Chapter 16: Kruskal's Algorithm

Section 16.1: Optimal, disjoint-set based implementation

We can do two things to improve the simple and sub-optimal disjoint-set subalgorithms:

1. **Path compression heuristic**: `findSet` does not need to ever handle a tree with height bigger than 2. If it ends up iterating such a tree, it can link the lower nodes directly to the root, optimizing future traversals;

```
subalgo findSet(v: a node):
    if v.parent != v
        v.parent = findSet(v.parent)
    return v.parent
```

2. **Height-based merging heuristic**: for each node, store the height of its subtree. When merging, make the taller tree the parent of the smaller one, thus not increasing anyone's height.

```
subalgo unionSet(u, v: nodes):
    vRoot = findSet(v)
    uRoot = findSet(u)

    if vRoot == uRoot:
        return

    if vRoot.height < uRoot.height:
        vRoot.parent = uRoot
    else if vRoot.height > uRoot.height:
        uRoot.parent = vRoot
    else:
        uRoot.parent = vRoot
        uRoot.height =  uRoot.height + 1
```

This leads to `O(alpha(n))` time for each operation, where `alpha` is the inverse of the fast-growing Ackermann function, thus it is very slow growing, and can be considered `O(1)` for practical purposes.

This makes the entire Kruskal's algorithm `O(m log m + m) = O(m log m)`, because of the initial sorting.

Note

Path compression may reduce the height of the tree, hence comparing heights of the trees during union operation might not be a trivial task. Hence to avoid the complexity of storing and calculating the height of the trees the resulting parent can be picked randomly:

```
subalgo unionSet(u, v: nodes):
    vRoot = findSet(v)
    uRoot = findSet(u)

    if vRoot == uRoot:
        return
    if random() % 2 == 0:
        vRoot.parent = uRoot
    else:
        uRoot.parent = vRoot
```

In practice this randomised algorithm together with path compression for `findSet` operation will result in

comparable performance, yet much simpler to implement.

Section 16.2: Simple, more detailed implementation

In order to efficiently handle cycle detection, we consider each node as part of a tree. When adding an edge, we check if its two component nodes are part of distinct trees. Initially, each node makes up a one-node tree.

```
algorithm kruskalMST(G: a graph)
    sort Gs edges by their value
    MST = a forest of trees, initially each tree is a node in the graph
    for each edge e in G:
        if the root of the tree that e.first belongs to is not the same
        as the root of the tree that e.second belongs to:
            connect one of the roots to the other, thus merging two trees

    return MST, which now a single-tree forest
```

Section 16.3: Simple, disjoint-set based implementation

The above forest methodology is actually a disjoint-set data structure, which involves three main operations:

```
subalgo makeSet(v: a node):
    v.parent = v    <- make a new tree rooted at v


subalgo findSet(v: a node):
    if v.parent == v:
        return v
    return findSet(v.parent)

subalgo unionSet(v, u: nodes):
    vRoot = findSet(v)
    uRoot = findSet(u)

    uRoot.parent = vRoot

algorithm kruskalMST(G: a graph):
    sort Gs edges by their value
    for each node n in G:
        makeSet(n)
    for each edge e in G:
        if findSet(e.first) != findSet(e.second):
            unionSet(e.first, e.second)
```

This naive implementation leads to `O(n log n)` time for managing the disjoint-set data structure, leading to `O(m*n log n)` time for the entire Kruskal's algorithm.

Section 16.4: Simple, high level implementation

Sort the edges by value and add each one to the MST in sorted order, if it doesn't create a cycle.

```
algorithm kruskalMST(G: a graph)
    sort Gs edges by their value
    MST = an empty graph
    for each edge e in G:
        if adding e to MST does not create a cycle:
            add e to MST
```

```
    return MST
```