Problem Set 4

Collaborated with: Steven Brotz, David Kawashima, Jessica Choi.
Used one (1) late token.

Problem 1

a. Calculating the stationary distribution using a system of equations and Wolfram, we get:

$$\pi = (\frac{7}{22}, \frac{7}{22}, \frac{4}{11})$$

If we take P^n as $n \to \infty$, we see that it approaches:

$$\begin{bmatrix} \frac{7}{22} & \frac{7}{22} & \frac{4}{11} \\ \frac{7}{22} & \frac{7}{22} & \frac{4}{11} \\ \frac{7}{22} & \frac{7}{22} & \frac{4}{11} \end{bmatrix}$$

Thus, because each row in P^n for large n is the same, and we know that $\sum \pi(i) = 1$, and thus $\sum \pi_0(i) = 1$, for any starting π_0, we have that:

$$\lim_{n\to\infty} \pi_0 P^n = \pi = (\frac{7}{22}, \frac{7}{22}, \frac{4}{11})$$

Thus, π does not depend on π_0, meaning that our stationary distribution does not depend on our starting distribution at all. In other words, the stationary distribution represents the steady state/ equilibrium.

b. Calculating the stationary distribution using a system of equations and Wolfram, we get:

$$\pi = (\frac{7}{16}, \frac{1}{8}, \frac{1}{16}, \frac{3}{8})$$

If we take P^n as $n \to \infty$, we see that it approaches:

$$\begin{bmatrix} 0 & 0.25 & 0 & 0.75 \\ 0.875 & 0 & 0.125 & 0 \\ 0 & 0.25 & 0 & 0.75 \\ 0.875 & 0 & 0.125 & 0 \end{bmatrix}$$

From this, we can see that because not all of the rows are equivalent, that $\lim_{n\to\infty} \pi_0 P^n$ does not converge.

Problem 2

a. We write $\tilde{r}$ as follows:

$$\tilde{r} = rG \qquad G = \alpha P + (1-\alpha)\begin{bmatrix} \frac{1}{n+1} & \frac{1}{n+1} & \cdots \\ \frac{1}{n+1} & \ddots & \\ \vdots & \cdots & \frac{1}{n+1} \end{bmatrix}$$

Expanding this out, we have:

$$(\tilde{r}_1, \tilde{r}_2, \dots, \tilde{r}_n, \tilde{r}_{n+1}) = (r_1, \dots, r_{n+1})\left(\begin{bmatrix} \alpha P_{1,1} & \alpha P_{1,2} & \cdots & \alpha P_{1,n+1} \\ \vdots & \ddots & & \vdots \\ \alpha P_{n+1,1} & \cdots & \cdots & \alpha P_{n+1,n+1} \end{bmatrix} + \begin{bmatrix} \frac{1-\alpha}{n+1} & \cdots & \frac{1-\alpha}{n+1} \\ \vdots & \ddots & \\ \frac{1-\alpha}{n+1} & \cdots & \frac{1-\alpha}{n+1} \end{bmatrix}\right)$$

Looking at each page rank:

$$\tilde{r}_1 = r_1\left(\alpha P_{1,1} + \frac{1-\alpha}{n+1}\right) + r_2\left(\alpha P_{2,1} + \frac{1-\alpha}{n+1}\right) + \dots + r_{n+1}\left(\alpha P_{n+1,1} + \frac{1-\alpha}{n+1}\right) = \frac{1-\alpha}{n+1}\sum_{i=1}^{n+1} r_i + \alpha \sum_{i=1}^{n+1} r_i P_{i,1}$$

$$\vdots$$

$$\tilde{r}_{n+1} = r_1\left(\alpha P_{1,n+1} + \frac{1-\alpha}{n+1}\right) + r_2\left(\alpha P_{2,n+1} + \frac{1-\alpha}{n+1}\right) + \dots + r_{n+1}\left(\alpha P_{n+1,n+1} + \frac{1-\alpha}{n+1}\right) = \frac{1-\alpha}{n+1}\sum_{i=1}^{n+1} r_i + \alpha \sum_{i=1}^{n+1} r_i P_{i,n+1}$$

$\tilde{r}_{n+1}$ is precisely the page we add which is X so we can now solve for the page rank of X.

$$X = \tilde{r}_{n+1} = \frac{1-\alpha}{n+1}\sum_{i=1}^{n+1} r_i + \alpha \sum_{i=1}^{n+1} r_i P_{i,n+1} = \frac{1-\alpha}{n+1} + \alpha X$$

because X has no out-degree, so therefore it points to itself.

