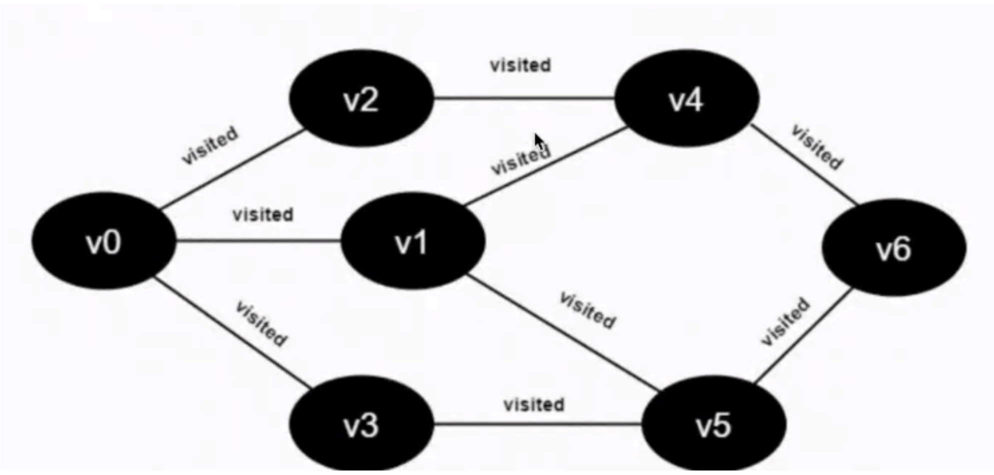
- 准备工作：创建一个visited数组，用来记录已被访问过的定点；创建一个队列，用来存放每一层的定点；初始化图G
- 从图中的v0开始访问，将对应的visited[v0]数组的值设置为true，同时将v0入队列。
- 只要队列不空，则重复如下操作
  - 队头定点u出队
  - 依次检查u的所有邻接定点w，若visited[w]的值为false，则访问w，并将visited[w]置为true，同时将w入队



- v0的全部邻接点均被访问完毕，将队头元素v2出队，开始访问v2所有邻接点。开始访问v2邻接点v0，判断visited[0],因为其值为1，不进行访问。继续访问v2邻接点v4，判断visited[4]，因为其值为0，访问v4



