Daily Coding Problem #246

Problem

This problem was asked by Dropbox.

Given a list of words, determine whether the words can be chained to form a circle. A word `X` can be placed in front of another word `Y` in a circle if the last character of `X` is same as the first character of `Y`.

For example, the words `['chair', 'height', 'racket', touch', 'tunic']` can form the following circle: `chair --> racket --> touch --> height --> tunic --> chair`.

Solution

It will be helpful to examine this problem as a graph. Consider each starting letter as a vertex, and each word as an edge that connects the starting letter and ending letter of that word. We can represent this in an adjacency list like so:

```
def make_graph(words):
    graph = defaultdict(list)

    for word in words:
        graph[word[0]].append(word[-1])
```

```
    return graph
```

Finding out whether we can chain these strings together is equivalent to determining whether there is a cycle that goes through all edges, or in other words an Eulerian cycle.

An efficient method of solving this takes advantage of the fact that a graph has a Eulerian cycle if and only if it satisfies the following two properties:

- For each vertex, its in-degree equals its out-degree.
- The graph is strongly connected.

The in-degree of a vertex refers to how many edges are directed into that vertex, while the out-degree refers to how many edges are directed out from that vertex. For our problem, this corresponds to how many words end and start with a given letter, respectively. To check that these values are equal, we can simply iterate over our graph and count them up.

```
def are_degrees_equal(graph):
    in_degree = defaultdict(int)
    out_degree = defaultdict(int)

    for key, values in graph.items():
        for v in values:
            out_degree[key] += 1
            in_degree[v] += 1

    return in_degree == out_degree
```

It remains to determine whether the graph is strongly connected. In other words, we must determine whether every node is reachable from every other node. Instead of trying a depth-first search from every node, there is a straightforward algorithm we can use. Choosing a vertex at random, we first see if we can visit all other vertices from it. Then, we reverse all edges in the graph and see if this is still possible.

```
def find_component(graph, visited, current_word):
    visited.add(current_word)

    for neighbor in graph[current_word]:
```

```
        if neighbor not in visited:
            find_component(graph, visited, neighbor)

    return visited

def is_connected(graph):
    start = list(graph)[0]
    component = find_component(graph, set(), start)

    reversed_graph = defaultdict(list)
    for key, values in graph.items():
        for v in values:
            reversed_graph[v].append(key)
    reversed_component = find_component(graph, set(), start)

    return component == reversed_component == graph.keys()
```

Creating the graph and checking the in-degree and out-degree of each vertex are both `O(N)`, where `N` is the number of words. Determining whether the graph is connected involves two depth-first searches, each with time complexity `O(V + E)`. But since the number of edges or vertices can be no greater than the number of strings, the whole algorithm runs in `O(N)` time.

The full solution is as follows:

```
from collections import defaultdict

def find_component(graph, visited, current_word):
    visited.add(current_word)

    for neighbor in graph[current_word]:
        if neighbor not in visited:
            find_component(graph, visited, neighbor)

    return visited

def is_connected(graph):
    start = list(graph)[0]
    component = find_component(graph, set(), start)
```

```
    reversed_graph = defaultdict(list)
    for key, values in graph.items():
        for v in values:
            reversed_graph[v].append(key)
    reversed_component = find_component(graph, set(), start)

    return component == reversed_component == graph.keys()

def are_degrees_equal(graph):
    in_degree = defaultdict(int)
    out_degree = defaultdict(int)

    for key, values in graph.items():
        for v in values:
            out_degree[key] += 1
            in_degree[v] += 1


    return in_degree == out_degree

def make_graph(words):
    graph = defaultdict(list)

    for word in words:
        graph[word[0]].append(word[-1])

    return graph

def can_chain(words):
    graph = make_graph(words)

    degrees_equal = are_degrees_equal(graph)
    connected = is_connected(graph)

    return degrees_equal and connected
```