第 9 章 图

Abstract

在生命旅途中，我们就像是一个个节点，被无数看不见的边相连。
每一次的相识与相离，都在这张巨大的网络图中留下独特的印记。

9.1 图

图（graph）是一种非线性数据结构，由顶点（vertex）和边（edge）组成。我们可以将图 G 抽象地表示为一组顶点 V 和一组边 E 的集合。以下示例展示了一个包含 5 个顶点和 7 条边的图。

$$
\begin{aligned}
V & = \{ 1, 2, 3, 4, 5 \} \\
E & = \{ (1,2), (1,3), (1,5), (2,3), (2,4), (2,5), (4,5) \} \\
G & = \{ V, E \}
\end{aligned}
$$

如果将顶点看作节点，将边看作连接各个节点的引用（指针），我们就可以将图看作一种从链表拓展而来的数据结构。如图 9-1 所示，**相较于线性关系（链表）和分治关系（树），网络关系（图）的自由度更高**，因而更为复杂。

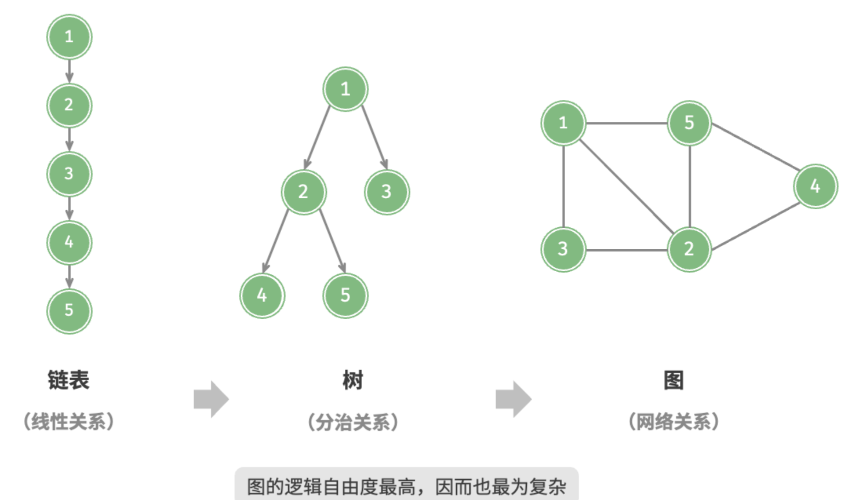


图 9-1 链表、树、图之间的关系

9.1.1 图的常见类型与术语

根据边是否具有方向，可分为无向图（undirected graph）和有向图（directed graph），如图 9-2 所示。

- 在无向图中，边表示两顶点之间的“双向”连接关系，例如微信或 QQ 中的“好友关系”。
- 在有向图中，边具有方向性，即 $A \rightarrow B$ 和 $A \leftarrow B$ 两个方向的边是相互独立的，例如微博或抖音上的“关注”与“被关注”关系。

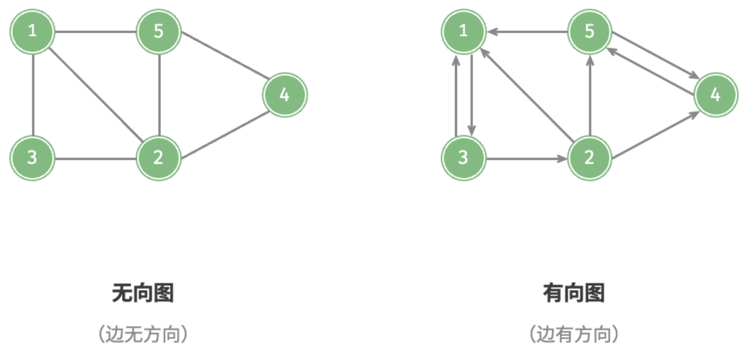


图 9-2 有向图与无向图

根据所有顶点是否连通，可分为**连通图**（connected graph）和**非连通图**（disconnected graph），如图 9-3 所示。

- 对于连通图，从某个顶点出发，可以到达其余任意顶点。
- 对于非连通图，从某个顶点出发，至少有一个顶点无法到达。

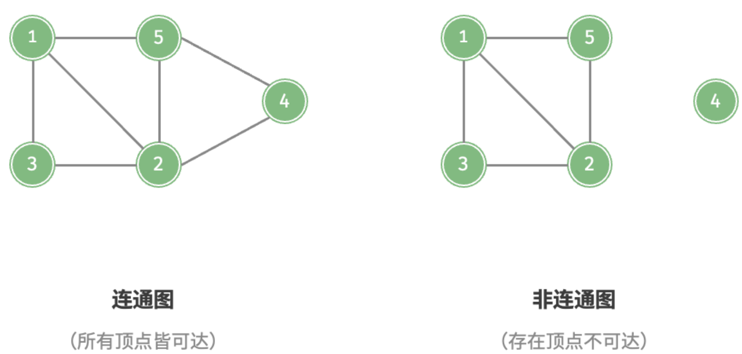


图 9-3 连通图与非连通图

我们还可以为边添加“权重”变量，从而得到如图 9-4 所示的**有权图**（weighted graph）。例如在《王者荣耀》等手游中，系统会根据共同游戏时间来计算玩家之间的“亲密度”，这种亲密度网络就可以用有权图来表示。

