

26 | 图：深度优先遍历（DFS）与广度优先遍历（BFS）

2023-04-12 王健伟 来自北京

《快速上手C++数据结构与算法》



你好，我是王健伟。

上节课我们讲述了邻接矩阵、邻接表、十字链表、邻接多重表、边集数组共 5 种数据结构，解决了对图进行存储的问题。接下来的问题，就是对图进行遍历了。

所谓图的遍历，就是指从图中任意一个顶点出发访遍图中其余**顶点**，且使每个顶点**都被访问**且**只被访问一次**。

shikey.com转载分享

图的遍历比树的遍历复杂很多，因为图中可能存在回路（环），因此遍历过程中需要标记每个已访问过的顶点以避免同一个顶点被访问多次。不过不用有什么负担，我会对照着代码给你讲透，这样既能理解，又方便你之后自己编写代码。

图的遍历通常分为两种：深度优先遍历、广度优先遍历。本节我们以**邻接表**作为图的存储结构来讲解这两种遍历。

深度优先遍历

深度优先遍历也称为深度优先搜索，英文名是 Depth First Search (DFS)。这种遍历其实是一个递归的过程，沿着每一个分支路径进行深入访问，很像一棵树的前序遍历。

你可以想象成进入到一个迷宫中，迷宫中有很多岔路，选择任意一条路走进去，一直到发现走不通的时候退回到上一个岔路口并重新选择一条路走进去，直到走遍所有关键节点。

针对前面用邻接表这种存储方式实现的图，我们可以在其中继续增加代码来实现对图的深度优先遍历。你可以看下相关的代码。

 复制代码

```
1 //深度优先遍历
2 void DepthFirstSearch(const T& tmpv) //tmpv代表从该顶点出发开始遍历，即tmpv是开始遍历的顶点
3 {
4     bool vVArray[MaxVertices_size]; //顶点是否被访问过的标记，false没有被访问过，true被访问过
5     for (int i = 0; i < MaxVertices_size; ++i) //开始时所有顶点都没有被访问过
6     {
7         vVArray[i] = false;
8     } //end for
9
10    int idx = GetVertexIdx(tmpv);
11    DepthFirstSearch(idx, vVArray);
12 }
13 void DepthFirstSearch(int idx, bool vVArray[])
14 {
15     cout << m_VertexArray[idx].data <<"-->"; //输出顶点数据（顶点值）
16     vVArray[idx] = true; //标记该顶点已经被访问过
17
18     int idx2 = GetFirstNeighbor(idx); //获取第一个邻接顶点的下标，B的第一个邻接顶点是F
19     while (idx2 != -1)
20     {
21         // (1) 继续沿着深度访问节点
22         if (vVArray[idx2] == false)
23         {
24             //没访问过，则进行递归访问
25             DepthFirstSearch(idx2, vVArray);
26         }
27
28         // (2) 找其他的邻接顶点（广度方向走）
29         idx2 = GetNextNeighbor(idx, idx2); //获取某个顶点（下标为idx）的邻接顶点（下标为idx2）
30     } //end while
31     return;
```

之后，修改 main 主函数中的代码，将一些删除图中顶点、边的代码注释掉并增加如下代码以测试图的深度优先遍历。

[复制代码](#)

```
1 gm2.DepthFirstSearch('B');
2 cout <<"nullptr"<< endl;
```

上面的代码表示从顶点 B 开始进行图的深度优先遍历，整个代码的执行就会是这样的。

```
0 A: -->3-->2-->1-->nullptr
1 B: -->5-->4-->0-->nullptr
2 C: -->5-->0-->nullptr
3 D: -->5-->0-->nullptr
4 E: -->1-->nullptr
5 F: -->3-->2-->1-->nullptr
图中有顶点6个，边7条！
-----
B-->F-->D-->A-->C-->E-->nullptr
```

结果中，最后一行就是深度优先遍历的结果。从结果中不难看到深度优先遍历的顺序，我们尝试把它梳理出来。

顶点 B（下标 1）作为遍历的开始顶点，肯定会最先被访问。

取得顶点 B 的第一个邻接顶点 F（下标 5），因该顶点没被访问过所以进行访问。

取得顶点 F 的第一个邻接顶点 D（下标 3），因该顶点没被访问过所以进行访问。

取得顶点 D 的第一个邻接顶点 F（下标 5），因为该顶点已被访问过，所以取得顶点 D 的下一个（第二个）邻接顶点 A（下标 0），因该顶点没被访问过所以进行访问。

