9.3 图的遍历

树代表的是“一对多”的关系，而图则具有更高的自由度，可以表示任意的“多对多”关系。因此，我们可以把树看作图的一种特例。显然，**树的遍历操作也是图的遍历操作的一种特例**。

图和树都需要应用搜索算法来实现遍历操作。图的遍历方式也可分为两种：广度优先遍历和深度优先遍历。

9.3.1 广度优先遍历

广度优先遍历是一种由近及远的遍历方式，从某个节点出发，始终优先访问距离最近的顶点，并一层层向外扩张。如图 9-9 所示，从左上角顶点出发，首先遍历该顶点的所有邻接顶点，然后遍历下一个顶点的所有邻接顶点，以此类推，直至所有顶点访问完毕。

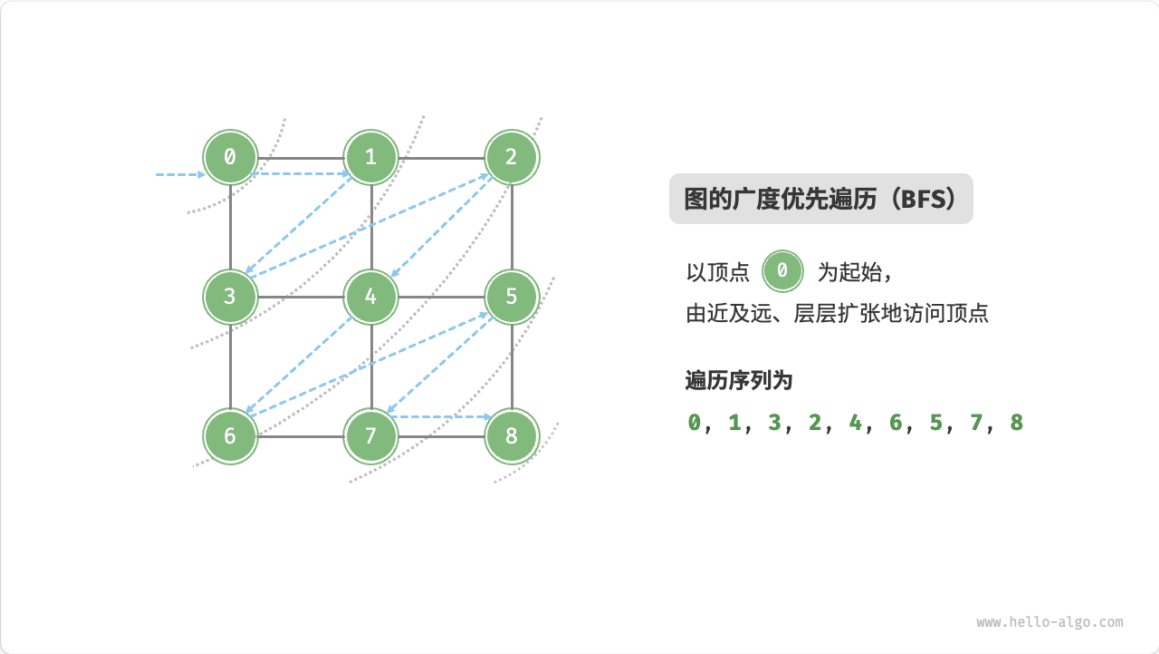


图 9-9 图的广度优先遍历

1. 算法实现

BFS 通常借助队列来实现，代码如下所示。队列具有“先入先出”的性质，这与 BFS 的“由近及远”的思想异曲同工。

1. 将遍历起始顶点 `startVet` 加入队列，并开启循环。
2. 在循环的每轮迭代中，弹出队首顶点并记录访问，然后将该顶点的所有邻接顶点加入到队列尾部。
3. 循环步骤 `2.` ，直到所有顶点被访问完毕后结束。

为了防止重复遍历顶点，我们需要借助一个哈希集合 `visited` 来记录哪些节点已被访问。

Tip

哈希集合可以看作一个只存储 `key` 而不存储 `value` 的哈希表，它可以在 $O(1)$ 时间复杂度下进行 `key` 的增删查改操作。根据 `key` 的唯一性，哈希集合通常用于数据去重等场景。

Python

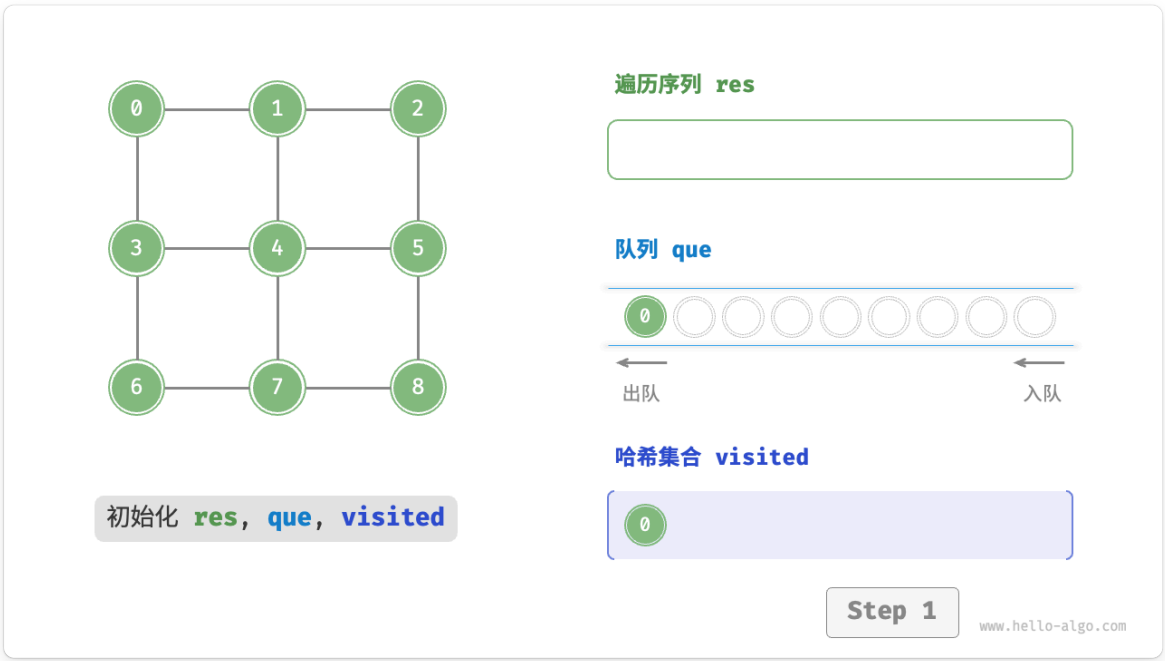
graph_bfs.py

```python
def graph_bfs(graph: GraphAdjList, start_vet: Vertex) -> list[Vertex]:
    """广度优先遍历"""
    # 使用邻接表来表示图，以便获取指定顶点的所有邻接顶点
    # 顶点遍历序列
    res = []
    # 哈希集合，用于记录已被访问过的顶点
    visited = set[Vertex]([start_vet])
    # 队列用于实现 BFS
    que = deque[Vertex]([start_vet])
    # 以顶点 vet 为起点，循环直至访问完所有顶点
    while len(que) > 0:
        vet = que.popleft()  # 队首顶点出队
        res.append(vet)  # 记录访问顶点
        # 遍历该顶点的所有邻接顶点
        for adj_vet in graph.adj_list[vet]:
            if adj_vet in visited:
                continue  # 跳过已被访问的顶点
            que.append(adj_vet)  # 只入队未访问的顶点
            visited.add(adj_vet)  # 标记该顶点已被访问
    # 返回顶点遍历序列
    return res
```

