<https://leetcode.com/problems/is-graph-bipartite/>

There is an **undirected** graph with n nodes, where each node is numbered between 0 and n - 1. You are given a 2D array graph, where graph[u] is an array of nodes that node u is adjacent to. More formally, for each v in graph[u], there is an undirected edge between node u and node v. The graph has the following properties:

* There are no self-edges (graph[u] does not contain u).
* There are no parallel edges (graph[u] does not contain duplicate values).
* If v is in graph[u], then u is in graph[v] (the graph is undirected).
* The graph may not be connected, meaning there may be two nodes u and v such that there is no path between them.

A graph is **bipartite** if the nodes can be partitioned into two independent sets A and B such that **every** edge in the graph connects a node in set A and a node in set B.

Return true *if and only if it is* ***bipartite***.

**Example 1:**

Input: graph = [[1,2,3],[0,2],[0,1,3],[0,2]]

Output: false

Explanation: There is no way to partition the nodes into two independent sets such that every edge connects a node in one and a node in the other.

**Example 2:**

Input: graph = [[1,3],[0,2],[1,3],[0,2]]

Output: true

Explanation: We can partition the nodes into two sets: {0, 2} and {1, 3}.

**Constraints:**

* graph.length == n
* 1 <= n <= 100
* 0 <= graph[u].length < n
* 0 <= graph[u][i] <= n - 1
* graph[u] does not contain u.
* All the values of graph[u] are **unique**.
* If graph[u] contains v, then graph[v] contains u.

**Attempt 1: 2022-11-26**

**Solution 1:  DFS (10min)**

class Solution {

public boolean isBipartite(int[][] graph) {

int n = graph.length;

int[] colors = new int[n];

for(int i = 0; i < n; i++) {

if(colors[i] == 0 && !helper(i, graph, colors, 1)) {

return false;

}

}

return true;

}

private boolean helper(int i, int[][] graph, int[] colors, int color) {

if(colors[i] != 0) {

return colors[i] == color;

}

colors[i] = color;

for(int neighbour : graph[i]) {

if(!helper(neighbour, graph, colors, -color)) {

return false;

}

}

return true;

}

}

https://leetcode.com/problems/is-graph-bipartite/discuss/115487/Java-Clean-DFS-solution-with-Explanation/247133

It should be O(V+E), same as BFS since in addition to the usual BFS, we're only doing constant amount of work.

Reposting from below comment:

"BFS takes O(V+E) in case of an adjacency list.

We take O(V) time as each of the vertex is pushed once into the queue.

Now, for each vertex, we scan all its neighbours and do some constant amount of work to mark the colour. To calculate the number of edges traversed in each vertex is non-trivial, however since this is an undirected graph, we observe that each edge will be visited twice i.e if there is an edge between i and j, it will be visited once in i's list and once in j's list. Thus, this gives a complexity of O(2E) which is O(E).

Hence, the overall complexity is O(V+E)."

**Refer to**

<https://www.cnblogs.com/grandyang/p/8519566.html>

原来输入数组中的graph[i]，表示顶点i所有相邻的顶点，比如对于例子1来说，顶点0和顶点1，3相连，顶点1和顶点0，2相连，顶点2和结点1，3相连，顶点3和顶点0，2相连。这道题让我们验证给定的图是否是二分图，所谓二分图，就是可以将图中的所有顶点分成两个不相交的集合，使得同一个集合的顶点不相连。为了验证是否有这样的两个不相交的集合存在，我们采用一种很机智的染色法，大体上的思路是要将相连的两个顶点染成不同的颜色，一旦在染的过程中发现有两连的两个顶点已经被染成相同的颜色，说明不是二分图。这里我们使用两种颜色，分别用1和-1来表示，初始时每个顶点用0表示未染色，然后遍历每一个顶点，如果该顶点未被访问过，则调用递归函数，如果返回false，那么说明不是二分图，则直接返回false。如果循环退出后没有返回false，则返回true。在递归函数中，如果当前顶点已经染色，如果该顶点的颜色和将要染的颜色相同，则返回true，否则返回false。如果没被染色，则将当前顶点染色，然后再遍历与该顶点相连的所有的顶点，调用递归函数，如果返回 false 了，则当前递归函数的返回 false，循环结束返回 true