取得顶点 A 的第一个邻接顶点 D（下标 3），因为该顶点已被访问过，所以取得顶点 A 的下一个（第二个）邻接顶点 C（下标 2），因该顶点没被访问过所以进行访问。

取得顶点 C 的第一个邻接顶点 F（下标 5），因为该顶点已被访问过，所以取得顶点 C 的下一个（第二个）邻接顶点 A（下标 0），因为该顶点已被访问过，所以取得顶点 C 的下一个（第三个）邻接顶点，但此时顶点 C 已经没有下一个邻接顶点了，因此返回到顶点 A 的处理流程。

取得顶点 A 的下一个（第三个）邻接顶点 B（下标 1），因为该顶点已被访问过，所以取得顶点 A 的下一个（第四个）邻接顶点，但此时顶点 A 已经没有下一个邻接顶点了，因此返回到顶点 D 的处理流程。

取得顶点 D 的下一个（第三个）邻接顶点，但此时顶点 D 已经没有下一个邻接顶点，因此返回到顶点 F 的处理流程。

取得顶点 F 的下一个（第二个）邻接顶点 C（下标 2），因为该顶点已经被访问过，所以取得顶点 F 的下一个（第三个）邻接顶点 B（下标 1），因为该顶点已经被访问过，所以取得顶点 F 的下一个（第四个）邻接顶点，但此时顶点 F 已经没有下一个邻接顶点了，因此返回到顶点 B 的处理流程。

取得顶点 B 的下一个邻接顶点（第二个）邻接点 E（下标 4），因为该顶点没被访问过所以进行访问。

取得顶点 E 的第一个邻接顶点 B（下标 1），但因为该顶点已被访问过，所以取得顶点 E 的下一个邻接顶点，但此时顶点 E 已经没有下一个邻接顶点了，因此返回到顶点 B 的处理流程。

取得顶点 B 的下一个邻接顶点（第三个）邻接顶点 A（下标 0），但因为该顶点已被访问过，所以取得顶点 B 的下一个邻接顶点，但此时顶点 B 已经没有下一个邻接顶点了，因此整个遍历过程结束。

整个深度优先遍历的顺序示意图如图 1 所示：

shikey.com转载分享

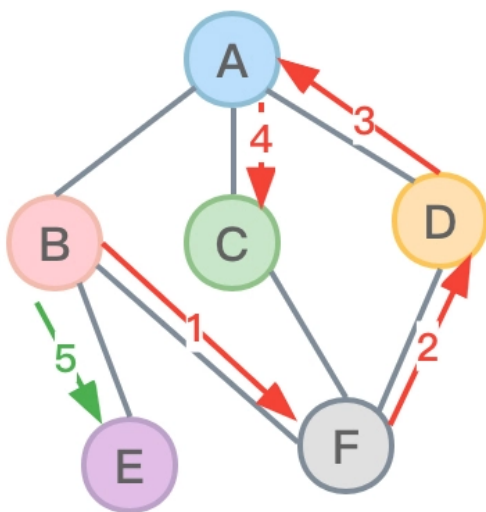


图1 一个图采用邻接表存储时以顶点B作为起点进行深度优先遍历的顶点遍历顺序示意图

在图 1 中，一共有 6 个顶点，并且通过 5（顶点数 -1，即 $6-1$ ）条边（图中标记了数字的边）遍历了所有这 6 个顶点，如果仅保留图中这 5 条边，则得到了一棵树（没有回路存在），如图 2 所示，这个树就叫做该图所对应的深度优先生成树：

