数据结构与算法

Data Structure and Algorithm

极夜酱

目录

Part I

基础篇

Chapter 1 图

1.1 图

1.1.1 图（Graph）

你的微信中有若干好友，而你的好友又有若干好友。许许多多的用户组成了一个多对多的关系网，这个关系网就是数据结构中的图。

再例如使用地图导航功能时，导航会根据你的出发地和目的地规划最佳的地铁换乘路线。许许多多的地铁站组成的交通网络也可以认为是图。

图是一种比树更为复杂的数据结构。树的结点之间是一对多的关系，并且存在父与子的层级划分。而图的顶点之间是多对多关系，并且所有顶点都是平等的，无所谓谁是父子。

在图中，最基本的单元是顶点（vertex），相当于树中的结点。顶点之间的关联关系被称为边（edge）。图中包含一组顶点和一组边，通常用 V 表示顶点集合，用 E 表示边集合。边可以看作是顶点对，即 $(v, w) \in E$，$v, w \in V$。

在有些图中，每一条边并不是完全等同的。例如地铁线路，站与站之间的距离都有可能不同。因此图中会涉及边的权重（weight），涉及到权重的图被称为带权图（weighted graph），也称为网络。

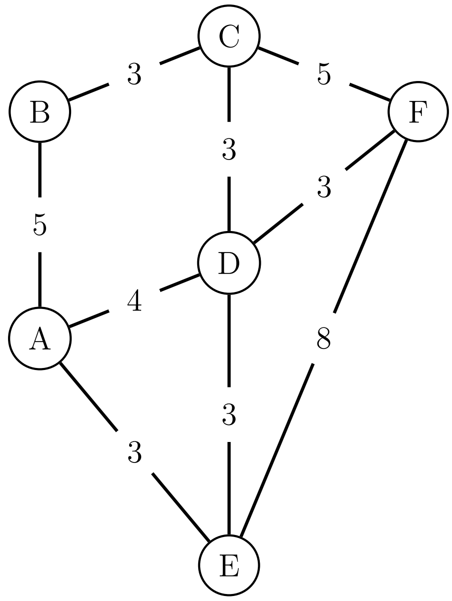


图 1.1: 带权图

还有一种图，顶点之间的关联并不是完全对称的。拿微信举例，你的好友列表里有我，但我的好友列表里未必有你。

这样一来，顶点之间的边就有了方向的区分，这种带有方向的图被称为有向图（directed graph）。有向边可以使用 <v, w> 表示从 v 指向 w 的边。

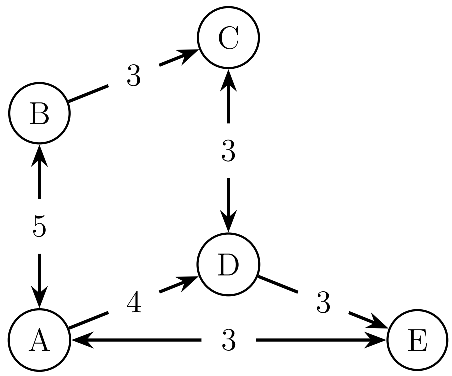


图 1.2: 有向图

相应地，在 QQ 中，只要我把你从好友里删除，你在自己的好友列表里就看不到我了。因此 QQ 的好友关系可以认为是一个没有方向区分的图，这种图被称为无向图（undirected graph）。

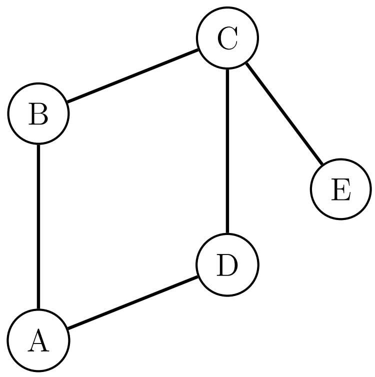
1.2 图的表示

1.2.1 邻接矩阵（Adjacency Matrix）

拥有 n 个顶点的图，它所包含的边的数量最多是 n(n-1) 条，因此，要表达各个顶点之间的关联关系，最清晰易懂的方式是使用邻接矩阵 G[N][N]。

对于无向图来说，如果顶点之间有关联，那么邻接矩阵中对应的值为 1；如果顶点之间没有关联，那么邻接矩阵中对应的值为 0。

$$G[i][j] = \begin{cases} 1 & < v_i, v_j > \text{是 G 中的边} \\ 0 & < v_i, v_j > \text{不是 G 中的边} \end{cases}$$