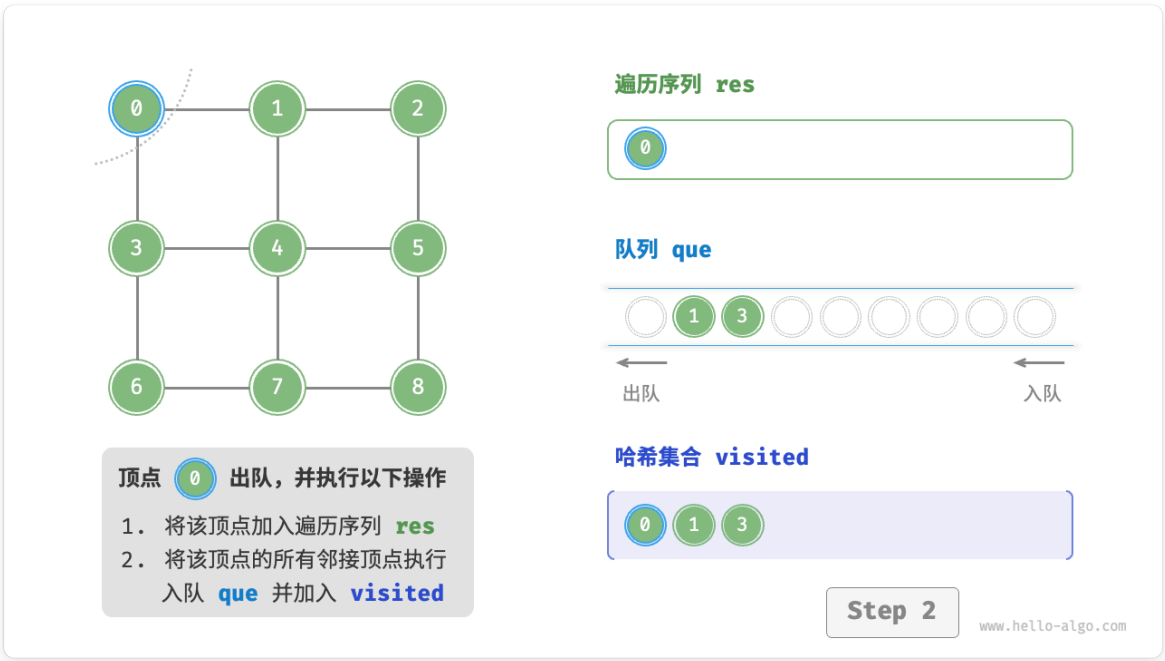
C++

graph_bfs.cpp

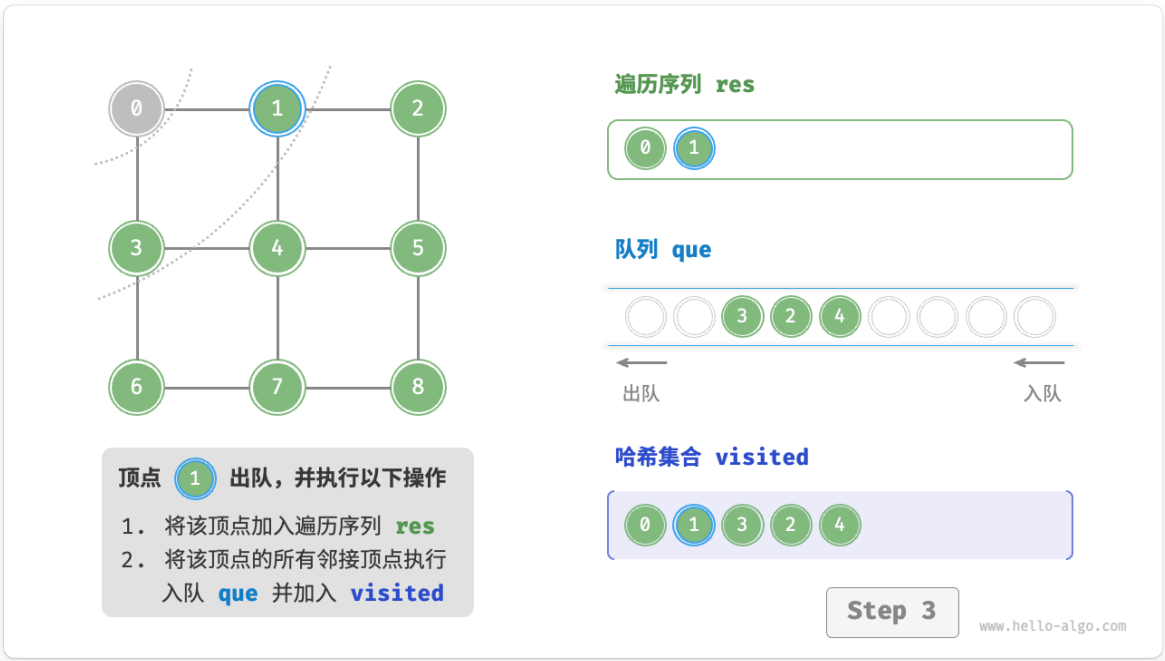
```cpp
/* 广度优先遍历 */
// 使用邻接表来表示图，以便获取指定顶点的所有邻接顶点
vector<Vertex *> graphBFS(GraphAdjList &graph, Vertex *startVet) {
    // 顶点遍历序列
    vector<Vertex *> res;
    // 哈希集合，用于记录已被访问过的顶点
    unordered_set<Vertex *> visited = {startVet};
    // 队列用于实现 BFS
    queue<Vertex *> que;
    que.push(startVet);
    // 以顶点 vet 为起点，循环直至访问完所有顶点
    while (!que.empty()) {
        Vertex *vet = que.front();
        que.pop();          // 队首顶点出队
        res.push_back(vet); // 记录访问顶点
        // 遍历该顶点的所有邻接顶点
        for (auto adjVet : graph.adjList[vet]) {
            if (visited.count(adjVet))
                continue;           // 跳过已被访问的顶点
            que.push(adjVet);       // 只入队未访问的顶点
            visited.emplace(adjVet); // 标记该顶点已被访问
        }
    }
    // 返回顶点遍历序列
```


0
1
2
3
4
5
6
7
8
初始化 res, que, visited
遍历序列 res
队列 que
出队
入队
哈希集合 visited
Step 1
www.hello-algo.com

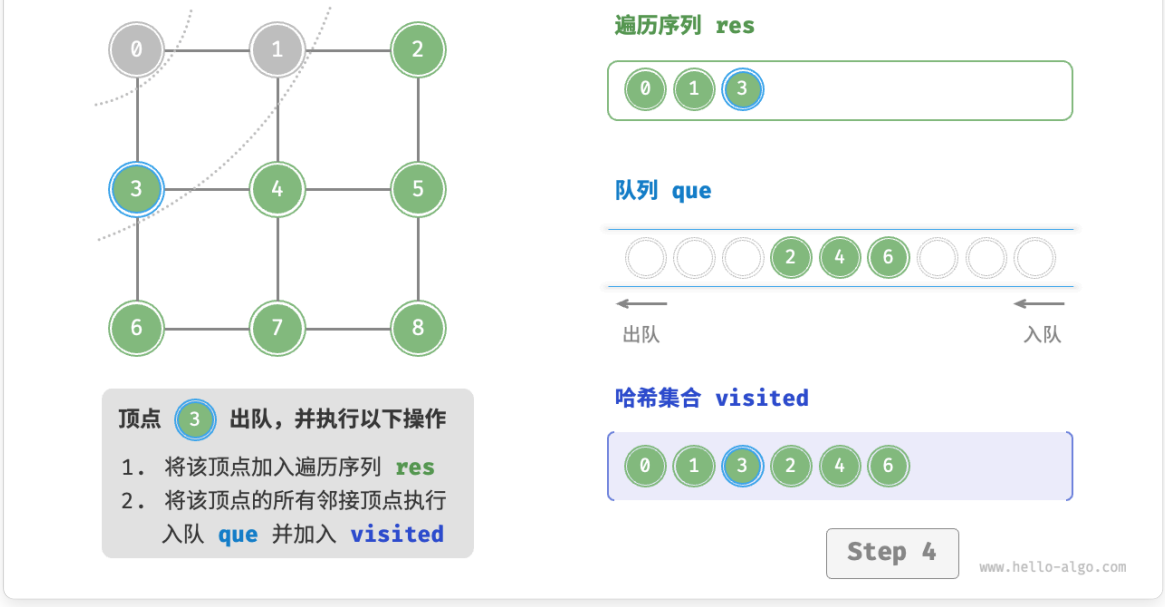
<2>


0
1
2
3
4
5
6
7
8
顶点 0 出队，并执行以下操作
1. 将该顶点加入遍历序列 res
2. 将该顶点的所有邻接顶点执行入队 que 并加入 visited
遍历序列 res
队列 que
出队
入队
哈希集合 visited
Step 2
www.hello-algo.com

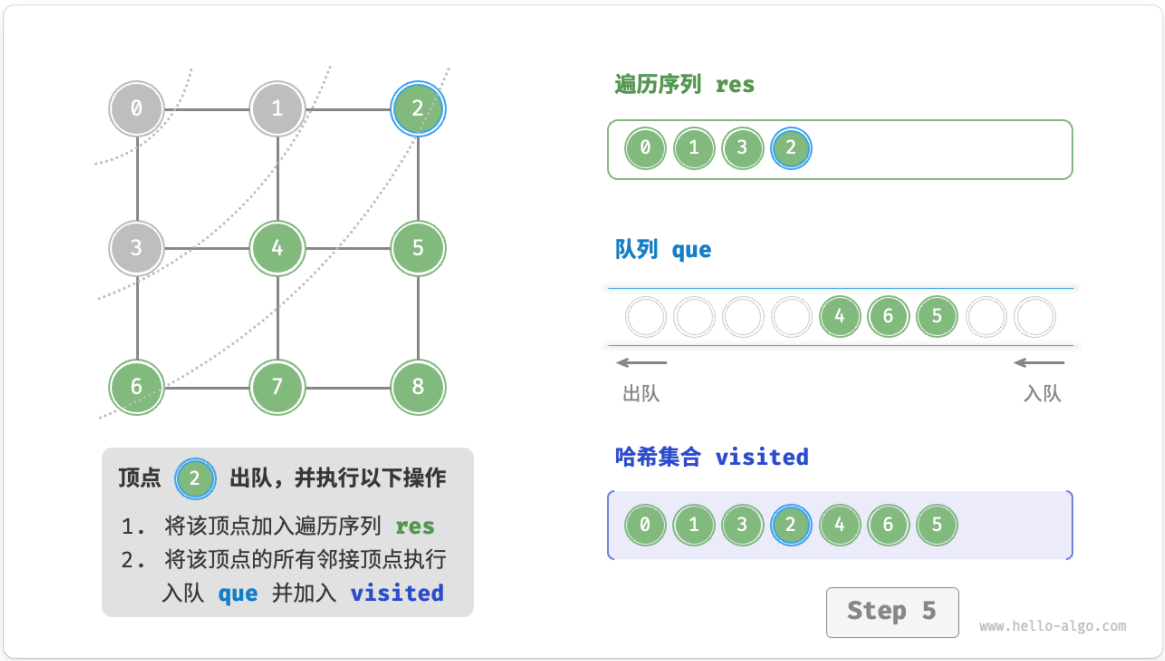
<3>


0
1
2
3
4
5
6
7
8
顶点 1 出队，并执行以下操作
1. 将该顶点加入遍历序列 res
2. 将该顶点的所有邻接顶点执行入队 que 并加入 visited
遍历序列 res
0 1
队列 que
3 2 4
出队
入队
哈希集合 visited
0 1 3 2 4
Step 3
www.hello-algo.com

