**Abstract:**

This investigation considers Google's PageRank algorithm and how the Perron-Frobenius theorem applies to the ranking mechanism. It further investigates how weights on the edges affect final rankings of the pages. Specifically, what is the minimum weight that must be add/removed before the first change in ranking occurs? After rewriting the code for the PageRank algorithm in Python, the research shows how adding or subtracting any weight to an edge that is tied in its pre-weight rankings causes an immediate change in ranking.

## I. The Perron-Frobenius Theorem

Linear Algebra is a branch of mathematics dedicated to the study of vectors and linear equations. It is a powerful tool, allowing for the analysis of rotations in space, least squares fittings, determination of a circle passing through three given points and the solving of many other problems across various fields of study. Within linear algebra, there exist special vectors that do not change their direction when multiplied by a matrix A, called eigenvectors. The associated eigenvalue, indicates how the vector x grew or shrunk when being multiplied by the matrix A. An eigenvalue $\lambda$ is called dominant if it has an algebraic multiplicity of 1 (simple) and such that for all other eigenvalues $\mu$ of A, including complex eigenvalues, the magnitude of $\lambda$ is greater than the magnitude of $\mu$ such that $||\mu|| < ||\lambda||$. In the 20$^{th}$ century, Oskar Perron and Georg Frobenius outlined a specific property pertaining to dominant eigenvalues. The final version of their theorem for column stochastic matrices (matrices who's column entries sum to one) is as follows: "Given A, a non-negative and irreducible square matrix. Then there is a positive real eigenvalue $\lambda$ with algebraic multiplicity 1 and such that all other eigenvalues have magnitude less than or equal to $\lambda$. Furthermore the eigenvector (unique up to scaling) corresponding to $\lambda$ is positive." However, because A is stochastic, $\lambda$ must be equal to 1. Additionally, if A is aperiodic, meaning that the greatest common divisor of all the return times (the expected number of steps in a walk before returning to the initial vertex) is 1, and the matrix is irreducible, then, the magnitude of all eigenvalues is less than the magnitude of the dominant eigenvalue, $\lambda$ (1 for a stochastic matrix). Hence, for a stochastic matrix, the eigenvalues are as follows: $1 \geq |\lambda_1| \geq |\lambda_2| \geq \ldots \geq |\lambda_{n-1}|$.

## II. The Power Method

When combined with the power method, this theorem has powerful implications. Eigenvalues and vectors have particular properties that allow matrices to be raised to a power with ease. The eigenvectors for a matrix A are unchanged when A is raised to power. In other words, the eigenvectors of A and $A^k$ are the same. If we write A as $A=S\Lambda S^{-1}$ using spectral decomposition, we see that $A^k= ( S\Lambda S^{-1})* ( S\Lambda S^{-1})*\ldots* ( S\Lambda S^{-1})$ where $( S\Lambda S^{-1})$ is multiplied by itself k times. Then, $A= S\Lambda^k S^{-1}$. If we let $A=P\Lambda P^{-1}$ and $v = c_1v_1+ c_2v_2+\ldots+ c_nv_n$, then $A^kv = (P\Lambda^k P^{-1})v$. Consider $Pe_j=v_j$ where $e_j$ is the column vector of all zeros with a on in the jth row. Then, $P^{-1}v_j$ can be rewritten as $e_j=P^{-1}v_j$. Hence,

$$P\Lambda^k(c_1e_1+ c_2e_2+\ldots+ c_ne_n)\rightarrow\ Pc_1e_1=c_1v_1\lambda^k$$

Considering the Perron-Frobineus Theorem, if A is aperiodic, all of the terms other than the first collapse to zero as k approaches infinity. Then since Thus, if A is stochastic then $\lambda=1$, so $A^kv \rightarrow c_1v_1$. Thus, using the power method, the stabilized eigenvector for the dominant eigenvalue 1 can be estimated quickly and efficiently.

