图的存储与遍历

dbywsc

2025/8

目录

1 图的相关概念

2 图的存储方式

3 图的遍历

4 拓扑排序

5 习题

图论 (Graph theory) 是数学的一个分支，图是图论的主要研究对象[1]。图 (Graph) 是由若干给定的顶点及连接两顶点的边构成的图形，这种图形通常用来描述某些事物之间的某种特定关系。顶点用于代表事物，连接两顶点的边则用于表示两个事物间具有这种关系。

[1] 在计算机科学，或者在算法竞赛中，图论的一些概念或者符号表示与数学中的图论不完全一样。

图是一个二元组 $G=(V(G),E(G))$ 。其中 $V(G)$ 是非空集，称为**点集** (vertex set)，对于 V 中点每个元素，我们称其为**顶点**(vertex) 或者**节点** (node)，简称为 **点**；$E(G)$ 为各点之间**边** (edge) 的集合，称为**边集** (edge set)。
常用 $G=(V,E)$ 来表示图。
图分为**无向图** (undirected graph)，**有向图** (directed graph) 和**混合图** (mixed graph)。
若 G 为无向图，则 E 中的每一个元素为一个无序二元组 (u,v)，称作**无向边** (undirected edge)，其中 $u,v\in V$。设 $e=(u,v)$，则称 u 和 v 为 e 的**端点** (endpoint)。
若 G 为有向图，则 E 中的每一个元素为一个有序二元组 (u,v)，称作**有向边** (directed edge) 或者 **弧** (arc)，有时也写作 $u\to v$。设 $e=u\to v$，则 u 为 e 的**起点** (tail)，v 为 e 的**终点** (head)。起点和终点称为 e 的端点。并称 u 为 e 的直接前驱，v 为 e 的直接后继。

若 G 为混合图，则其中既有有向边，又有无向边。
若 G 的每一条边 $e_k = (u_k, v_k)$ 都被赋予一个数作为这个边的**权**，则称 G 为**带权图**。如果这个图的权均为正实数，则称图为**正权图**。
图 G 的点数 $|V(G)|$ 称为图的**阶**。
与一个点 v 关联的边的条数称为**度**(degree)，记做 $d(v)$。特别地，对于边 (u, v)，则每条这样的边要对 $d(v)$ 产生 2 的贡献。

途径(walk)：是连接一连串顶点的点的序列，可以为有限或者无限长度。形式化地说，一条有限途径 w 是一个边的序列 $e_1, e_2, \dots, e_k$ ，是的存在一个顶点序列 $v_0, v_1, v_2, \dots v_k$ ，满足 $e = (v_{i-1}, v_i)$ ，其中，$i \in [1, k]$ 。这样的途径可以简写为 $v_0 \to v_1 \to v_2 \to \dots \to v_k$ 。通常来说，边的数量 k 被称为这条边的 **长度** (如果边是带权的，长度则为权重之和)。

迹 (trail)：对于一条途径 w , 若 $e_1, e_2, \dots, e_k$ 两两互不相同，则称 w 是一条迹。

简单路径 (simple path) （又称**路径** (path))：对于一条迹 w ，若其连接点的序列中点两两不同，则称 w 是一条路径。

回路 (circuit)：对于一条迹 w，若 $v_0 = v_k$，则称 w 是一条回路。

环/圈(cycle)（又称**简单回路/简单环** (simple curcuit))：对于一条回路 w，若 $v_0 = v_k$ 是点序列中唯一重复出现的点对，则称 w 是一个环。

自环(loop)：对 E 中对边 $e=(u,v)$，若 $u=v$，则 e 被称作自环。
重边(multiple edge)：若 E 中存在两个完全相同的元素（边）e_1, e_2，则它们被称作（一组）重边。
简单图 (simple graph)：若一张图中没有重边和自环，则它被称为简单图，否则，称它为 **多重图** (multi graph)。

通常有四种存图方式，分别为：直接存边，邻接矩阵，邻接表，链式前向星[2]。
本节中，用 n 指代图的点数，用 m 指代图的边数，用 $d^+(u)$ 指代点 u 的出度，即以 u 为出发点的边数。

[2]链式前向星是 OI 中的说法，它更正式的表达叫邻接链表

直接存边即定义一个结构体数组，结构体中存储边的属性，比如起点、终点、权值等。

```
struct Edge {
    int u, v, w;
};
std::vector<Edge> edges(m);
```

