题目地址（886. 可能的二分法）

https://leetcode-cn.com/problems/possible-bipartition/

题目描述

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。

每个人都可能不喜欢其他人，那么他们不应该属于同一组。

形式上，如果 dislikes[i] = [a, b]，表示不允许将编号为 a 和 b 的人归入同一组。

当可以用这种方法将每个人分进两组时，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
示例 3：

输入：N = 5, dislikes = [[1,2],[2,3],[3,4],[4,5],[1,5]]
输出：false

提示：

1 <= N <= 2000
0 <= dislikes.length <= 10000
dislikes[i].length == 2
1 <= dislikes[i][j] <= N
dislikes[i][0] < dislikes[i][1]
对于dislikes[i] == dislikes[j] 不存在 i != j

前置知识

- 图的遍历
- DFS

标签

- 图

难度

- 中等

入选理由

- 二分图，会这一道题就够了

公司

- 暂无

思路

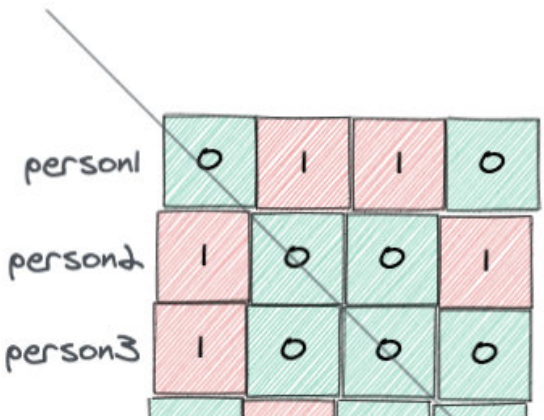
这是一个图的问题。解决这种问题一般是要遍历图才行的，这也是图的套路。 那么遍历的话，你要有一个合适的数据结构。比较常见的图存储方式是邻接矩阵和邻接表。

而我们这里为了操作方便（代码量），直接使用邻接矩阵。由于是互相不喜欢，不存在一个喜欢另一个，另一个不喜欢一个的情况，因此这是无向图。而无向图邻接矩阵实际上是会浪费空间，具体看我下方画的图。

而题目给我们的二维矩阵并不是现成的邻接矩阵形式，因此我们需要自己生成。

我们用 1 表示互相不喜欢（dislike each other）， 0 表示没有互相不喜欢。

```
graph = [[0] * N for i in range(N)]
for a, b in dislikes:
    graph[a - 1][b - 1] = 1
    graph[b - 1][a - 1] = 1
```



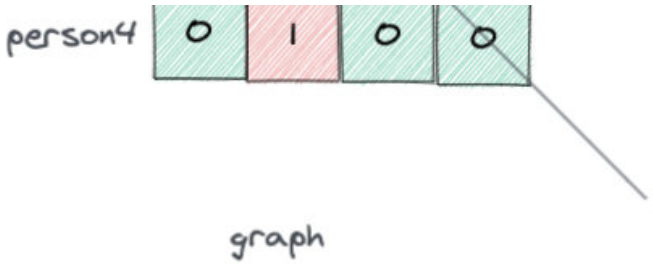


image.png

同时可以用 hashmap 或者数组存储 N 个人的**分组情况**， 业界关于这种算法一般叫染色法，因此我们命名为 colors，其实对应的本题叫 groups 更合适。

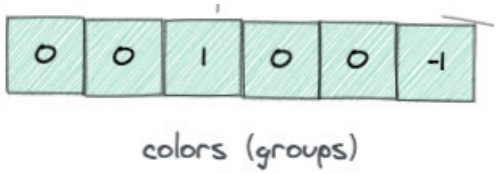


image.png

我们用：

- 0 表示**未分组**
- 1 表示**分组 1**
- -1 表示**分组 2**

> 之所以用 0，1，-1，而不是 0，1，2 是因为我们会在不能分配某一组的时候尝试分另外一组，这个时候有其中一组转变为另外一组就可以直接乘以-1，而 0，1，2 这种就稍微麻烦一点而已。

具体算法：

- 遍历每一个人，尝试给他们进行分组。这里可以分组 1，也可以分组 2，都没有关系。这里我们就分配组 1.