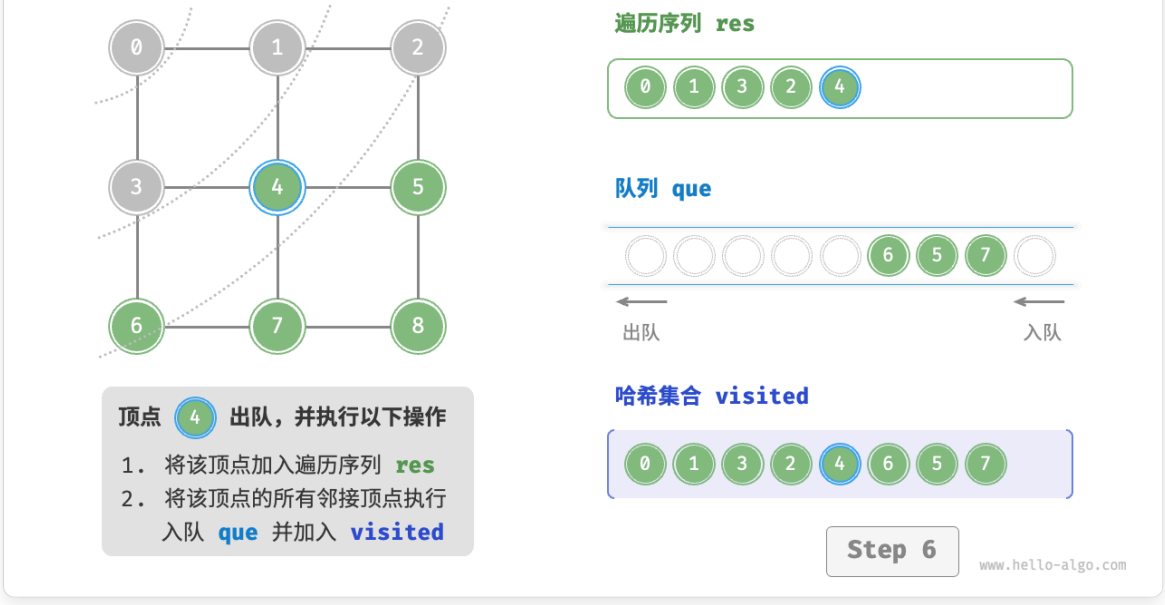
<4>


0
1
2
3
4
5
6
7
8
顶点 3 出队，并执行以下操作
1. 将该顶点加入遍历序列 res
2. 将该顶点的所有邻接顶点执行入队 que 并加入 visited
遍历序列 res
0 1 3
队列 que
2 4 6
出队
入队
哈希集合 visited
0 1 3 2 4 6
Step 4
www.hello-algo.com

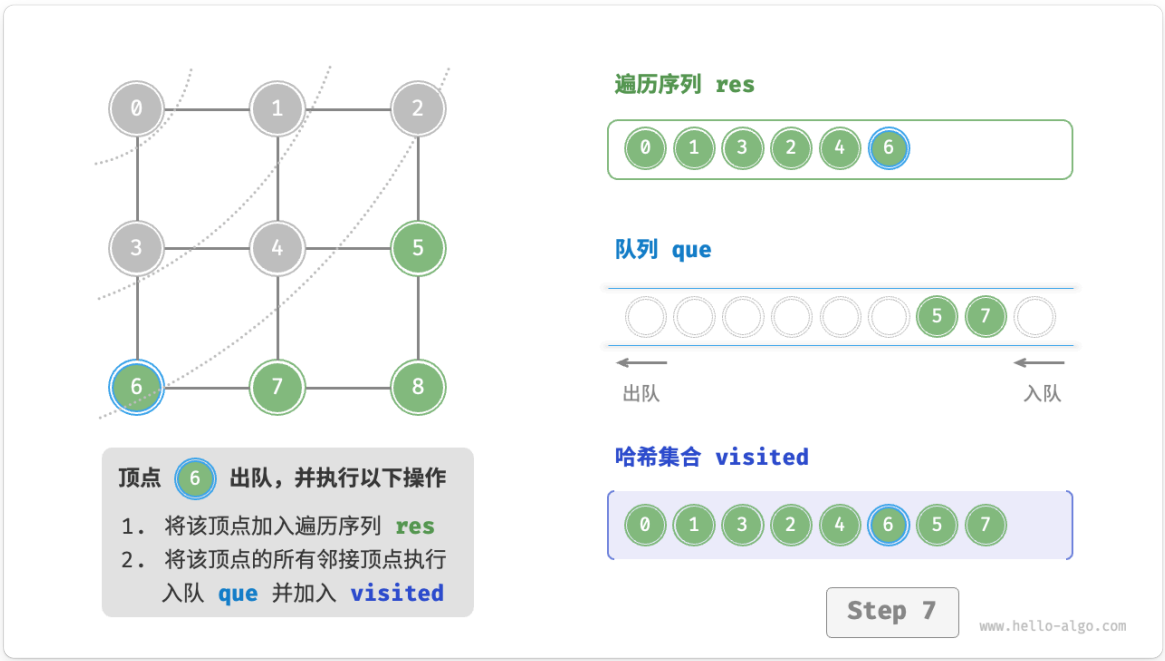
<5>


0
1
2
3
4
5
6
7
8
顶点 2 出队，并执行以下操作
1. 将该顶点加入遍历序列 res
2. 将该顶点的所有邻接顶点执行入队 que 并加入 visited
遍历序列 res
0 1 3 2
队列 que
4 6 5
出队
入队
哈希集合 visited
0 1 3 2 4 6 5
Step 5
www.hello-algo.com

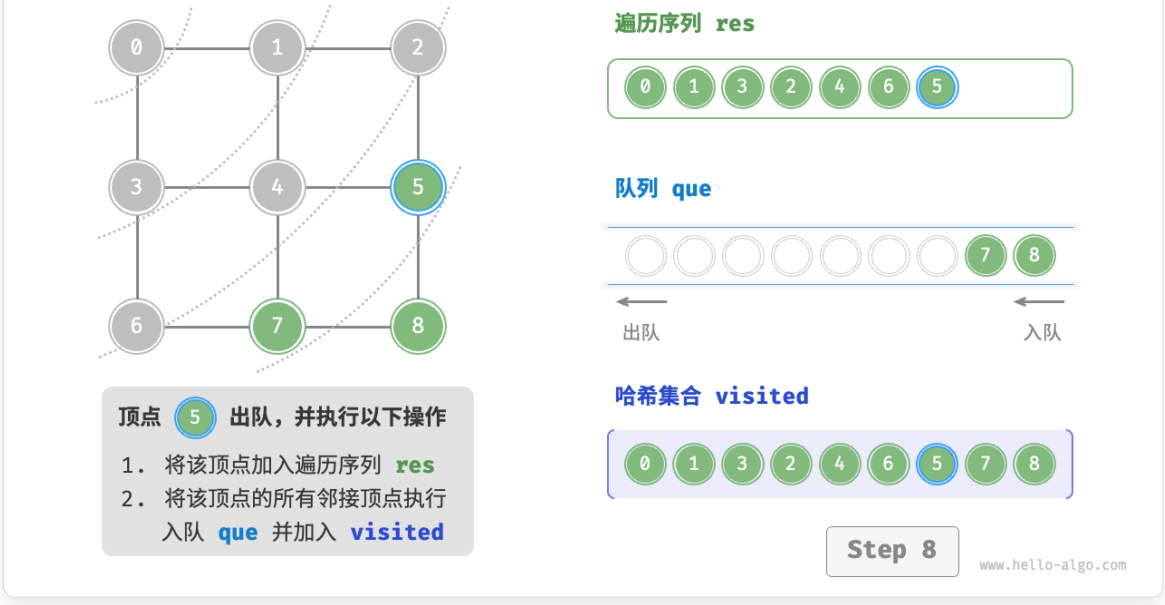
<6>


0
1
2
3
4
5
6
7
8
顶点 4 出队，并执行以下操作
1. 将该顶点加入遍历序列 res
2. 将该顶点的所有邻接顶点执行入队 que 并加入 visited
遍历序列 res
0 1 3 2 4
队列 que
6 5 7
出队
入队
哈希集合 visited
0 1 3 2 4 6 5 7
Step 6
www.hello-algo.com