复杂度：
查询是否存在某条边：$O(m)$ 。
遍历一个点的所有出边：$O(m)$ 。
遍历整张图：$O(nm)$ 。
空间复杂度：$O(m)$ 。
直接存边的效率比较低下，一般不会使用这种做法。但是在 *Kruskal* 算法中，由于需要对所有的边按边权排序，因此需要使用直接存边。在某些题目中，如果涉及多次建图的操作，需要重复使用这些边，也可以先将所有的边存下来。

邻接矩阵即使用一个二维数组 adj 来存边，使用 $adj[u][v]=1$ 来标记存在一条边 $e=(u,v)$，如果为 0 则表示不存在，如果是带权边，则可以使用 $adj[u][v]=w$ 表示存在一条边 $u\to v$ ，且边权为 w 。对于无向边，我们直接令 $adj[u][v]=1$ 并且 $adj[v][u]=1$，即存储两条两个方向的边。
复杂度:
查询是否存在某条边: $O(1)$ 。
遍历一个点的所有出边： $O(n)$ 。
遍历整张图： $O(n^2)$ 。
空间复杂度： $O(n^2)$ 。
邻接矩阵无法处理需要存储重边的情况。它的优势是可以 $O(1)$ 地查询一条边是否存在，但是总体来说，它的时空复杂度都很高。只能在小规模内使用，并且一般用于处理稠密图[3]。在应用上， *Floyd* 算法需要依赖邻接矩阵。

[3]边的条数 $|E|$ 远远小于 $|v|^2$ 的图称为**稀疏图**，如果两者接近，则称为**稠密图**。

邻接表的实现是使用一个支持动态增加元素的数据结构构成的数组 (例如 $std :: vector < int > adj[n+1]$) 来存边，其中 $adj[u]$ 存储了点 u 的所有出边。
对于无向边，显然可以通过存储两条有向边的方式实现。
对于带权图，我们可以额外定义一个表示边的数据结构，存储终点和权值，可以使用 $std :: pair < int, int >$ 来实现。

存储无权图：

```cpp
std::vector<int> adj[N];
int main(void) {
    for(int i = 1; i <= m; i++) {
        int u, v; std::cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
}
```

存储带权图：

```cpp
typedef std::pair<int, int> PII;
std::vector<PII> adj[N];
int main(void) {
    for(int i = 1; i <= m; i++) {
        int u, v, w; std::cin >> u >> v >> w;
        G[u].emplace_back(v, w);
    }
}
```

复杂度:

查询是否存在 u 到 v 的边: $O(d^+(u))$ （如果实现做了排序，可以使用二分查找，时间复杂度为 $O(\log(d^+(u)))$ ）。

遍历点 u 点所有出边： $O(d^+(u))$ 。

遍历整张图： $O(n+m)$ 。

空间复杂度： $O(m)$ 。

相对来说，领接表的时空复杂度都很优秀，可以处理大部分问题，接下来的所有图论算法我们都将使用领接表演示。

链式前向星（邻接链表）本质上是使用链表实现的领接表。在 *OI* 中被广泛使用。
复杂度：
查询是否存在 u 到 v 的边：$O(d^+(u))$。
遍历点 u 点所有出边：$O(d^+(u))$。
遍历整张图：$O(n+m)$。
空间复杂度：$O(m)$。
由于使用了静态链表实现，常数上来说速度稍快于邻接表。
同样能够处理绝大部分问题，由于存边时带了编号，有时会非常有用，比如在网络流算法中就会使用这种写法。同时，在 *Java* 等常数较慢的语言中是最佳的选择。[4]

[4]对这种方法感兴趣的同学可以查阅 OI-wiki

```
// head[u] 和 cnt 的初始值都为 -1
void add(int u, int v) {
  nxt[++cnt] = head[u];  // 当前边的后继
  head[u] = cnt;         // 起点 u 的第一条边
  to[cnt] = v;           // 当前边的终点
}

// 遍历 u 的出边
for (int i = head[u]; ~i; i = nxt[i]) {  // ~i 表示 i != -1
  int v = to[i];
}
```