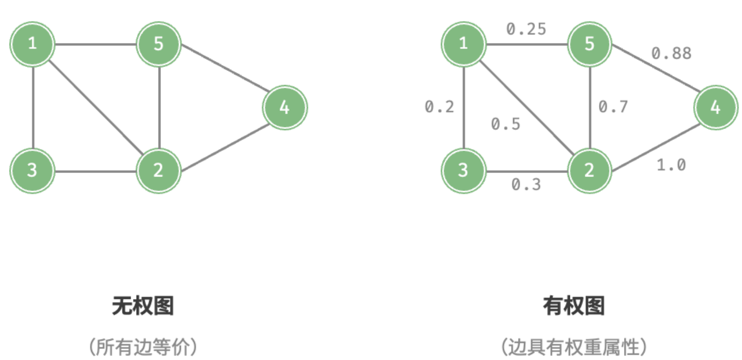


图 9-4　有权图与无权图

图数据结构包含以下常用术语。

- **邻接（adjacency）**：当两顶点之间存在边相连时，称这两顶点"邻接"。在图 9-4 中，顶点 1 的邻接顶点为顶点 2、3、5。
- **路径（path）**：从顶点 A 到顶点 B 经过的边构成的序列被称为从 A 到 B 的"路径"。在图 9-4 中，边序列 1-5-2-4 是顶点 1 到顶点 4 的一条路径。
- **度（degree）**：一个顶点拥有的边数。对于有向图，**入度（in-degree）**表示有多少条边指向该顶点，**出度（out-degree）**表示有多少条边从该顶点指出。

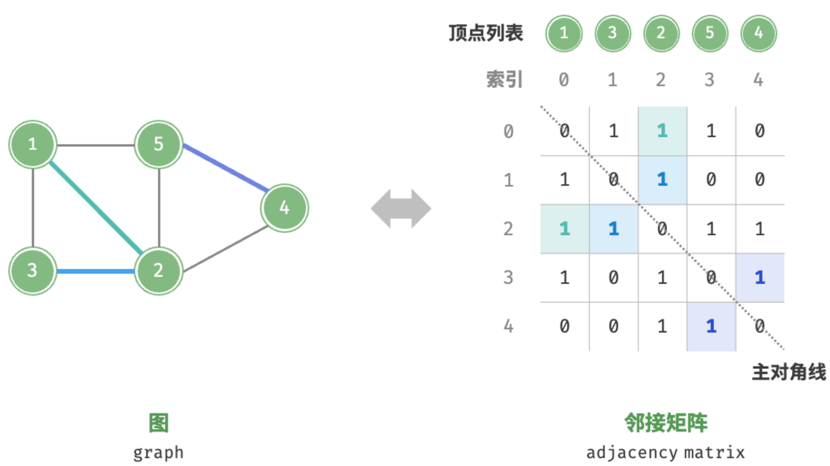
9.1.2 图的表示

图的常用表示方式包括"邻接矩阵"和"邻接表"。以下使用无向图进行举例。

1. 邻接矩阵

设图的顶点数量为 n ，**邻接矩阵（adjacency matrix）**使用一个 $n \times n$ 大小的矩阵来表示图，每一行（列）代表一个顶点，矩阵元素代表边，用 1 或 0 表示两个顶点之间是否存在边。

如图 9-5 所示，设邻接矩阵为 M、顶点列表为 V ，那么矩阵元素 $M[i, j] = 1$ 表示顶点 $V[i]$ 到顶点 $V[j]$ 之间存在边，反之 $M[i, j] = 0$ 表示两顶点之间无边。



	0	1	2	3	4
0	0	1	1	1	0
1	1	0	1	0	0
2	1	1	0	1	1
3	1	0	1	0	1
4	0	0	1	1	0

图 9-5　图的邻接矩阵表示

邻接矩阵具有以下特性。

- 在简单图中，顶点不能与自身相连，此时邻接矩阵主对角线元素没有意义。
- 对于无向图，两个方向的边等价，此时邻接矩阵关于主对角线对称。
- 将邻接矩阵的元素从 1 和 0 替换为权重，则可表示有权图。

使用邻接矩阵表示图时，我们可以直接访问矩阵元素以获取边，因此增删查改操作的效率很高，时间复杂度均为 $O(1)$ 。然而，矩阵的空间复杂度为 $O(n^2)$ ，内存占用较多。

2.　邻接表

邻接表（adjacency list）使用 n 个链表来表示图，链表节点表示顶点。第 i 个链表对应顶点 i ，其中存储了该顶点的所有邻接顶点（与该顶点相连的顶点）。图 9-6 展示了一个使用邻接表存储的图的示例。

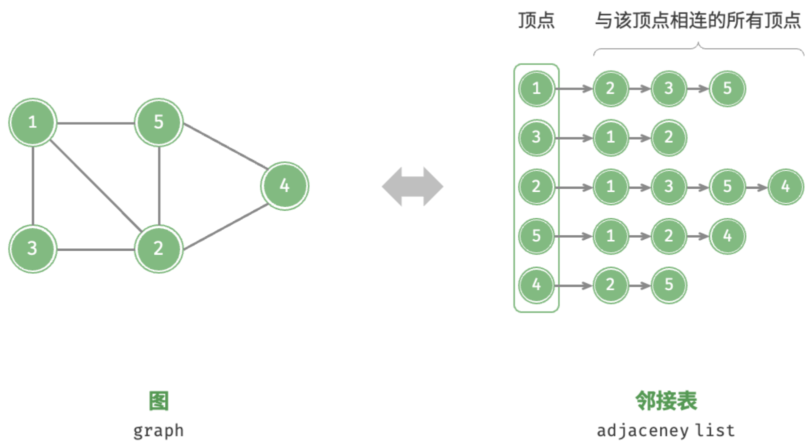


图 9-6　图的邻接表表示

邻接表仅存储实际存在的边，而边的总数通常远小于 n^2 ，因此它更加节省空间。然而，在邻接表中需要通过遍历链表来查找边，因此其时间效率不如邻接矩阵。

观察图 9-6 ，**邻接表结构与哈希表中的“链式地址”非常相似，因此我们也可以采用类似的方法来优化效率**。比如当链表较长时，可以将链表转化为 AVL 树或红黑树，从而将时间效率从 $O(n)$ 优化至 $O(\log n)$ ；还可以把链表转换为哈希表，从而将时间复杂度降至 $O(1)$ 。

