Daily Coding Problem #218

Problem

This problem was asked by Yahoo.

Write an algorithm that computes the reversal of a directed graph. For example, if a graph consists of `A -> B -> C`, it should become `A <- B <- C`.

Solution

Our first instinct might be to traverse the graph, finding all the edges. For each edge, we can swap its vertices and add the updated edge to a new graph.

For example, if we find an edge directed from `A` to `B`, with a weight of `5`, we would add a corresponding edge to our transposed graph with the same weight, but from `B` to `A`. Traversing the graph can be done with depth-first search.

```
from collections import defaultdict

def helper(graph, node, visited, transpose):
    if node not in visited:
```

```
        visited.add(node)

        for n, weight in graph[node]:
            transpose[n] += [(node, weight)]
            helper(graph, n, visited, transpose)

    return transpose

def reverse(graph):
    start = graph.keys()[0]
    transpose = helper(graph, start, set(), defaultdict(list))
    return transpose
```

This will take `O(|V| + |E|)` time, and building up the reversed graph will use `O(|V| + |E|)` space.

However, there is a simpler approach. If we already have our graph represented as an adjacency list, all we need to do is generate a new adjacency list with swapped keys and values.

In other words, suppose we our graph is represented as `{0: [2, 3], 1: [3, 4], 2: [1]}`. We iterate through each key k, and every value v. If v is not a key in our transposed graph, we add it, along with the value k. Otherwise, we append v to the existing list of k's edges. In this example the first few edges would be `{2: 0, 3: 0, 3: 1, ...}`.

```
def reverse(graph):
    transpose = defaultdict(list)

    for node in graph:
        for neighbor, weight in graph[node]:
            transpose[neighbor].append((node, weight))

    return transpose
```

The time and space complexity is unchanged from above.