Daily Coding Problem #294

Problem

This problem was asked by Square.

A competitive runner would like to create a route that starts and ends at his house, with the condition that the route goes entirely uphill at first, and then entirely downhill.

Given a dictionary of places of the form `{location: elevation}`, and a dictionary mapping paths between some of these locations to their corresponding distances, find the length of the shortest route satisfying the condition above. Assume the runner's home is location `0`.

For example, suppose you are given the following input:

```
elevations = {0: 5, 1: 25, 2: 15, 3: 20, 4: 10}
paths = {
    (0, 1): 10,
    (0, 2): 8,
    (0, 3): 15,
    (1, 3): 12,
    (2, 4): 10,
    (3, 4): 5,
    (3, 0): 17,
    (4, 0): 10
}
```

]

In this case, the shortest valid path would be `0 -> 2 -> 4 -> 0`, with a distance of 28.

Solution

Since we must go uphill before going downhill, there must be some point `p` on our route with maximum elevation. Therefore, we can solve this problem by finding the shortest route from `0` to `p`, and from `p` back to `0`, for all possible choices of `p`.

Each part of the method above can be solved in `O(E + V log V)` time using Dijkstra's algorithm, where `E` is the number of paths and `V` is the numbers of locations. In fact, though, there is a more efficient algorithm. If we consider only uphill or only downhill path segments, it will be impossible to form a cycle, so we are dealing with a directed acyclic graph. As a result, for each of these two subproblems, we can use a topological sort to determine in what order to visit the locations, and use this ordering to find the minimal cost path.

Topological sort ensures that for any directed edge `(u, v)` in our graph, `u` will come before `v` in our ordering. We can implement it using depth-first search, appending each newly seen to a stack. Assume for this function we have access to `edges`, a dictionary mapping each start location to a list of `(end, distance)` tuples.

```python
def helper(v, visited, stack, edges):
    visited[v] = True

    for neighbor, _ in edges[v]:
        if not visited[neighbor]:
            helper(neighbor, visited, stack, edges)

    stack.append(v)

def toposort(edges, num_vertices):
    visited = [False for _ in range(num_vertices)]
    stack = []

    for v in range(num_vertices):
        if not visited[v]:
            helper(v, visited, stack, edges)
    return stack
```

Implemented this way, topological sort will take `O(V + E)` time.

This stack will help us compute the shortest path between a source and any other destination. In particular, we can pop successive elements off the stack, and for each one update the distances between any of its neighbors and the source, if applicable.

```
def get_distances(edges, stack):
    dist = [float('inf') for _ in range(len(stack))]
    dist[0] = 0

    while stack:
        curr = stack.pop()

        for neighbor, distance in edges[curr]:
            if dist[neighbor] > dist[curr] + distance:
                dist[neighbor] = dist[curr] + distance

    return dist[1:]
```

We must call this method twice: once to compute the shortest paths between the runner's house and any uphill location, and a second time (with reversed edges) to compute the shortest downhill paths between those locations and the runner's house.

Each of these calls will take `O(V + E)` time, since we must traverse the entire graph in the worst case.

Finally, since we know all the possible ways to get to and from the highest locations, we can sum up each pair of distances and return the minimum overall. The main function will be as follows:

```
def shortest_route(elevations, paths):
    uphill_edges = defaultdict(list)

    downhill_edges = defaultdict(list)
    all_edges = defaultdict(list)

    for (start, end), distance in paths.items():
        all_edges[start].append((end, distance))
        if elevations[start] < elevations[end]:
            uphill_edges[start].append((end, distance))
        else:
            downhill_edges[end].append((start, distance))

    num_vertices = len(all_edges.keys())
    stack = toposort(all_edges, num_vertices)

    uphill_distances = get_distances(uphill_edges, list(stack))
    downhill_distances = get_distances(downhill_edges, list(stack))

    return min(x + y for x, y in zip(uphill_distances, downhill_distances))
```

The total running time of this algorithm will be `O(V + E)`, as this is an upper bound for each of our component functions. We have used a few extra dictionaries to store the uphill and downhill edges for the sake of clarity, but the space complexity will still be `O(V + E)`.