## III. PageRank

PageRank is a methodology used by Google to determine the importance or relevance of a page by estimating the likelihood of landing on a page on a random walk. It can be described simply as a voting system, where a link to a page counts as a vote in support of importance. Each page has some value which also determines the value of its votes. The total value of votes from a page are spread evenly among all outgoing links. Then to compute the importance or rank, the fractions of votes are summed and then dampened to prevent any other pages from having too much influence. However, the initial value of a page that is spread out among its votes is dependent upon the links leading to that page seemingly creating an issue with any circular network. These relationships can be represented as a stochastic matrix A, computed by normalizing a matrix with entries corresponding to edges of the directed digraph of the network. A then represents the system of linear equations that determine each ranking. However, the issue surrounding a circular network is not yet resolved. Additionally, if the network is disconnected, the ranking system would fail. In order to solve these problems,

consider a column stochastic nxn matrix III where the entries are 1/n. Then , let Q=(1-α)A + αIII, where 0≤ α ≤1, creating a weighted average of the matrices. For α in (0,1], Q is column stochastic because it is a convex combination of two column stochastic matrices. Here α represents the probability that one "jumps" from one page to another without following a link, hence connecting the previously separate subgraphs. This "jumping" parameter ensures that Q is connected and aperiodic and hence the power method is applicable. With this representation, the problem can be solved by finding an eigenvector corresponding to the eigenvalue 1 using the power method and the Perron-Frobinius method discussed above.

**IV. Investigative Question**

This investigation considers the PageRank methodology employed by Google and the relationship between weighted edges and final rankings. Specifically, the paper considers the range of weights on an edge that maintain initial unweighted rankings. In other words, how big (or small) must weight on an edge be to alter the final rankings? To answer this question, both the PageRank algorithm and weighting steps needed to be developed.

**V. Description of the Code**

A. *Implementing PageRank Algorithm*

The code pertaining to this question first read in an adjacency matrix, and then normalized said matrix by dividing each term by the sum of its column. Then, in order to create the matrix Q as discussed above, the III matrix was created by inserting n columns of n entries, where n is the length of the adjacency matrix and each entry in 1/n. Thus III is an n by n matrix filled with the reciprocal of n. From this III matrix, the matrix Q was created by multiplying the III matrix by the jumping parameter alpha and multiplying the normalized adjacency matrix A by one minus alpha and then summing the two matrices in a convex combination to create the stochastic matrix Q. Then, to use the power method, Q is first multiplied by an n by one vector of 1/n, referred to here as v1. Then, if the difference between this new vector (the result of multiplying Q by the vector of 1/n), referred to here at v2, is less than 0.1% of the original vector, the code enters a loop. Within the loop, v1 is saved as v2 and the Q is multiplied by the new v1 on the right to find the new value of v2. This loop continues while the difference in

v1-v2 is greater than 0.1% of v1. Once the loop is broken, the resulting v2 is the final ranking vector of the initial adjacency matrix.

B. *Creating an Upper Bound*

To test the range of weights that maintain this initial ranking, the code first reads in the edge to be ranked and creates a vector ranking each node in the original rank (the one created above without weights). It creates this vector using the numpy argsort command with type "mergesort." Argsort returns the vector where the first entry is the least important node and the last is the most important. Mergesort ensures that given a tie, the original order is preserved so that the code does not observe false changes in rank due to reordering of ties. The initial rank is saved as the previous rank and enters a while loop. The loop first computes the weight to be added, incrementing the weight by a specified step (here 0.1) with each iteration. This weight is then added to the value in the matrix representing the edge to be weighted. If this value is less then 10, the adjacency matrix is renormalized, a new Q computed and a new final ranking vector reached. Then, using the argsort command, a vector of ranks is produced. This loop continues as long as the rank vector remains unchanged. The restriction on the value being less then 10 however is put in place to avoid adding weights indefinitely without any changes in rank. If the value is greater than or equal to 10, the function prints a message stating that there were too many steps and returns the weight from the last step and exits the method and thus the loop.

C. *Creating a Lower Bound*

This described function computes the upper bound on a weight that maintain the ranking. The process for determining the lower bound is nearly identical. Instead of incrementing the weight by the step with each iteration, the step is subtracted from the weight each time and then added to the matrix entry. Then, instead of comparing the entry to 10 to prevent infinite steps, the entry is compared to zero, where the method returns the weight and breaks the loop if the value is less than or equal to zero to prevent non positive entries. However, if the entry is greater than zero, it normalizes the adjacency matrix, computes Q, and computes a final ranking vector just as described above. This method returns the lower bound

for the range of weights that maintain the initial ranking. The code outputs this range in addition to the initial vector of importance from the unweighted matrix, the changes in rank from both the lower and upper bound and the changes in the importance vector for both the lower and upper bound.