Now, solving for X:

$$X = \frac{1-\alpha}{n+1} + \alpha X \Rightarrow X - \alpha X = \frac{1-\alpha}{n+1} \Rightarrow X(1-\alpha) = \frac{1-\alpha}{n+1} \Rightarrow \boxed{X = \frac{1}{n+1}}$$

B/C X has no out or in degree, it cannot give or take rank to / from the other nodes. Thus we simply have to rescale and normalize the page ranks of the other n pages. Hence:

$$\boxed{\tilde{r} = r \cdot \frac{n}{n+1}}$$

$$\boxed{X = \frac{1}{n+1}}$$

Scanned by CamScanner

b. We first calculate the Page Rank of Y. Similar to how we calculated X in part a, we get that

$$Y = \frac{1-\alpha}{n+2} \sum_{i=1}^{n+2} r_i + \alpha \sum_{i=1}^{n+2} r_i P_{i,n+2}$$

B/c Y has an out-degree of 1 and nothing else points to Y, we know that $\alpha \sum_{i=1}^{n+2} r_i P_{i,n+2} = 0$ and so:

$$Y = \frac{1-\alpha}{n+2} \sum_{i=1}^{n+2} r_i = \boxed{\frac{1-\alpha}{n+2}}$$

Now, for X, we not only have a link to itself (b/c it has no out-degree), we also get rank from Y.

$$X = \frac{1-\alpha}{n+2} + \alpha X + \alpha Y \Rightarrow (1-\alpha) X = \frac{1-\alpha}{n+2} + \frac{\alpha(1-\alpha)}{n+2}$$ (subst. for Y)

$$\boxed{X = \frac{1+\alpha}{n+2}}$$

This shows that the pagerank of X $\boxed{\text{does improve}}$ b/c before the addition of Y, $X = \frac{1}{n+1}$. and for large n:

$$\frac{1+\alpha}{n+2} > \frac{1}{n+1}, \quad n \to \infty.$$

Now, X and Y are still isolated from the rest of the graph of n nodes, so for the original n nodes, we simply rescale/normalize to account for these two new nodes.

$$\boxed{F = r \cdot \frac{n}{n+2}}$$

c. The setup would be as follows:

$$\begin{tikzcd} & X & \\ Y \arrow[ur] & & Z \arrow[ul] \end{tikzcd}$$

We know that b/c these three nodes are isolated from the rest of the graph, the only way for X to gain rank is for other pages to point to it, as we saw in part b. Hence, we should have both Y and Z contribute their rank to X. Likewise to maximize X's rank, we do not want X to share its rank by pointing anywhere, so we keep it so that X has the max. in-degree and min. out-degree (0). Mathematically, b/c X, Y, and Z are isolated from the rest of the n nodes, $X+Y+Z = 3/(n+3)$. Thus, to maximize X, we minimize Y and Z by setting $y = z = \frac{1-d}{n+3}$ and pointing them both to X to have a rank of $\frac{1+2d}{n+3}$.
↑max.

d. No, adding links from page X to other pages will not improve the PageRank of X, but actually decrease it. Increasing the out-degree of X will only mean that X will potentially be giving rank to other pages. Similarly, adding links from Y or Z to older, popular pages will mean that Y and Z will not be able to donate their full ranks to X, but can only donate a portion of it b/c they are connected to additional pages now. Mathematically, based off of part c, we see that $x+y+z \le \frac{3}{(n+3)}$, given that now X, Y, and Z may have links to other pages. To maximize X, we should abide w/ the setup in c.

e. From the previous parts, we see that adding additional pages and making them point to X will boost the rank of X. However, doing such will only boost the page rank of X marginally, especially for larger and larger n. To increase the rank of X more significantly, we should try to get popular pages that already have high rank to point to X. In this case, a portion of the popular pages' ranks will be transferred to X which can boost the rank of X.

Problem 3

- Degree $\not\propto$ closeness

We present the following counterexample:

From degree centrality, both B and C have the same order of difference (both have $C_D(B) = C_D(C) = 2/4 = 1/2$. However, if we look at closeness, we have that

$$C_C(B) = \frac{4}{1+1+2+3} = \frac{4}{7} \quad \text{but} \quad C_C(C) = \frac{4}{2+1+1+2} = \frac{4}{6} = 2/3.$$

- Degree $\not\propto$ Betweenness. We use the same counterexample as above. Both B and C have the same degree centralities. Now, we calculate their betweenness centralities.

$$C_B(B) = \frac{1+1+1}{6} = \frac{1}{2} \quad \text{but} \quad C_B(C) = \frac{1+1+1+1}{6} = \frac{2}{3}.$$

- Degree $\propto$ PageRank

To show proportionality, we introduce a scalar constant $c = \frac{1}{2|E|}$ such that $r_i = \frac{1}{2|E|} d_i$.