Try to color all nodes with alternating colors. If we came back to an already-colored node with a different color, then we return false. This is a typical application of DFS/BFS cycle detection described in Robert Sedgwick's Algorithms (4th ed.) book. Slight explanation of correctness: why is it that we can be sure that as long as we have a cycle, where we try to color one node with a different color the second time, we know for sure the graph is bad? This is because the node just prior to the conflicting node must have the same color as the conflicting node itself. And as long as such a cycle exists, there is no way you can bipartite-partition all nodes within this cycle because, well, these two just won't do. What if there are no cycles? Well, cycle-free graphs are always bipartite. Multiple DFS is required since the nodes might not be fully connected. Each DFS will color one independent subgraph. If one DFS discovers that one subgraph is okay, we can safely move on: it does not matter anymore because this subgraph and its consisting nodes won't further affect any other independent subgraph.

class Solution {

public boolean isBipartite(int[][] graph) {

int[] colors = new int[graph.length];

for (int i = 0; i < graph.length; i++) {

// If node not paint on any color yet, but not able to paint

// any color further is not a bipartite graph

// Start paint with color = 1

if (colors[i] == 0 && !helper(i, graph, colors, 1)) {

return false;

}

}

return true;

}

private boolean helper(int node, int[][] graph, int[] colors, int color) {

// If node painted, check if it match the color try to paint on it

// if not match means not bipartite graph

if (colors[node] != 0) {

return colors[node] == color;

}

colors[node] = color;

for (int next: graph[node]) {

// The adjacent nodes paint with different color as -1

if (!helper(next, graph, colors, -color)) {

return false;

}

}

return true;

}

}

**Solution 2:  BFS (10min)**

class Solution {

public boolean isBipartite(int[][] graph) {

int n = graph.length;

int[] colors = new int[n];

// For each un-painted node start a BFS (similar like for

// each un-painted node start a DFS)

for(int i = 0; i < n; i++) {

if(colors[i] == 0) {

Queue<Integer> q = new LinkedList<Integer>();

q.offer(i);

colors[i] = 1;

while(!q.isEmpty()) {

int cur = q.poll();

for(int next : graph[cur]) {

// If adjacent node of current node not painted

// paint it with different color than current node

// e.g adjacent node is 0, current node is 1, then

// paint adjacent node as -1

if(colors[next] == 0) {

q.offer(next);

colors[next] = -colors[cur];

// If adjacent node already painted, but the color

// same as current node, then circle happen, return

// false, colors[next] == colors[cur] and

// colors[next] != -colors[cur] both works

} else if(colors[next] == colors[cur]) {

return false;

}

}

}

}

}

return true;

}

}

https://leetcode.com/problems/is-graph-bipartite/discuss/115487/Java-Clean-DFS-solution-with-Explanation/247133

It should be O(V+E), same as BFS since in addition to the usual BFS, we're only doing constant amount of work.

Reposting from below comment:

"BFS takes O(V+E) in case of an adjacency list.

We take O(V) time as each of the vertex is pushed once into the queue.

Now, for each vertex, we scan all its neighbours and do some constant amount of work to mark the colour. To calculate the number of edges traversed in each vertex is non-trivial, however since this is an undirected graph, we observe that each edge will be visited twice i.e if there is an edge between i and j, it will be visited once in i's list and once in j's list. Thus, this gives a complexity of O(2E) which is O(E).

Hence, the overall complexity is O(V+E)."

**Refer to**

<https://www.cnblogs.com/grandyang/p/8519566.html>

我们再来看一种迭代的解法，整体思路还是一样的，还是遍历整个顶点，如果未被染色，则先染色为1，然后使用 BFS 进行遍历，将当前顶点放入队列 queue 中，然后 while 循环 queue 不为空，取出队首元素，遍历其所有相邻的顶点，如果相邻顶点未被染色，则染成和当前顶点相反的颜色，然后把相邻顶点加入 queue 中，否则如果当前顶点和相邻顶点颜色相同，直接返回 false，循环退出后返回 true

class Solution {

public boolean isBipartite(int[][] graph) {

int[] colors = new int[graph.length];

for(int i = 0; i < graph.length; i++) {

// For each un-painted node start a BFS (similar like for

// each un-painted node start a DFS)

if(colors[i] == 0) {

Queue<Integer> queue = new LinkedList<Integer>();

queue.offer(i);

colors[i] = 1;

while(!queue.isEmpty()) {

int cur = queue.poll();

for(int next : graph[cur]) {

// If adjacent node of current node not painted

// paint it with different color than current node

// e.g adjacent node is 0, current node is 1, then

// paint adjacent node as -1

if(colors[next] == 0) {

queue.offer(next);

colors[next] = -colors[cur];

// If adjacent node already painted, but the color

// same as current node, then circle happen, return

// false, colors[next] == colors[cur] and

// colors[next] != -colors[cur] both works

} else if(colors[next] == colors[cur]) {

return false;

}

}

}

}

}

return true;

}

}