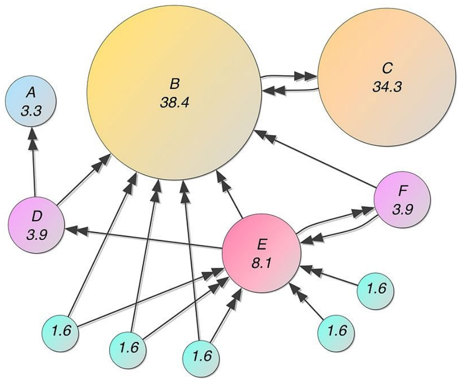


Lecture 17:
PageRank

CS 111: Intro to Computational Science
Spring 2023

Ziad Matni, Ph.D.
Dept. of Computer Science, UCSB

Administrative

- New homework (Assignment 9) out later today
  - PageRank --- might be complex, so start early!

- Grades for Quiz 6 are on Canvas

- Preliminary slides available for today's lecture
- Demonstration will be recorded

Final Exam

- **Wednesday, June 14th from 9:00 AM – 11:00 AM**
  - ***If you are late to the exam, I will not let you take it.***
  - **Arrive 10 minutes early** as I will assign seating for each of you
- DSP students: register now!
- The exam is ***comprehensive***
  - Study: all lectures, all demo codes, all assignments
- I have Practice Questions 4U! Available on Canvas! See the “Final Exam” part in “Modules”
- The ULAs will do a Review Session on **Friday, June 9th** from **10:00 AM to around 12:00 PM**
  - The location will be announced on Piazza later

Final Exam – What to Bring With?

- Your UCSB IDs (MUST have this or I may not let you take the exam)
  - pen/pencils (I prefer pencil, but if you're going to use pen, be neat)
  - eraser(s) (& sharpener??)
  - You are allowed a SIMPLE, NON-GRAPHING calculator (*optional*)

- 1 *OPTIONAL* sheet of paper for notes
  - HAS TO BE 8.5"x11" – nothing else will be allowed
  - Both sides ok, can fill it out in any way you like (print it, write it in ink, pencil, crayon…)
  - **You MUST hand that in to me after the exam** (put your name on it too!)

- DO NOT BRING:
  - Any computers, any phones/smart watches, any other written/printed material

MORE INFO! MOOOOOooooooOOORE!

- Ok – let’s go to Canvas...

PageRank

- Algorithm famously used by Google Search to ***rank*** web pages in their search engine results

Simplified Form:

$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{L(v)}$$

Consider a webpage **u**

Consider the set of all webpages *linking to* **u** (**B**$_u$)

- PR(**u**) = PageRank value for a page **u**
- PR(**v**) = same for **v**, where **v** is contained in the set **B**$_u$
- L(**v**) = number *links from* page **v**

