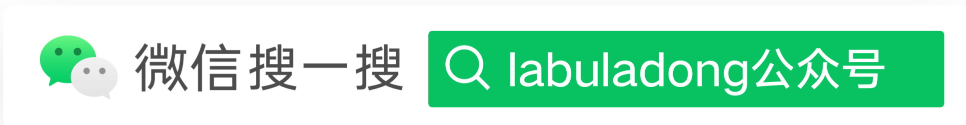
二分图判定



通知：数据结构精品课 V1.6 持续更新中，第八期打卡挑战 开始报名，算法私教课 开始预约。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

牛客	LeetCode	力扣	难度
-	785. Is Graph Bipartite?	785. 判断二分图	
-	886. Possible Bipartition	886. 可能的二分法	
-	-	剑指 Offer II 106. 二分图	

————

我之前写了好几篇图论相关的文章：

图遍历算法

名流问题

并查集算法计算连通分量

环检测和拓扑排序

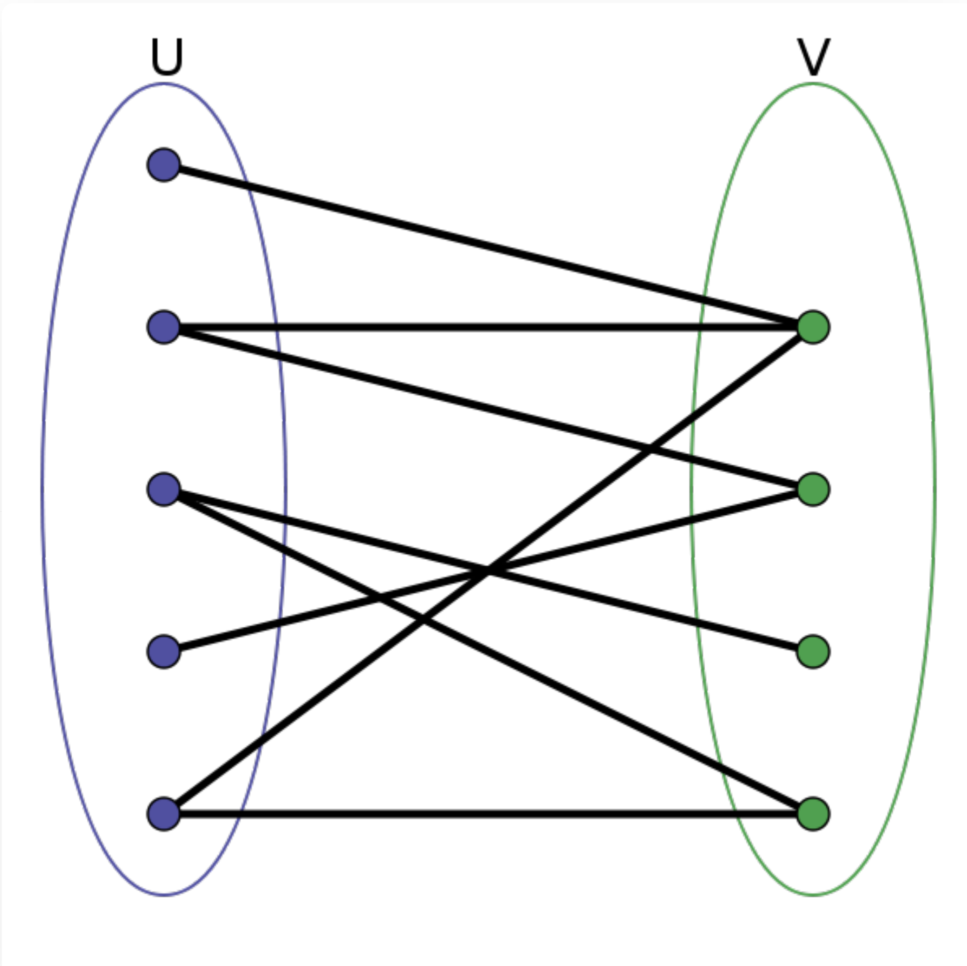
Dijkstra 最短路径算法

今天继续来讲一个经典图论算法：二分图判定。

二分图简介

在讲二分图的判定算法之前，我们先来看下百度百科对「二分图」的定义：

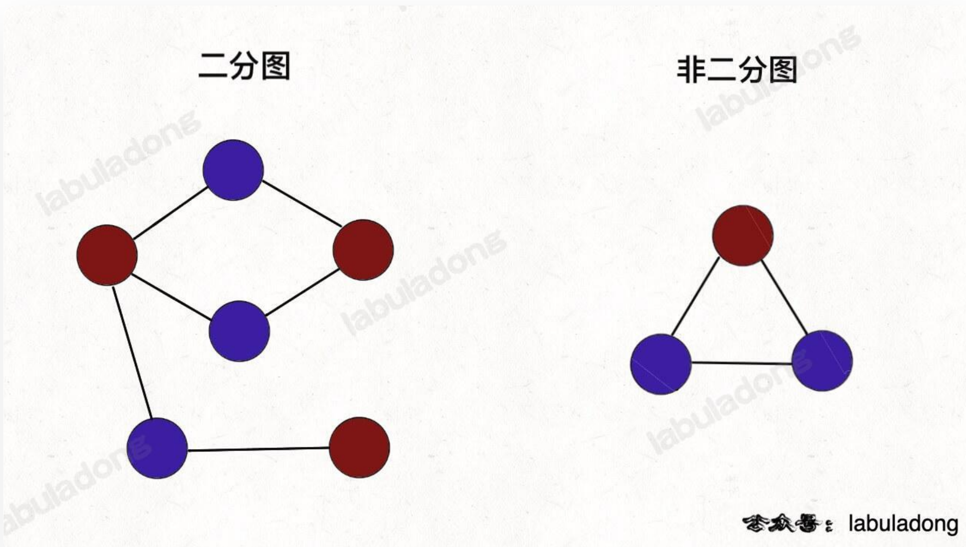
