图的存储相关知识补充

我们在数据结构课上已经学习了使用指针维护链表对图进行存储。但是考虑到指针写法的复杂性，且同学们在使用指针时也更容易写错或是让程序难以调试，因此这种写法并不是十分适合在算法中使用，本文将对一些常用的图存储方法进行补充。考虑到篇幅限制，本文将仅仅介绍在算法中最常用的几种建图方法，对于更多的图存储方法，大家可以自行查阅资料进行学习。

邻接矩阵

对于一个有 n 个点的图，我们可以使用一个 `g[n][n]` 的数组来存边。若 `g[u][v] = 1`，则表示存在一条从点 u 到点 v 的边，若为 0 则表示不存在。对于带权的图，我们也可以直接用 `g[u][v]` 表示 u 到 v 的边的边权。

对于邻接矩阵，我们能够以 $O(1)$ 的时间复杂度查找是否存在某条边。以 $O(n)$ 的时间遍历一个点的所有出边。

但是对于邻接矩阵而言最大的问题是它在稀疏图上的效率很低，尤其是在点数较多的情况下。例如一个有 10^5 个点的图，我们需要开一个 `g[100000][100000]` 的数组，对于我们一般的算法问题而言，这种空间开销显然是不可接受的。

> 对于 accoding 上最常见设置的空间上限为 `64M`，一个 int 类型的大小为 `4B`，因此我们最多能够存储 `64M / 4B = 16 * 1024 * 1024` 个 int 类型的数，也即大约 $16 \cdot 10^6$ 个 int。

因此对于邻接矩阵，我们能接受的 n 的数量级大约为 10^3，通常可以在 Floyd 算法（时间复杂度 $O(n^3)$ 求得任意两点间的最小距离）中使用。

```cpp
// 可存储 N 个点的邻接矩阵
int g[N][N];

// 添加一条 u 到 v 的边，这条边的权值为 w
g[u][v] = w;

// 遍历点 u 的出边
for (int i = 0; i < N; i++) {
    if (g[u][i]) {
        // do something ...
    }
}
```

```cpp
// dfs 遍历图
int vis[N]; // 表示一个点是否被 visit 过
void dfs(int u) {
    if (vis[u])
        return;
    vis[u] = 1;
    for (int i = 0; i < N; i++) {
        if (g[u][i]) {
            dfs(i);
        }
    }
}
```

邻接表

由于邻接矩阵在稀疏图中花费了大量的无用空间表示不存在的边，我们考虑对其进行优化。我们使用可以动态增加元素的数据结构来维护，例如使用 c++ stl 中提供的 vector，我们可以使用 `vector<int> g[n]` 来存储有 n 个点的图。或者很熟悉 c++ stl 已经大量使用 vector 的同学可能会使用二维 `vector<vector<int>> g(n)` 来初始化一个有 n 个点的邻接表。

无论哪种方式，此时我们的 `g[u]` 是一个 vector，其中存储了点 u 的出边。

```cpp
// 可存储 N 个点的邻接表
vector<int> g[N];

// 添加一条从 u 到 v 的边
// 对一个 vector 使用 push_back 方法向其末尾加入一个数 v
g[u].push_back(v);

// 遍历点 u 的出边
for (int i = 0; i < g[u].size(); i++) {
    int v = g[u][i];    // v 即是我们指向的那个点
    // do something ...
}

// dfs 遍历图
int vis[N]; // 表示一个点是否被 visit 过
void dfs(int u) {
    if (vis[u])
        return;
    vis[u] = 1;
    for (int i = 0; i < g[u].size(); i++) {
        dfs(g[u][i]);
    }
```

```cpp
}

// 在 c++11 中已支持更加现代化的写法来遍历一个 vector， 大家可能在 java 中
// 已经使用过类似的写法
// 例如 g[u] = {2, 3, 5}，则 v 会直接依次遍历 2, 3, 5
for (int v : g[u]) {
    dfs(v);
}

// 如果除了想要存储边，还希望存储边权，我们可以使用 struct 或者 stl 的
// pair / array 进行存储，下面以 struct 为例（相信会使用
// pair / array 的同学肯定也看得懂 struct 的写法）
struct Edge {
    int to;     // 表示这条边指向的点
    int weight; // 表示这条边的边权
};

vector<Edge> g[N];

// 添加一条从 u 到 v 权值为 w 的边
g[u].push_back({v, w});

// 遍历点 u 的出边
for (int i = 0; i < g[u].size(); i++) {
    int to = g[u][i].to, weight = g[u][i].weight;
    // do something ...
}
```

该方法存各种图都比较适合，除了有特殊需求（例如想要 $O(1)$ 的查找是否有一条边时，可以使用邻接矩阵）的情况下，大部分问题都可以使用该方法进行建图。该方法也十分地简单好写，非常推荐大家学习使用该方法！

链式前向星

该方法本质上是一个链表维护的邻接表，只是使用数组进行模拟。

首先给出一个指针的写法，便于大家后续的理解。

```cpp
struct Edge {
    int to;             // 表示本条边指向的点
    struct Edge *nxt;   // 表示本条边的下一条边。
}

// head[u] 表示点 u 指出的第一条边，对应上述邻接表中的 g[u] 这个 vector 的
// 第一个元素（或者可以说最后一个元素）
struct Edge *head[N];
```

```cpp
// 添加一条从 u 到 v 的边
void add(int u, int v) {
    struct Edge *e = (struct Edge *) malloc(sizeof(struct Edge));
    e->to = v;
    e->nxt = head[u];
    head[u] = e;
}
```

由于指针写法不易调试，写起来也更加复杂，我们使用三个核心数组 `head[N]`，`nxt[N]`，`to[N]` 进行模拟。

```cpp
int head[N], nxt[N], to[N];
int cnt;

// 添加一条从 u 到 v 的边
void add(int u, int v) {
    // 表示新添加的这条边为第 cnt 条边，这条边的下一个为 head[u]，
    nxt[++cnt] = head[u];   // 对应上述的 e->nxt = head[u];
    head[u] = cnt;          // 对应上述的 head[u] = e;
    to[cnt] = v;            // 对应上述的 e->to = v;
}

// 遍历点 u 的出边
for (int e = head[u]; e != 0; e = nxt[i]) {
    int v = to[i];
    // do something ...
}
```

该写法在图的存储的空间复杂度和图的遍历的时间复杂度上都与上述邻接表相同，也适合用于各种图的存储。