Daily Coding Problem Blog

Daily Coding Problem #234

Problem

This problem was asked by Microsoft.

Recall that the minimum spanning tree is the subset of edges of a tree that connect all its vertices with the smallest possible total edge weight. Given an undirected graph with weighted edges, compute the *maximum* weight spanning tree.

Solution

Finding the maximum spanning tree is actually not too different from finding the minimum spanning tree. One algorithm we can use is as follows:

- First, sort the edges by decreasing order of weight.
- Then, for each edge, check if adding it to the solution set will turn the tree into a cycle. If not, add it.

By the end, we will either have a maximum spanning tree, or we will find that such a tree does not exist.

```
def max_spanning_tree(graph):
    """
    Keep adding edges of maximal weight, as long as they do not make a cycle.
    Graph is a dict containing a list of vertices and a list of (v1, v2, weight)
edges.
    """

    tree = set()
    n_vertices = len(graph['vertices'])
    graph['edges'].sort(key=lambda x: x[2], reverse=True)

    for edge in graph['edges']:
        if not makes_cycle(tree, edge):
            tree.add(edge)

    return tree if len(tree) == n_vertices - 1 else None
```

The hard part is still unanswered: how do we define `makes_cycle`? One way would be to perform a depth-first search through the tree, seeing if we can start with one vertex of the edge and get to the other. If so, adding the new edge would create a loop, and we should return `False`.

However, since performing DFS is `O(V + E)`, and we must do this for each edge, this algorithm is `O(E * (V + E))`.

We can improve on this by using a disjoint-set data structure to store our edges and vertices. To see how this works, imagine that to start, there are a bunch of people (representing vertices) sitting in separate rooms (representing sets). At each turn, we pick two people, `A`, and `B` (representing an edge). If they are not in the same room, we move one to join the other. If they are already in the same room, adding such an edge would create a cycle, since there would be two ways to get from `A` to `B` on an undirected graph.

Translating this to the problem at hand, let us keep track of the set each vertex is assigned to using a list of parents. At first, `parents[0] == 0` `parents[1] == 1`, and so on. When we come across our first edge, say, `(3, 4)`, we assign `parents[3] = 4`, since we now know that v_3 is in the same set as v_4. We might next come across the edge `(2, 3)`, causing us to assign `parents[2] = 3`. After these two operations, `parents` would look like this: `[0, 1, 3, 4, 4, ...]`.

We can continue this process for each edge. As long as both vertices in the edge are not already in the same set, we add the edge to our solution.

To find out which set each vertex belongs to, we just need to trace the path of its ancestors. That is, suppose we wanted to find where vertex `2` is. We would look at `parent[2]`, which directs us to `parent[3]`, which directs us to `parent[4]`. Since `parent[4] == 4`, we cannot go any further, so we should return 4.

```
class DisjointSet:
    def __init__(self, n):
        self.parents = [i for i in range(n)]

    def find(self, v):
        while v != self.parents[v]:
            v = self.parents[v]
        return v

    def join(self, v1, v2):
        s1 = self.find(v1)
        s2 = self.find(v2)
        self.parents[s1] = s2

def max_spanning_tree(graph):
    """
    Keep adding edges of maximal weight if they join together disjoint sets of
vertices.
    Graph is a dict containing a list of vertices and a list of (v1, v2, weight)
edges.
    """
    tree = set()
    n = len(graph['vertices'])
    ds = DisjointSet(n)
    graph['edges'].sort(key=lambda x: x[2], reverse=True)

    for edge in graph['edges']:
        if ds.find(edge[0]) != ds.find(edge[1]):
            tree.add(edge)
            ds.join(edge[0], edge[1])

    return tree if len(tree) == n - 1 else None
```

You might ask, how is this any more efficient that the original solution? If we perform a series of joins that moves v_2 to set `1`, v_3 to set `2`, and so on, all the way up to moving v_N to set `N - 1`, then `find(`v_N`)` will be `O(N)`.

This is true, but fortunately it can be fixed. Instead of arbitrarily reassigning one vertex to be the parent of the other, we should make the parent be the set that already has more

children. That way, it will be more likely that the vertex will be closer to the root, so it will take around `O(log N)` steps to find the root of a given vertex.

But we can reduce this even further! Note that when we first perform `find` on a vertex, it may take several steps to get to the root. But once we are done, we know exactly what the root should be. So if we make the root the direct parent of all the vertices involved in those steps, the next `find` operation for any of them will be `O(1)`.

Together, these two optimizations would look like this:

```
class DisjointSet:
    def __init__(self, n):
        self.parents = [i for i in range(n)]
        self.sizes = [1] * n

    def find(self, v):
        root = v
        while root != self.parents[root]:
            root = self.parents[root]

        step = v
        while step != root:
            step, self.parents[step] = self.parents[step], root
        return root

    def join(self, v1, v2):
        s1 = self.find(v1)
        s2 = self.find(v2)

        small, big = (s1, s2) if self.sizes[s1] < self.sizes[s2] else (s2, s1)
        self.parents[small] = big
        self.sizes[big] += self.sizes[small]

def max_spanning_tree(graph):
    """
    Keep adding edges of maximal weight if they join together disjoint sets of
vertices.
    Graph is a dict containing a list of vertices and a list of (v1, v2, weight)
edges.
    """
    tree = set()
    n = len(graph['vertices'])
    ds = DisjointSet(n)
    graph['edges'].sort(key=lambda x: x[2], reverse=True)

    for edge in graph['edges']:
        if ds.find(edge[0]) != ds.find(edge[1]):
            tree.add(edge)
            ds.join(edge[0], edge[1])

    return tree if len(tree) == n - 1 else None
```

The time complexity of this is `O(E * log E)`, since the `find` and `join` operations are now dominated by the time it takes to initially sort the list of edges.