Lec 1. 图的概念和存储结构

图论入门与进阶课程

by Alan233

Outline

1. 图论相关概念

1.1 图 (Graph)

图是一个二元组 $G = (V, E)$, 其中 V 称为**点集** (Vertex Set), E 称为**边集** (Edge Set).

1.1 图 (Graph)

图是一个二元组 $G = (V, E)$, 其中 V 称为**点集** (Vertex Set), E 称为**边集** (Edge Set).

对于 V 中的每个元素, 我们称其为**顶点** (Vertex) 或**节点** (Node).

1.1 图 (Graph)

图是一个二元组 $G = (V, E)$, 其中 V 称为**点集** (Vertex Set), E 称为**边集** (Edge Set).

对于 V 中的每个元素, 我们称其为**顶点** (Vertex) 或**节点** (Node).

图一般分为三种：**无向图** (Undirected Graph)、**有向图** (Directed Graph)、**混合图** (Mixed Graph).

1.1 图 (Graph)

图是一个二元组 $G = (V, E)$, 其中 V 称为**点集** (Vertex Set), E 称为**边集** (Edge Set).

对于 V 中的每个元素, 我们称其为**顶点** (Vertex) 或**节点** (Node).

图一般分为三种：**无向图** (Undirected Graph)、**有向图** (Directed Graph)、**混合图** (Mixed Graph).

- **无向图**：E 中的每个元素 $e = (u, v)$, 称为**无向边** (Undirected Edge), 其中 $u, v \in V$ 且无序, 并称 u, v 为 e 的**端点** (endpoint).

1.1 图 (Graph)

图是一个二元组 $G = (V, E)$, 其中 V 称为**点集** (Vertex Set), E 称为**边集** (Edge Set).

对于 V 中的每个元素, 我们称其为**顶点** (Vertex) 或**节点** (Node).

图一般分为三种：**无向图** (Undirected Graph)、**有向图** (Directed Graph)、**混合图** (Mixed Graph).

- **无向图**：E 中的每个元素 $e = (u, v)$, 称为**无向边** (Undirected Edge), 其中 $u, v \in V$ 且无序, 并称 u, v 为 e 的**端点** (endpoint).
- **有向图**：E 中的每个元素 $e = (u \to v)$, 称为**有向边** (Directed Edge), 其中 $u, v \in V$, 并称 u 为 e 的**起点** (head), v 为 e 的**终点** (tail).

1.2 相邻 (Adjacent)

1.2 相邻 (Adjacent)

在无向图 $G = (V, E)$ 中, 若点 v 是边 e 的**端点**, 则称 v 和 e 是**关联的** (incident). 对于两点 u 和 v, 若边 $(u, v) \in E$, 则称 u 和 v 是**相邻的** (adjacent).

1.2 相邻 (Adjacent)

在无向图 $G = (V, E)$ 中, 若点 v 是边 e 的**端点**, 则称 v 和 e 是**关联的** (incident). 对于两点 u 和 v, 若边 $(u, v) \in E$, 则称 u 和 v 是**相邻的** (adjacent).

一个顶点 $v \in V$ 的**邻域** (neighborhood) 是所有与点 v 相邻的顶点集合, 记作 N(v).

1.3 简单图 (Simple Graph)

1.3 简单图 (Simple Graph)

首先, 我们定义两个术语:

1.3 简单图 (Simple Graph)

首先, 我们定义两个术语:

- **自环** (loop): 若 $e = (v, v) \in E$, 称 e 为自环.

1.3 简单图 (Simple Graph)

首先, 我们定义两个术语:

- **自环** (loop): 若 $e = (v, v) \in E$, 称 e 为自环.
- **重边** (multiple edge): 若 E 中存在两个完全相同的元素 e_1, e_2, 称它们为重边 (可以理解为 E 是个可重无序二元组集).

1.3 简单图 (Simple Graph)

首先, 我们定义两个术语:

- **自环** (loop): 若 $e = (v, v) \in E$, 称 e 为自环.
- **重边** (multiple edge): 若 E 中存在两个完全相同的元素 e_1, e_2, 称它们为重边 (可以理解为 E 是个可重无序二元组集).