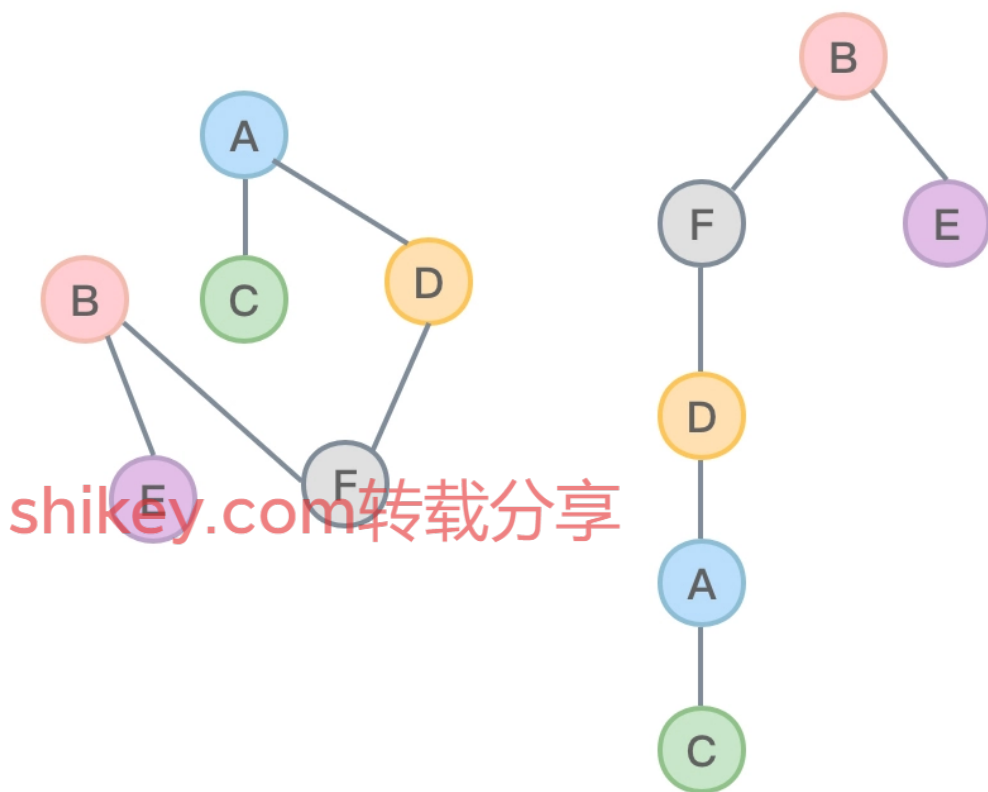


图2 一个图采用邻接表存储时对应的深度优先生成树（两幅图画的是同一棵树）


图 2 这棵深度优先生成树是对原图进行深度优先遍历的过程得到的。显然，遍历的顺序决定了深度优先生成树的样子。而在以邻接表作为图的存储结构时，由于边节点结构的插入方式可以采用头插或者尾插方式（前面范例采用的是头插方式），所以同一个图的邻接表存储方式并不唯一。也就是说，用邻接表存储图所得到的深度优先生成树并不唯一。另外，若用邻接矩阵存储图，那么因为得到的邻接矩阵是唯一的，所以得到的深度优先生成树应该也是唯一的，你可以自行尝试。

不难看到，只要图是连通的，就可以以任意顶点为起始点进行遍历。

如果是非连通图，就需要对它的连通分量分别进行深度优先遍历，直至图中所有顶点都被访问到为止。

最简单的判断是否是连通图的方法是进行一次深度优先遍历后再扫描一次代码中的 `vVArray` 数组（标记图中顶点是否被访问过的数组），如果在该数组中发现其对应位置的顶点真实存在但却没被访问过，则可以从这个顶点出发再次调用 `DepthFirstSearch` 成员函数。如此反复，最终即可对非连通图的所有连通分量实现完整遍历。要做到这些，只需要修改 `DepthFirstSearch` 成员函数即可。

我们看下修改后的完整代码。

 复制代码

```
1 //深度优先遍历
2 void DepthFirstSearch(const T& tmpv) //tmpv代表从该顶点出发开始遍历，即tmpv是开始遍历的顶点
3 {
4     bool vVArray[MaxVertices_size]; //顶点是否被访问过的标记，false没有被访问过，true被访问过
5     for (int i = 0; i < MaxVertices_size; ++i) //开始时所有顶点都没有被访问过
6     {
7         vVArray[i] = false;
8     } //end for
9
10    int idx = GetVertexIdx(tmpv);
11    DepthFirstSearch(idx, vVArray);
12
13    //如果是非连通图，则继续遍历其他子图
14    bool iffndnovisited; //是否找到了没被访问的顶点
15    int idxnovisited; //没被访问的顶点的下标
16    lblloop:
17    iffndnovisited = false;
```



```

18     idxnovisited = -1;
19     for (int i = 0; i < m_numVertices; ++i)
20     {
21         if (vVArray[i] == false)
22         {
23             iffindhovisited = true; //标记找到了没被访问的顶点
24             idxnovisited = i; //记录没被访问的顶点的下标
25             break;
26         }
27     }
28     if (iffindhovisited == true) //找到了没被访问的顶点
29     {
30         DepthFirstSearch(idxnovisited, vVArray);
31         goto lblloop;
32     }
33 }