之前我们已经接触过了 *DFS* 和 *BFS* 算法，事实上，这两个算法本来就是一种图上遍历的算法。

与搜索中的 *DFS* 不同，图上 *DFS* 是有对应模版的：

```
void dfs(int u) {
    vis[u] = true;
    for(auto v : G[u]) {
        if(!vis[v]) dfs(v);
    }
}
```

DFS 的时间复杂度为 $O(m+n)$ ，空间复杂度为 $O(n)$ ，同时，由于 *DFS* 递归依赖栈空间，因此，栈空间的空间复杂度也为 $O(n)$ 。
对于一张图，对其进行 *DFS* 遍历，得到的节点的顺序称为 **DFS 序** 。
对于联通图来说，DFS 序通常不唯一。

BFS 是图论中最重要、最基础的算法之一。事实上，之前我们就已经给出过图上 *BFS* 的模版。

```
void bfs(int start) {
    queue<int> q;
    q.push(start); vis[start] = 1;
    while(q.size()) {
        auto u = q.front(); q.pop();
        for(auto v : G[u]) {
            if(!vis[v]) {
                vis[v] = 1; q.push(v);
            }
        }
    }
}
```

时间复杂度为 $O(n+m)$ ，空间复杂度为 $O(n)$ 。相对于 *DFS* 序，*BFS* 过程中访问到的节点的顺序叫做 *BFS* 序，*BFS* 序通常不唯一。

现在，来练习一道模版题：P5318 【深基 18. 例 3】查找文献

拓扑排序(Topological sorting) 要解决的问题是如何给一个**有向无环图**（简称 DAG）排序。

与传统意义上的排序不同，这里的排序指的是针对节点之间的依赖关系做一个线性的排序。比如要想学习算法竞赛，你需要学习编程语言、高等数学、离散数学、数据结构等等，它们之间的依赖关系是，先学会了高等数学，才能学习离散数学，先学习了编程语言，才能学习数据结构，而离散数学和数据结构之间又存在关联...... 如果我们把每个课程看作一个点，把它们之间的关系作为边，那我们刚刚就相当于是在做拓扑排序。

因此我们可以说，在一个 DAG 中，我们将途中的顶点以线性的方式进行排序，使得对任何一个顶点 u 到 v 的有向边 (u, v)，都可以有 u 在 v 的前面。

还有给定一个 DAG，如果从 i 到 j 有边，则认为 j 依赖于 i。如果 i 到 j 有路径（i 可达 j），则称 j 间接依赖于 i。

拓扑排序的目标是将所有节点排序，使得排在前面的节点不能依赖于排在后面的节点。

构造拓扑排序的步骤是:
1. 从图中的选择一个入度为 0 [5] 的点。
2. 输出并删除这个点，连带删除这个点所有的出边。
拓扑排序的实现有 *DFS* 和 *BFS* 两种，我们在这里只介绍 *BFS* 实现。
BFS 实现的拓扑排序又叫 *Kahn* 算法，初始状态下，集合 S 装着所有入度为 0 的点，L 是一个空列表。每次从 S 中取出一个点（可以随便去）u 放入 L，然后将 u 和 u 的所有出边删除，此时可能会有新的点入度变成了 0 ，将它们也放入 S 中。不断重复以上过程，直到 S 为空，如果此时图中还存在边，说明存在环路，这张图并非 DAG ，因此拓扑序不存在。否则，此时依次输出 L 中的点，就是要求的拓扑序。
时间复杂度：由于本质上是一个 *BFS* ，因此复杂度为 $O(E+V)$ 。

[5] 如果存在一条有向边 $u \to v$ ，那么这条边为 u 增加了 1 的出度，为 v 增加了 1 的入度。

```cpp
bool toposort(void) {
    std::vector<int> L;
    std::queue<int> s;
    for(int i = 1; i <= n; i++)
        if(in[i] == 0) s.push(i);
    while(s.size()) {
        auto u = s.front(); s.pop();
        L.push_back(u);
        for(auto v : G[u]) {
            if(--in[v] == 0) s.push(v);
        }
    }
    if(L.size() == n) {
        for(auto i : L) std::cout << i << " ";
        return true;
    }
    return false;
}
```

B3644 【模板】拓扑排序 / 家谱树 s

P3916 图的遍历
P2853 [USACO06DEC] Cow Picnic S
P1347 排序
P1983 [NOIP 2013 普及组] 车站分级
P1038 [NOIP 2003 提高组] 神经网络
P1807 最长路
P4017 最大食物链计数