9.1.3　图的常见应用

如表 9-1 所示，许多现实系统可以用图来建模，相应的问题也可以约化为图计算问题。

表 9-1　现实生活中常见的图

	顶点	边	图计算问题
社交网络	用户	好友关系	潜在好友推荐
地铁线路	站点	站点间的连通性	最短路线推荐
太阳系	星体	星体间的万有引力作用	行星轨道计算

9.2 图的基础操作

图的基础操作可分为对“边”的操作和对“顶点”的操作。在“邻接矩阵”和“邻接表”两种表示方法下，实现方式有所不同。

9.2.1 基于邻接矩阵的实现

给定一个顶点数量为 n 的无向图，则各种操作的实现方式如图 9-7 所示。

- **添加或删除边**：直接在邻接矩阵中修改指定的边即可，使用 $O(1)$ 时间。而由于是无向图，因此需要同时更新两个方向的边。
- **添加顶点**：在邻接矩阵的尾部添加一行一列，并全部填 0 即可，使用 $O(n)$ 时间。
- **删除顶点**：在邻接矩阵中删除一行一列。当删除首行首列时达到最差情况，需要将 $(n-1)^2$ 个元素“向左上移动”，从而使用 $O(n^2)$ 时间。
- **初始化**：传入 n 个顶点，初始化长度为 n 的顶点列表 `vertices` ，使用 $O(n)$ 时间；初始化 $n \times n$ 大小的邻接矩阵 `adjMat` ，使用 $O(n^2)$ 时间。

图 9-7 邻接矩阵的初始化、增删边、增删顶点

以下是基于邻接矩阵表示图的实现代码：

```c
// === File: graph_adjacency_matrix.c ===

/* 基于邻接矩阵实现的无向图结构体 */
typedef struct {
    int vertices[MAX_SIZE];
    int adjMat[MAX_SIZE][MAX_SIZE];
    int size;
} GraphAdjMat;

/* 构造函数 */
GraphAdjMat *newGraphAdjMat() {
    GraphAdjMat *graph = (GraphAdjMat *)malloc(sizeof(GraphAdjMat));
    graph->size = 0;
    for (int i = 0; i < MAX_SIZE; i++) {
        for (int j = 0; j < MAX_SIZE; j++) {
            graph->adjMat[i][j] = 0;
        }
    }
    return graph;
}

/* 析构函数 */
void delGraphAdjMat(GraphAdjMat *graph) {
    free(graph);
}

/* 添加顶点 */
void addVertex(GraphAdjMat *graph, int val) {
    if (graph->size == MAX_SIZE) {
        fprintf(stderr, " 图的顶点数量已达最大值\n");
        return;
    }
    // 添加第 n 个顶点，并将第 n 行和列置零
    int n = graph->size;
    graph->vertices[n] = val;
    for (int i = 0; i <= n; i++) {
        graph->adjMat[n][i] = graph->adjMat[i][n] = 0;
    }
    graph->size++;
}

/* 删除顶点 */
void removeVertex(GraphAdjMat *graph, int index) {
    if (index < 0 || index >= graph->size) {
```

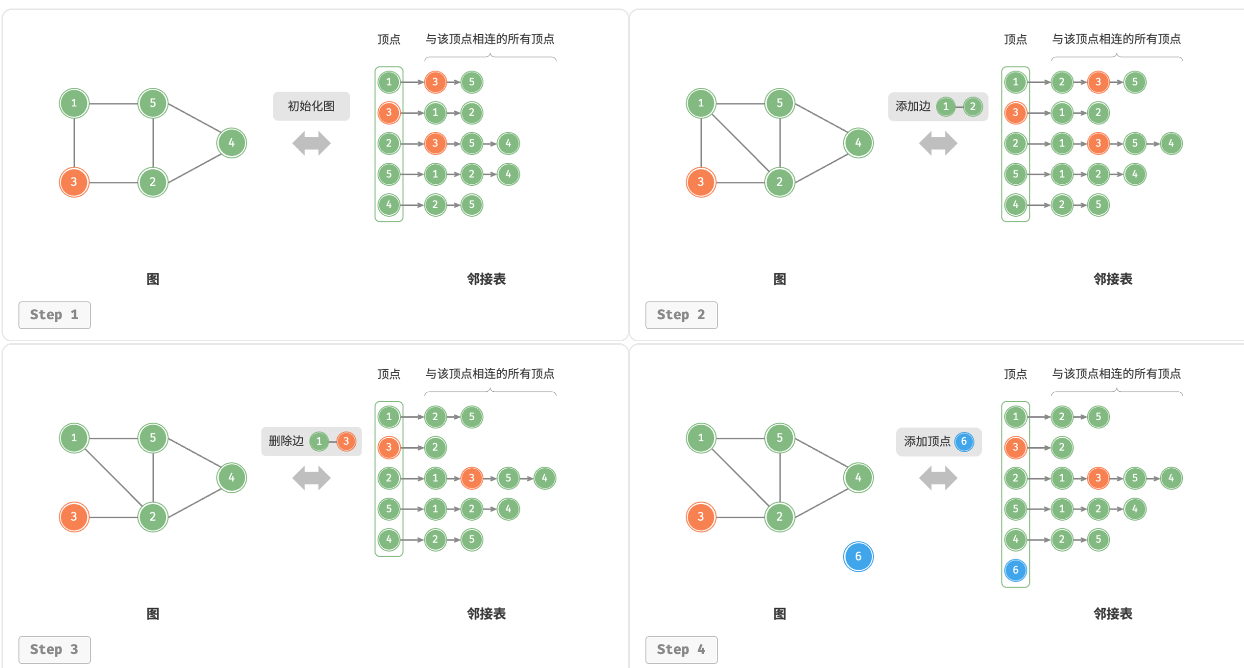
```
        fprintf(stderr, " 顶点索引越界\n");
        return;
    }
    // 在顶点列表中移除索引 index 的顶点
    for (int i = index; i < graph->size - 1; i++) {
        graph->vertices[i] = graph->vertices[i + 1];
    }
    // 在邻接矩阵中删除索引 index 的行
    for (int i = index; i < graph->size - 1; i++) {
        for (int j = 0; j < graph->size; j++) {
            graph->adjMat[i][j] = graph->adjMat[i + 1][j];
        }
    }
    // 在邻接矩阵中删除索引 index 的列
    for (int i = 0; i < graph->size; i++) {
        for (int j = index; j < graph->size - 1; j++) {
            graph->adjMat[i][j] = graph->adjMat[i][j + 1];
        }
    }
    graph->size--;
}

/* 添加边 */
// 参数 i, j 对应 vertices 元素索引
void addEdge(GraphAdjMat *graph, int i, int j) {
    if (i < 0 || j < 0 || i >= graph->size || j >= graph->size || i == j) {
        fprintf(stderr, " 边索引越界或相等\n");
        return;
    }
    graph->adjMat[i][j] = 1;
    graph->adjMat[j][i] = 1;
}

/* 删除边 */
// 参数 i, j 对应 vertices 元素索引
void removeEdge(GraphAdjMat *graph, int i, int j) {
    if (i < 0 || j < 0 || i >= graph->size || j >= graph->size || i == j) {
        fprintf(stderr, " 边索引越界或相等\n");
        return;
    }
    graph->adjMat[i][j] = 0;
    graph->adjMat[j][i] = 0;
}

/* 打印邻接矩阵 */
void printGraphAdjMat(GraphAdjMat *graph) {
    printf(" 顶点列表 = ");
```

```
    printArray(graph->vertices, graph->size);
    printf(" 邻接矩阵 =\n");
    for (int i = 0; i < graph->size; i++) {
        printArray(graph->adjMat[i], graph->size);
    }
}
```

