DSC 190 - Homework 04

Due: Thursday, February 11

Write your solutions to the following problems by either typing them up or handwriting them on another piece of paper. Unless otherwise noted by the problem's instructions, show your work or provide some justification for your answer. Homeworks are due via Gradescope on Thursday at 11:59 p.m.

Problem 0.

As you know, this is the first time this course is being offered. As such, your feedback is incredibly important! I'd like to hear your thoughts about DSC 190 – good and bad. If you fill out the anonymous Google Form at https://forms.gle/Wehdz17SsMNf6bSU7, you'll receive two points extra credit on this homework.

Note that the form is anonymous, so the only way I'll know who filled it out is if you tell me! You can do so by answering this question with a sentence that says you filled it out.

Programming Problem 1.

In a file named `dsf.py`, create a class named `DisjointSetForest`. This class should have `.find_set`, `.make_set`, and `.union` methods as described in lecture, as well as the following methods and attributes:

- `.size_of_set(x)`: return the size of the set containing `x` as an integer. This should be done efficiently in $O(\alpha(n))$ amortized time.
- `.number_of_sets`: an integer attribute containing the current number of disjoint sets (not elements!) contained in the collection.

For methods that require elements to be specified, such as `.find_set`, `.union`, and `.size_of_set`, a `ValueError` should be raised if the element does not exist within the collection.

Starter code is available on the course webpage. Note that you may need to modify some of the existing methods, such as `.make_set`!

Solution:

```
class DisjointSetForest:

    def __init__(self):
        self._parent = []
        self._rank = []
        self._size_of_set = []
        self.number_of_sets = 0

    def make_set(self):
        """Create a new singleton set.

        Returns
        -------
        int
            The id of the element that was just inserted.

        """
        # get the new element's "id"
```

```
        x = len(self._parent)
        self._parent.append(None)
        self._rank.append(0)
        self._size_of_set.append(1)
        self.number_of_sets += 1
        return x

    def find_set(self, x):
        """Find the representative of the element.

        Parameters
        ----------
        x : int
            The element to look for.

        Returns
        -------
        int
            The representative of the set containing x.

        Raises
        ----
        ValueError
            If x is not in the collection.

        """
        try:
            parent = self._parent[x]
        except IndexError:
            raise ValueError(f'{x} is not in the collection.')

        if self._parent[x] is None:
            return x
        else:
            root = self.find_set(self._parent[x])
            self._parent[x] = root
            return root

    def union(self, x, y):
        """Union the sets containing x and y, in-place.

        Parameters
        ----------
        x, y : int
            The elements whose sets should be unioned.

        Raises
        ------
        ValueError
            If x or y are not in the collection.

        """
        x_rep = self.find_set(x)
```

```
        y_rep = self.find_set(y)

        if x_rep == y_rep:
            return

        if self._rank[x_rep] > self._rank[y_rep]:
            self._parent[y_rep] = x_rep
            self._size_of_set[x_rep] += self._size_of_set[y_rep]
        else:
            self._parent[x_rep] = y_rep
            self._size_of_set[y_rep] += self._size_of_set[x_rep]
            if self._rank[x_rep] == self._rank[y_rep]:
                self._rank[y_rep] += 1

        self.number_of_sets -= 1

    def size_of_set(self, x):
        """The size of the set containing x.

        Parameters
        ----------
        x : int
            The element whose set will be sized.

        Returns
        -------
        int
            The size of the set containing x.

        Raises
        ------
        ValueError
            If x is not in the collection.

        """
        return self._size_of_set[self.find_set(x)]
```

Problem 1.

Consider the following model for iteratively generating a random connected graph with n nodes. Start with a graph containing n nodes and no edges. Next, randomly select an edge from the uniform distribution on the set of all possible edges and add it to the graph (if it isn't already in the graph). Repeat this process until the graph is connected[1].

How many edges will the resulting graph have? It is a random number, of course, but it must be at least $n - 1$. Let's visualize the distribution of this quantity through an empirical experiment, *a la* DSC 10.

Write a function to simulate the above procedure and return the number of edges in the resulting graph, assuming $n = 100$. Run your function 10,000 times and plot a histogram of the results. Include your figure, as well as your code. Your code should be fast enough so that all 10,000 trials take less than 1 minute or so in total.

