Daily Coding Problem #72

Problem

This problem was asked by Google.

In a directed graph, each node is assigned an uppercase letter. We define a path's value as the number of most frequently-occurring letter along that path. For example, if a path in the graph goes through "ABACA", the value of the path is 3, since there are 3 occurrences of 'A' on the path.

Given a graph with n nodes and `m` directed edges, return the largest value path of the graph. If the largest value is infinite, then return null.

The graph is represented with a string and an edge list. The `i`-th character represents the uppercase letter of the `i`-th node. Each tuple in the edge list `(i, j)` means there is a directed edge from the `i`-th node to the `j`-th node. Self-edges are possible, as well as multi-edges.

For example, the following input graph:

```
ABACA
```

```
[(0, 1),
 (0, 2),
 (2, 3),
 (3, 4)]
```

Would have maximum value 3 using the path of vertices `[0, 2, 3, 4]`, `(A, A, C, A)`.

The following input graph:

```
A
```

```
[(0, 0)]
```

Should return null, since we have an infinite loop.

Solution

The naive solution here would be to try every single path from every vertex, count up each path's value and keep track of the maximum value we've seen.

To do this, we can use DFS to try every path as well as return null if we come across a cycle. The Counter module in Python is quite handy in this case:

```
from collections import Counter

def max_path(s, lst):
    adj = [[] for v in s]
    # Build adjacency list
    for u, v in lst:
        adj[u].append(v)

    maximum_path = 0
    # Try every path from node v.
    for v in range(len(s)):
        stack = [(s[v], set([v]), v)] # Every item in the stack has form (path_string, visited, current_node)
        while stack:
            path_string, visited, current_node = stack.pop()
            # Count value of current path and update maximum_path if necessary
            cnt = Counter(path_string)
            _, path_val = cnt.most_common(1)[0]
            maximum_path = max(maximum_path, path_val)
            for neighbour in adj[current_node]:
                if neighbour in visited:
                    # There is a cycle.
                    return None
                stack.append((path_string + s[neighbour], visited.union([neighbour]), neighbour))
    return maximum_path
```

However, this would be terribly slow. DFS is O(V + E), where V and E are the sizes of the vertices and edges. Since we also evaluate the current path each time, our algorithm is O(V * (V + E)). Let's try to improve this runtime.

Notice that we're recomputing the whole path on each iteration. This is inefficient since, for example, only one character could change, so we should only need to increment that one character. This sounds like a good problem for dynamic programming.

Furthermore, notice that since we're using the alphabet of uppercase characters, we have a fixed number (26) of potential values that contribute to the longest chain.

Let's keep a matrix of size N by 26. `A[i][j]` will contain the maximum value of the path that can be made from the character `i` (where `i` will index into the alphabet, so A = 0, B = 1, etc. Then we'll use the following recurrence to keep track of the path with the largest value:

- When we get to a node v, we'll do DFS on all its neighbours.
- Then `A[v][j]` will be the maximum of all `A[neighbour][j]` for all its neighbours.
- Then, we also need to count the current node too, so increment `A[v][current_char]` by one, where `current_char` is the current node's assigned letter.

We will use DFS, like before, to actually search the graph as well as determining if we have a cycle.

```
VISITED = 0
UNVISITED = 1
VISITING = 2

def max_path(s, lst):
    adj = [[] for v in s]
    # Build adjacency list
    for u, v in lst:
        adj[u].append(v)

    # Create matrix cache
    dp = [[0 for _ in range(26)] for _ in range(len(s))]
    state = {v: UNVISITED for v in range(len(s))}

    def dfs(v):
        state[v] = VISITING
        for neighbour in adj[v]:
            if state[neighbour] == VISITING:
                # We have a cycle
```

```
                    return True
                dfs(neighbour)
                for i in range(26):
                    dp[v][i] = dp[neighbour][i]
            current_char = ord(s[v]) - ord('A')
            dp[v][current_char] += 1
            state[v] = VISITED

        # Run DFS on graph
        for v in range(len(s)):
            if state[v] == UNVISITED:
                has_cycle = dfs(v)
                if has_cycle:
                    return None

        return max(max(v for v in node) for node in dp)
```

This will now just run in O(V + E) time, same as DFS.