> 二分图的顶点集可分割为两个互不相交的子集，图中每条边依附的两个顶点都分属于这两个子集，且两个子集内的顶点不相邻。



其实图论里面很多术语的定义都比较拗口，不容易理解。我们甭看这个死板的定义了，来玩个游戏吧：

给你一幅「图」，请你用两种颜色将图中的所有顶点着色，且使得任意一条边的两个端点的颜色都不相同，你能做到吗？

这就是图的「双色问题」，其实这个问题就等同于二分图的判定问题，如果你能够成功地将图染色，那么这幅图就是一幅二分图，反之则不是：



在具体讲解二分图判定算法之前，我们先来说说计算机大佬们闲着无聊解决双色问题的目的是什么。

首先，二分图作为一种特殊的图模型，会被很多高级图算法（比如最大流算法）用到，不过这些高级算法我们不是特别有必要去掌握，有兴趣的读者可以自行搜索。

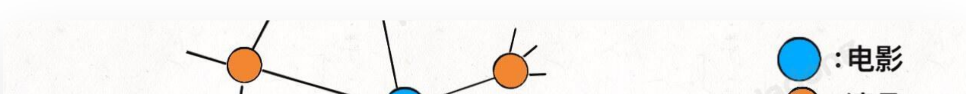
从简单实用的角度来看，二分图结构在某些场景可以更高效地存储数据。

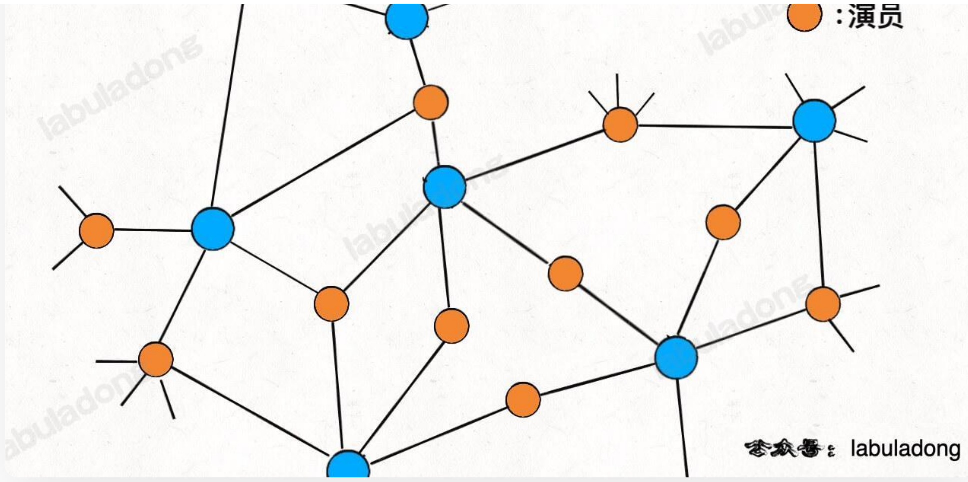
比如说我们需要一种数据结构来储存电影和演员之间的关系：某一部电影肯定是由多位演员出演的，且某一位演员可能会出演多部电影。你使用什么数据结构来存储这种关系呢？