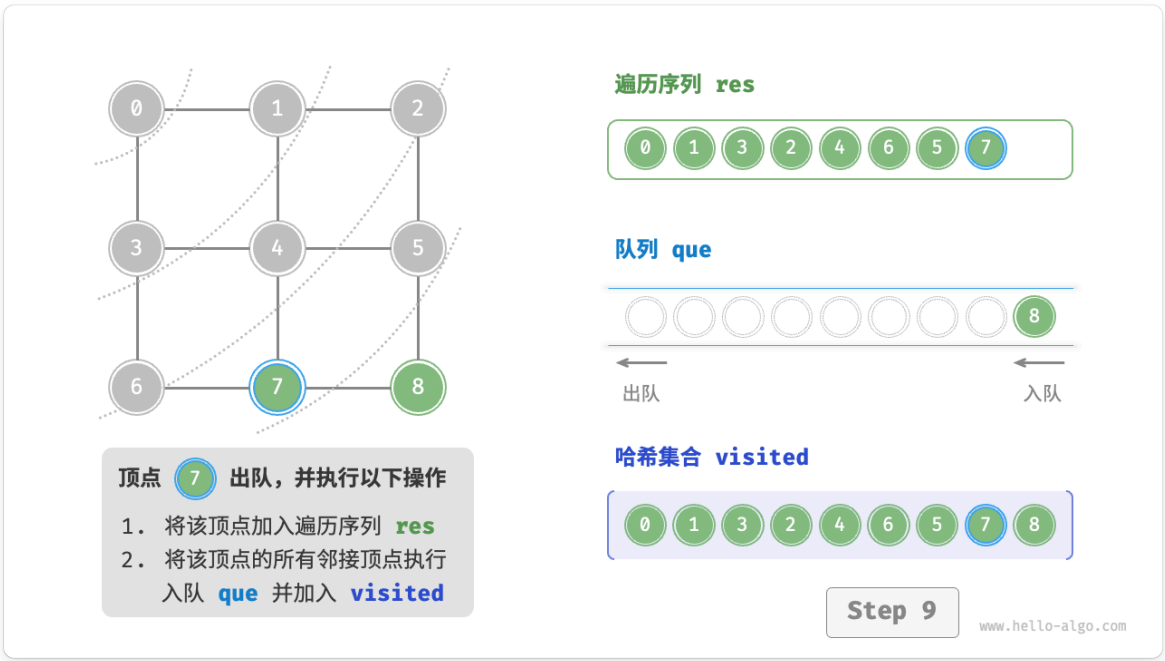
<7>


0
1
2
3
4
5
6
7
8
顶点 6 出队，并执行以下操作
1. 将该顶点加入遍历序列 res
2. 将该顶点的所有邻接顶点执行入队 que 并加入 visited
遍历序列 res
0 1 3 2 4 6
队列 que
5 7
出队
入队
哈希集合 visited
0 1 3 2 4 6 5 7
Step 7
www.hello-algo.com

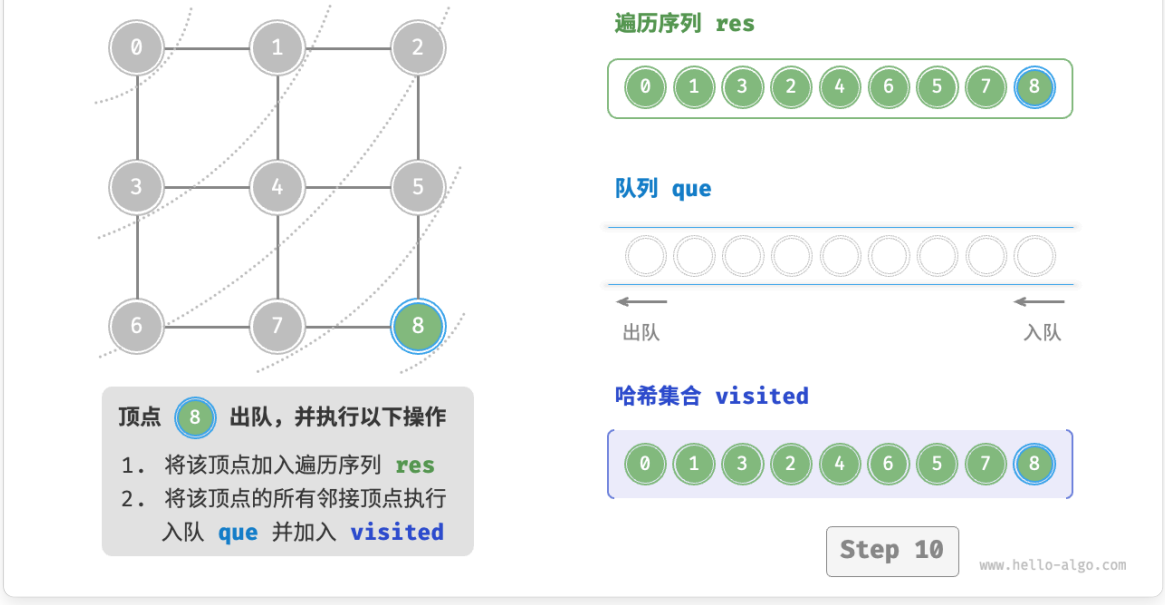
<8>


0
1
2
3
4
5
6
7
8
顶点 5 出队，并执行以下操作
1. 将该顶点加入遍历序列 res
2. 将该顶点的所有邻接顶点执行入队 que 并加入 visited
遍历序列 res
0 1 3 2 4 6 5
队列 que
7 8
出队
入队
哈希集合 visited
0 1 3 2 4 6 5 7 8
Step 8
www.hello-algo.com

<9>


0
1
2
3
4
5
6
7
8
顶点 7 出队，并执行以下操作
1. 将该顶点加入遍历序列 res
2. 将该顶点的所有邻接顶点执行入队 que 并加入 visited
遍历序列 res
0 1 3 2 4 6 5 7
队列 que
8
出队
入队
哈希集合 visited
0 1 3 2 4 6 5 7 8
Step 9
www.hello-algo.com

<10>


0
1
2
3
4
5
6
7
8
顶点 8 出队，并执行以下操作
1. 将该顶点加入遍历序列 res
2. 将该顶点的所有邻接顶点执行入队 que 并加入 visited
遍历序列 res
0 1 3 2 4 6 5 7 8
队列 que
出队
入队
哈希集合 visited
0 1 3 2 4 6 5 7 8
Step 10
www.hello-algo.com

<11>

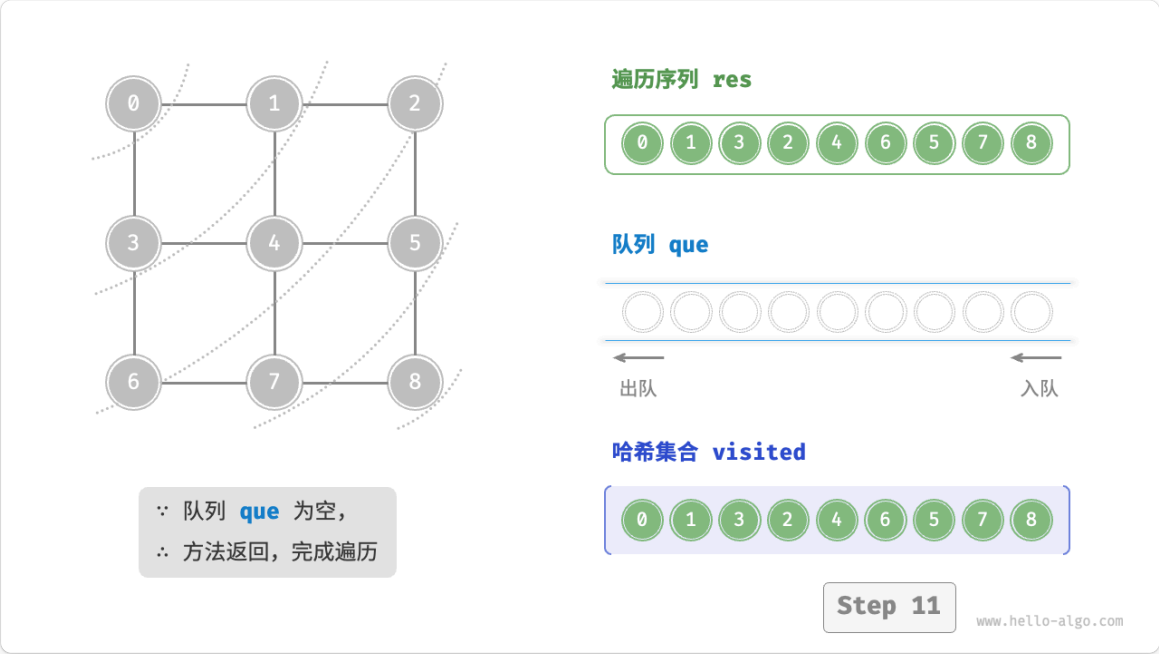


图 9-10　图的广度优先遍历步骤

广度优先遍历的序列是否唯一？

不唯一。广度优先遍历只要求按“由近及远”的顺序遍历，**而多个相同距离的顶点的遍历顺序允许被任意打乱**。以图 9-10 为例，顶点 1、3 的访问顺序可以交换，顶点 2、4、6 的访问顺序也可以任意交换。

