

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。

初始化邻接矩阵

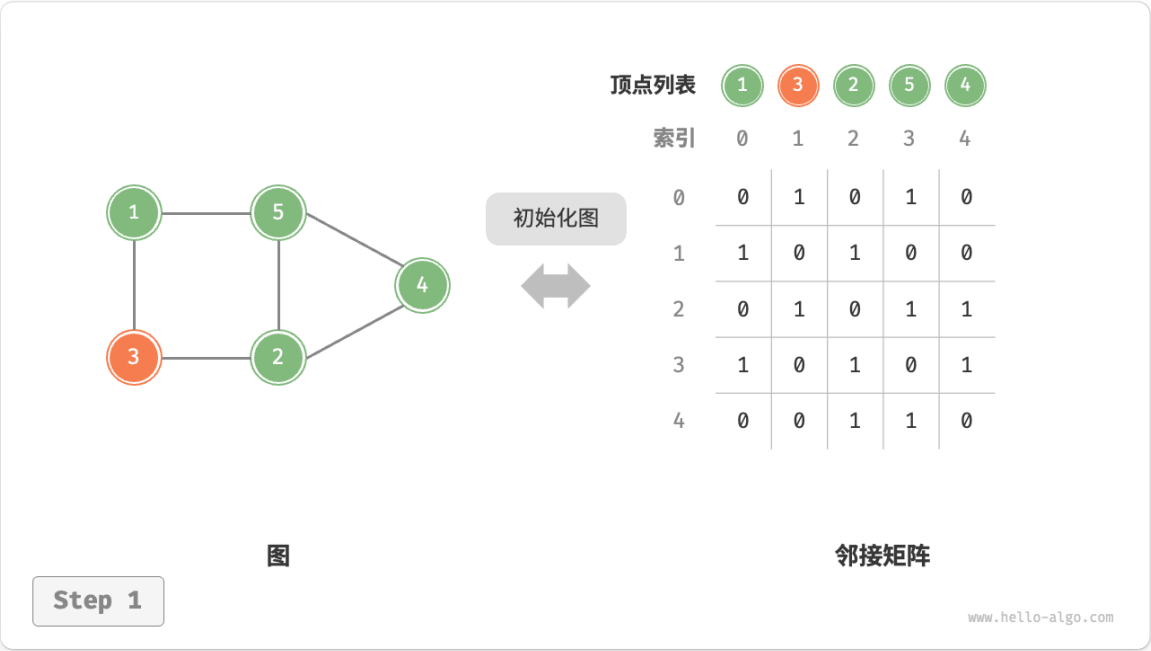


图 9-7 邻接矩阵的初始化、增删边、增删顶点

以下是基于邻接矩阵表示图的实现代码：

Python

graph_adjacency_matrix.py

```
class GraphAdjMat:
    """基于邻接矩阵实现的无向图类"""

    def __init__(self, vertices: list[int], edges: list[list[int]]):
        """构造方法"""
        # 顶点列表，元素代表“顶点值”，索引代表“顶点索引”
        self.vertices: list[int] = []
        # 邻接矩阵，行列索引对应“顶点索引”
        self.adj_mat: list[list[int]] = []
        # 添加顶点
        for val in vertices:
            self.add_vertex(val)
        # 添加边
        # 请注意，edges 元素代表顶点索引，即对应 vertices 元素索引
        for e in edges:
            self.add_edge(e[0], e[1])

    def size(self) -> int:
        """获取顶点数量"""
        return len(self.vertices)

    def add_vertex(self, val: int):
        """添加顶点"""
        n = self.size()
        # 向顶点列表中添加新顶点的值
        self.vertices.append(val)
        # 在邻接矩阵中添加一行
        new_row = [0] * n
        self.adj_mat.append(new_row)
        # 在邻接矩阵中添加一列
        for row in self.adj_mat:
            row.append(0)

    def remove_vertex(self, index: int):
        """删除顶点"""
        if index >= self.size():
            raise IndexError()
        # 在顶点列表中移除索引 index 的顶点
        self.vertices.pop(index)
        # 在邻接矩阵中删除索引 index 的行
        self.adj_mat.pop(index)
        # 在邻接矩阵中删除索引 index 的列
        for row in self.adj_mat:
            row.pop(index)

    def add_edge(self, i: int, j: int):
        """添加边"""
```

```

        # 参数 i, j 对应 vertices 元素索引
        # 索引越界与相等处理
        if i < 0 or j < 0 or i >= self.size() or j >= self.size() or i ==
j:
            raise IndexError()
        # 在无向图中, 邻接矩阵关于主对角线对称, 即满足 (i, j) == (j, i)
        self.adj_mat[i][j] = 1
        self.adj_mat[j][i] = 1

    def remove_edge(self, i: int, j: int):
        """删除边"""
        # 参数 i, j 对应 vertices 元素索引
        # 索引越界与相等处理
        if i < 0 or j < 0 or i >= self.size() or j >= self.size() or i ==
j:
            raise IndexError()
        self.adj_mat[i][j] = 0
        self.adj_mat[j][i] = 0

    def print(self):
        """打印邻接矩阵"""