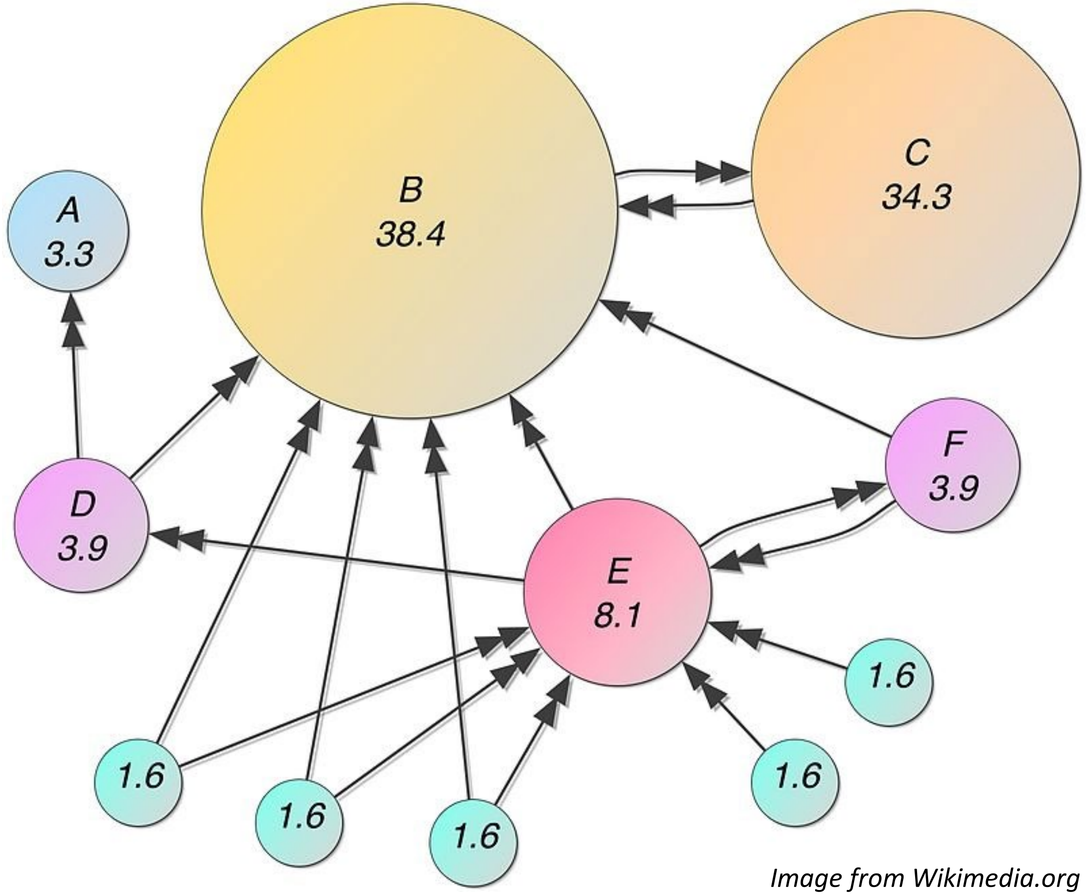


Image from Wikimedia.org

- Looks at the **network** of documents on the Web, **analyzes** how they're connected, and **ranks them** in importance to one another

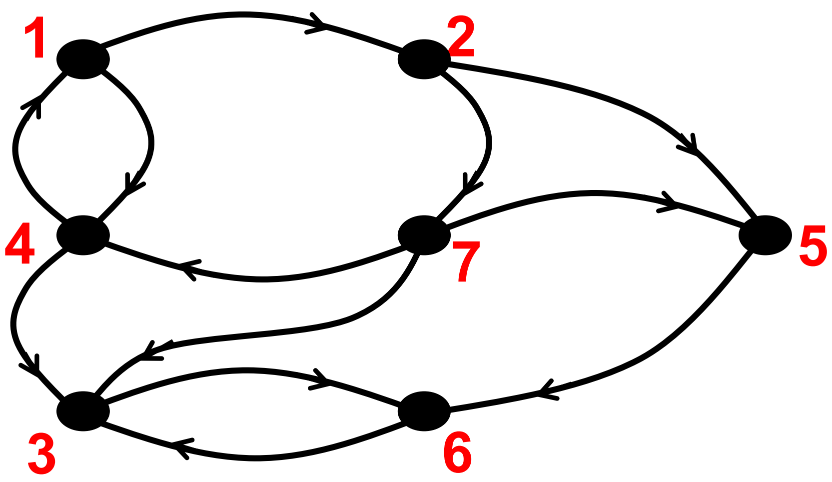
Google and the Random Surfer

How does Google figure out which web pages are most important?

