图（Graph）数据结构

杨其臻

Jul 23, 2025

图作为一种强大的数据结构，在现实世界中无处不在。社交网络中的人际关系、地图导航中的路径规划、推荐系统中的用户行为建模，都依赖于图的抽象能力。在计算机科学领域，图常被视为数据结构的“天花板”之一，因为它能高效处理复杂的关联关系。本文的目标是系统讲解图的核心概念，包括图的定义、分类和存储方式，并手把手实现两种基础存储结构：邻接矩阵和邻接表。我们还将实现关键算法如深度优先搜索（DFS）和广度优先搜索（BFS），分析其适用场景，并提供完整的 Python 代码示例。通过这些内容，读者将构建对图的系统认知，为实际项目应用奠定基础。

1　图的基础理论

图在数学和编程中的核心定义是一致的：它由顶点（Vertex 或 Node）和边（Edge）组成。抽象表示为 $G = (V, E)$，其中 V 是顶点集合，E 是边集合。例如，在社交网络中，用户是顶点，好友关系是边；在地图导航中，地点是顶点，道路是边。图的分类基于多个维度：有向图与无向图的区别在于边是否有方向性（如网页链接是有向的，好友关系通常是无向的）；加权图与无权图则涉及边是否携带权重（如导航距离是加权，好友关系是无权）。关键概念包括连通图（所有顶点相互可达）、环（路径形成闭环）、度（顶点的连接数，入度/出度用于有向图），以及稀疏图与稠密图（边数远少于或接近顶点数的平方），这些概念直接影响存储结构的选择。

2　图的存储结构：如何用代码表示图?

图的存储结构决定了算法的效率和适用性。邻接矩阵使用二维数组表示顶点间的边关系。例如，`matrix[i][j]` 存储顶点 `i` 到 `j` 的边权重。以下 Python 实现展示其核心逻辑：

```
class GraphMatrix:
    def __init__(self, num_vertices):
        self.matrix = [[0] * num_vertices for _ in range(num_vertices)]

    def add_edge(self, v1, v2, weight=1):
        self.matrix[v1][v2] = weight
        # 若是无向图，需对称添加：self.matrix[v2][v1] = weight
```

在解读这段代码时，`__init__` 方法初始化一个大小为 $V \times V$ 的矩阵（V 是顶点数），所有元素初始化为 0 表示无边。`add_edge` 方法添加边：参数 `v1` 和 `v2` 指定起点和终点，`weight` 默认为 1（无权图）。关键点在于注释部分：对于无向图，必须对称设置 `matrix[v2][v1]`，以确保双向关系。邻接矩阵的空间复杂度为 $O(V^2)$，适

合稠密图；优点是快速（$O(1)$）判断边存在性；缺点是空间浪费于稀疏图，且添加/删除顶点需重建矩阵，成本较高。

邻接表则更高效地处理稀疏图，使用数组加链表结构，每个顶点维护其邻接点列表。以下实现展示核心逻辑：

```
class GraphList:
    def __init__(self, num_vertices):
        self.adj_list = [[] for _ in range(num_vertices)]

    def add_edge(self, v1, v2, weight=1):
        self.adj_list[v1].append((v2, weight)) # 存储目标节点和权重
        # 无向图需反向添加：self.adj_list[v2].append((v1, weight))
```

代码解读：`__init__` 创建大小为 V 的列表，每个元素是空列表。`add_edge` 将目标顶点 `v2` 和权重添加到 `v1` 的邻接表中；注释强调无向图需反向添加以保持对称。邻接表空间复杂度为 $O(V+E)$（E 是边数），查询邻接点仅需 $O(\text{degree}(V))$（顶点的度数），但无法 $O(1)$ 判断任意边存在性。总结对比：邻接矩阵在稠密图中优势明显，而邻接表在稀疏图中更节省空间；添加顶点时，邻接表开销低；遍历邻接点时，邻接表基于度数更高效。

3 图的遍历：探索图的基础算法

遍历算法是图分析的基石。深度优先搜索（DFS）采用“一条路走到底”的策略，使用栈实现回溯机制。它适用于拓扑排序、连通分量检测等场景。以下是基于邻接表的 Python 实现：

```
def dfs(graph, start):
    visited = set()
    stack = [start]
    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            visited.add(vertex)
            for neighbor, _ in graph.adj_list[vertex]:
                if neighbor not in visited:
                    stack.append(neighbor)
    return visited
```

代码解读：`visited` 集合记录已访问顶点，避免重复。`stack` 初始化为起始点；循环中弹出顶点，若未访问则标记，并遍历其邻接点（`graph.adj_list[vertex]` 返回邻接列表）。关键点在于 `stack.append(neighbor)` 将未访问邻居压栈，确保深度优先。时间复杂度为 $O(V+E)$，空间复杂度 $O(V)$。

广度优先搜索（BFS）采用“层层推进”策略，使用队列实现最短路径基础。它适用于无权图最短路径或社交网络扩展。实现如下：

```
from collections import deque
def bfs(graph, start):
    visited = set([start])
```

```
    queue = deque([start])
    while queue:
        vertex = queue.popleft()
        for neighbor, _ in graph.adj_list[vertex]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
    return visited
```

解读：visited 和队列初始包含起始点。循环中，queue.popleft() 取出顶点，遍历其邻接点；未访问邻居被标记并加入队列尾部（queue.append(neighbor)）。这保证了层级遍历，时间复杂度同样 $O(V+E)$。BFS 与 DFS 的核心差异在于遍历顺序：BFS 找到最短路径（无权图），DFS 更易检测环或深度结构。

4 图的应用场景与进阶算法预告

图在工程中广泛应用。路径规划依赖 Dijkstra 算法（加权最短路径）或 A*算法（启发式搜索）；网络分析中，PageRank 算法通过链接结构评估网页重要性；社交网络使用 BFS 扩展好友推荐（如三级好友关系）。这些应用凸显图的建模能力。进阶方向包括最小生成树算法（如 Prim 或 Kruskal 用于网络优化）、拓扑排序（处理任务依赖）、以及强连通分量算法（如 Tarjan 算法分析子图结构）。掌握这些算法能解决复杂问题如芯片布线或数据流分析。

本文系统回顾了图的关键点：邻接矩阵和邻接表作为存储结构，需根据图类型（稠密或稀疏）选择；DFS 和 BFS 是基础遍历算法，前者适合深度探索，后者用于最短路径。图的重要性在于其普适性：从微观的芯片设计到宏观的宇宙网络建模，都能高效处理关联关系。建议下一步在项目中实践，如使用 NetworkX 图库或挑战 LeetCode 题目（如课程调度问题），以深化理解。