2. 复杂度分析

时间复杂度：所有顶点都会入队并出队一次，使用 $O(|V|)$ 时间；在遍历邻接顶点的过程中，由于是无向图，因此所有边都会被访问 2 次，使用 $O(2|E|)$ 时间；总体使用 $O(|V| + |E|)$ 时间。

空间复杂度：列表 `res` ，哈希集合 `visited` ，队列 `que` 中的顶点数量最多为 $|V|$ ，使用 $O(|V|)$ 空间。

9.3.2　深度优先遍历

深度优先遍历是一种优先走到底、无路可走再回头的遍历方式。如图 9-11 所示，从左上角顶点出发，访问当前顶点的某个邻接顶点，直到走到尽头时返回，再继续走到尽头并返回，以此类推，直至所有顶点遍历完成。

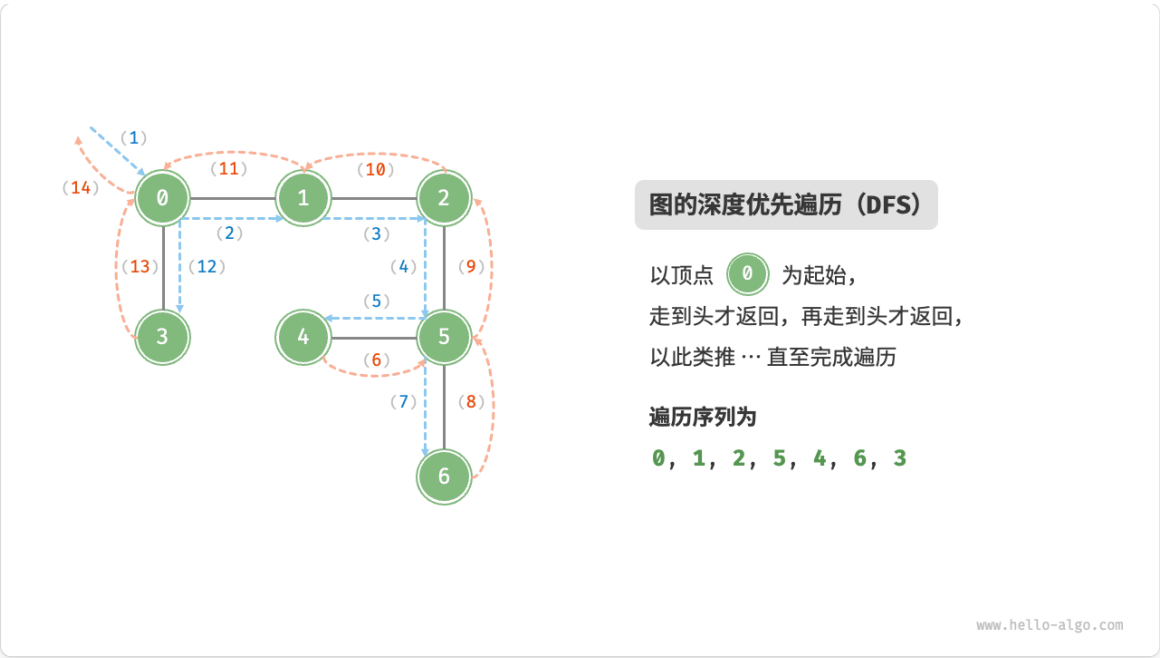


图 9-11 图的深度优先遍历

1. 算法实现

这种“走到尽头再返回”的算法范式通常基于递归来实现。与广度优先遍历类似，在深度优先遍历中，我们也需要借助一个哈希集合 `visited` 来记录已被访问的顶点，以避免重复访问顶点。

Python

graph_dfs.py

```python
def dfs(graph: GraphAdjList, visited: set[Vertex], res: list[Vertex], vet: Vertex):
    """深度优先遍历辅助函数"""
    res.append(vet)  # 记录访问顶点
    visited.add(vet)  # 标记该顶点已被访问
    # 遍历该顶点的所有邻接顶点
    for adjVet in graph.adj_list[vet]:
        if adjVet in visited:
            continue  # 跳过已被访问的顶点
        # 递归访问邻接顶点
        dfs(graph, visited, res, adjVet)

def graph_dfs(graph: GraphAdjList, start_vet: Vertex) -> list[Vertex]:
    """深度优先遍历"""
    # 使用邻接表来表示图，以便获取指定顶点的所有邻接顶点
    # 顶点遍历序列
    res = []
    # 哈希集合，用于记录已被访问过的顶点
    visited = set[Vertex]()
    dfs(graph, visited, res, start_vet)
    return res
```

C++

graph_dfs.cpp

可视化运行

Python 3.11 Frames Objects

```
1  class Vertex:
2      """顶点类"""
3      def __init__(self, val: int)
4          self.val = val
5
6  def vals_to_vets(vals: list[int]
7      """输入值列表 vals ，返回顶点列
8      return [Vertex(val) for val
9
10 class GraphAdjList:
11     """基于邻接表实现的无向图类"""
12
13     def __init__(self, edges: li
14         """构造方法"""
15         self.adj_list = dict[Ver
16         for edge in edges:
17             self.add_vertex(edge
18             self.add_vertex(edge
19             self.add_edge(edge[0
20
```

Global frame

Edit Code & Get AI Help

line that just executed

next line to execute

< Prev Next >

Click to Start Visualization

Visualized with pythontutor.com

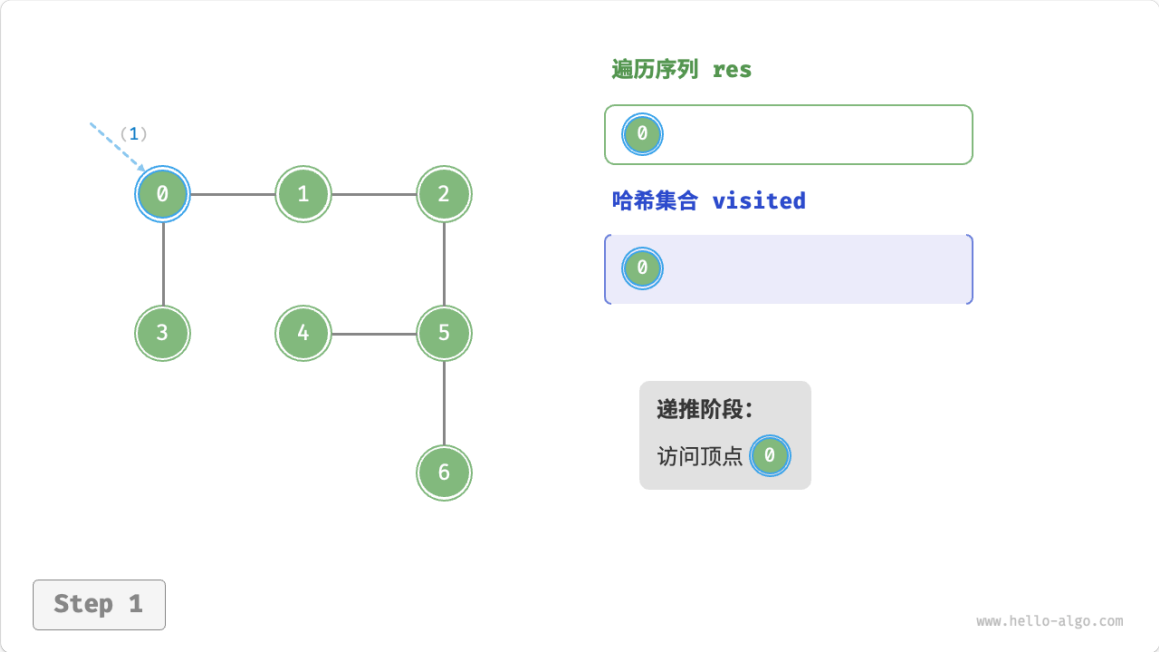
全屏观看 >

深度优先遍历的算法流程如图 9-12 所示。

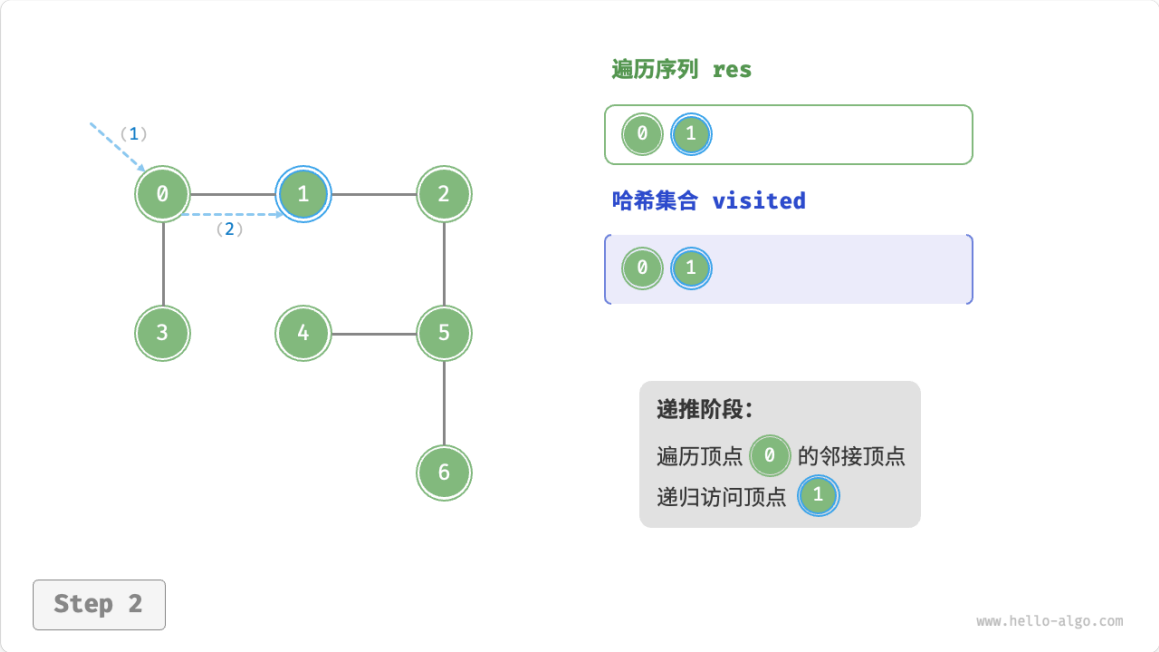
- **直虚线代表向下递推**，表示开启了一个新的递归方法来访问新顶点。
- **曲虚线代表向上回溯**，表示此递归方法已经返回，回溯到了开启此方法的位置。