**VI. Example and Future Work**

The attached code reads in an example matrix for the network depicted in the appendix. In this example, the home page has the highest rank, followed by the product page. When weighting one of the edges going to any of the pages of equal importance, such as the group of reviews or group of external sites, there is an immediate change in rank for any weight since they were all equal to begin with. This process could then be used to break ties among webpages be creating a rule around weighting edges (for example, weighting edges linking to edu pages more). Future work might consider the weights on a broader range of the number line rather than in a narrow interval and how much a page's rank can be pushed up or down using this weights before it reaches a ceiling.

**VII. Works Cited**

Bryan, Kurt, and Tanya Leise. *THE $25,000,000,000∗ EIGENVECTOR THE LINEAR ALGEBRA BEHIND GOOGLE*. www.rose-hulman.edu/~bryan/googleFinalVersionFixed.pdf.

Katznelson, Yitzhak, and Yonatan R. Katznelson. *A (Terse) Introduction to Linear Algebra*. American Mathematical Society, 2008.

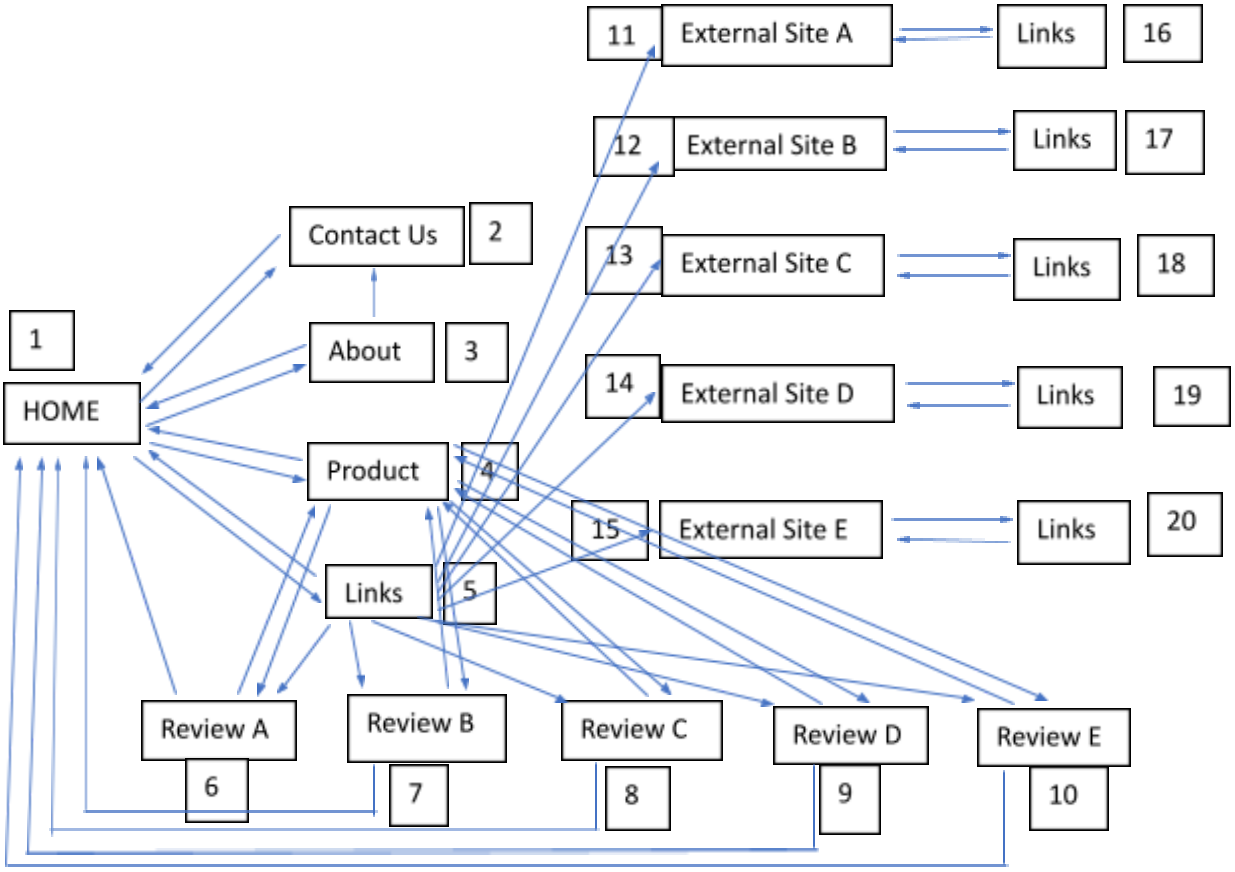
“Lecture 17: Perron-Frobenius Theory.” *Stanford*, 2008, stanford.edu/class/ee363/lectures/pf.pdf.