Thus, we get that

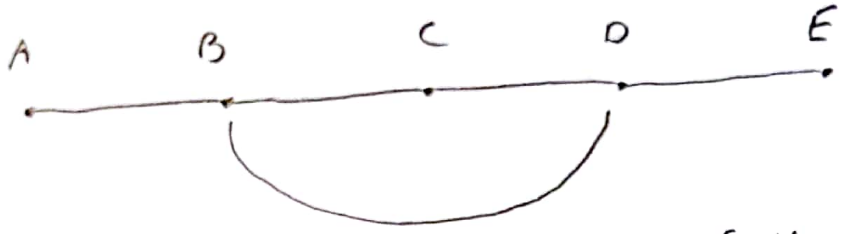
(1) $r_i \geq 0$ for all i

(2) $\sum_{i=1}^{n} r_i = \frac{1}{2|E|} \sum_{i=1}^{n} d_i = 1$ (total degree = 2 · number of edges)

(3) $\sum_{j \in N(i)} \frac{r_j}{d_j} = \sum_{j \in N(i)} \frac{1}{2|E|} = \frac{1}{2|E|} d_i = r_i$

Thus we have shown that degree is proportional to PageRank

- Closeness $\not\propto$ Betweenness. We use the following graph as a counterexample and focus on nodes B and C.



Let's calculate closeness centrality first.

$$C_c(B) = \frac{4}{1+1+1+2} = \frac{4}{7}, \quad C_c(C) = \frac{4}{2+1+1+2} = \frac{2}{3}$$

Now, let's calculate betweenness centrality.

$$C_B(B) = \frac{1+1+1+1}{6} = \frac{2}{3}, \quad C_B(C) = \frac{0}{6} = 0.$$

From above, based on closeness, we have that C ranks higher than B, but based on betweenness, C ranks lower than B.

- Closeness $\not\propto$ PageRank: B/C degree $\propto$ PageRank and degree $\not\propto$ closeness, PageRank $\not\propto$ closeness.

- Betweenness $\not\propto$ PageRank: B/C degree $\propto$ PageRank and degree $\not\propto$ betweenness, PageRank $\not\propto$ betweenness.

b. Degree centrality : Should use this when you care about direct connections. If you want to see who is popular on a social network where edges denote friendships, then you could use this.

closeness centrality : Useful for identifying "broadcasters" or people that can quickly influence the entire network. This might be useful for placing post offices b/c post offices have to be able to route information throughout the entire network efficiently.

Betweenness centrality : Useful for finding individuals who can influence the flow around a system. Potentially useful for identifying potential traffic bottlenecks on certain roads / intersections that connect other routes together.

PageRank : This identifies nodes whose influence goes beyond their direct connections into the larger network. In addition to the application of search engines, PageRank can also be used to identify very important publications / authors of research literature where an edge means a citation.

Problem 4

a.

```
# CommonFriends

function main(){
        # Define the input
        listOfFriendships = list of (Person, [List of Friends])

        # Run MapReduce on the list listOfFriendship
        listOfOutputs = MapReduce(listOfFriendship)

        # Output result
        Print contents of listOfOutputs on console.
}

function MapReduce(listOfFriendships){

        # Map step:
        For each(key, value) pair in listOfFriendship, run Map(key, value)

        # Collector step:
        Collect all outputs from Map() in the form of (key, value)
                pairs in a new list called listOfMapOutputs
        Make a list of distinct keys in listOfMapOutputs
        For each unique key, concatenate the corresponding value lists

        For each such (key, listOfValues), run Reduce() on them.
        Concatenate the results from all calls to Reduce() into a
                list and return it.

}

function Map(person, listOfFriends){
        # key = person, values = listOfFriends for that person

        For each friend in listOfFriends{
                # Sort so that pairs of people get emitted in
                # lexicographical order
                EmitInteremediate(sorted(person, friend),
                        lstOfFriends)
                }
        }
}
```

```
function Reduce(pair, listValues){

        # For input ((A,B), [lst1, lst2]),
        # To get mutual friends, we simply
        # take the intersection of lst1
        # and lst2 to get mutual friends of
        # A and B

        mutual_friends = []
        lst1 = listValues[0]
        lst2 = listValues[1]
        for friend in lst1{
                if friend in lst2 {
                        add friend to mutual_friends
                }
        }
        return (pair, mutual_friends)
}
```