- v2的全部邻接点均被访问完毕。将队头元素v1出队，开始访问v1的所有邻接点。开始访问v1邻接点v0，因为visited[0]值为0，访问v5



- v5的全部邻接点均被访问完毕，将队头元素v6出队，开始访问v6的所有邻接点。开始访问v6邻接点v4，因为visited[4]的值为1，不进行访问。继续

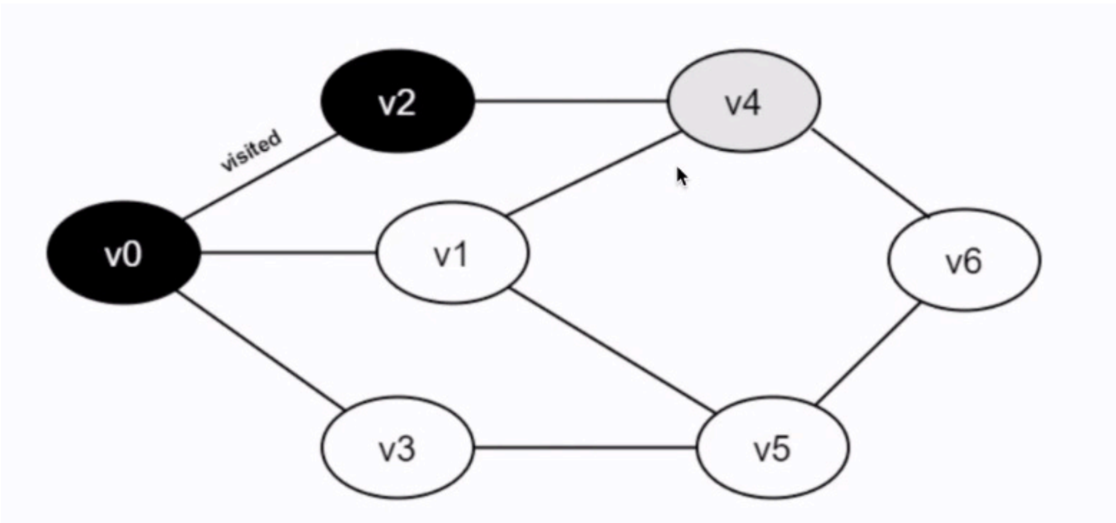
访问v6邻接点v5，因为visited[5]的值为1，不进行访问。

DFS

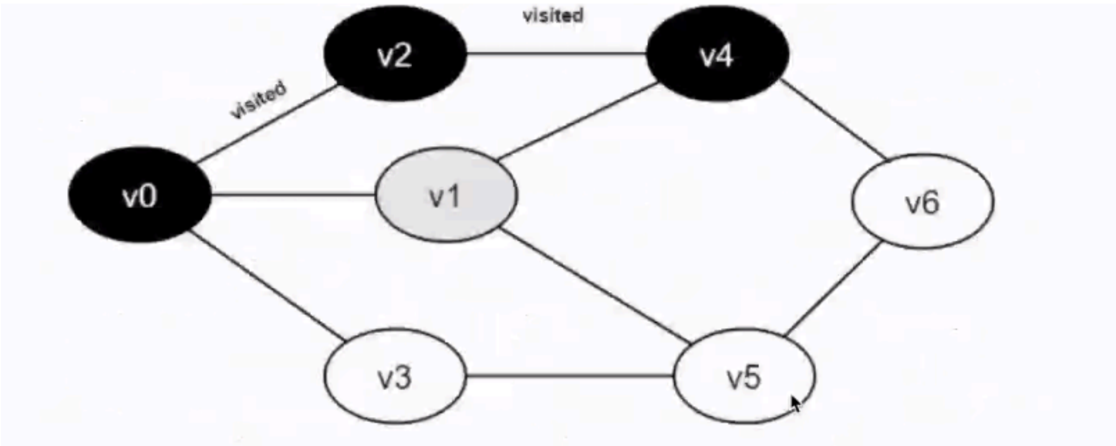
DFS遍历类似于树的先序遍历，是树的先序遍历的推广。

具体步骤：准备工作：创建一个visited数组，用于记录所有被访问过的定点。

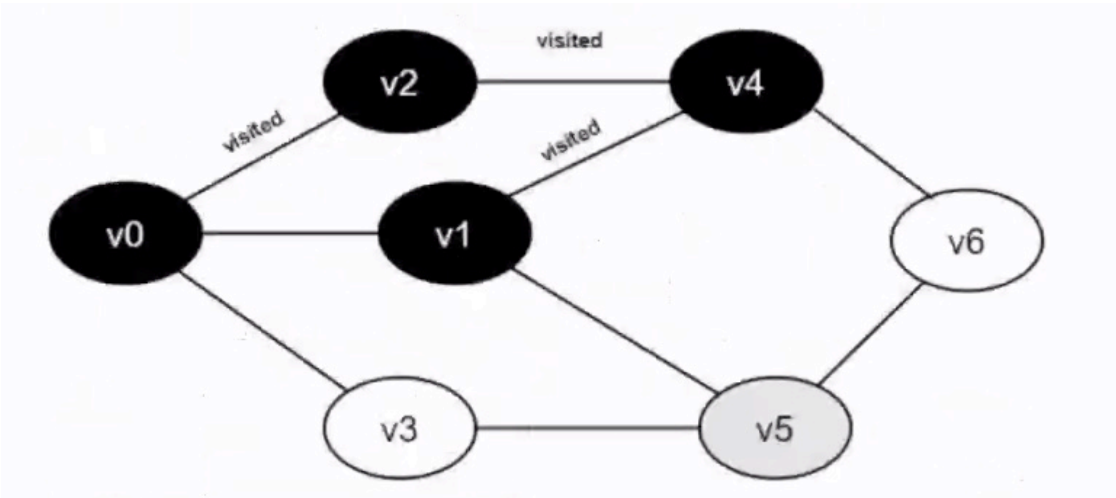
1. 从图中v0触发，访问v0。
2. 找出v0的第一个未被访问的邻接点。访问该顶点。以该顶点为新顶点，重复此步骤，直至刚访问过的定点没有未被访问的邻接点为止。
3. 返回前一个访问过的仍有未被访问邻接的定点，继续访问该顶点的下一个未被访问的邻接点。
4. 重复2，3步骤，直至所有顶点均被访问，搜索结束