既然是存储映射关系，最简单的不就是使用哈希表嘛，我们可以使用一个 `HashMap<String, List<String>>` 来存储电影到演员列表的映射，如果给一部电影的名字，就能快速得到出演该电影的演员。

但是如果给出一个演员的名字，我们想快速得到该演员演出的所有电影，怎么办呢？这就需要「反向索引」，对之前的哈希表进行一些操作，新建另一个哈希表，把演员作为键，把电影列表作为值。

显然，如果用哈希表存储，需要两个哈希表分别存储「每个演员到电影列表」的映射和「每部电影到演员列表」的映射。但如果用「图」结构存储，将电影和参演的演员连接，很自然地就成为了一幅二分图：





每个电影节点的相邻节点就是参演该电影的所有演员，每个演员的相邻节点就是该演员参演过的所有电影，非常方便直观。

再举个近在眼前的例子，我的网站每篇文章末尾都会有引用该文章的题目，这就是二分图的应用：

「文章」和「题目」就可以抽象成「图」中两种不同的节点类型，文章和题目之间的引用可以抽象成边，这就是标准的二分图模型，只要查询任意一个「文章」节点即可得到关联的「题目」节点，查询任意一个「题目」节点，即可得到关联的「文章」节点。

其实生活中不少实体的关系都能自然地形成二分图结构，所以在某些场景下图结构也可以作为存储键值对的数据结构（符号表）。

好了，接下来进入正题，说说如何判定一幅图是否是二分图。

二分图判定思路

判定二分图的算法很简单，就是用代码解决「双色问题」。

说白了就是遍历一遍图，一边遍历一边染色，看看能不能用两种颜色给所有节点染色，且相邻节点的颜色都不相同。

既然说到遍历图，也不涉及最短路径之类的，当然是 DFS 算法和 BFS 皆可了，DFS 算法相对更常用些，所以我们先来看看如何用 DFS 算法判定双色图。

首先，基于 学习数据结构和算法的框架思维 写出图的遍历框架：

```
/* 二叉树遍历框架 */
void traverse(TreeNode root) {
```

```java
    if (root == null) return;
    traverse(root.left);
    traverse(root.right);
}

/* 多叉树遍历框架 */
void traverse(Node root) {
    if (root == null) return;
    for (Node child : root.children)
        traverse(child);
}

/* 图遍历框架 */
boolean[] visited;
void traverse(Graph graph, int v) {
    // 防止走回头路进入死循环
    if (visited[v]) return;
    // 前序遍历位置，标记节点 v 已访问
    visited[v] = true;
    for (TreeNode neighbor : graph.neighbors(v))
        traverse(graph, neighbor);
}
```

因为图中可能存在环，所以用 `visited` 数组防止走回头路。

这里可以看到我习惯把 return 语句都放在函数开头，因为一般 return 语句都是 base case，集中放在一起可以让算法结构更清晰。

其实，如果你愿意，也可以把 if 判断放到其它地方，比如图遍历框架可以稍微改改：

```java
/* 图遍历框架 */
boolean[] visited;
void traverse(Graph graph, int v) {
    // 前序遍历位置，标记节点 v 已访问
    visited[v] = true;
    for (int neighbor : graph.neighbors(v)) {
        if (!visited[neighbor]) {
            // 只遍历没标记过的相邻节点
            traverse(graph, neighbor);
        }
    }
}
```

这种写法把对 `visited` 的判断放到递归调用之前，和之前的写法唯一的不同就是，你需要保证调用 `traverse(v)` 的时候，`visited[v] == false`。

为什么要特别说这种写法呢？因为我们判断二分图的算法会用到这种写法。

回顾一下二分图怎么判断，其实就是让 `traverse` 函数一边遍历节点，一边给节点染色，尝试让每对相邻节点的颜色都不一样。

所以，判定二分图的代码逻辑可以这样写：

```
/* 图遍历框架 */
void traverse(Graph graph, boolean[] visited, int v) {
    visited[v] = true;
    // 遍历节点 v 的所有相邻节点 neighbor
    for (int neighbor : graph.neighbors(v)) {
        if (!visited[neighbor]) {
            // 相邻节点 neighbor 没有被访问过
            // 那么应该给节点 neighbor 涂上和节点 v 不同的颜色
            traverse(graph, visited, neighbor);
        } else {
            // 相邻节点 neighbor 已经被访问过
            // 那么应该比较节点 neighbor 和节点 v 的颜色
            // 若相同，则此图不是二分图
        }
    }
}
```