        print("顶点列表 =", self.vertices)
        print("邻接矩阵 =")
        print_matrix(self.adj_mat)

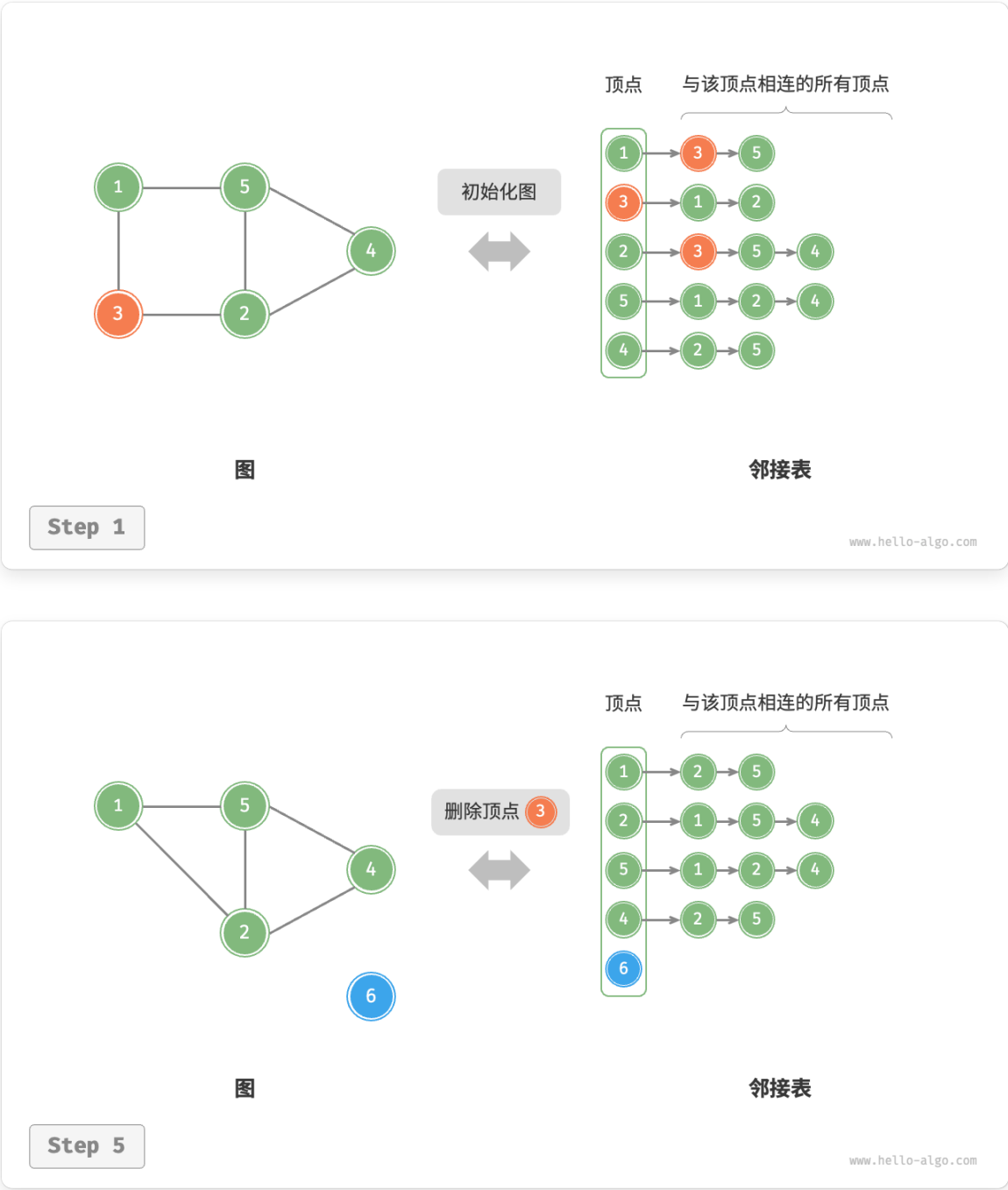
```

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。

初始化邻接表



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。

Python

graph_adjacency_list.py

```
class GraphAdjList:
    """基于邻接表实现的无向图类"""

    def __init__(self, edges: list[list[Vertex]]):
        """构造方法"""
        # 邻接表, key: 顶点, value: 该顶点的所有邻接顶点
        self.adj_list = dict[Vertex, list[Vertex]]()
        # 添加所有顶点和边
        for edge in edges:
            self.add_vertex(edge[0])
            self.add_vertex(edge[1])
            self.add_edge(edge[0], edge[1])

    def size(self) -> int:
        """获取顶点数量"""
        return len(self.adj_list)

    def add_edge(self, vet1: Vertex, vet2: Vertex):
        """添加边"""
        if vet1 not in self.adj_list or vet2 not in self.adj_list or vet1
        == vet2:
            raise ValueError()
        # 添加边 vet1 - vet2
        self.adj_list[vet1].append(vet2)
        self.adj_list[vet2].append(vet1)

    def remove_edge(self, vet1: Vertex, vet2: Vertex):
        """删除边"""
        if vet1 not in self.adj_list or vet2 not in self.adj_list or vet1
        == vet2:
            raise ValueError()
        # 删除边 vet1 - vet2
        self.adj_list[vet1].remove(vet2)
        self.adj_list[vet2].remove(vet1)

    def add_vertex(self, vet: Vertex):
        """添加顶点"""
        if vet in self.adj_list:
            return
        # 在邻接表中添加一个新链表
        self.adj_list[vet] = []

    def remove_vertex(self, vet: Vertex):
        """删除顶点"""
```

```

    if vet not in self.adj_list:
        raise ValueError()
    # 在邻接表中删除顶点 vet 对应的链表
    self.adj_list.pop(vet)
    # 遍历其他顶点的链表，删除所有包含 vet 的边
    for vertex in self.adj_list:
        if vet in self.adj_list[vertex]:
            self.adj_list[vertex].remove(vet)

def print(self):
    """打印邻接表"""
    print("邻接表 =")
    for vertex in self.adj_list:
        tmp = [v.val for v in self.adj_list[vertex]]
        print(f"{vertex.val}: {tmp},")

```

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。