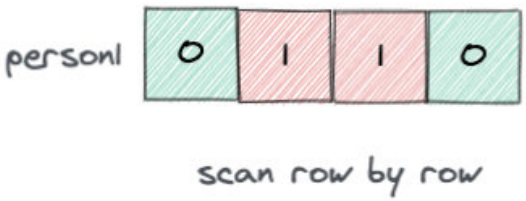


image.png

- 遍历这个人讨厌的人，尝试给他们分另外一组，如果不可以分配另外一组，则返回 false。直到最后都没有找到无法分配的组我们就返回 true。

那问题的关键在于如何判断“不可以分配另外一组”呢？

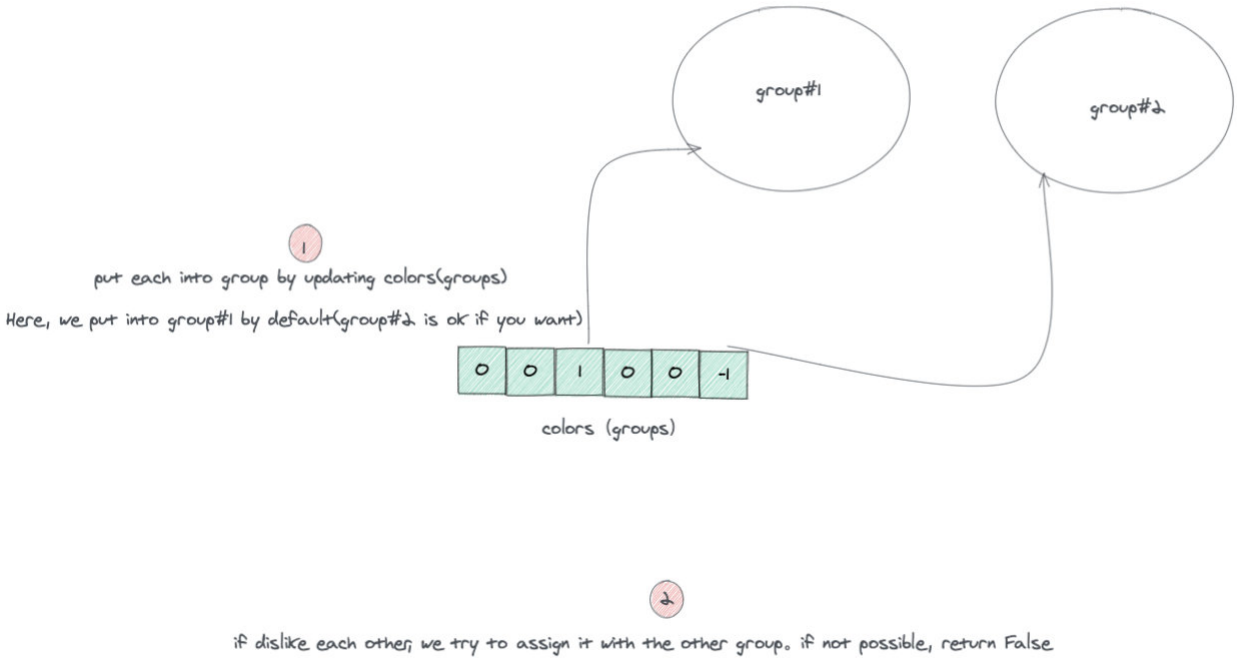


image.png

实际上，我们已经用 colors 记录了分组信息，对于每一个人如果分组确定了，我们就更新 colors，那么对于一个人如果分配了一个组，并且他讨厌的人也被分组之后，**分配的组和它只能是一组**，那么“就是不可以分配另外一组”。

代码表示就是：

```
# 其中j 表示当前是第几个人，N表示总人数。 dfs的功能就是根据colors和graph分配组，true表示可以分，false表示不可以，具体代码见代码区。
if colors[j] == 0 and not self.dfs(graph, colors, j, -1 * color, N)
```