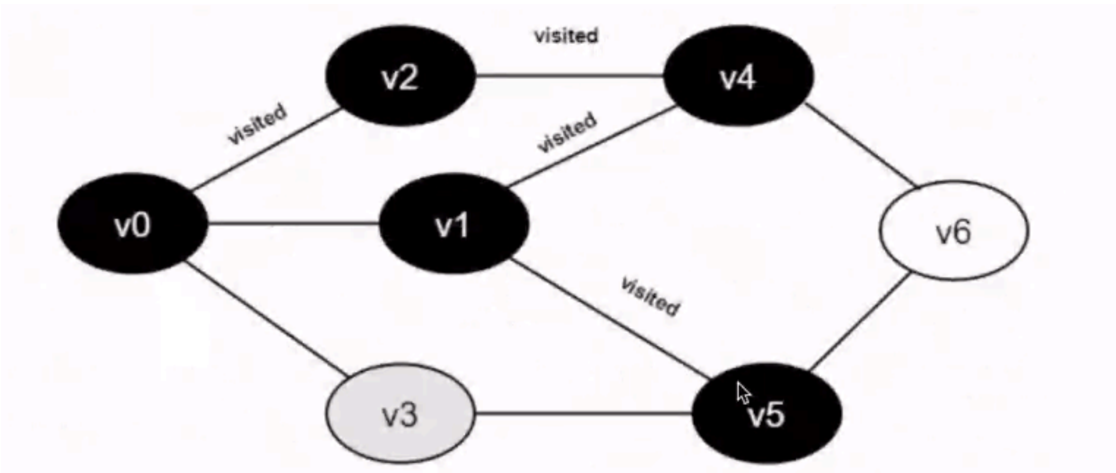
访问v2的邻接点v0,判断visited[0],其值为1.不访问。继续访问v2的邻接点v4，判断visited[4],其值为0，访问v4