简单图 (Simple Graph): 若图 G 中没有自环和重边, 则为简单图.

1.3 简单图 (Simple Graph)

首先, 我们定义两个术语:

- **自环** (loop): 若 $e = (v, v) \in E$, 称 e 为自环.
- **重边** (multiple edge): 若 E 中存在两个完全相同的元素 e_1, e_2, 称它们为重边 (可以理解为 E 是个可重无序二元组集).

简单图 (Simple Graph): 若图 G 中没有自环和重边, 则为简单图.

相对应地, 若图 G 中有自环或重边, 则为**多重图** (Multigraph).

1.4 度数 (Degree)

1.4 度数 (Degree)

与一个顶点 v 关联的边的条数称为该顶点的**度** (degree), 记作 $\mathrm{deg(v)}$.

1.4 度数 (Degree)

与一个顶点 v 关联的边的条数称为该顶点的**度** (degree), 记作 deg(v).

对于无向简单图, 显然有 $\deg(v) = |N(v)|$.

1.4 度数 (Degree)

与一个顶点 v 关联的边的条数称为该顶点的**度** (degree), 记作 deg(v).

对于无向简单图, 显然有 $\deg(v) = |N(v)|$.

Theorem 1.4.1 (图论基本定理): 对于任何无向图 $G = (V, E)$, 有

$$\sum_{v \in V} \deg(v) = 2|E|$$

1.5 路径 (Path)

1.5 路径 (Path)

若顶点序列 $(v_0, v_1, \dots, v_k)$ 满足 $\forall 0 \le i < k, (v_i, v_{i+1}) \in E$, 则称 $v_0 \to v_1 \to \dots \to v_k$ 为一条**路径** (Path).

1.5 路径 (Path)

若顶点序列 $(v_0, v_1, ..., v_k)$ 满足 $\forall 0 \leq i < k, (v_i, v_{i+1}) \in E$, 则称 $v_0 \to v_1 \to ... \to v_k$ 为一条**路径** (Path).

- 如果 $v_0, v_1, ..., v_k$ 中的点两两不同, 则称其为**简单路径**.

1.5 路径 (Path)

若顶点序列 $(v_0, v_1, ..., v_k)$ 满足 $\forall 0 \le i < k, (v_i, v_{i+1}) \in E$, 则称 $v_0 \to v_1 \to ... \to v_k$ 为一条**路径** (Path).

- 如果 $v_0, v_1, ..., v_k$ 中的点两两不同, 则称其为**简单路径**.
- 如果 $v_0 = v_k$, 则称其为一条**回路** (Circuit) 或者**环** (Cycle).

2. 图论存储结构

2.1 邻接矩阵

2.1 邻接矩阵

最直截了当的方法是用一个二维数组 `adj` 来存边.

2.1 邻接矩阵

最直截了当的方法是用一个二维数组 `adj` 来存边.

若 $u \to v$ 有边, 则 `adj[u][v] = 1`, 否则 `adj[u][v] = 0`. 当然, 如果边本身有边权 (即每条边具有一个权值), 则可以用 `adj[u][v]` 记录对应的边权.

2.1 邻接矩阵

最直截了当的方法是用一个二维数组 `adj` 来存边.

若 $u \to v$ 有边, 则 `adj[u][v] = 1`, 否则 `adj[u][v] = 0`. 当然, 如果边本身有边权 (即每条边具有一个权值), 则可以用 `adj[u][v]` 记录对应的边权.

```
for (int i = 1; i <= m; i++) {
  int u, v;
  cin >> u >> v;
  adj[u][v] = true;
}
```

2.1 邻接矩阵

2.1 邻接矩阵

- 🍏 复杂度分析:

2.1 邻接矩阵

🍏 复杂度分析：

- 查询是否存在某条边 $O(1)$;

2.1 邻接矩阵

复杂度分析：

- 查询是否存在某条边 $O(1)$;
- 遍历一个点的所有出边 $O(n)$;

2.1 邻接矩阵

🍏 复杂度分析:

- 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 $O(n^2)$;

2.1 邻接矩阵

🍏 复杂度分析:

- 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 $O(n^2)$.

2.1 邻接矩阵

复杂度分析:

- 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 $O(n^2)$.