9.2.2 基于邻接表的实现

设无向图的顶点总数为 n、边总数为 m，则可根据图 9-8 所示的方法实现各种操作。

- **添加边**：在顶点对应链表的末尾添加边即可，使用 $O(1)$ 时间。因为是无向图，所以需要同时添加两个方向的边。
- **删除边**：在顶点对应链表中查找并删除指定边，使用 $O(m)$ 时间。在无向图中，需要同时删除两个方向的边。
- **添加顶点**：在邻接表中添加一个链表，并将新增顶点作为链表头节点，使用 $O(1)$ 时间。
- **删除顶点**：需遍历整个邻接表，删除包含指定顶点的所有边，使用 $O(n + m)$ 时间。
- **初始化**：在邻接表中创建 n 个顶点和 $2m$ 条边，使用 $O(n + m)$ 时间。



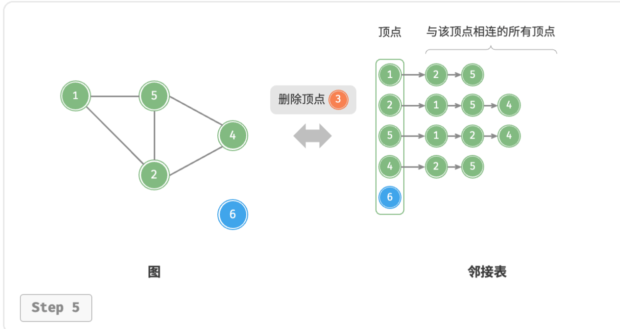


图 9-8 邻接表的初始化、增删边、增删顶点

以下是邻接表的代码实现。对比图 9-8，实际代码有以下不同。

- 为了方便添加与删除顶点，以及简化代码，我们使用列表（动态数组）来代替链表。
- 使用哈希表来存储邻接表，`key` 为顶点实例，`value` 为该顶点的邻接顶点列表（链表）。