“Networks.” *More than Just a Web Search Algorithm: Google's PageRank in Non-Internet Contexts : Networks Course Blog for INFO 2040/CS 2850/Econ 2040/SOC 2090*, 2014, blogs.cornell.edu/info2040/2014/11/03/more-than-just-a-web-search-algorithm-googles-pagerank-in-non-internet-contexts/.

Rogers, Ian. “Pagerank Explained Correctly with Examples.” *Princeton University*, The Trustees of Princeton University, www.cs.princeton.edu/~chazelle/courses/BIB/pagerank.htm.

**VIII. Appendix**

*Sample Network*:



*Copy of Code*:

```
f = open ( 'exampleMatrix1.txt', 'r')
lines = f.readlines()
l = []
for line in lines:
    if len(line) == 1:
        break
    l.append([ float(num) for num in line.split(',') ])
import numpy as np
np.set_printoptions(precision=20)
a1=np.asarray(l)
n=int(len(a1))
alpha = float(input("Enter the parameter"))
def normalize_matrix(a):
    b = a/a.sum(axis=0,keepdims=1)
    b.sum(0) # Verify
    return b
b1 = normalize_matrix(a1)
def cast_to_matrix (b):
    m = np.asmatrix(b)
    return m
m1 = cast_to_matrix(b1)
def reciprocal (i):
    return 1.0 / i
def create_III (b):
    n = len(b)
    v = np.array([(reciprocal(n))])
    i = 1
    while i < n:
```

```
        v = np.insert(v, 0, (reciprocal(n)))
        i = i+1
    j = 1
    III = v
    while j<n:
        III = np.vstack((III, v))
        j = j+1
    return III
def create_Q (alpha, III, P):
    term1 = alpha*III
    term2 = (1.0-alpha)*P
    Q = term1 + term2
    return Q
def multiply_vector (b):
    n = len(b)
    v = np.array([(reciprocal(n))])
    i = 1
    while i < n:
        v = np.insert(v, 0, (reciprocal(n)))
        i = i+1
    v0 = v.reshape(-1, 1)
    v1=cast_to_matrix(v0)
    v2 = np.matmul(b,v1)
    array = np.absolute(v1-v2)
    pct = 0.001
    array2 = pct*v1
    test=np.greater(array, array2)
    i=1
```

```
    while any(x==True for x in test):
        v1 = v2
        v2 = np.matmul(b,v1)
        array = np.absolute(v1-v2)
        array2 = pct*v1
        test=np.greater(array, array2)
        i= i+1
    v3=np.asarray(v2)
    return v3
def change_rank_upper (vFinal, a1, alpha, III):
    col=int(input("Enter the column"))-1
    row = int(input("Enter the row"))-1
    vFinal=np.around(vFinal, decimals=8)
    rank = np.argsort(vFinal,kind='mergesort',axis=0)
    weight = 0
    step = 0.01
    a2=a1.copy()
    prevRank = rank
    while (np.array_equal(prevRank, rank)):
        weight = weight + step
        a2[row][col]=a1[row][col]+weight
        if (a2[row][col]<10):
            b1 = normalize_matrix(a2)
            Q = create_Q(alpha, III, b1)
            vFinal2 = multiply_vector(Q)
            vFinal2=np.around(vFinal2, decimals=8)
            prevRank= rank
            rank = np.argsort(vFinal2,kind='mergesort',axis=0)
        else:
```

```
            print("Too Many Steps")
            print(prevRank.reshape(1,-1))
            print(rank.reshape(1,-1))
            print(vFinal.reshape(1,-1))
            print(vFinal2.reshape(1,-1))
            return weight-step
    print("Change in Rank")
    print(prevRank.reshape(1,-1))
    print(rank.reshape(1,-1))
    print(vFinal.reshape(1,-1))
    print(vFinal2.reshape(1,-1))
    return weight-step
def change_rank_lower (vFinal, a1, alpha, III):
    col=int(input("Enter the column"))-1
    row = int(input("Enter the row"))-1
    vFinal=np.around(vFinal, decimals=8)
    vFinal2=vFinal.copy()
    rank = np.argsort(vFinal,kind='mergesort',axis=0)
    weight = 0
    step=0.01
    a2=a1.copy()
    prevRank = rank
    while (np.array_equal(prevRank, rank)):
        weight = weight - step
        a2[row][col]=a1[row][col]+weight
        if (a2[row][col]>0):
            b1 = normalize_matrix(a2)
            Q = create_Q(alpha, III, b1)
            vFinal2 = multiply_vector(Q)
```

```
            vFinal2=np.around(vFinal2, decimals=8)
            prevRank= rank
            rank = np.argsort(vFinal2,kind='mergesort',axis=0)
        else:
            print("Value below zero")
            print(prevRank.reshape(1, -1))
            print(rank.reshape(1,-1))
            print(vFinal.reshape(1,-1))
            print(vFinal2.reshape(1,-1))
            return weight+step
    print("Change in rank")
    print(prevRank.reshape(1,-1))
    print(rank.reshape(1,-1))
    print(vFinal.reshape(1,-1))
    print(vFinal2.reshape(1,-1))
    return weight+step
III= create_III(b1)
Q = create_Q(alpha, III, b1)
vFinal = multiply_vector(Q)
print("Final vector for Q: ")
print(vFinal.reshape(1,-1))
weight_upper = change_rank_upper(vFinal, a1, alpha, III)
weight_lower = change_rank_lower(vFinal, a1, alpha, III)
print("Weight range: ")
print(weight_upper)
print(weight_lower)
```

