Daily Coding Problem #255

Problem

This problem was asked by Microsoft.

The transitive closure of a graph is a measure of which vertices are reachable from other vertices. It can be represented as a matrix `M`, where `M[i][j] == 1` if there is a path between vertices `i` and `j`, and otherwise `0`.

For example, suppose we are given the following graph in adjacency list form:

```
graph = [
    [0, 1, 3],
    [1, 2],
    [2],
    [3]
]
```

The transitive closure of this graph would be:

```
[1, 1, 1, 1]
[0, 1, 1, 0]
[0, 0, 1, 0]
[0, 0, 0, 1]
```

Given a graph, find its transitive closure.

Solution

One algorithm we can use to solve this is a modified version of Floyd-Warshall.

Traditionally Floyd-Warshall is used for finding the shortest path between all vertices in a weighted graph. It works in the following way: for any pair of nodes `(i, j)`, check to see if there is an intermediate vertex k such that the cost of getting from `i` to k to `j` is less than the current cost of getting from `i` to `j`. This is generalized by examining each possible choice of k, and updating every `(i, j)` cost that can be improved.

In our case, we are concerned not with costs but simply with whether it is possible to get from `i` to `j`. So we can start with a boolean matrix `reachable` filled with zeros, except for the connections given in our adjacency matrix.

Then, for each intermediate node k, and for each connection `(i, j)`, if `reachable[i][j]` is zero but there is a path from `i` to k and from k to `j`, we should change it to one.

```
def closure(graph):
    n = len(graph)
    reachable = [[0 for _ in range(n)] for _ in range(n)]

    for i, v in enumerate(graph):
        for neighbor in v:
            reachable[i][neighbor] = 1

    for k in range(n):
        for i in range(n):
            for j in range(n):
                reachable[i][j] |= (reachable[i][k] and reachable[k][j])



    return reachable
```

Since we are looping through three levels of vertices, this will take $O(V^3)$ time. Our matrix uses $O(V^2)$ space.

An alternative method is to perform a depth-first search starting from each vertex. Initially, we will have a `reachable` matrix which is set to zero for all pairs of vertices. Then, for each vertex `i`, we recursively find all vertices adjacent to `i`, and adjacent to those adjacent to `i`, and so on. For any reachable vertex `j` found in this way, we update `reachable[i][j]` to be one.

```
def helper(reachable, graph, i, j):
    reachable[i][j] = 1

    for v in graph[j]:
        if reachable[i][v] == 0:
            reachable = helper(reachable, graph, i, v)

    return reachable

def closure(graph):
    n = len(graph)
    reachable = [[0 for _ in range(n)] for _ in range(n)]

    for i in range(n):
        reachable = helper(reachable, graph, i, i)

    return reachable
```

The time complexity of depth-first search is `O(V + E)`, so this algorithm will take `O(V * (V + E))`. In the case where our graph is maximally dense, $E = V^2$, so this will be similar to above.

Privacy Policy

Terms of Service

Press