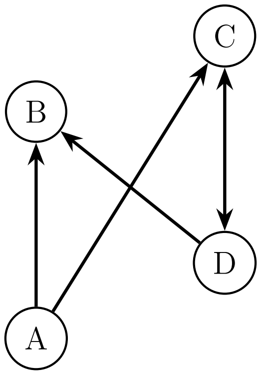
	A	**B**	**C**	**D**	**E**
A	0	1	0	1	0
B	1	0	1	0	0
C	0	1	0	1	1
D	1	0	1	0	0
E	0	0	1	0	0

表 1.1: 无向图邻接矩阵

需要注意的是，邻接矩阵从左上到右下的一条对角线上的元素值必然是 0，因为任何一个顶点与它自身是没有连接的。同时，无向图对应的邻接矩阵是一个对称

矩阵，假如 A 和 B 有关联，那么 B 和 A 也必定有关联。

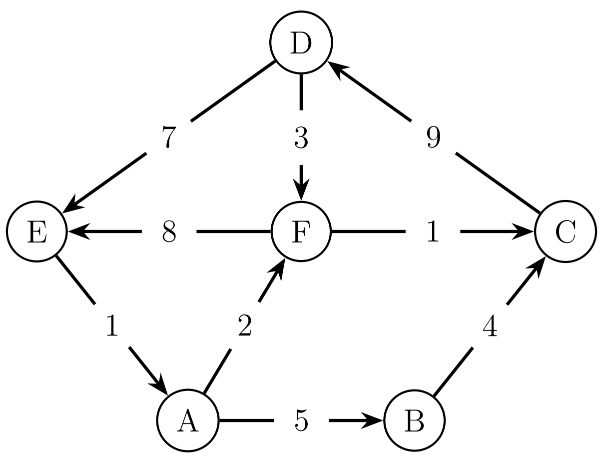
但是对于有向图的邻接矩阵，不一定是一个对称矩阵，假如 A 可以达到 B，从 B 未必能达到 A。



	A	**B**	**C**	**D**
A	0	1	1	0
B	0	0	0	0
C	0	0	0	1
D	0	1	1	0

表 1.2: 有向图邻接矩阵

对于网络，只要把邻接矩阵对应位置的值定义为边 $< v_i, v_j >$ 的权重即可。



	A	**B**	**C**	**D**	**E**	**F**
A	∞	5	∞	∞	∞	2
B	∞	∞	4	∞	∞	∞
C	∞	∞	∞	9	∞	∞
D	∞	∞	∞	∞	7	3
E	1	∞	∞	∞	∞	∞
F	∞	∞	1	∞	8	∞

表 1.3: 带权图邻接矩阵

对于带权图，如果 v_i 和 v_j 之前没有边应该将权值设为 ∞。

邻接矩阵的优点：

1. 简单、直观。
2. 可以快速查到一个顶点和另一顶点之间的关联关系。
3. 方便计算任一顶点的度，对于有向图，从顶点发出的边数为出度，指向顶点的边数为入度。

邻接矩阵的缺点：

1. 浪费空间，对于稀疏图（点很多而边很少）有大量无效元素。但对于稠密图（特别是完全图）还是很合算的。
2. 浪费时间，统计稀疏图中边的个数，也就是计算邻接矩阵中元素 1 的个数。