- **An important page is one that lots of important pages "point to".**
- Start at any web page and follow links at random. Forever.
- You'll eventually see "important" pages more often than you do unimportant ones!
- If you hit a "dangling node", that is, one where there is *no* outgoing link, you will "absorb the random surfer" and set the PageRank of other pages to 0.
    - To avoid this, we'll add "virtual edges" directed from dangling nodes to *all* the other nodes.
    - Allows us to continue the 'hunt' for "important pages"

Analyzing the Web with Graphs and Matrices

Graph



Matrix

Rows: indegree

Columns: outdegree

	1	2	3	4	5	6	7
1				1			
2	1						
3				1		1	1
4	1						1
5		1					1
6			1		1		
7		1					

- Graph nodes are web pages
- Arrows between nodes are links between web pages
- Matrix entries are links from "column" pages to "row" pages
- PageRank comes from doing linear algebra on the matrix
- Google matrix has 130×10^{12}+ rows/columns *(that was back in 2016, so it's grown* ***A LOT*** *since then)*

The Random Surfer Model:
Defining the Adjacency Matrix E

Let:

- **W** = set of web pages; **n** = # of web pages in **W**;
  E = the nxn *adjacency* matrix for **W**

  *(FYI, Matrix E is what we were calling Matrix **A** before…)*

- For a large **W**,
  n can typically be in the billions (10^9) and in the trillions (10^{12})

- **E** will be **huge and sparse** with $\mathbf{e_{ij}} = \begin{cases} \mathbf{1} \text{ if } \boldsymbol{i} \text{ is linked to } \boldsymbol{j}, \\ \mathbf{0} \text{ otherwise} \end{cases}$

The Random Surfer Model:
*Defining the Link Matrix **LM***

E e_{ij}

```
[[0. 0. 1. 1.]
 [1. 0. 0. 0.]
 [1. 1. 0. 1.]
 [1. 1. 0. 0.]]
```

- **E** = the *n*x*n* adjacency matrix for **W**

- So, as explained before:

$$\mathbf{r_i} = \Sigma_j\, e_{ij} \quad \text{(in-degree of } j^{th} \text{ page)}$$

and

$$\mathbf{c_j} = \Sigma_i\, e_{ij} \quad \text{(out-degree of } j^{th} \text{ page)}$$

  - Recall: a node with **outdegree = 0** is called "**dangling**"

LM e_{ij}/c_j

```
[[0.         0.         1.         0.5        ]
 [0.33333333 0.         0.         0.         ]
 [0.33333333 0.5        0.         0.5        ]
 [0.33333333 0.5        0.         0.         ]]
```

- We can now define what we call a "**Link Matrix**", **LM = E / outdegree**
  - It's matrix **E**, but the links are given weight based on number of out-going links
  - Note that the sum of the columns is always = 1 (so, in other words, they are normalized)

The Random Surfer Model: *Defining the Markov / PageRank Matrix* ***M***

- Define: **p** = the probability that the "random walk" follows a link (typ. **p** = 0.85)
- We'll call **m** = **1 – p**, *the prob. that an arbitrary page is chosen* (typ. **m** = 0.15)

- Let's define a matrix **M** with elements: $\mathbf{x_{ij}} = p.(\mathbf{e_{ij}}/\mathbf{c_j}) + \delta$ where: $\delta = m/n$
  - Note 1: $\mathbf{e_{ij}}/\mathbf{c_j}$ are the elements in Link Matrix **LM**… (recall: $\mathbf{c_j}$ is the outdegree)
  - Note 2: δ will be a small number, since *n* is usually a big number
  - Note 3: Matrix **M** will have *most* of its entries ($\mathbf{x_{ij}}$) = δ

Markov / PageRank Matrix **M**

- ***Again***: **M** has elements: $\mathbf{x_{ij}} = p.(\mathbf{e_{ij}}/\mathbf{c_j}) + \delta$ where: $\delta = m/n$

- **M** is called the "**transition probability matrix of the Markov Chain**"
- Sometimes referred to as the "**Markov matrix**" or simply the "**PageRank matrix**"
- It's a ***column stochastic*** **matrix**, meaning:
  - The sum of elements in any column will equal 1
  - The matrix's largest eigenvalue will be 1

The Random Surfer Model: **Example**

- **M** has elements: $x_{ij} = p.(e_{ij}/c_j) + \delta$ where: $\delta = m/n$
- Example of a small matrix:
  - n = 4 pages
  - p = 0.85 ➔ m = 1 – p = 0.15
  - δ = 0.15/4 = 0.0375