为了加深理解，建议将图 9-12 与代码结合起来，在脑中模拟（或者用笔画下来）整个 DFS 过程，包括每个递归方法何时开启、何时返回。

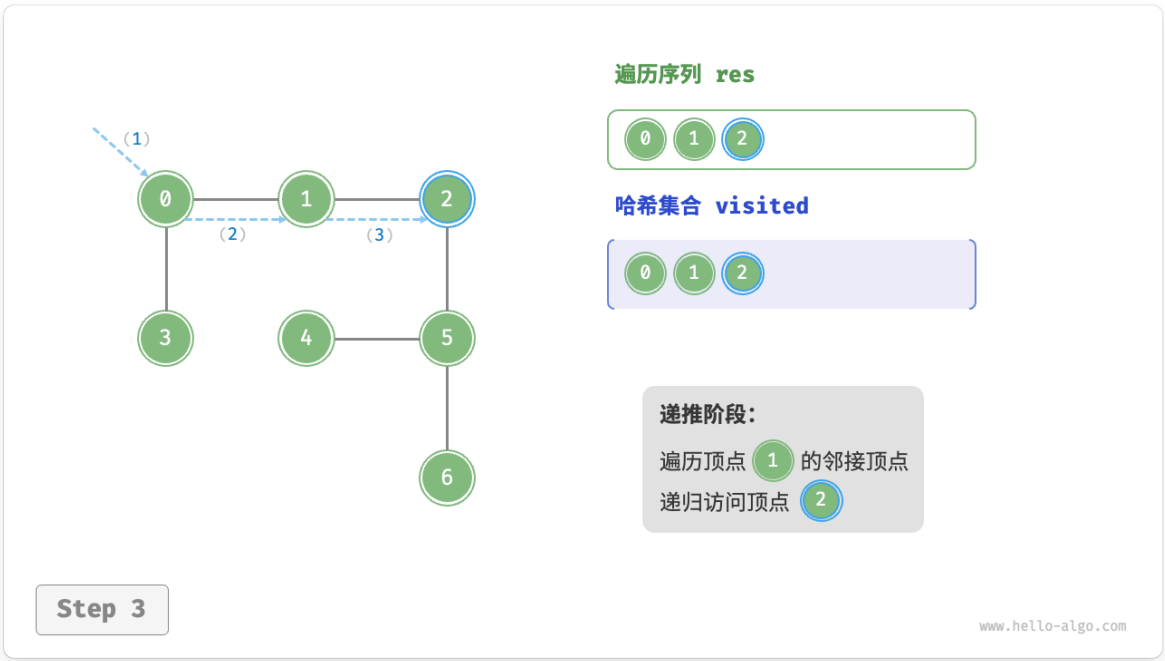
<1>


(1)
0
1
2
3
4
5
6
遍历序列 res
0
哈希集合 visited
0
递推阶段：
访问顶点 0
Step 1
www.hello-algo.com

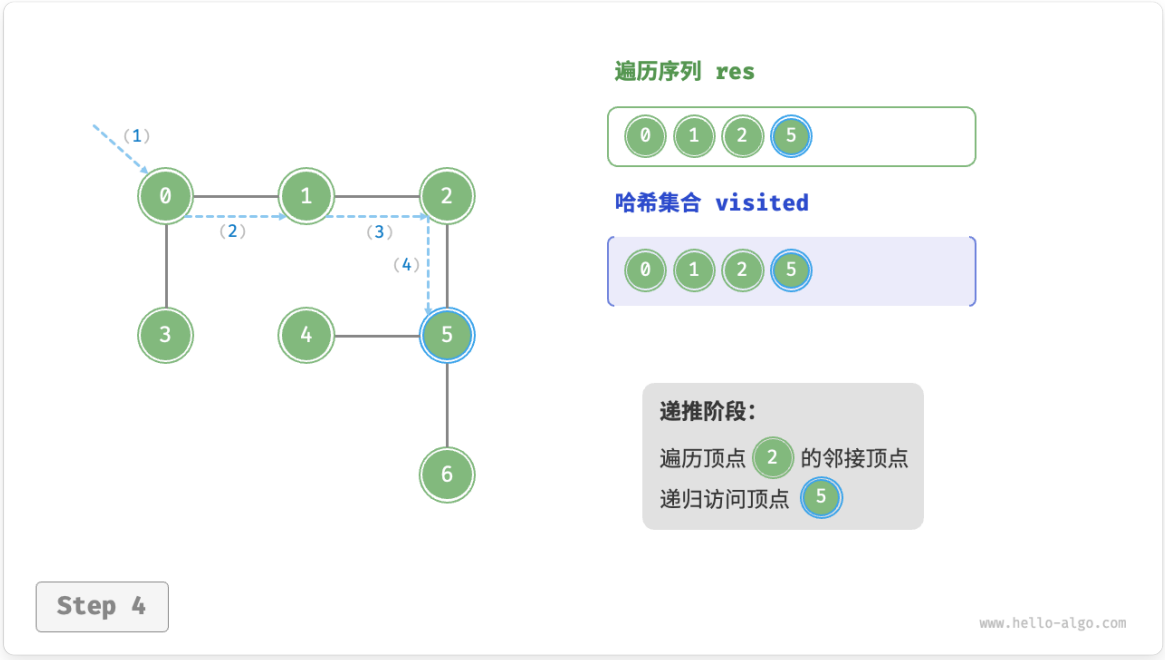
<2>


(1)
(2)
0
1
2
3
4
5
6
遍历序列 res
0 1
哈希集合 visited
0 1
递推阶段：
遍历顶点 0 的邻接顶点
递归访问顶点 1
Step 2
www.hello-algo.com

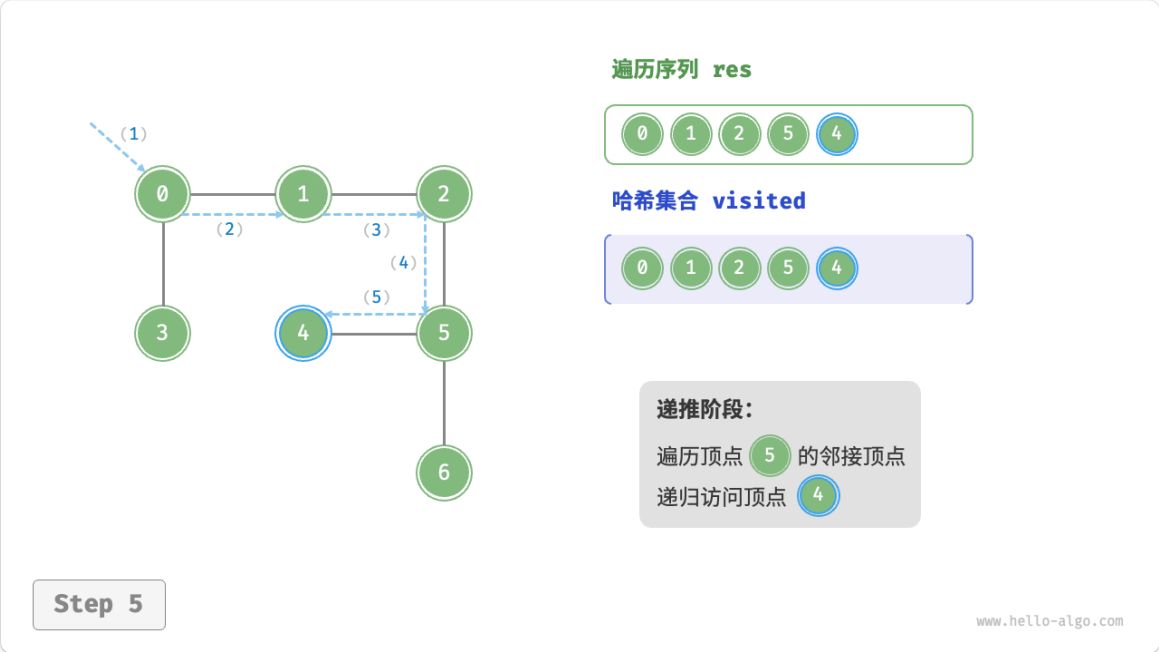
<3>


(1)
0
1
2
(2)
(3)
3
4
5
6
遍历序列 res
0 1 2
哈希集合 visited
0 1 2
递推阶段:
遍历顶点 1 的邻接顶点
递归访问顶点 2
Step 3
www.hello-algo.com

