Daily Coding Problem #207

Problem

This problem was asked by Dropbox.

Given an undirected graph G, check whether it is bipartite. Recall that a graph is bipartite if its vertices can be divided into two independent sets, U and V, such that no edge connects vertices of the same set.

Solution

Another way of phrasing this problem is that we want to know if it is possible to assign two colors to the graph vertices such that there are no edges between vertices of the same color. We can solve this using a modified breadth first search. In particular, each time we visit a node in the graph, we mark all its neighbors as having the opposite color. If a vertex's neighbor already has a color, and it is the same color as the vertex itself, we know the graph is not bipartite. Else, if we are able to visit all vertices and assign colors without conflict, the graph is indeed bipartite.

Let's walk through a simple example. Here we will represent the graph with an adjacency list, mapping each vertex to a list of its neighbors:

```
graph = {
    0: [1, 2],
    1: [0, 2],
    2: [0, 1, 3],
    3: [2]
}
```

We start our BFS by selecting a start vertex, let's say `0`, and assigning it a color value of 1. Following this, we assign each of its neighbors, `1` and `2`, the value -1, and append them to our queue. At this point a mapping from vertices to colors would look like this:

```
{
    0: 1,
    1: -1,
    2: -1,
    3: 0
}
```

When we visit the next element on the queue, `1`, we see that one of its neighbors is `2`. But since have already assigned these two nodes the same color, we know this graph cannot be bipartite.

Here is how we could implement this:

```
def is_bipartite(graph):
    start = graph.keys()[0]
    queue = [start]
    colors = [0 for _ in range(len(graph))]
    colors[start] = 1

    while queue:
        vertex = queue.pop(0)
```

```
            for neighbor in graph[vertex]:
                if colors[neighbor] == 0:
                    colors[neighbor] = -colors[vertex]
                    queue.append(neighbor)
                elif colors[neighbor] == colors[vertex]:
                    return False

        return True
```

This works if all the vertices in the graph are connected. If not, however, this algorithm could terminate without examining a non-bipartite subgraph. As an example, imagine if the above graph were actually a small part of a larger graph with vertices 5, 6, 7 and so on.

To handle this case, we can modularize our method into a helper function, and keep calling it on vertices that have not yet been visited:

```
def helper(graph, start, colors):
    queue = [start]
    colors[start] = 1

    while queue:
        vertex = queue.pop(0)

        for neighbor in graph[vertex]:
            if colors[neighbor] == 0:
                colors[neighbor] = -colors[vertex]
                queue.append(neighbor)
            elif colors[neighbor] == colors[vertex]:
                return False

    return True

def is_bipartite(graph):
    colors = [0 for _ in range(len(graph))]
    for vertex in graph.keys():
        if colors[vertex] == 0:
```

```
            if not helper(graph, vertex, colors):
                return False

    return True
```

The time complexity of this algorithm is the same as that of breadth first search: `O(|V| + |E|)`, since in the worst case we must explore every vertex and edge of our graph.