*Output from Example Network*:

```
Enter the parameter0.15
Final vector for Q:
[[ 0.11967533330365255628  0.0469318462295194791   0.03293366142167736266
   0.07766573953849359446  0.03293366142167736266  0.021048811339214369
   0.021048811339214369    0.021048811339214369    0.021048811339214369
   0.021048811339214369    0.05917371304575824026  0.05917371304575824026
   0.0591737130457582472   0.05917371304575826108  0.05917371304575826108
   0.05774942723202283051  0.05774942723202284439  0.05774942723202284439
   0.05774942723202284439  0.05774942723202285133]]
Enter the column5
Enter the row8
Change in Rank
[[ 5  6  7  8  9  2  4  1 15 16 17 18 19 10 11 12 13 14  3  0]]
[[ 5  6  8  9  7  2  4  1 15 16 17 18 19 10 11 12 13 14  3  0]]
[[ 0.11967532999999999643  0.04693184999999999718  0.03293365999999999655
   0.077665739999999997    0.03293365999999999655  0.02104881000000000113
   0.02104881000000000113  0.02104881000000000113  0.02104881000000000113
   0.02104881000000000113  0.05917370999999999742  0.05917370999999999742
   0.05917370999999999742  0.05917370999999999742  0.05917370999999999742
   0.05774942999999999743  0.05774942999999999743  0.05774942999999999743
   0.05774942999999999743  0.057749429999999999743]]
[[ 0.11969180999999999571  0.04693684000000000051  0.03293716000000000005
   0.07768000999999999379  0.03293716000000000005  0.02104879000000000125
   0.02104879000000000125  0.02107422000000000115  0.02104879000000000125
   0.02104879000000000125  0.05916633000000000309  0.05916633000000000309
   0.05916633000000000309  0.05916633000000000309  0.05916633000000000309
   0.05774320000000000147  0.05774320000000000147  0.05774320000000000147
   0.05774320000000000147  0.057743200000000000147]]
Enter the column5
Enter the row8
Change in rank
[[ 5  6  7  8  9  2  4  1 15 16 17 18 19 10 11 12 13 14  3  0]]
[[ 7  5  6  8  9  2  4  1 15 16 17 18 19 10 11 12 13 14  3  0]]
[[ 0.11967532999999999643  0.04693184999999999718  0.03293365999999999655
   0.077665739999999997    0.03293365999999999655  0.02104881000000000113
   0.02104881000000000113  0.02104881000000000113  0.02104881000000000113
   0.02104881000000000113  0.05917370999999999742  0.05917370999999999742
   0.05917370999999999742  0.05917370999999999742  0.05917370999999999742
   0.05774942999999999743  0.05774942999999999743  0.05774942999999999743
   0.05774942999999999743  0.057749429999999999743]]
[[ 0.11965882999999999381  0.04692684999999999912  0.03293014999999999831
   0.07765144999999999686  0.03293014999999999831  0.02104883000000000101
   0.02104883000000000101  0.02102336000000000135  0.02104883000000000101
   0.02104883000000000101  0.05918111000000000205  0.05918111000000000205
   0.05918111000000000205  0.05918111000000000205  0.05918111000000000205
   0.05775567000000000201  0.05775567000000000201  0.05775567000000000201
   0.05775567000000000201  0.057755670000000000201]]
Weight range:
0.0
0.0
```