Hint: you can use DFS/BFS to determine whether the graph is connected, but if you do your code will take

[1] This random graph model is related to (but not the same as) the famous Erdös-Renyi random graph model. While simple, the E-R model has been the subject of a lot of theoretical analysis.

forever to run... Is there a better way of keeping track of connected components?

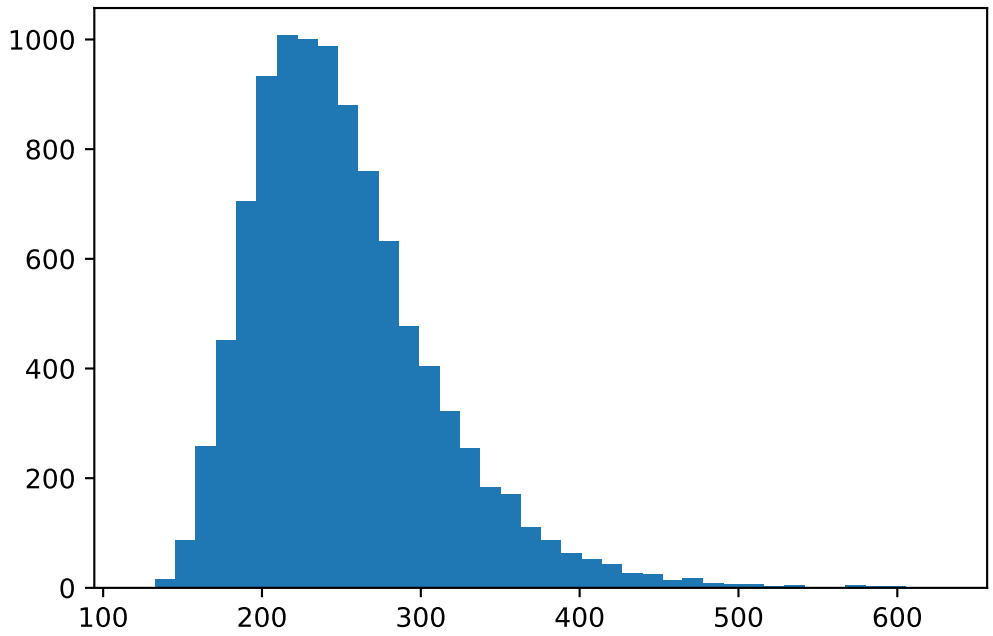
Solution: We'll use a disjoint set forest to keep track of the connected components. If we tried to use a graph data structure and DFS/BFS, it would take forever to run. Here's some code for running the experiment once:

```
def experiment(n):
    dsf = DisjointSetForest()
    for i in range(n): dsf.make_set()

    count = 0
    seen = np.zeros((n, n))
    while dsf.number_of_sets > 1:
        u = random.choice(range(n))
        v = random.choice(range(n))
        if seen[u, v] or u == v:
            continue
        seen[u, v] = seen[v, u] = 1
        dsf.union(u, v)
        count += 1

    return count
```

When we run this 10,000 times and plot a histogram, we see the following:



It looks like this might be a log-normal distribution...

Problem 2.

Determine whether each of the following statements is True or False. If the statement is False, provide a counterexample.

a) In the activity selection problem, assuming unique finish times, any optimal solution must contain the event with the earliest finish time.

Solution: This is false. Suppose the events are $[0, 2], [0, 1]$, and $[3, 4]$. Then both $\{[0, 1], [3, 4]\}$ and $\{[0, 2], [3, 4]\}$ are optimal. Namely, the second solution is optimal but does not contain the event with earliest finish time.

b) In the minimal spanning tree problem, assuming unique edge weights, any optimal solution must contain the graph edge whose weight is the smallest.

Solution: This is True.

You didn't need to provide an argument, but here's one in case you were curious. Suppose (u, v) is the lightest edge in a graph, and let T^* be the MST. Suppose (u, v) is not in T^*. Let e be any edge in T^* along the path from u to v. Removing this edge disconnected T^* into two connected components, one containing u and the other v. Adding the edge (u, v) re-connects the tree, and since (u, v) is the lightest edge, doing so decreases the total edge weight of T^*. Therefore T^* wasn't optimal (this is a proof by contradiction).