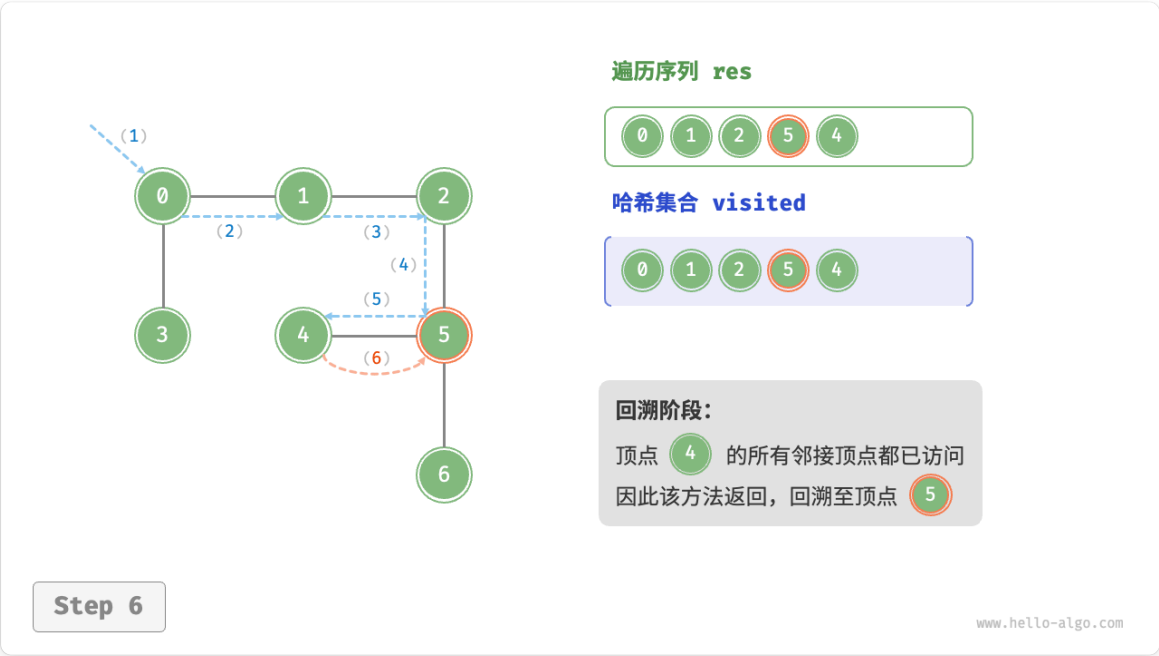
<4>


(1)
0
1
2
(2)
(3)
(4)
3
4
5
6
遍历序列 res
0 1 2 5
哈希集合 visited
0 1 2 5
递推阶段:
遍历顶点 2 的邻接顶点
递归访问顶点 5
Step 4
www.hello-algo.com

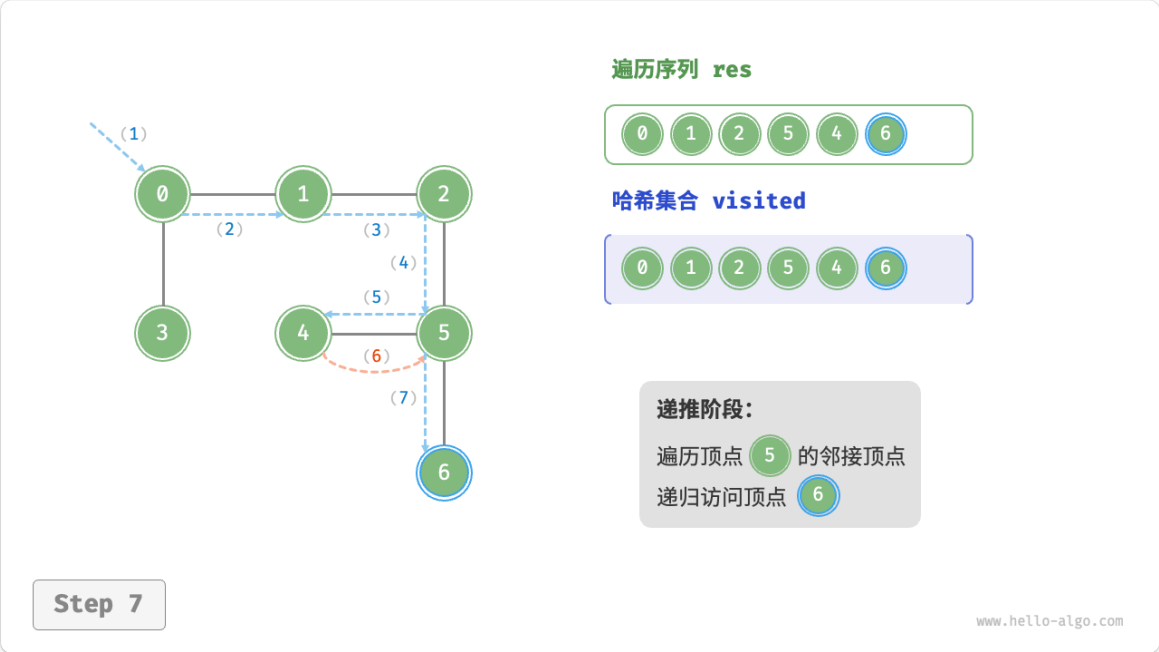
<5>


(1)
0
1
2
(2)
(3)
(4)
(5)
3
4
5
6
遍历序列 res
0 1 2 5 4
哈希集合 visited
0 1 2 5 4
递推阶段:
遍历顶点 5 的邻接顶点
递归访问顶点 4
Step 5
www.hello-algo.com

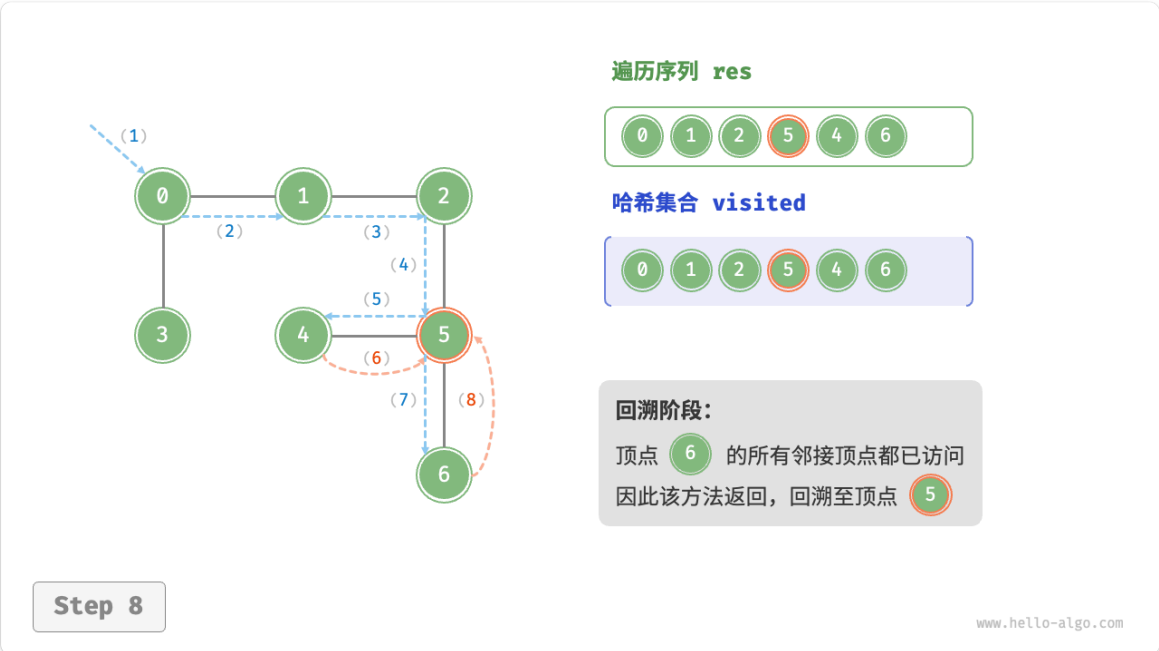
<6>


(1)
0
1
2
(2)
(3)
(4)
(5)
3
4
5
(6)
6
遍历序列 res
0 1 2 5 4
哈希集合 visited
0 1 2 5 4
回溯阶段:
顶点 4 的所有邻接顶点都已访问
因此该方法返回，回溯至顶点 5
Step 6
www.hello-algo.com