- For comparison: Example of a large matrix:
  - n = 3 billion pages (3×10^9)
  - p = 0.85 ➔ m = 1 – p = 0.15
  - $\delta = 0.15/(3\times10^9) = 5 \times 10^{-11}$

E e_{ij}

Adjacency Matrix

```
[[0. 0. 1. 1.]
 [1. 0. 0. 0.]
 [1. 1. 0. 1.]
 [1. 1. 0. 0.]]
```

Link Matrix

LM e_{ij}/c_j

```
[[0.         0.         1.         0.5       ]
 [0.33333333 0.         0.         0.        ]
 [0.33333333 0.5        0.         0.5       ]
 [0.33333333 0.5        0.         0.        ]]
```

M $p.(e_{ij}/c_j) + \delta$

```
[[0.0375     0.0375     0.8875     0.4625    ]
 [0.32083333 0.0375     0.0375     0.0375    ]
 [0.32083333 0.4625     0.0375     0.4625    ]
 [0.32083333 0.4625     0.0375     0.0375    ]]
```

Markov Matrix

Dangling Nodes

- These are nodes that do not have any out-going links
  - i.e. whose out-degree = 0
  - When looking at the adjacency matrix, the column is all zeros…

- These can be problematic computationally
  - Divide-by-zero errors down the road
  - If encountered, modify that node to one where it has links to *everything!*
  - *WHY DO THIS?*
    - When you encounter a "dangling" node, it stops you from moving forward, *sooooo…*

Summary of Types of Matrices We've Used So Far…

- The Adjacency Matrix, **E**
  - Spells out all links in a network
- The Link Matrix, **LM**
  - Shows the links weighted by node out-degree
- The Markov Matrix (or PageRank Matrix), **M**
  - Further weighs the nodes in probabilistic terms

Using the PageRank Matrix in Python

```
def make_M_from_E(E, m = 0.15):
    # Make the PageRank matrix from the adjacency matrix (E) of a graph.
    n = E.shape[0]
    outdegree = np.sum(E, axis=0)       # meaning add each column, return these sums in a vector
    for j in range(n):                  # this is a provision to see if we have “dangling” vertices in E
        if outdegree[j] == 0:           # if vertex is “dangling”, i.e. no outdegree for that column
            E[:, j] = np.ones(n)        # then make that column in E be all 1s
            E[j, j] = 0                 # and also make sure that the diagonal value in that col. is a 0
    LM = E / outdegree                  # create the Link matrix from E
    mS = m * np.ones((n,n)) / n         # create column of normalized 1s, multiplied by the ‘m’ probability
    M = (1 - m) * LM + mS               # the Markov matrix is thus calculated based on the Link matrix
    return M
```

Last Step!

- We want to calculate **PageRank values** for all the nodes in the graph
  - So we can now rank pages!

- Last Step: Find the ***eigenvector*** of matrix **M** that's associated with the largest *eigenvalue* of **M**
  - Note-to-remember1: `npla.eig()` will associate λ_0 with v_0, λ_1 with v_1, etc…
  - Note-to-remember2: Eventual PageRank values always have to be **real** numbers

- Can find this in one of 2 ways:
  - Computationally harder way (ok to do for smaller **M**): use **npla.eig(M)** function
  - Computationally easier way (must do this for larger **M**): use an iterative method (of course!) ☺

Iterative Power Method

- Based on **Mx = λx** , where $\lambda = 1$
- Where **x** – the eigenvector associated with $\lambda = 1$ – ***is the PageRank solution***
  - <u>Condition</u>: **x** is a normalized vector (i.e. has a magnitude of 1)
  - The proof for this uses the *Perron-Frobenius theorem* (which we will **not** go over or cover)

```
x = some_initial_value_vector (usually, chose x = all 1s)
for i in range(Number_of_iterations):
    x = M @ x
    x = x / npla.norm(x)
# will approximate iteratively the value for x
```

Quick! To the Python-mobile!

Your TO DOs!

- New (last) Assignment 09 is out today and due by next week Wednesday

</LECTURE>