另外，我们在邻接表中使用 `Vertex` 类来表示顶点，这样做的原因是：如果与邻接矩阵一样，用列表索引来区分不同顶点，那么假设要删除索引为 i 的顶点，则需遍历整个邻接表，将所有大于 i 的索引全部减 1，效率很低。而如果每个顶点都是唯一的 `Vertex` 实例，删除某一顶点之后就无须改动其他顶点了。

```c
// === File: graph_adjacency_list.c ===

/* 节点结构体 */
typedef struct AdjListNode {
    Vertex *vertex;           // 顶点
    struct AdjListNode *next; // 后继节点
} AdjListNode;

/* 查找顶点对应的节点 */
AdjListNode *findNode(GraphAdjList *graph, Vertex *vet) {
    for (int i = 0; i < graph->size; i++) {
        if (graph->heads[i]->vertex == vet) {
            return graph->heads[i];
        }
    }
    return NULL;
}

/* 添加边辅助函数 */
void addEdgeHelper(AdjListNode *head, Vertex *vet) {
    AdjListNode *node = (AdjListNode *)malloc(sizeof(AdjListNode));
    node->vertex = vet;
    // 头插法
    node->next = head->next;
    head->next = node;
}
```

```c
/* 删除边辅助函数 */
void removeEdgeHelper(AdjListNode *head, Vertex *vet) {
    AdjListNode *pre = head;
    AdjListNode *cur = head->next;
    // 在链表中搜索 vet 对应节点
    while (cur != NULL && cur->vertex != vet) {
        pre = cur;
        cur = cur->next;
    }
    if (cur == NULL)
        return;
    // 将 vet 对应节点从链表中删除
    pre->next = cur->next;
    // 释放内存
    free(cur);
}

/* 基于邻接表实现的无向图类 */
typedef struct {
    AdjListNode *heads[MAX_SIZE]; // 节点数组
    int size;                     // 节点数量
} GraphAdjList;

/* 构造函数 */
GraphAdjList *newGraphAdjList() {
    GraphAdjList *graph = (GraphAdjList *)malloc(sizeof(GraphAdjList));
    if (!graph) {
        return NULL;
    }
    graph->size = 0;
    for (int i = 0; i < MAX_SIZE; i++) {
        graph->heads[i] = NULL;
    }
    return graph;
}

/* 析构函数 */
void delGraphAdjList(GraphAdjList *graph) {
    for (int i = 0; i < graph->size; i++) {
        AdjListNode *cur = graph->heads[i];
        while (cur != NULL) {
            AdjListNode *next = cur->next;
            if (cur != graph->heads[i]) {
                free(cur);
            }
            cur = next;
```

```c
        }
        free(graph->heads[i]->vertex);
        free(graph->heads[i]);
    }
    free(graph);
}

/* 查找顶点对应的节点 */
AdjListNode *findNode(GraphAdjList *graph, Vertex *vet) {
    for (int i = 0; i < graph->size; i++) {
        if (graph->heads[i]->vertex == vet) {
            return graph->heads[i];
        }
    }
    return NULL;
}

/* 添加边 */
void addEdge(GraphAdjList *graph, Vertex *vet1, Vertex *vet2) {
    AdjListNode *head1 = findNode(graph, vet1);
    AdjListNode *head2 = findNode(graph, vet2);
    assert(head1 != NULL && head2 != NULL && head1 != head2);
    // 添加边 vet1 - vet2
    addEdgeHelper(head1, vet2);
    addEdgeHelper(head2, vet1);
}

/* 删除边 */
void removeEdge(GraphAdjList *graph, Vertex *vet1, Vertex *vet2) {
    AdjListNode *head1 = findNode(graph, vet1);
    AdjListNode *head2 = findNode(graph, vet2);
    assert(head1 != NULL && head2 != NULL);
    // 删除边 vet1 - vet2
    removeEdgeHelper(head1, head2->vertex);
    removeEdgeHelper(head2, head1->vertex);
}

/* 添加顶点 */
void addVertex(GraphAdjList *graph, Vertex *vet) {
    assert(graph != NULL && graph->size < MAX_SIZE);
    AdjListNode *head = (AdjListNode *)malloc(sizeof(AdjListNode));
    head->vertex = vet;
    head->next = NULL;
    // 在邻接表中添加一个新链表
    graph->heads[graph->size++] = head;
}
```

```c
/* 删除顶点 */
void removeVertex(GraphAdjList *graph, Vertex *vet) {
    AdjListNode *node = findNode(graph, vet);
    assert(node != NULL);
    // 在邻接表中删除顶点 vet 对应的链表
    AdjListNode *cur = node, *pre = NULL;
    while (cur) {
        pre = cur;
        cur = cur->next;
        free(pre);
    }
    // 遍历其他顶点的链表，删除所有包含 vet 的边
    for (int i = 0; i < graph->size; i++) {
        cur = graph->heads[i];
        pre = NULL;
        while (cur) {
            pre = cur;
            cur = cur->next;
            if (cur && cur->vertex == vet) {
                pre->next = cur->next;
                free(cur);
                break;
            }
        }
    }
    // 将该顶点之后的顶点向前移动，以填补空缺
    int i;
    for (i = 0; i < graph->size; i++) {
        if (graph->heads[i] == node)
            break;
    }
    for (int j = i; j < graph->size - 1; j++) {
        graph->heads[j] = graph->heads[j + 1];
    }
    graph->size--;
    free(vet);
}
```

9.2.3 效率对比

设图中共有 n 个顶点和 m 条边，表 9-2 对比了邻接矩阵和邻接表的时间效率和空间效率。

表 9-2 邻接矩阵与邻接表对比

	邻接矩阵	邻接表（链表）	邻接表（哈希表）
判断是否邻接	$O(1)$	$O(m)$	$O(1)$
添加边	$O(1)$	$O(1)$	$O(1)$
删除边	$O(1)$	$O(m)$	$O(1)$
添加顶点	$O(n)$	$O(1)$	$O(1)$
删除顶点	$O(n^2)$	$O(n+m)$	$O(n)$
内存空间占用	$O(n^2)$	$O(n+m)$	$O(n+m)$

观察表 9-2，似乎邻接表（哈希表）的时间效率与空间效率最优。但实际上，在邻接矩阵中操作边的效率更高，只需一次数组访问或赋值操作即可。综合来看，邻接矩阵体现了“以空间换时间”的原则，而邻接表体现了“以时间换空间”的原则。

