Daily Coding Problem #270

Problem

This problem was asked by Twitter.

A network consists of nodes labeled `0` to `N`. You are given a list of edges `(a, b, t)`, describing the time `t` it takes for a message to be sent from node `a` to node `b`. Whenever a node receives a message, it immediately passes the message on to a neighboring node, if possible.

Assuming all nodes are connected, determine how long it will take for every node to receive a message that begins at node `0`.

For example, given `N = 5`, and the following edges:

```
edges = [
    (0, 1, 5),
    (0, 2, 3),
    (0, 5, 4),
    (1, 3, 8),
    (2, 3, 1),
    (3, 5, 10),
    (3, 4, 5)
]
```

You should return 9, because propagating the message from `0 -> 2 -> 3 -> 4` will take

You should return 9, because propagating the message from `0 -> 2 -> 3 -> 4` will take that much time.

Solution

To help organize our input, we can think of the network nodes as vertices on a graph, and each connection as an edge. We will use a helper class that maps each node to a list of tuples of the form `(neighbor, time)`.

```
class Network:
    def __init__(self, N, edges):
        self.vertices = range(N + 1)
        self.edges = edges

    def make_graph(self):
        graph = {v: [] for v in network.vertices}

        for u, v, w in network.edges:
            graph[u].append((v, w))

        return graph
```

Finding the shortest amount of time it will take for each node to receive the message, then, is equivalent to finding the shortest path between vertex `0` and all other vertices. For this we can use Dijkstra's algorithm.

We will first create a dictionary, `times`, mapping each node to the minimum amount of time it takes for a message to propagate to it. Initially this will be infinite for all nodes except the start node, which is zero.

Then, we consider an unvisited node with the smallest propagation time. For each of its neighbors, we replace the propagation time for that neighbor with the time it would take

to go through the current node to get to that neighbor, if the latter is smaller. We continue this process until we have visited all nodes.

In the end, the largest value in our dictionary will represent the time it will take for the last node to get the message.

```
def propagate(network):
```

```
    graph = network.make_graph()
    times = {node: float('inf') for node in graph}
    times[0] = 0

    q = list(graph)
    while q:
        u = min(q, key=lambda x: times[x])
        q.remove(u)
        for v, time in graph[u]:
            times[v] = min(times[v], times[u] + time)

    return max(times.values())
```

Since we must find the minimum value of our dictionary for each unexamined node, and there are `N` nodes, this will take $O(N^2)$ time.

For sparse graphs, we can improve on this by using a priority queue, ordering each node by propagation time. To start, this queue will just hold node zero, with value zero.

Starting from `0`, then, each time we encounter a new neighbor, we add it to the queue, with value equal to the sum of the time from node zero to the current node, and from the current node to the neighbor. Whenever we pop a node off the queue that does not exist in our `times` dictionary, we add it with the corresponding value.

```
def propagate(network):
    graph = network.make_graph()
    times = {}

    q = [(0, 0)]
    while q:
        u, node = heapq.heappop(q)
        if node not in times:
            times[node] = u

            for neighbor, v in graph[node]:
                if neighbor not in times:
                    heapq.heappush(q, (u + v, neighbor))

    return max(times.values())
```

It takes $O(\log E)$ time to pop or push an element from the heap, and we must do this for

each edge, so the complexity of this algorithm is `O(E log E)`.