如果你能看懂上面这段代码，就能写出二分图判定的具体代码了，接下来看两道具体的算法题来实操一下。

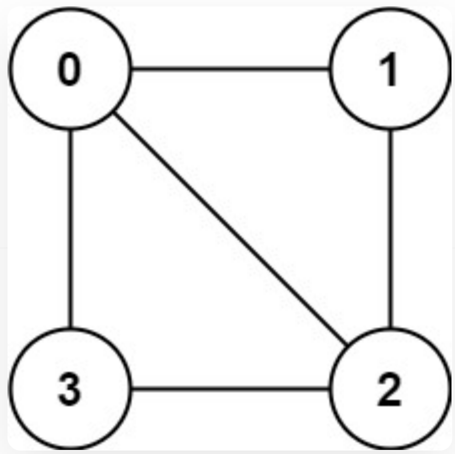
题目实践

力扣第 785 题「判断二分图」就是原题，题目给你输入一个 邻接表 表示一幅无向图，请你判断这幅图是否是二分图。

函数签名如下：

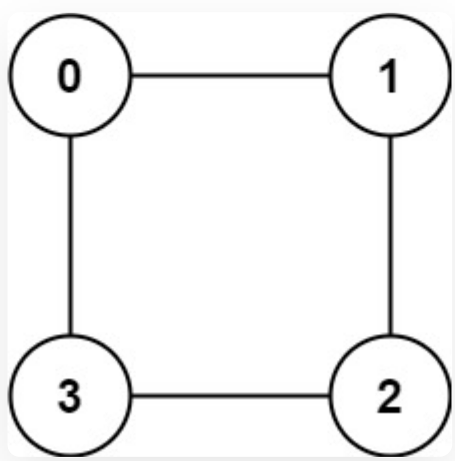
```java
boolean isBipartite(int[][] graph);
```

比如题目给的例子，输入的邻接表 `graph = [[1,2,3],[0,2],[0,1,3],[0,2]]`，也就是这样一幅图：



显然无法对节点着色使得每两个相邻节点的颜色都不相同，所以算法返回 false。

但如果输入 `graph = [[1,3],[0,2],[1,3],[0,2]]`，也就是这样一幅图：



如果把节点 `{0, 2}` 涂一个颜色，节点 `{1, 3}` 涂另一个颜色，就可以解决「双色问题」，所以这是一幅二分图，算法返回 true。

结合之前的代码框架，我们可以额外使用一个 `color` 数组来记录每个节点的颜色，从而写出解法代码：

```java
// 记录图是否符合二分图性质
private boolean ok = true;
// 记录图中节点的颜色，false 和 true 代表两种不同颜色
private boolean[] color;
// 记录图中节点是否被访问过
private boolean[] visited;

// 主函数，输入邻接表，判断是否是二分图
public boolean isBipartite(int[][] graph) {
    int n = graph.length;
    color =  new boolean[n];
    visited =  new boolean[n];
    // 因为图不一定是联通的，可能存在多个子图
    // 所以要把每个节点都作为起点进行一次遍历
    // 如果发现任何一个子图不是二分图，整幅图都不算二分图
    for (int v = 0; v < n; v++) {
        if (!visited[v]) {
            traverse(graph, v);
        }
    }
    return ok;
}

// DFS 遍历框架
private void traverse(int[][] graph, int v) {
    // 如果已经确定不是二分图了，就不用浪费时间再递归遍历了
    if (!ok) return;

    visited[v] = true;
    for (int w : graph[v]) {
        if (!visited[w]) {
            // 相邻节点 w 没有被访问过
            // 那么应该给节点 w 涂上和节点 v 不同的颜色
            color[w] = !color[v];
            // 继续遍历 w
            traverse(graph, w);
        } else {
            // 相邻节点 w 已经被访问过
            // 根据 v 和 w 的颜色判断是否是二分图
            if (color[w] == color[v]) {
                // 若相同，则此图不是二分图
                ok = false;
```

```java
            }
        }
    }
}
```