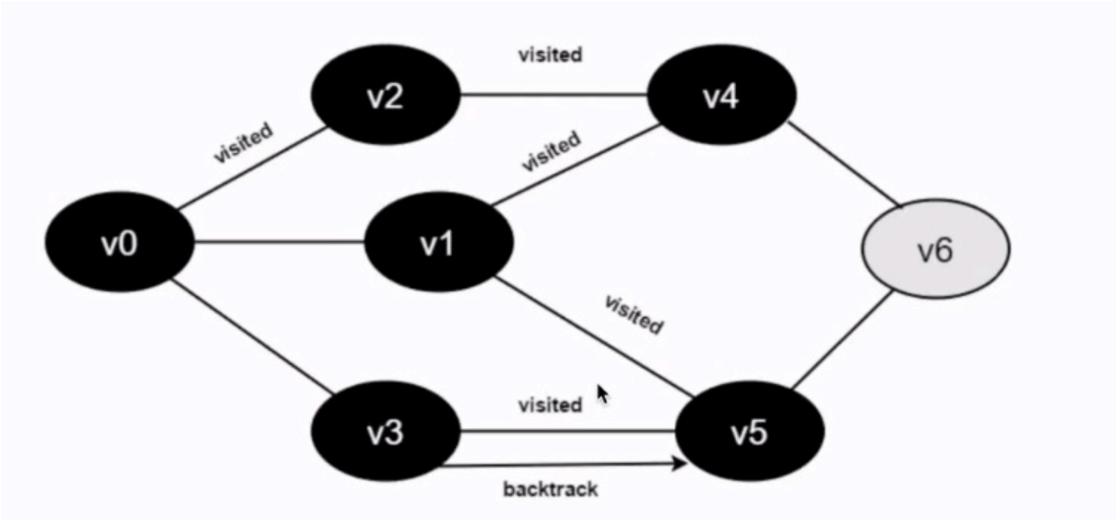
访问v4的邻接点v1，判断visited[1],其值为0，访问v1



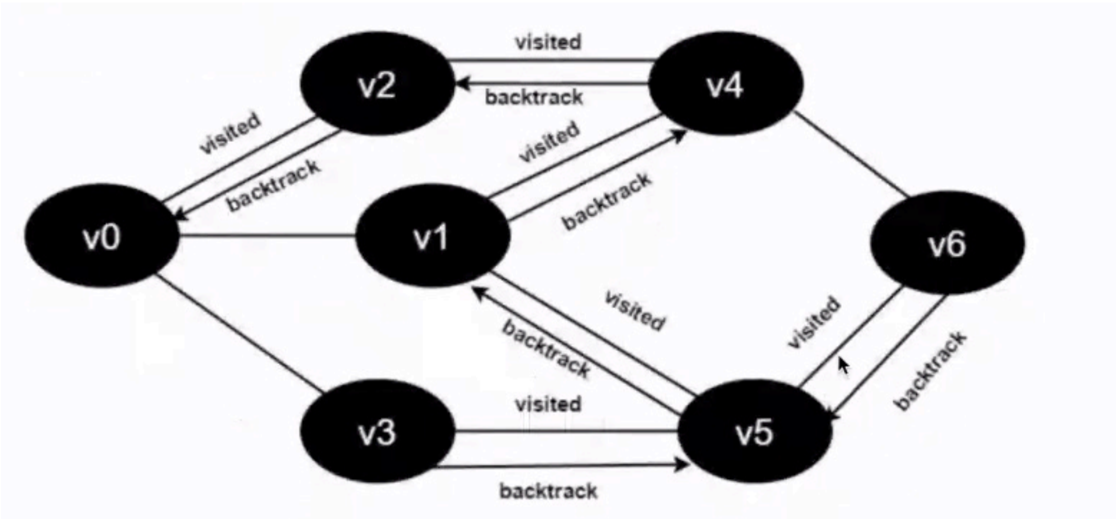
访问v1的邻接点v0，判断visited[0],其值为1，不访问。继续访问v1的邻接点v4，判断visited[4],其值为1，不访问。继续访问v1的邻接点v5，判断visited[5]，其值为0，不访问v5.



访问v5的邻接点v1，判断visited[1],其值为1，不访问。继续访问v5的邻接点v3，判断visited[3],其值为0，访问v3.



访问v3的邻接点v0，判断visited[0],其值为1，不访问。继续访问v3的邻接点v5，判断visited[5]，其值为1，不访问。v3所有的邻接点均已被访问，回溯到上一个顶点v5，遍历v5所有邻接点v6，判断visited[6]，其值为0，访问v6.



v5所有邻接点均已被访问，回溯到上一个顶点v1.v1所有邻接点均已被访问，回溯到上一个顶点v4，遍历v4剩余邻接点v6.v4所有邻接点均已被访问，回溯到上一个顶点v2。v2所有邻接点均已被访问，回溯到其上一个顶点v1，遍历v1剩余邻接点v3。v1所有邻接点都已被访问，搜索结束