```

如果一个图是非连通的，那么对他的连通分量分别进行深度优先遍历就会得到多棵深度优先生成树，合在一起组成深度优先生成森林。

广度优先遍历

广度优先遍历也称为广度优先搜索，英文名为 Breadth First Search（BFS）。这种遍历很像一棵二叉树的层序遍历或者说树的广度优先遍历。

二叉树的层序遍历是从树的根节点开始，按照从上到下从左到右的顺序对节点逐个遍历。图的层序遍历也类似。在图 1 中，从顶点 B 出发进行广度优先遍历的步骤应该是这样的。

访问顶点 B。


通过顶点 B 可以找到与之相邻且未曾访问过的顶点 E、F、A，即访问顶点 E、F、A。

从顶点 E、F、A 出发，再找到与这些顶点相邻且未曾访问过的顶点 C、D。

不难看到，图的广度优先遍历是一种从近到远的搜索策略——先查找离起始顶点最近的顶点，依次向外搜索。

在实现二叉树的层序遍历代码时，借助了链式队列来完成，在实现图的广度优先遍历时也类似。把以往讲解的链式队列相关的代码复制到本项目当前的 MyProject.cpp 文件中来。针对

前面用邻接表这种存储方式实现的图，我们可以在其中继续增加代码，实现对图的广度优先遍历。

 复制代码

```
1 //广度优先遍历
2 void BreadthFirstSearch(const T& tmpv) //tmpv代表从该顶点出发开始遍历，即tmpv是开始遍历
3 {
4     bool vVArray[MaxVertices_size]; //顶点是否被访问过的标记，false没有被访问过，true被访问
5     for (int i = 0; i < MaxVertices_size; ++i) //开始时所有顶点都没有被访问过
6     {
7         vVArray[i] = false;
8     } //end for
9
10    int idx = GetVertexIdx(tmpv);
11    cout << m_VertexArray[idx].data <<"-->";
12    vVArray[idx] = true; //标记该顶点已经被访问过
13
14    LinkQueue<int> lnobj; //借助队列实现遍历
15    lnobj.Enqueue(idx); //先把起始顶点下标入队
16    while (!lnobj.IsEmpty()) //循环判断队列是否为空
17    {
18        lnobj.DeQueue(idx); //出队列
19
20        int idx2 = GetFirstNeighbor(idx);
21        while (idx2 != -1)
22        {
23            if (vVArray[idx2] == false)
24            {
25                //没访问过
26                cout << m_VertexArray[idx2].data <<"-->";
27                vVArray[idx2] = true; //标记该顶点已经被访问过
28                lnobj.Enqueue(idx2); //入队
29            }
30            idx2 = GetNextNeighbor(idx, idx2); //获取某个顶点（下标为idx）的邻接顶点（下标为
31        } //end while
32    } //end while
33 }
```

shiskey.com转载分享

在 main 主函数中，再加入代码测试广度优先遍历。


```
1 gm2.BreadthFirstSearch('B');  
2 cout <<"nullptr"<< endl;
```

新增代码的执行结果如下：

```
B--->F--->E--->A--->D--->C--->nullptr
```

从结果中不难看到，广度优先遍历的顺序是这样的。

1. 顶点 B（下标 1）作遍历的开始顶点，肯定会最先被访问。然后把顶点 B 入队。
2. 从队列中取出队头元素顶点 B，针对顶点 B，有这样一系列的操作。

取得顶点 B 的第一个邻接顶点 F（下标 5），因为该顶点没被访问过所以进行访问。然后把顶点 F 入队。

取得顶点 B 的邻接顶点 F 的下一个邻接顶点 E（下标 4），因为该顶点没被访问过所以进行访问。然后把顶点 E 入队。

取得顶点 B 的邻接顶点 E 的下一个邻接顶点 A（下标 0），因为该顶点没被访问过所以进行访问。然后把顶点 A 入队。

取得顶点 B 的邻接顶点 A 的下一个邻接顶点，但此时顶点 B 已经没有下一个邻接顶点，因此重新到队列中取队头元素。

3. 从队列中取出队头元素 F，针对顶点 F，会有什么样的操作呢？

shikey.com 转载分享

取得顶点 F 的第一个邻接顶点 D（下标 3），因为该顶点没被访问过所以进行访问。然后把顶点 D 入队。

取得顶点 F 的邻接顶点 D 的下一个邻接顶点 C（下标 2），因为该顶点没被访问过所以进行访问。然后把顶点 C 入队。

取得顶点 F 的邻接顶点 C 的下一个邻接顶点 B（下标 1），因为该顶点已被访问过，所以取得顶点 F 的邻接顶点 B 的下一个邻接顶点，但此时顶点 F 已经没有下一个邻接顶点，因