关键点

- 二分图
- 染色法
- 图的建立和遍历
- colors 数组

代码

代码支持：Python，CPP，JS，Java

Python Code:

```python
class Solution:
    def dfs(self, graph, colors, i, color, N):
        colors[i] = color
        for j in range(N):
            # dislike eachother
            if graph[i][j] == 1:
                if colors[j] == color:
                    return False
                if colors[j] == 0 and not self.dfs(graph, colors, j, -1 * color, N):
                    return False
        return True

    def possibleBipartition(self, N: int, dislikes: List[List[int]]) -> bool:
        graph = [[0] * N for i in range(N)]
        colors = [0] * N
        for a, b in dislikes:
            graph[a - 1][b - 1] = 1
            graph[b - 1][a - 1] = 1
        for i in range(N):
            if colors[i] == 0 and not self.dfs(graph, colors, i, 1, N):
                return False
        return True
```

CPP Code:

```cpp
class Solution {
    vector<vector<int>> G;
    vector<int> _colors;
public:
    bool possibleBipartition(int n, vector<vector<int>>& dislikes)
    {
        G = vector<vector<int>>(n);
        for (const auto& d : dislikes)
        {
            G[d[0] - 1].push_back(d[1] - 1);
            G[d[1] - 1].push_back(d[0] - 1);
        }
        _colors = vector<int>(n, 0); // 0: 没颜色, 1: 染成红色, -1: 染成蓝色
        for (int i = 0; i < n; i++)
            if (_colors[i] == 0 && !dfs(i, 1))
                return false;
        return true;
    }
    bool dfs(int cur, int color)
```

```
    {
        _colors[cur] = color;
        for (int next : G[cur])
        {
            if (_colors[next] == color)
                return false;
            if (_colors[next] == 0 && !dfs(next, -color))
                return false;
        }
        return true;
    }
};
```

JS Code:

```
const possibleBipartition = (N, dislikes) => {
  let graph = [...Array(N + 1)].map(() => Array()), // 动态创建二维数组
    colors = Array(N + 1).fill(-1);

  // build the undirected graph
  for (const d of dislikes) {
    graph[d[0]].push(d[1]);
    graph[d[1]].push(d[0]);
  }

  const dfs = (cur, color = 0) => {
    colors[cur] = color;
    for (const nxt of graph[cur]) {
      if (colors[nxt] !== -1 && colors[nxt] === color) return false; // conflict
      if (colors[nxt] === -1 && !dfs(nxt, color ^ 1)) return false;
    }
    return true;
  };

  for (let i = 0; i < N; ++i) if (colors[i] === -1 && !dfs(i, 0)) return false;

  return true;
};
```

Java Code:

```
class Solution {
    ArrayList<Integer>[] graph;
    Map<Integer, Integer> color;

    public boolean possibleBipartition(int N, int[][] dislikes) {
        graph = new ArrayList[N+1];
        for (int i = 1; i <= N; ++i)
```

```
            graph[i] = new ArrayList();

        for (int[] edge: dislikes) {
            graph[edge[0]].add(edge[1]);
            graph[edge[1]].add(edge[0]);
        }

        color = new HashMap();
        for (int node = 1; node <= N; ++node)
            if (!color.containsKey(node) && !dfs(node, 0))
                return false;
        return true;
    }

    public boolean dfs(int node, int c) {
        if (color.containsKey(node))
            return color.get(node) == c;
        color.put(node, c);

        for (int nei: graph[node])
            if (!dfs(nei, c ^ 1))
                return false;
        return true;
    }
```

复杂度分析

另外 V 和 E 分别为图中的点和边的数目。

- 时间复杂度：由于 colors[i] == 0 才会进入 dfs，而 colors[i] 会在 dfs 中被染色且不撤销染色，因此每个点最多被处理一次，并且每个点的边也最多处理一次，因此时间复杂度为 $\mathrm{O}(\mathrm{V}+\mathrm{E})$
- 空间复杂度： Python 代码使用了邻接矩阵，因此空间复杂度为 $\mathrm{O}(\mathrm{V}^2)$，而 CPP 代码使用了类似邻接表的结果，因此空间复杂度为 $\mathrm{O}(\mathrm{V}+\mathrm{E})$。

> Python 代码也轻松实现 v + e 的空间复杂度，这个问题留给大家来完成。

相关问题

- 785. 判断二分图

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 45K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

上一页 下一页