图论相关问题代码实现技巧

Lazurite Ayane

November 6, 2024

邻接矩阵

- 邻接矩阵适合存储**稠密图**。
- 使用二维数组保存边权即可，如果边不存在可以置边权为很大的数 $+\infty$。
- 实现时需要注意**重边**与**自环**。对于最短路问题，可以在重复的边中选择边权最小的一条保留。

Floyd

- Floyd 算法适合使用邻接矩阵：
- ```
  const int oo = 1e9;
  int d[N][N]; // N 个结点的邻接矩阵
  // 将邻接矩阵初始化为 +∞，对角线为 0
  for (int k = 0; k < n; k++) // Floyd
      for (int i = 0; i < n; i++)
          for (int j = 0; j < n; j++)
              d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
  ```
- 注意三重循环的顺序。

* 尽管按照错误的循环顺序重复执行三次之后得到的结果仍然是正确的，但是能不出错就别出错。
  ```

STL

- C++ 语言标准模板库（Standard Template Library）。注意 C 语言不支持。
- 在图论中常用到：
  - 向量（变长数组）：`vector`
  - 优先队列（堆）：`priority_queue`
- 集合（`set`）、映射（`map`）等 STL 也可以让部分题目的代码实现更简洁，但图论中不太常用。
- 平常使用的 `sort`, `lower_bound`, `min` 等 STL 有实现。

向量 vector

- 可以当作不定长的数组使用。
- 在图论中可以用来方便地实现**邻接表**。
- 需要 vector 头文件：
  - `#include <vector>`

基本操作 (1)

- 初始化元素为 int 类型的向量 v：
  - `vector<int> v;`
  - 元素类型可以是任意类型，例如结构体或是 vector。
- 访问 v 的第 i 个元素：`v[i]`
- 获取 v 的长度：
  - `v.size()`
  - 注意：返回值类型是**无符号整数**。
- 改变 v 的长度：
  - `v.resize(count);`

基本操作 (2)

- 在 v 的末尾插入元素：
  - `v.push_back(value);`
  - `v.emplace_back(value);`
  - `emplace_back` 在原地构建元素，复杂度常数更小。
  - **均摊**复杂度 $O(1)$。
- 删除 v 末尾的元素：
  - `v.pop_back();`
- 清空 v 的所有内容：
  - `v.clear();`
- 其余操作可以参考 OI-Wiki 或 cppreference。

关于空间

- vector 所占用的空间**不一定是**元素数量与元素类型所需空间的乘积，一般来说会大于这个值。
- 因此，慎重用 vector 定义高维数组，可能会导致内存超限。

简单邻接表

- 邻接表适合存储**稀疏图**。
- 对于每个结点，使用一个 vector 保存与之相连的边。
- 假设图中总共至多有 N 个结点，每条边不含边权。可以这样实现邻接表：
  - `vector<int> g[N]; // N 个结点的邻接表`