此重新到队列中取队头元素。

4. 之后，从队列中取出队头元素 E，针对顶点 E，取得顶点 E 的第一个邻接顶点 B（下标 1），因为该顶点已被访问过，所以取得顶点 E 的邻接顶点 B 的下一个邻接顶点，但此时顶点 E 已经没有下一个邻接顶点，因此重新到队列中取队头元素。
5. 从队列中取出队头元素 A，针对顶点 A，取得顶点 A 的第一个邻接顶点 D（下标 3），因为该顶点已被访问过，所以取得顶点 A 的邻接顶点 D 的下一个邻接顶点 C（下标 2），因为该顶点已被访问过，所以取得顶点 A 的邻接顶点 C 的下一个邻接顶点 B（下标 1），因为该顶点已被访问过，所以取得顶点 A 的邻接顶点 B 的下一个邻接顶点，但此时顶点 A 已经没有下一个邻接顶点，因此重新到队列中取队头元素。
6. 从队列中取出队头元素 D，针对顶点 D，取得顶点 D 的第一个邻接顶点 F（下标 5），因为该顶点已被访问过，所以取得顶点 D 的邻接顶点 F 的下一个邻接顶点 A（下标 0），因为该顶点已被访问过，所以取得顶点 D 的邻接顶点 A 的下一个邻接顶点，但此时顶点 D 已经没有下一个邻接顶点，因此重新到队列中取队头元素。
7. 从队列中取出队头元素 C，针对顶点 C，取得顶点 C 的第一个邻接顶点 F（下标 5），因为该顶点已被访问过，所以取得顶点 C 的邻接顶点 F 的下一个邻接顶点 A（下标 0），因为该顶点已被访问过，所以取得顶点 C 的邻接顶点 A 的下一个邻接顶点，但此时顶点 C 已经没有下一个邻接顶点，因此重新到队列中取队头元素。
8. 从队列中取出队头元素，但因为此时队列已经为空，因此整个遍历过程结束。

整个广度优先遍历的顺序示意图如图 3 所示：

shikey.com转载分享

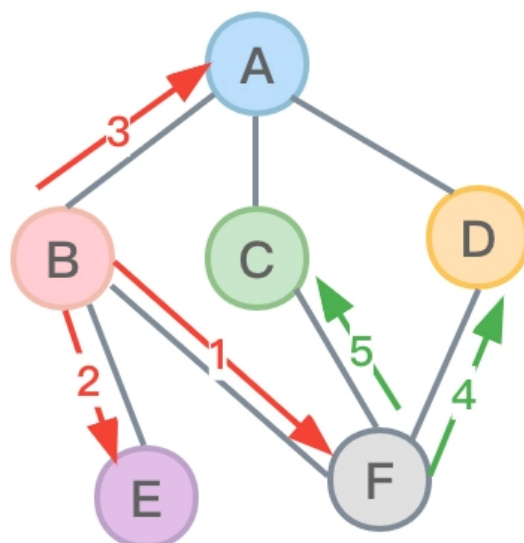


图3 一个图采用邻接表存储时以顶点B作为起点进行广度优先遍历的顶点遍历顺序示意图

在图 3 中，一共有 6 个顶点，并且通过 5（顶点数 -1，即 6-1）条边（图中标记了数字的边）遍历了所有这 6 个顶点，如果仅保留图中这 5 条边，则得到了一棵树（没有回路存在），如图 4 所示，这个树就叫做该图所对应的**广度优先生成树**。