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。

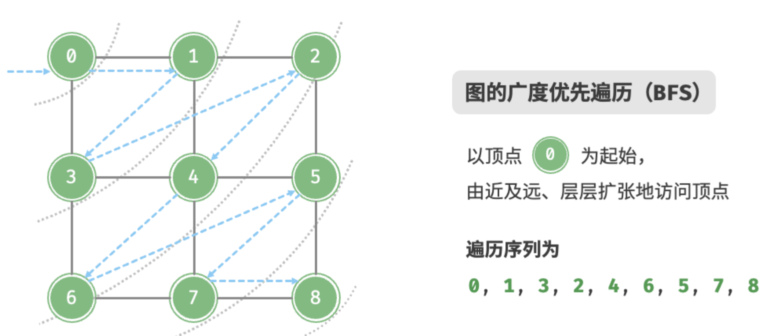


图 9-9　图的广度优先遍历

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。

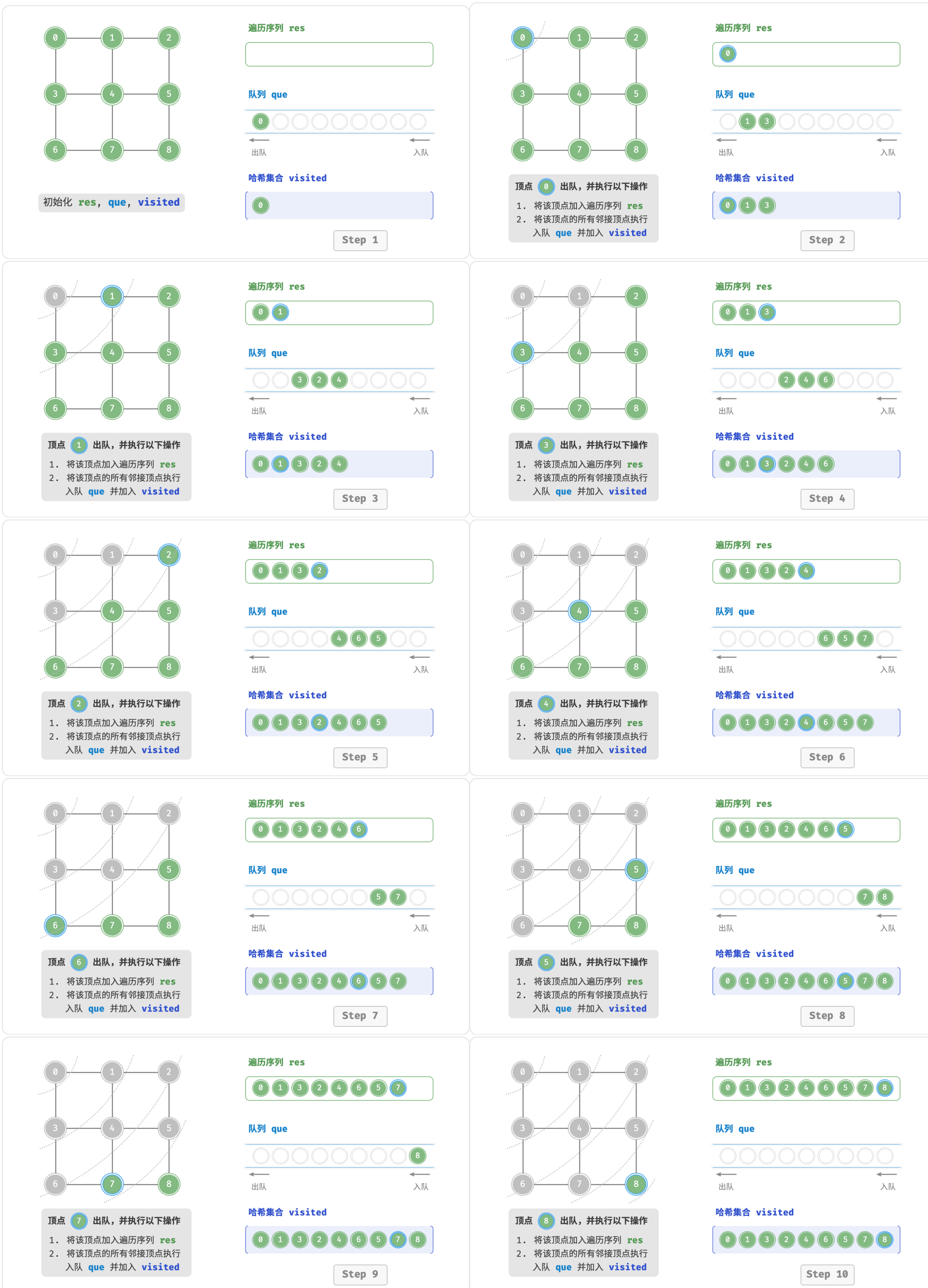
为了防止重复遍历顶点，我们需要借助一个哈希集合 `visited` 来记录哪些节点已被访问。

> **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。

```c
// === File: graph_bfs.c ===

/* 节点队列结构体 */
typedef struct {
    Vertex *vertices[MAX_SIZE];
    int front, rear, size;
} Queue;

/* 构造函数 */
Queue *newQueue() {
    Queue *q = (Queue *)malloc(sizeof(Queue));
    q->front = q->rear = q->size = 0;
    return q;
}

/* 判断队列是否为空 */
int isEmpty(Queue *q) {
    return q->size == 0;
}

/* 入队操作 */
void enqueue(Queue *q, Vertex *vet) {
    q->vertices[q->rear] = vet;
    q->rear = (q->rear + 1) % MAX_SIZE;
    q->size++;
}

/* 出队操作 */
Vertex *dequeue(Queue *q) {
    Vertex *vet = q->vertices[q->front];
    q->front = (q->front + 1) % MAX_SIZE;
    q->size--;
```

```c
    return vet;
}

/* 检查顶点是否已被访问 */
int isVisited(Vertex **visited, int size, Vertex *vet) {
    // 遍历查找节点，使用 O(n) 时间
    for (int i = 0; i < size; i++) {
        if (visited[i] == vet)
            return 1;
    }
    return 0;
}

/* 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
void graphBFS(GraphAdjList *graph, Vertex *startVet, Vertex **res, int *resSize, Vertex **visited, int
↪ *visitedSize) {
    // 队列用于实现 BFS
    Queue *queue = newQueue();
    enqueue(queue, startVet);
    visited[(*visitedSize)++] = startVet;
    // 以顶点 vet 为起点，循环直至访问完所有顶点
    while (!isEmpty(queue)) {
        Vertex *vet = dequeue(queue); // 队首顶点出队
        res[(*resSize)++] = vet;      // 记录访问顶点
        // 遍历该顶点的所有邻接顶点
        AdjListNode *node = findNode(graph, vet);
        while (node != NULL) {
            // 跳过已被访问的顶点
            if (!isVisited(visited, *visitedSize, node->vertex)) {
                enqueue(queue, node->vertex);             // 只入队未访问的顶点
                visited[(*visitedSize)++] = node->vertex; // 标记该顶点已被访问
            }
            node = node->next;
        }
    }
    // 释放内存
    free(queue);
}
```