用途

2.1 邻接矩阵

🍏 复杂度分析:

- 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 $O(n^2)$.

🔖 用途

- 只适用于没有重边的情况 (或者重新定义, `adj[u][v]` 表示 $u \to v$ 的边数);

2.1 邻接矩阵

🍏 复杂度分析:

- 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 $O(n^2)$.

🔖 用途

- 只适用于没有重边的情况 (或者重新定义, `adj[u][v]` 表示 $u \to v$ 的边数);
- 可以 $O(1)$ 查询一条边是否存在;

2.1 邻接矩阵

🍏 复杂度分析:

- 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 $O(n^2)$.

🔖 用途

- 只适用于没有重边的情况 (或者重新定义, `adj[u][v]` 表示 $u \to v$ 的边数);
- 可以 $O(1)$ 查询一条边是否存在;
- 适用于**稠密图**. 对于**稀疏图**, 空间将无法承受.

2.1 邻接矩阵

那么, 这个**稠密图**的限度是多少呢❓

2.1 邻接矩阵

那么, 这个**稠密图**的限度是多少呢❓

当 $n \leq 5000$ 时, 二维数组 `bool adj[N][N]` 自然是开得下的, 并且在之后遍历的过程也是 $O(n^2)$ 的, 肯定没问题.

2.1 邻接矩阵

那么, 这个**稠密图**的限度是多少呢❓

当 $n \leq 5000$ 时, 二维数组 `bool adj[N][N]` 自然是开得下的, 并且在之后遍历的过程也是 $O(n^2)$ 的, 肯定没问题.

但这就是**稠密图**的极限了吗? 其实不然.

2.1 邻接矩阵

那么, 这个**稠密图**的限度是多少呢❓

当 $n \leq 5000$ 时, 二维数组 `bool adj[N][N]` 自然是开得下的, 并且在之后遍历的过程也是 $O(n^2)$ 的, 肯定没问题.

但这就是**稠密图**的极限了吗? 其实不然.

其实 STL 库中提供了一个名为 `bitset` 的容器. `bitset<N> S` 定义了一个长度为 N 的 `01` 串, `S[i]` 就表示第 i 位的值.

2.1 邻接矩阵

那么, 这个**稠密图**的限度是多少呢❓

当 $n \leq 5000$ 时, 二维数组 `bool adj[N][N]` 自然是开得下的, 并且在之后遍历的过程也是 $O(n^2)$ 的, 肯定没问题.

但这就是**稠密图**的极限了吗? 其实不然.

其实 STL 库中提供了一个名为 `bitset` 的容器. `bitset<N> S` 定义了一个长度为 N 的 `01` 串, `S[i]` 就表示第 i 位的值.

定义 `bitset<N> adj[N]` 的空间花销为 $\frac{N^2}{8\times2^{20}}$ MB, 在遍历的过程中我们可以调用 `_find_first()` 及 `_find_next(int)` 成员函数, 做到时间复杂度 $O\left(\frac{n^2}{\omega}\right)$, 通常取 $\omega = 64$ (从运行效率来看, ω 接近 256).

2.2 邻接表

2.2 邻接表

之前我们的存储效率低, 是因为 `bool adj[N][N]` 中存在很多 “无用” 的位置, 我们没必要记录它们.

2.2 邻接表

之前我们的存储效率低, 是因为 `bool adj[N][N]` 中存在很多 “无用” 的位置, 我们没必要记录它们.

STL 库中有一个 “动态数组” 称为 `vector`, 我们可以定义 `vector<int> adj[N]` 来存储边, 其中 `adj[u]` 中存储的是点 u 的所有出边的信息.

2.2 邻接表

之前我们的存储效率低, 是因为 `bool adj[N][N]` 中存在很多 “无用” 的位置, 我们没必要记录它们.

STL 库中有一个 “动态数组” 称为 `vector`, 我们可以定义 `vector<int> adj[N]` 来存储边, 其中 `adj[u]` 中存储的是点 u 的所有出边的信息.

```
for (int i = 1; i <= m; i++) {
  int u, v;
  cin >> u >> v;
  adj[u].push_back(v);
}
```

2.2 邻接表

2.2 邻接表

- 🍏 复杂度分析:

2.2 邻接表

🍏 复杂度分析：

- 查询是否存在某条边 $O(\deg^+(u))$.