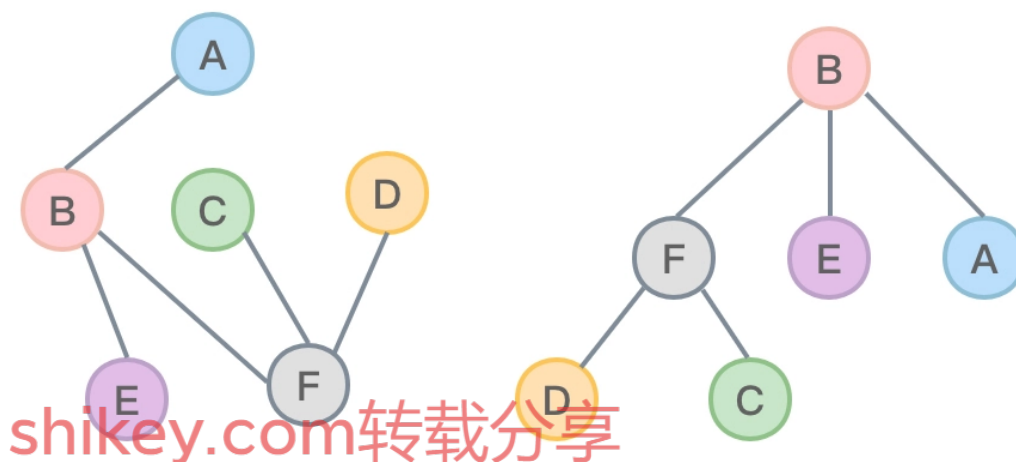


图4 一个图采用邻接表存储时对应的广度优先生成树（两幅图画的是同一棵树）

图 4 这棵广度优先生成树是对原图进行广度优先遍历的过程而得到的。显然，遍历顺序决定了广度优先生成树的样子。而在以邻接表作为图的存储结构时，由于边节点结构的插入方式可以采用头插或者尾插方式（前面范例采用的是头插方式），所以同一个图的邻接表存储方式不

唯一。也就是说，用邻接表存储图所得到的广度优先生成树并不唯一。另外，如果用邻接矩阵存储图，那么因为得到的邻接矩阵是唯一的，所以得到的广度优先生成树应该也是唯一的，你可以自行尝试。

如果一个图是非连通的，那么对它的连通分量分别进行广度优先遍历就会得到多棵广度优先生成树，合在一起组成广度优先生成森林。

深度与广度优先遍历的空间与时间复杂度

最后，还记得我们在最开始学习过的复杂度分析吗？我们在这里复习一下。

对于深度优先遍历，所需要的空间是 `vvArray` 数组和递归调用栈的最大深度，而递归调用栈的最大深度也不会超过图的顶点个数，因此对于深度优先遍历空间复杂度是 $O(|V|)$ 。

对于广度优先遍历，所需要的空间是 `vvArray` 数组、`LinkQueue` 队列，而这些存储空间的大小都不会超过顶点个数，因此对于广度优先遍历的空间复杂度是 $O(|V|)$ 。

而无论是深度优先遍历还是广度优先遍历，采用邻接表作为图的存储结构所需要的时间复杂度就是访问各个顶点以及邻接顶点所需要的时间之和，也就是 $O(|V| + |E|)$ 。

那么，如果采用邻接矩阵作为图的存储结构，那么深度优先遍历和广度优先遍历所需要的时间复杂度又是多少呢？因为查找每个顶点的邻接点都需要 $O(|V|)$ 的时间，所以 $|V|$ 个顶点的时间复杂度就是 $O(|V|^2)$ 。

小结

本节首先讲解并实现了图的深度优先遍历，也就是选择任意一条路走进去，一直到发现走不通的时候退回到上一个岔路口并重新选择一条路走进去，直到走遍所有关键节点。后面我也详细解说了深度优先遍历的顺序。

接着讲解并实现了图的广度优先遍历，也就是从近到远进行搜索，先查找离起始顶点最近的顶点，依次向外搜索，直到走遍所有关键节点。同样，我也详细解说了广度优先遍历的顺序。

最后，我们给出了深度和广度优先遍历的空间与时间复杂度。

空间复杂度：深度优先遍历和广度优先遍历的空间复杂度都是 $O(|V|)$ 。

时间复杂度：采用**邻接表**作为图的存储结构时深度优先遍历和广度优先遍历的时间复杂度都是 $O(|V|+|E|)$ 。而采用**邻接矩阵**作为图的存储结构时深度优先遍历和广度优先遍历的时间复杂度都是 $O(|V|^2)$ 。

图的深度优先遍历和广度优先遍历在算法面试题中经常出现，希望你对本节内容好好理解和掌握以从容地应对面试。

课后思考

请参照深度优先遍历的写法，书写广度优先遍历针对非连通图的遍历。

欢迎你在留言区分享自己的成果。如果觉得有所收获，也可以把课程分享给更多的朋友一起学习。我们下节课见！

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

精选留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

shikey.com转载分享