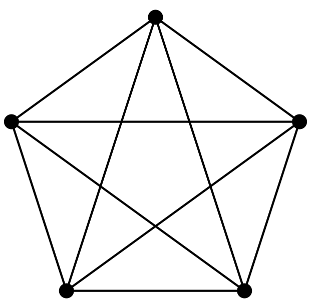
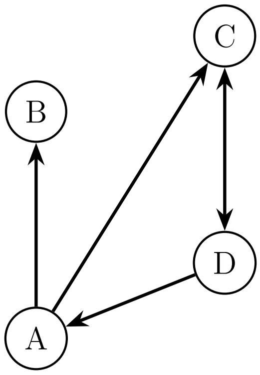


图 1.3: 完全图

1.2.2 邻接表（Adjacency List）

为了解决邻接矩阵占用空间的问题，人们想到了另一种图的表示方法——邻接表。在邻接表中，图的每一个顶点都是一个链表的头结点，其后连接着该顶点能够直接到达的相邻顶点。对于稀疏图而言，邻接表存储方式占用的空间比邻接矩阵要小得多。



0 → 1 → 2
1 →
2 → 3
3 → 1 → 2

图 1.4: 邻接表

通过遍历邻接表可以查找到所有能够到达的相邻顶点，但是对于逆向查找，即哪些顶点可以达到一个顶点就会很麻烦。

逆邻接表和邻接表是正好相反的，逆邻接表每一个顶点作为链表的头结点，后继结点所存储的是能够直接到达该顶点的相邻顶点。

0 →
1 → 0 → 3
2 → 0 → 3
3 → 2

图 1.5: 逆邻接表

可是，一个图要维护正反两个邻接表，也太麻烦了吧?

通过十字链表可以把邻接表和逆邻接表结合在一起。

$$
\begin{array}{r|c|l}
\hline
\leftarrow & 0 & \rightarrow 1 \rightarrow 2 \\
\hline
3 \leftarrow 0 \leftarrow & 1 & \rightarrow \\
\hline
3 \leftarrow 0 \leftarrow & 2 & \rightarrow 3 \\
\hline
2 \leftarrow & 3 & \rightarrow 1 \rightarrow 2 \\
\hline
\end{array}
$$

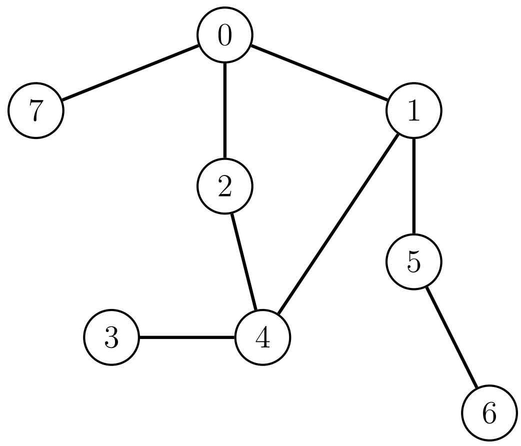
图 1.6: 十字链表

1.3 图的遍历

1.3.1 深度优先搜索（DFS, Depth First Search）

深度优先搜索是一种一头扎到底的遍历方法，选择一条支路，尽可能不断地深入，如果遇到死路就回退，回退过程中如果遇到没探索的支路，就进入该支路继续深入。

例如有一个小镇，你知道小镇的每个地方与每条路。小镇的每个地方都藏有可以实现愿望的光玉，现在你要出发去收集小镇上所有的光玉。你的出生点在 0 号位置，你需要一个地点都不遗漏地走完整个小镇，才能收集完所有光玉。



二叉树的先序遍历本质上也可以认为是图的深度优先遍历。要想实现回溯，可以利用栈的先进后出的特性，也可以采用递归的方式，因为递归本身就是基于方法调用栈来实现的。

Algorithm 1 深度优先搜索

```
procedure DFS(Vertex V)
    isVisited[V] = true
    for v in V do
        if !isVisited[v] then dfs(v)
        end if
    end for
end procedure
```

1.3.2 广度优先搜索（BFS, Breath First Search）

除了深度优先搜索一头扎到底的方法以外，还有一种方法就是首先把从源点相邻的顶点遍历，然后再遍历稍微远一点的顶点，再去遍历更远一点的顶点。

二叉树的层次遍历本质上也可以认为是图的广度优先遍历，需要借助队列来实现重放。