b.

```
# High school days

function main(){
        # Define the input
        listOfFilenames = list of filenames containing test scores
                of all students
        listKeyValues[] = list of pairs (filename, fileScores)
                where fileScores is a string containing the scores,
                separated by spaces, of the file with name 'filename'


        # Run MapReduce on the list listOfFilenames
        listOfOutputs = MapReduce(listKeyValues)

        # Output result
        Print contents of listOfOutputs on console.
}

function MapReduce(listKeyValues){

        # Map step:
        For each(key, value) pair in listKeyValues, run Map(key, value)

        # Collector step:
```

```
        Collect all outputs from Map() in the form of (key, value)
        pairs in a new list called listOfMapOutputs.
        Make a list of distinct keys in listOfMapOutputs.
        For each unique key, create a list of the values
        to create (key, listOfValues) pairs.

        # Reduce step:
        For each such (key, listOfValues), run Reduce() on them.

        # Return the results
        Concatenate the results of all calls to Reduce() into a list.
        Sort this list in reverse (decreasing) order.
        Output this list.
}

function Map(filename, fileScores){
        # key = filename, values = scores in that filename

        # Assuming scores are separated by spaces
        score[] = Split fileScores by " ".

        for each score in scores {
                EmitIntermediate(score, 1)
        }
}

function Reduce(score, listValues){
        # key = word and value = list of 1's
        # no. of ones for each score is the
        # no. of occurrences of the word
        # in all files.
        return ([score * len(listValues)])

        # Return: list consisting of that score repeated
        # the number of times as the no. of occurrences
        # in all the files
}
```

c. The logic is as follows. Notice that if we take the unit circle centered at (0,0), it has an area equal to π. If we look at the square with corners at (-1,-1), (-1, 1), (1, -1), and (1, 1), such a square has side length = 2, so this square will have an area of 4. If we generate two numbers from our random generator (which generates numbers from [-1, +1]) and treat those two numbers as (x, y) coordinates, we are guaranteed that this point will lie in the square described above. Now to calculate π, we look at the proportion of points that not only lie in the square, but also lie in the circle. Then, if we divide the number of points that lie in the circle by the total number of points generated, it should come out to around $\frac{\pi}{4}$. Thus, if we multiply this proportion of points that lie in the circle by 4, we should be able to approximate π.

```
# Good old pi

function main(){
        # Define the input
        # Generate n (in this case we just say 1000)pairs of points
        let n = 1000
        lstOfPoints = []
        for i from 1 to n{
                let x = rand()
                let y = rand()
                Add (x, y) to lstOfPoints
        }

        # Run MapReduce on this list of points
        listOfOutputs = MapReduce(lstOfPoints)

        # Output result
        Print contents of listOfOutputs on console.
}

function MapReduce(lstOfPoints){

        # Map step:
        For each point (x, y) in lstOfPoints, run Map((x,y))

        # Collector step:
        Collect all outputs from Map() in the form of (value)
        and concatenate all values into a new list called
        listOfMapOutputs.

        # Taking the sum of listOfMapOutputs gives us the
        # number of points lying in the circle. Dividing
        # this by n gives us proportion of all points that
        # lie within the circle

        let sum = listOfMapOutputs
        let proportion = sum / n

        # Multiply by 4 because as explained before,
```

```
        # the total square area is 4.

        let pi_approx = proportion * 4

        return pi_approx
}

function Map(point){
        # input = a single (x,y) point where
        # x and y are both between -1 and 1

        # Test if in circle. If in circle, we
        # return 1, else return 0

        If x^2 + y^2 <= 1 {
                EmitIntermediate(1)
        }
        else
        {
                EmitIntermediate(0)
        }

}
```

d.

```
# GaugeTheDistance

function main(){
        G[] = Adjacency list of the graph.
        # G[i] is a list of (neighbors, distance) tuple of node i.
        n = length(G)
        dist[] = list that will contain the distances
        from node 1 eventually. Initialize it arbitrarily.
        distUpdated[] = list that will contain distances from
        node 1 after each run of MapReduce in the
        following loop. Initialize it with $(0, \infty, \infty, \ldots, \inft

        while( NOT stoppingCriterion(dist, distUpdated)){
                dist = distUpdated
                inputs = ((i, dist[i]), G[i]) for all i
                distUpdated = MapReduce(inputs).
        }
        print the list dist[].
}
function stoppingCriterion(dist1, dist2){
```

```
        return dist1 == dist2
}

function Map((i, dist[i]), G[i]){
        # Input is a node and its distance from node 1
        # and its adjacency list. We output a tuple
        # containing (node, distance to i) based on its
        # connection/distance to the input node

        EmitIntermediate(i, dist[i])
        if (dist[i] != infty)
        {
                for each (neighbor, distance) in dist[i]
                {
                        EmitIntermediate(neighbor, distance + dist[i])
                }
        }
}

function Reduce(i, distancesFromi){
        # each node will have a list of distances,
        # take the minimum one
        return (i, min(distancesFromi))
}
```