  如果事先进行排序 `sort(adj[u].begin(), adj[u].end())`, 那么可以做到 $O(\log \deg^+(u))$;

2.2 邻接表

复杂度分析：

- 查询是否存在某条边 $O(\deg^+(u))$.

  如果事先进行排序 `sort(adj[u].begin(), adj[u].end())`，那么可以做到 $O(\log\deg^+(u))$;

- 遍历一个点的所有出边 $O(\deg^+(u))$;

2.2 邻接表

🍏 复杂度分析:

- 查询是否存在某条边 $O(\deg^+(u))$.

  如果事先进行排序 `sort(adj[u].begin(), adj[u].end())`, 那么可以做到 $O(\log\deg^+(u))$;
- 遍历一个点的所有出边 $O(\deg^+(u))$;
- 遍历整张图 $O(n+m)$;

2.2 邻接表

🍏 复杂度分析：

- 查询是否存在某条边 $O(\deg^+(u))$.

  如果事先进行排序 `sort(adj[u].begin(), adj[u].end())`, 那么可以做到 $O(\log\deg^+(u))$;
- 遍历一个点的所有出边 $O(\deg^+(u))$;
- 遍历整张图 $O(n+m)$;
- 空间复杂度 $O(n+m)$.

2.2 邻接表

复杂度分析:

- 查询是否存在某条边 $O(\deg^+(u))$.

  如果事先进行排序 `sort(adj[u].begin(), adj[u].end())`, 那么可以做到 $O(\log\deg^+(u))$;
- 遍历一个点的所有出边 $O(\deg^+(u))$;
- 遍历整张图 $O(n+m)$;
- 空间复杂度 $O(n+m)$.

用途

2.2 邻接表

复杂度分析:

- 查询是否存在某条边 $O(\deg^+(u))$.

  如果事先进行排序 `sort(adj[u].begin(), adj[u].end())`, 那么可以做到 $O(\log \deg^+(u))$;
- 遍历一个点的所有出边 $O(\deg^+(u))$;
- 遍历整张图 $O(n+m)$;
- 空间复杂度 $O(n+m)$.

用途

- 适合存储各种图, 一般是首选 (除非是稠密图结合 bitset);

2.3 链式前向星

2.3 链式前向星

相信大家都学过链表叭! 顾名思义, “链式前向星” 采用了链表结构, 每个点 u 均开了一个链表, 用 `head[u]` 表示头指针, `next[e]` 表示 e 这条边的下一个边编号, `to[e]` 表示 e 这条边的终点.

2.3 链式前向星

相信大家都学过链表叭! 顾名思义, “链式前向星”采用了链表结构, 每个点 u 均开了一个链表, 用 `head[u]` 表示头指针, `next[e]` 表示 e 这条边的下一个边编号, `to[e]` 表示 e 这条边的终点.

它的本质其实就是用链表实现邻接表, 因此应用场景跟邻接表相同.

2.3 链式前向星

相信大家都学过链表叭! 顾名思义, "链式前向星" 采用了链表结构, 每个点 u 均开了一个链表, 用 `head[u]` 表示头指针, `next[e]` 表示 e 这条边的下一个边编号, `to[e]` 表示 e 这条边的终点.

它的本质其实就是用链表实现邻接表, 因此应用场景跟邻接表相同.

```
void add(int u, int v) { // 加入一条 u->v 的边
  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
}
for (int i = head[u]; ~i; i = nxt[i]) { // 遍历 u 的所有出边
  int v = to[i]; // 当前边 i 的终点
}
```

2.4 习题

洛谷 P5318 查找文献

2.4.1: 给定一张 n 个点、m 条边的**有向图**, 求从 1 号点出发分别 DFS 和 BFS 能得到的字典序最小的序列.

$1 \leq n \leq 10^5, 1 \leq m \leq 10^6.$

🎈 DFS: 深度优先搜索

🚕 BFS: 广度优先搜索

2.4 习题

数据范围告诉我们, 这题若用**邻接矩阵**肯定不可取, 而用**邻接表**或者**链式前向星**则非常合适.