```
g[u].emplace_back(v); // 添加一条边 u → v
for (int i = 0; i < g[u].size(); i++) {
    int v = g[u][i]; // 遍历 u 的出边 u → v
    // ···
}
```

语法糖

- 实际上，可以使用语法糖简化遍历出边的实现：
  - `for (auto v : g[u]) {`
    `// 遍历 $u$ 的出边 $u \to v$`
    `// $\cdots$`
    `}`
  - 但是并不建议滥用 auto。

边权 (1)：结构体

- 对于具有边权或是其他信息的边，可以定义结构体以保存边的信息。
  - ```
    struct Edge {
        int to; // 边指向的点
        int weight; // 边权
    }
    ```
- 邻接表的实现变化不大：
  - ```
    vector<Edge> g[N]; // N 个结点的邻接表
    g[u].emplace_back({v, w});
      // 添加边权为 w 的一条边 u → v
    ```

pair

- 两个元素的有序对 $\langle x, y \rangle$ 可以使用 STL 的 pair 保存。
- pair $\langle x, y \rangle$ 之间的大小关系定义为：

$$\langle x_1, y_1 \rangle < \langle x_2, y_2 \rangle \iff x_1 < x_2 \vee (x_1 = x_2 \wedge y_1 < y_2)$$

- 第一个元素类型 T1，第二个元素类型 T2 的 pair：
  - `pair<T1, T2> p;`
- 创建一个 pair：
  - `p = make_pair(x, y);`
- 取 pair 的第一个元素：`p.first`
- 取 pair 的第二个元素：`p.second`

边权 (2)：pair

- 可以用 pair 实现邻接表。第一个元素保存边指向的点，第二个元素保存边权：
  - `vector<pair<int, int>> g[N];`
    `g[u].emplace_back(make_pair(v, w));`
    `// 添加边权为 w 的一条边 u → v`
- 遍历邻接表：
  - `for (auto e : g[u]) {`
    `int v = e.first, w = e.second;`
    `// 遍历 u 的出边 u → v，边权为 w`
    `}`

优先队列 priority_queue

- 可以当作堆使用。
- priority_queue 不是容器，而是容器适配器。之后可以看到在定义优先队列时使用了 vector。
- 在图论中可以用来方便地实现堆优化的 Dijkstra 算法。
- 需要 queue 头文件：
  - `#include <queue>`

基本操作 (1)

- 初始化元素为 int 类型的**大根堆** q：
  - `priority_queue<int> q;`
  - 等价于：

    ```
    priority_queue<int, vector<int>, less<int>> q;
    ```

  - 元素类型可以是任意类型，但需要支持比较运算符。
- 初始化元素为 int 类型的**小根堆** q：
  - `priority_queue<int, vector<int>, greater<int>> q;`

基本操作 (2)

- 向 q 中添加元素：
  - `q.push(value);`
  - 时间复杂度 $O(\log n)$。
- 获取 q 中堆顶元素：
  - `q.top()`
  - 时间复杂度 $O(1)$。
- 移除 q 中堆顶元素：
  - `q.pop();`
  - 时间复杂度 $O(\log n)$。

基本操作 (3)

- 获取 q 中元素数量：
  - `q.size()`
  - 与 vector 一样，返回值类型为无符号整数。
  - 时间复杂度 $O(1)$。
- 判断 q 是否为空：
  - `q.empty()`
  - 时间复杂度 $O(1)$。
- 优先队列不提供直接清空的方法。
- 其余操作可以参考 OI-Wiki 或 cppreference。

堆优化 Dijkstra (1)

假设结点数量不超过 N。先定义距离数组与所使用的堆：

- ```
  const long long oo = 1e18;
  long long d[N]; bool vis[N];
  priority_queue<pair<long long, int>> q;
    // 第一个元素表示距离的相反数，第二个元素表示点的编号
  ```

假设最短路的起点是 s。初始化距离数组与堆：

- ```
  for (int i = 0; i < n; i++)
      d[i] = oo, vis[i] = 0; // 初始化距离为 +∞
  d[s] = 0;
  q.push(make_pair(0, s)); // 当前 s 到 s 的距离为 0
  ```

堆优化 Dijkstra (2)

Dijkstra 算法主流程：

```
while (!q.empty()) {
    int u = q.top().second; // 找到当前堆中距离 s 最小的点 u
    q.pop(); // 从堆中删除 u
    if (vis[u]) continue; // 保证点 u 尚未用于更新
    vis[u] = 1;
    for (auto e : g[u]) { // 遍历 u 出发的每一条边
        int v = e.first, w = e.second;
        if (d[v] > d[u] + w) { // 如果可以松弛的话
            d[v] = d[u] + w; // 更新 s 到 v 的距离
            q.push(make_pair(-d[v], v)); // 并将 v 加入堆中
        }
    }
}
```

堆优化 Dijkstra (3)

- 代码中使用前文提到的邻接表存储图。
- 代码中的 `q.push(make_pair(-d[v], v));` 之所以是 `-d[v]`，是因为在定义 q 时方便起见定义了大根堆，而在 Dijkstra 中需要小根堆。直接将距离反号就能让 q 变成所需的小根堆。
- 由于一个点可能被加入堆中多次，所以需要 `vis` 去判断一个点是否已经被用于更新。
- 以上实现仅供参考，同学们可以有自己的实现方式。

Fin

- 存储图的方式除邻接矩阵与邻接表以外，还有链式前向星。因为有一定理解难度，此处不再赘述。有兴趣的同学可以参考 OI-Wiki。
- 尽管 C++ 语言有 STL，但上机或考试时可能会有**只能使用 C 语言的题目**。
- Q & A