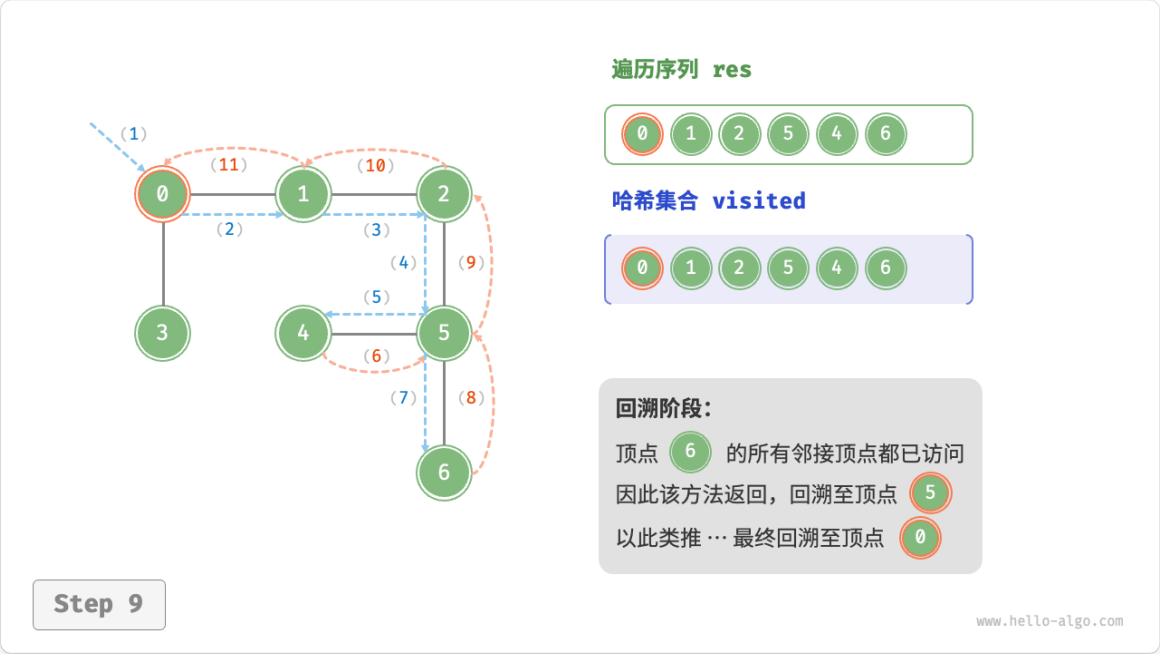
<7>


(1)
0
1
2
(2)
(3)
(4)
(5)
3
4
5
(6)
(7)
6
遍历序列 res
0 1 2 5 4 6
哈希集合 visited
0 1 2 5 4 6
递推阶段：
遍历顶点 5 的邻接顶点
递归访问顶点 6
Step 7
www.hello-algo.com

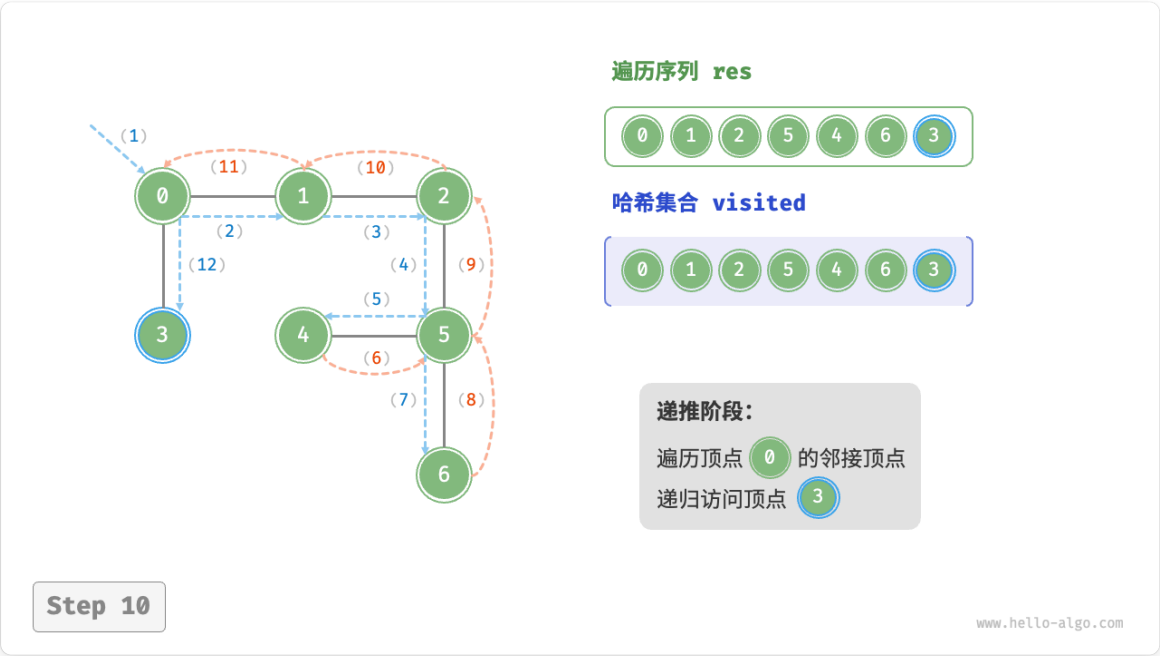
<8>


(1)
0
1
2
(2)
(3)
(4)
(5)
3
4
5
(6)
(7)
(8)
6
遍历序列 res
0 1 2 5 4 6
哈希集合 visited
0 1 2 5 4 6
回溯阶段：
顶点 6 的所有邻接顶点都已访问
因此该方法返回，回溯至顶点 5
Step 8
www.hello-algo.com

<9>


遍历序列 res
0 1 2 5 4 6
哈希集合 visited
0 1 2 5 4 6
回溯阶段：
顶点 6 的所有邻接顶点都已访问
因此该方法返回，回溯至顶点 5
以此类推 … 最终回溯至顶点 0
Step 9
www.hello-algo.com

<10>


遍历序列 res
0 1 2 5 4 6 3
哈希集合 visited
0 1 2 5 4 6 3
递推阶段：
遍历顶点 0 的邻接顶点
递归访问顶点 3
Step 10
www.hello-algo.com

<11>

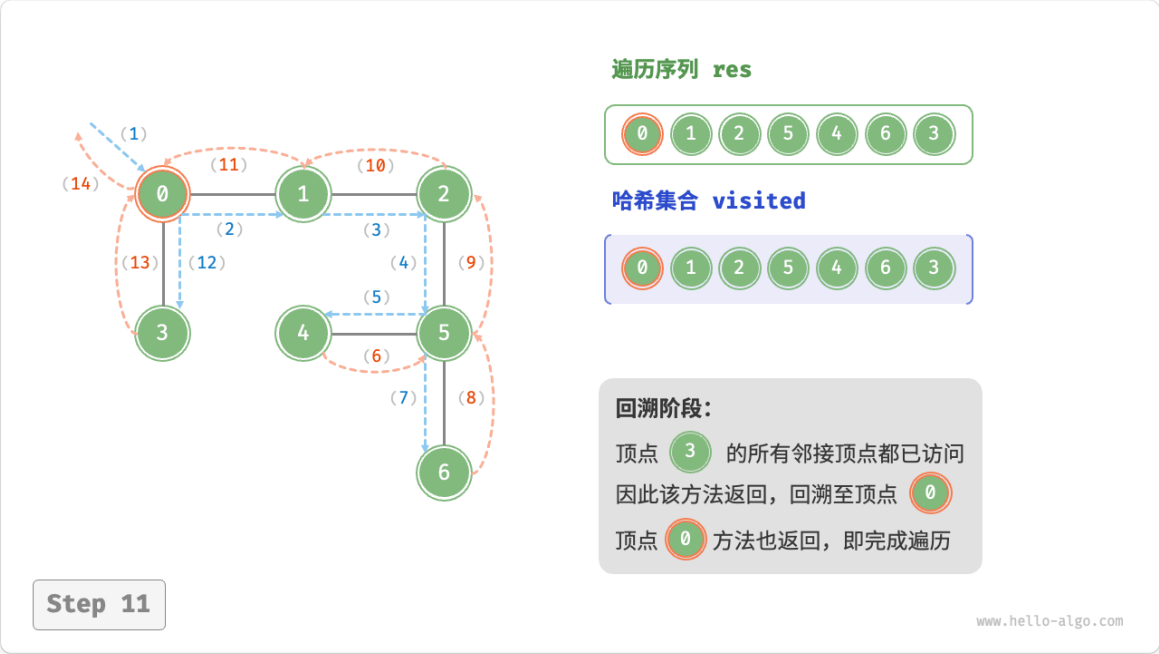


图 9-12　图的深度优先遍历步骤

深度优先遍历的序列是否唯一？

与广度优先遍历类似，深度优先遍历序列的顺序也不是唯一的。给定某顶点，先往哪个方向探索都可以，即邻接顶点的顺序可以任意打乱，都是深度优先遍历。

以树的遍历为例，“根 → 左 → 右”“左 → 根 → 右”“左 → 右 → 根”分别对应前序、中序、后序遍历，它们展示了三种遍历优先级，然而这三者都属于深度优先遍历。

2. 复杂度分析

时间复杂度：所有顶点都会被访问 1 次，使用 $O(|V|)$ 时间；所有边都会被访问 2 次，使用 $O(2|E|)$ 时间；总体使用 $O(|V| + |E|)$ 时间。

空间复杂度：列表 `res` ，哈希集合 `visited` 顶点数量最多为 $|V|$ ，递归深度最大为 $|V|$ ，因此使用 $O(|V|)$ 空间。

上一页 9.2 图基础操作
下一页 9.4 小结

欢迎在评论区留下你的见解、问题或建议