代码相对抽象，建议对照图 9-10 来加深理解。


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
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
Step 2
顶点 1 出队，并执行以下操作
Step 3
顶点 3 出队，并执行以下操作
Step 4
顶点 2 出队，并执行以下操作
Step 5
顶点 4 出队，并执行以下操作
Step 6
顶点 6 出队，并执行以下操作
Step 7
顶点 5 出队，并执行以下操作
Step 8
顶点 7 出队，并执行以下操作
Step 9
顶点 8 出队，并执行以下操作
Step 10

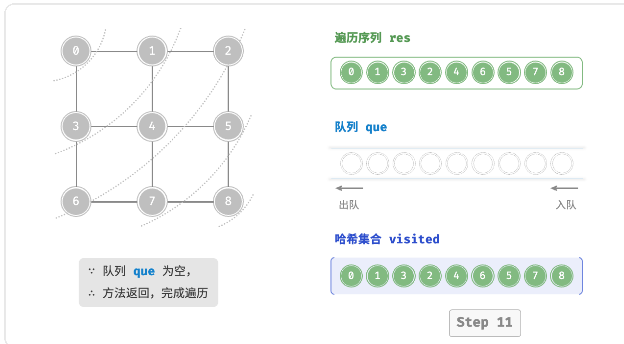


图 9-10　图的广度优先遍历步骤

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。

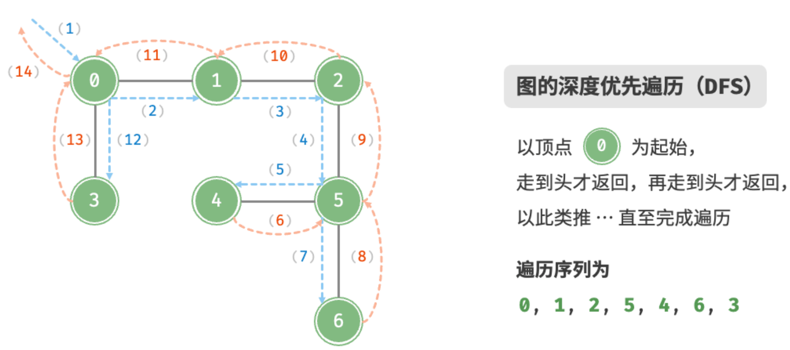


图 9-11　图的深度优先遍历

1. 算法实现

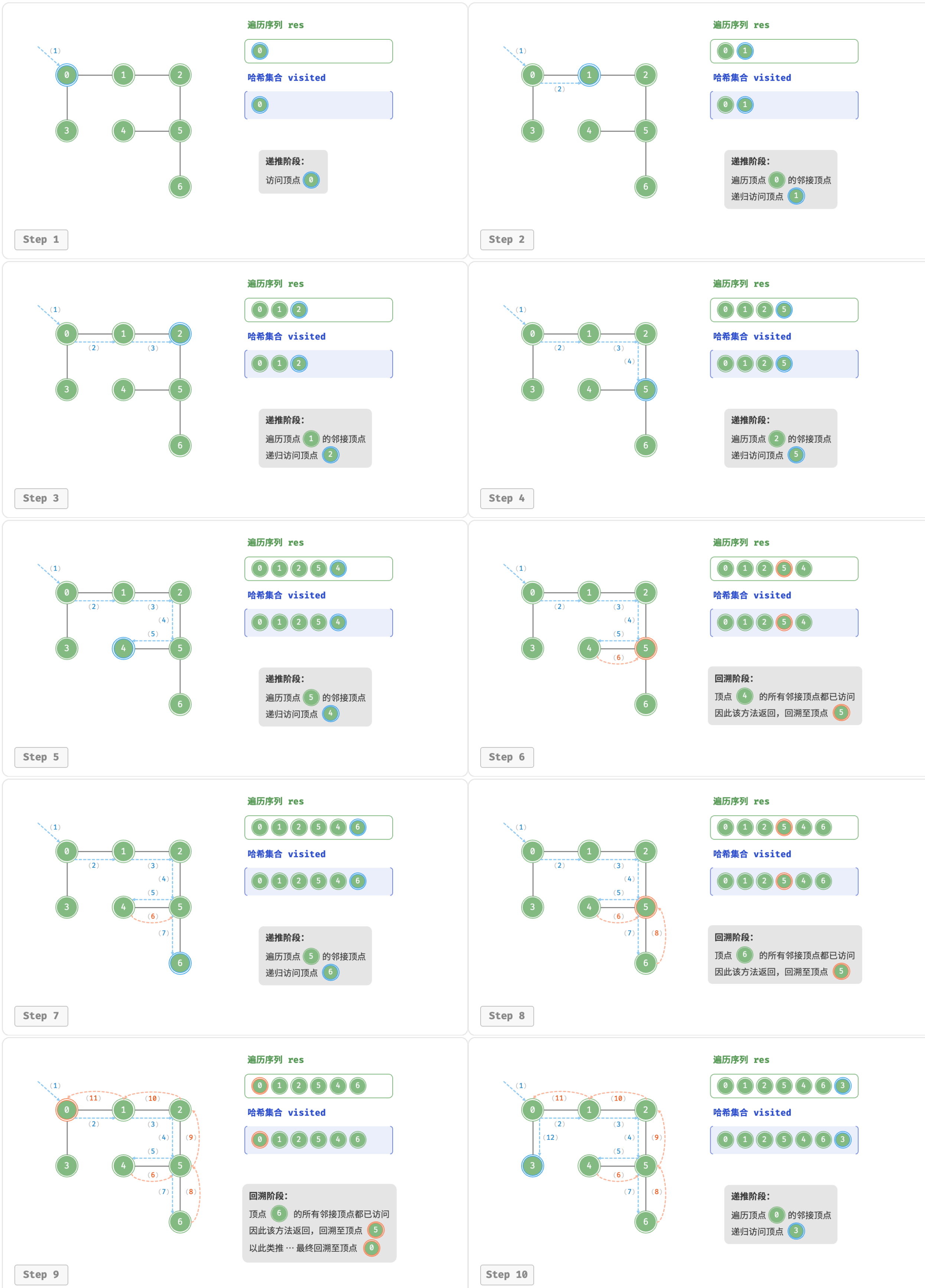
这种“走到尽头再返回”的算法范式通常基于递归来实现。与广度优先遍历类似，在深度优先遍历中，我们也需要借助一个哈希集合 `visited` 来记录已被访问的顶点，以避免重复访问顶点。

```c
// === File: graph_dfs.c ===

/* 检查顶点是否已被访问 */
int isVisited(Vertex **res, int size, Vertex *vet) {
    // 遍历查找节点，使用 O(n) 时间
    for (int i = 0; i < size; i++) {
        if (res[i] == vet) {
            return 1;
        }
    }
    return 0;
}

/* 深度优先遍历辅助函数 */
void dfs(GraphAdjList *graph, Vertex **res, int *resSize, Vertex *vet) {
    // 记录访问顶点
    res[(*resSize)++] = vet;
    // 遍历该顶点的所有邻接顶点
    AdjListNode *node = findNode(graph, vet);
    while (node != NULL) {
        // 跳过已被访问的顶点
        if (!isVisited(res, *resSize, node->vertex)) {
            // 递归访问邻接顶点
            dfs(graph, res, resSize, node->vertex);
        }
        node = node->next;
    }
}

/* 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
void graphDFS(GraphAdjList *graph, Vertex *startVet, Vertex **res, int *resSize) {
    dfs(graph, res, resSize, startVet);
}
```