这就是解决「双色问题」的代码，如果能成功对整幅图染色，则说明这是一幅二分图，否则就不是二分图。

接下来看一下 BFS 算法的逻辑：

```java
// 记录图是否符合二分图性质
private boolean ok = true;
// 记录图中节点的颜色，false 和 true 代表两种不同颜色
private boolean[] color;
// 记录图中节点是否被访问过
private boolean[] visited;

public boolean isBipartite(int[][] graph) {
    int n = graph.length;
    color =  new boolean[n];
    visited =  new boolean[n];

    for (int v = 0; v < n; v++) {
        if (!visited[v]) {
            // 改为使用 BFS 函数
            bfs(graph, v);
        }
    }

    return ok;
}

// 从 start 节点开始进行 BFS 遍历
private void bfs(int[][] graph, int start) {
    Queue<Integer> q = new LinkedList<>();
    visited[start] = true;
    q.offer(start);

    while (!q.isEmpty() && ok) {
        int v = q.poll();
        // 从节点 v 向所有相邻节点扩散
        for (int w : graph[v]) {
            if (!visited[w]) {
                // 相邻节点 w 没有被访问过
```

```java
                    // 那么应该给节点 w 涂上和节点 v 不同的颜色
                    color[w] = !color[v];
                    // 标记 w 节点，并放入队列
                    visited[w] = true;
                    q.offer(w);
                } else {
                    // 相邻节点 w 已经被访问过
                    // 根据 v 和 w 的颜色判断是否是二分图
                    if (color[w] == color[v]) {
                        // 若相同，则此图不是二分图
                        ok = false;
                    }
                }
            }
        }
    }
```

核心逻辑和刚才实现的 `traverse` 函数（DFS 算法）完全一样，也是根据相邻节点 `v` 和 `w` 的颜色来进行判断的。关于 BFS 算法框架的探讨，详见前文 BFS 算法框架 和 Dijkstra 算法模板，这里就不展开了。

最后再来看看力扣第 886 题「可能的二分法」：

886. 可能的二分法 labuladong 题解 思路

难度 中等 138

给定一组 `N` 人（编号为 `1, 2, ..., N`）， 我们想把每个人分进**任意**大小的两组。

每个人都可能讨厌其他人，那么他们不应该属于同一组。

形式上，如果 `dislikes[i] = [a, b]`，表示 `a` 和 `b` 互相讨厌对方，不应该把他们分进同一组。

当可以将所有人分进两组时，返回 `true`；否则返回 `false`。

示例 1：

```
输入：N = 4, dislikes = [[1,2],[1,3],[2,4]]
输出：true
解释：group1 [1,4], group2 [2,3]
```

示例 2：

```
输入：N = 3, dislikes = [[1,2],[1,3],[2,3]]
输出：false
```

函数签名如下:

```java
boolean possibleBipartition(int n, int[][] dislikes);
```

其实这题考察的就是二分图的判定:

如果你把每个人看做图中的节点，相互讨厌的关系看做图中的边，那么 `dislikes` 数组就可以构成一幅图;

又因为题目说互相讨厌的人不能放在同一组里，相当于图中的所有相邻节点都要放进两个不同的组;

那就回到了「双色问题」，如果能够用两种颜色着色所有节点，且相邻节点颜色都不同，那么你按照颜色把这些节点分成两组不就行了嘛。

所以解法就出来了，我们把 `dislikes` 构造成一幅图，然后执行二分图的判定算法即可:

```java
private boolean ok = true;
private boolean[] color;
private boolean[] visited;

public boolean possibleBipartition(int n, int[][] dislikes) {
    // 图节点编号从 1 开始
    color = new boolean[n + 1];
    visited = new boolean[n + 1];
    // 转化成邻接表表示图结构
    List<Integer>[] graph = buildGraph(n, dislikes);

    for (int v = 1; v <= n; v++) {
        if (!visited[v]) {
            traverse(graph, v);
        }
    }

    return ok;
}

// 建图函数
private List<Integer>[] buildGraph(int n, int[][] dislikes) {
    // 图节点编号为 1...n
```

```java
        List<Integer>[] graph = new LinkedList[n + 1];
        for (int i = 1; i <= n; i++) {
            graph[i] = new LinkedList<>();
        }
        for (int[] edge : dislikes) {
            int v = edge[1];
            int w = edge[0];
            // 「无向图」相当于「双向图」
            // v -> w
            graph[v].add(w);
            // w -> v
            graph[w].add(v);
        }
        return graph;
    }

    // 和之前的 traverse 函数完全相同
    private void traverse(List<Integer>[] graph, int v) {
        if (!ok) return;
        visited[v] = true;
        for (int w : graph[v]) {
            if (!visited[w]) {
                color[w] = !color[v];
                traverse(graph, w);
            } else {
                if (color[w] == color[v]) {
                    ok = false;
                }
            }
        }
    }
```

至此，这道题也使用 DFS 算法解决了，如果你想用 BFS 算法，和之前写的解法是完全一样的，可以自己尝试实现。

二分图的判定算法就讲到这里，更多二分图的高级算法，敬请期待。

▶ **引用本文的文章**