2.4 习题

数据范围告诉我们, 这题若用**邻接矩阵**肯定不可取, 而用**邻接表**或者**链式前向星**则非常合适.

相信在之前的课里, 大家对 DFS 和 BFS 已经有了一定的了解, 这里主要谈谈如何求“字典序最小”的序列.

2.4 习题

数据范围告诉我们, 这题若用**邻接矩阵**肯定不可取, 而用**邻接表**或者**链式前向星**则非常合适.

相信在之前的课里, 大家对 DFS 和 BFS 已经有了一定的了解, 这里主要谈谈如何求“字典序最小”的序列.

对于 DFS, 我们肯定希望先走点 1 能到的最小的点, 执行子问题, 回溯到 1 时, 走次小的点, 以此类推. 从整体上来看, 那就是将 `adj[1]` 内的点从小到大排序, 依次访问. 其余点同理.

2.4 习题

数据范围告诉我们, 这题若用**邻接矩阵**肯定不可取, 而用**邻接表**或者**链式前向星**则非常合适.

相信在之前的课里, 大家对 DFS 和 BFS 已经有了一定的了解, 这里主要谈谈如何求“字典序最小”的序列.

对于 DFS, 我们肯定希望先走点 1 能到的最小的点, 执行子问题, 回溯到 1 时, 走次小的点, 以此类推. 从整体上来看, 那就是将 `adj[1]` 内的点从小到大排序, 依次访问. 其余点同理.

对于 BFS, 从点 1 出发可以一次性遍历到所有终点, 显然也是将 `adj[1]` 内的点从小到大排序, 依次访问. 其余点同理.

2.4 习题

总的来看, 不论是 DFS 还是 BFS, 它都是通过将 `adj[u]` 排序实现了“字典序最小”.

2.4 习题

总的来看, 不论是 DFS 还是 BFS, 它都是通过将 `adj[u]` 排序实现了 "字典序最小".

由于每条边只会被访问至多一次, 因此总时间复杂度 $O(n + m)$.

2.4 习题

总的来看, 不论是 DFS 还是 BFS, 它都是通过将 `adj[u]` 排序实现了 “字典序最小” .

由于每条边只会被访问至多一次, 因此总时间复杂度 $O(n+m)$.

接下来, 我将通过代码, 手把手讲解一下这题的实现.

2.4 习题

洛谷 P3916 图的遍历

2.4.2: 给定一张 n 个点、m 条边的**有向图**, 对于每个点 v, 令 $A(v)$ 表示从点 v 出发, 能到达的编号最大的点. 试计算 $A(1), A(2), ..., A(n)$.

$1 \leq n, m \leq 10^5$.

2.4 习题

最直接的想法是: 分别从每个点 v 出发, 进行一通 DFS/BFS 得到编号最大的点.

2.4 习题

最直接的想法是: 分别从每个点 v 出发, 进行一通 DFS/BFS 得到编号最大的点.

正确性显然没问题, 但问题是在最坏情况下, 时间复杂度高达 $O(n^2)$, 无法接受.

2.4 习题

最直接的想法是: 分别从每个点 v 出发, 进行一通 DFS/BFS 得到编号最大的点.

正确性显然没问题, 但问题是在最坏情况下, 时间复杂度高达 $O(n^2)$, 无法接受.

我们可以采取逆向思维, 建立**反图**, 并从点 n 往点 1 依次遍历. 点 n 能到达的所有点 v, 它们的 $A(v) = n$; 接着点 $n-1$ 能到达的其余点 v', 它们的 $A(v') = n-1$, 依次类推.

2.4 习题

最直接的想法是: 分别从每个点 v 出发, 进行一通 DFS/BFS 得到编号最大的点.

正确性显然没问题, 但问题是在最坏情况下, 时间复杂度高达 $O(n^2)$, 无法接受.

我们可以采取逆向思维, 建立**反图**, 并从点 n 往点 1 依次遍历. 点 n 能到达的所有点 v, 它们的 $A(v) = n$; 接着点 $n-1$ 能到达的其余点 v', 它们的 $A(v') = n-1$, 依次类推.

同样由于每条边只会被访问至多一次, 因此总时间复杂度 $O(n+m)$.