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。


遍历序列 res
哈希集合 visited
递推阶段:
访问顶点 0
Step 1
遍历序列 res
哈希集合 visited
递推阶段:
遍历顶点 0 的邻接顶点
递归访问顶点 1
Step 2
遍历序列 res
哈希集合 visited
递推阶段:
遍历顶点 1 的邻接顶点
递归访问顶点 2
Step 3
遍历序列 res
哈希集合 visited
递推阶段:
遍历顶点 2 的邻接顶点
递归访问顶点 5
Step 4
遍历序列 res
哈希集合 visited
递推阶段:
遍历顶点 5 的邻接顶点
递归访问顶点 4
Step 5
遍历序列 res
哈希集合 visited
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
遍历序列 res
哈希集合 visited
递推阶段:
遍历顶点 5 的邻接顶点
递归访问顶点 6
Step 7
遍历序列 res
哈希集合 visited
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
遍历序列 res
哈希集合 visited
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
遍历序列 res
哈希集合 visited
递推阶段:
遍历顶点 0 的邻接顶点
递归访问顶点 3
Step 10

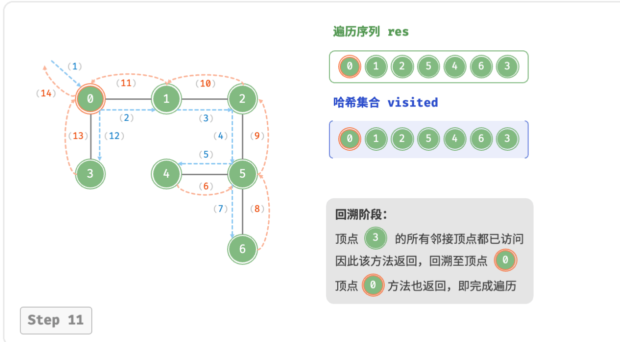


图 9-12　图的深度优先遍历步骤

> **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。

9.4　小结

1.　重点回顾

- 图由顶点和边组成，可以表示为一组顶点和一组边构成的集合。
- 相较于线性关系（链表）和分治关系（树），网络关系（图）具有更高的自由度，因而更为复杂。
- 有向图的边具有方向性，连通图中的任意顶点均可达，有权图的每条边都包含权重变量。
- 邻接矩阵利用矩阵来表示图，每一行（列）代表一个顶点，矩阵元素代表边，用 1 或 0 表示两个顶点之间有边或无边。邻接矩阵在增删查改操作上效率很高，但空间占用较多。
- 邻接表使用多个链表来表示图，第 i 个链表对应顶点 i ，其中存储了该顶点的所有邻接顶点。邻接表相对于邻接矩阵更加节省空间，但由于需要遍历链表来查找边，因此时间效率较低。
- 当邻接表中的链表过长时，可以将其转换为红黑树或哈希表，从而提升查询效率。
- 从算法思想的角度分析，邻接矩阵体现了“以空间换时间”，邻接表体现了“以时间换空间”。
- 图可用于建模各类现实系统，如社交网络、地铁线路等。
- 树是图的一种特例，树的遍历也是图的遍历的一种特例。
- 图的广度优先遍历是一种由近及远、层层扩张的搜索方式，通常借助队列实现。
- 图的深度优先遍历是一种优先走到底、无路可走时再回溯的搜索方式，常基于递归来实现。

2. Q & A

Q：路径的定义是顶点序列还是边序列？

维基百科上不同语言版本的定义不一致：英文版是“路径是一个边序列”，而中文版是“路径是一个顶点序列”。以下是英文版原文：In graph theory, a path in a graph is a finite or infinite sequence of edges which joins a sequence of vertices.

在本文中，路径被视为一个边序列，而不是一个顶点序列。这是因为两个顶点之间可能存在多条边连接，此时每条边都对应一条路径。

Q：非连通图中是否会有无法遍历到的点？

在非连通图中，从某个顶点出发，至少有一个顶点无法到达。遍历非连通图需要设置多个起点，以遍历到图的所有连通分量。

Q：在邻接表中，“与该顶点相连的所有顶点”的顶点顺序是否有要求？

可以是任意顺序。但在实际应用中，可能需要按照指定规则来排序，比如按照顶点添加的次序，或者按照顶点值大小的顺序等，这样有助于快速查找“带有某种极值”的顶点。