CMPUT 275 — Tangible Computing II — Winter 2018
Minimum Spanning Trees and Union Find — Version 1.0

Problem 1)
Find a minimum spanning tree in the graph on the last page using Kruskal's algorithm.

Problem 2)
What do you get if you run Kruskal's algorithm on a graph that is not connected?

Problem 3)
The following is Prim's algorithm for computing a minimum spanning tree of a connected graph $G = (V, E)$ with edge weights $c(u, v) \geq 0, e \in E$.

- Let v be any arbitrarily chosen vertex.
- Let $S := \{v\}$ and $T := \{\}$.
- While $S \neq V$
  - Among the edges with exactly one endpoint S, pick one with minimum weight (break ties arbitrarily). Call this edge (u, w) where w is the endpoint not in S.
  - Update $S := S \cup \{w\}$ and $T = T \cup \{(u, w)\}$.
- **return** T

Address these problems.

1. Trace the execution of this algorithm on the same graph as in problem 1, starting with the leftmost vertex.
2. Think of how to implement this as efficiently as possible. $O(|E| \cdot \log |E|)$ is possible using heaps.
3. Recall that a spanning tree is a minimum spanning tree if and only if for every edge uv not in T, $c(u, v) \geq c(x, y)$ for every edge xy lying on the $u - v$ path in the tree.

   How can you use this fact to demonstrate that Prim's algorithm finds a minimum spanning tree? If it helps make your argument simpler, you can assume that no two edges have the same weight (though it does work in general).

Research It Yourself
If you are curious, look up **Boruvka's Algorithm**. It is an $O(|E| \log |V|)$ algorithm that can actually be implemented to run in $O(|E|)$ time when used on graphs that can be drawn with no crossing edges.

Problem 4)
Write the output of each of the following commands (except the first) in the space provided just below the instruction. Recall the `union` operation returns `True` if there was a merge and `False` if they were already in the same set.

```
uf = UnionFind({1,2,3,4,5})
```

```
uf.union(1, 2)
```

```
uf.union(2, 3)
```

```
uf.find(1) == uf.find(2)
```

```
uf.find(1) == uf.find(4)
```

```
uf.union(1, 3)
```

```
uf.union(4, 5)
```

```
uf.union(1, 5)
```

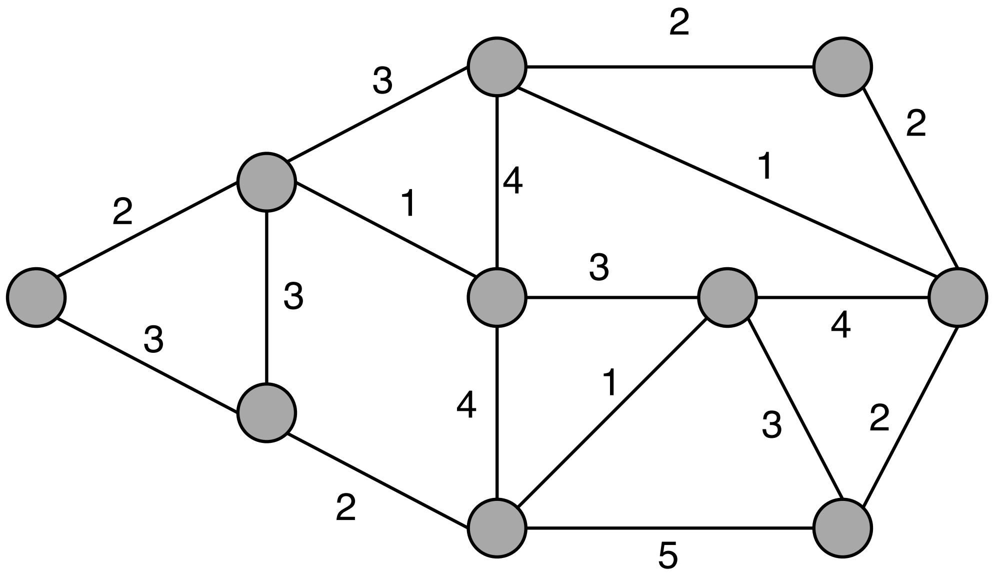
```
uf.union(2, 4)
```

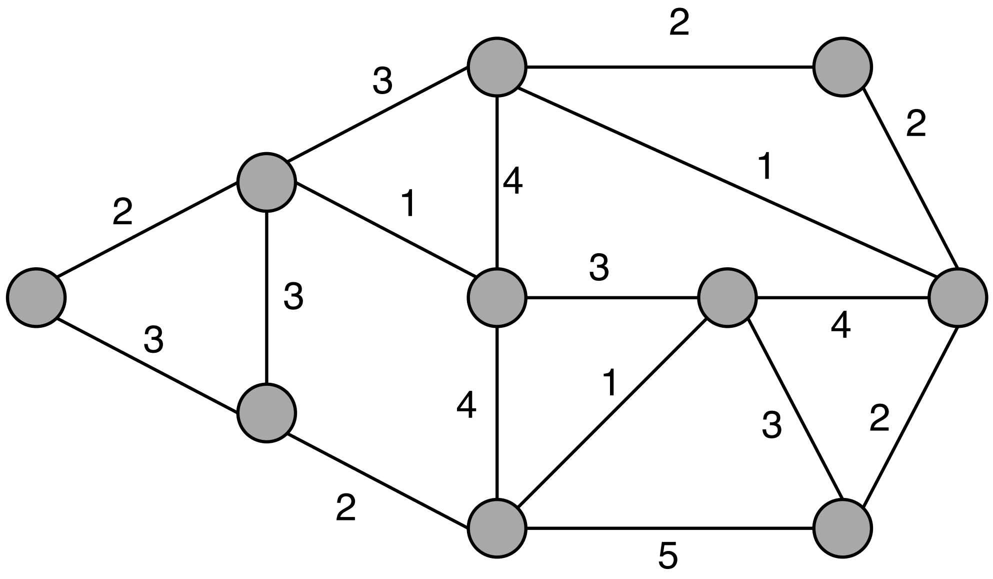
Also sketch the internal tree representation of the union-find data structure after each of these commands. Use path compression (i.e. the faster find method from the end of the slides).

Problem 5)
Do the same as in the previous exercise, except with the following commands. When `union(i,j)` is called where the representatives have the same rank, have the representative for `i` connect to the representative for `j`.

```
uf = UnionFind({1,2,3,4,5,6,7,8})

uf.union(1, 2)

uf.union(3, 4)

uf.union(5, 6)

uf.union(7, 8)

uf.union(2, 4)

uf.union(6, 8)

uf.union(4, 8)

uf.find(1)
```

Challenge Problem)
Generalize problem 5 to show that a single find operation can still take roughly $\log n$ time, even though, as discussed in class, the *average* running time of an operation is much less.


2
3
2
1
4
1
2
3
3
4
3
1
4
3
2
2
5


2
3
2
1
4
1
2
3
3
4
3
1
4
3
2
2
5