Daily Coding Problem #299

Problem

This problem was asked by Samsung.

A group of houses is connected to the main water plant by means of a set of pipes. A house can either be connected by a set of pipes extending directly to the plant, or indirectly by a pipe to a nearby house which is otherwise connected.

For example, here is a possible configuration, where `A`, `B`, and `C` are houses, and arrows represent pipes:

```
A <--> B <--> C <--> plant
```

Each pipe has an associated cost, which the utility company would like to minimize. Given an undirected graph of pipe connections, return the lowest cost configuration of pipes such that each house has access to water.

In the following setup, for example, we can remove all but the pipes from `plant` to `A`, `plant` to `B`, and `B` to `C`, for a total cost of 16.

```
pipes = {
    'plant': {'A': 1, 'B': 5, 'C': 20},
    'A': {'C': 15},
    'B': {'C': 10},
    'C': {}
```

```
    c : {}
}
```

Solution

When arrows appear in a problem, they are often a hint that a graph algorithm may be helpful. Thinking along these lines, we can consider each house as a vertex, and each pipe as an edge. Since we want to ensure that each house is connected in a single component of lowest cost, the solution will be to find a minimum spanning tree.

In a previous problem we achieved this using Kruskal's algorithm, so let us instead use Prim's algorithm this time around. Prim's is another greedy algorithm, whose key idea is this: starting with a given vertex, we can keep adding edges of minimal cost to a growing component, until these finally span every vertex.

To see how this works, let's work through this algorithm on our example above. Our initial vertex can be the plant, since we know all water must ultimately be sourced from there. In the following diagram we can see the edges we must add in order, and the connected component at any given time:

```
 Component Vertices | Edge to Add | Previous Map
-------------------------------------------------
{plant}             | (plant, A)  | prev[A] = plant
{plant, A}          | (plant, B)  | prev[B] = plant
{plant, A, B}       | (B, C)      | prev[C] = B
{plant, A, B, C}    |             |
```

Note that each time we add a vertex, we must take note of where the edge came from, so we are able to recreate the path in the end.

To efficiently figure out which edge to add at any given time, we will add all edges of each recently visited vertex to a min-heap, so that finding the lowest cost edge will just mean popping an element of the heap.

```
import heapq

def min_spanning_tree(pipes):
    heap = [(0, ('plant', 'plant'))]
    costs = {}; prev = {}
```

```
    seen = set()
    vertices = pipes.keys()

    while len(seen) != len(vertices):
        cost, edge = heapq.heappop(heap)
        u, v = edge

        if v not in seen:
            if cost < costs.get(v, float('inf')):
                costs[v] = cost
                prev[v] = u

            for neighbor, cost in pipes[v].items():
                heapq.heappush(heap, (cost, (v, neighbor)))

            seen.add(v)

    path = {v: [] for v in vertices}
    for u, v in prev.items():
        path[v].append(u)

    return path, sum(costs.values())
```

In the worst case we will need to add and remove each edge to our heap. Since each of these operations is `O(log N)`, the overall time and space complexity will be `O(N log N)`, where `N` is the number of pipes.