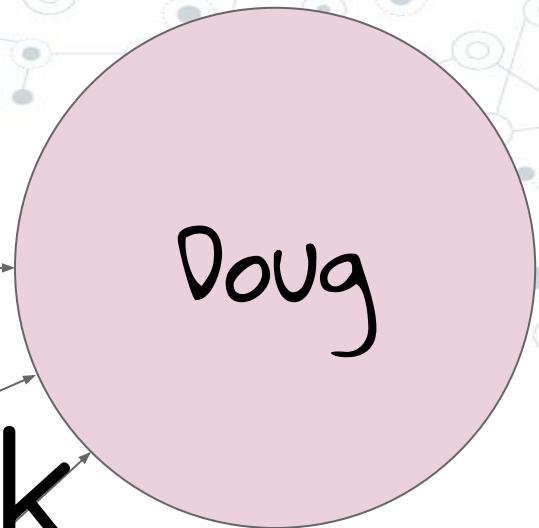
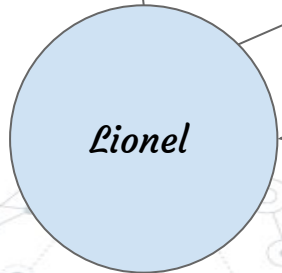


Lucy

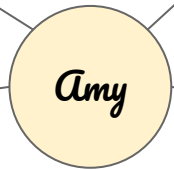


Doug

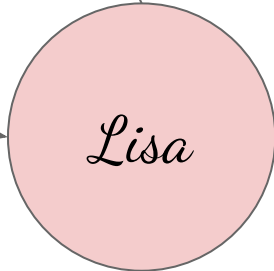
# PageRank



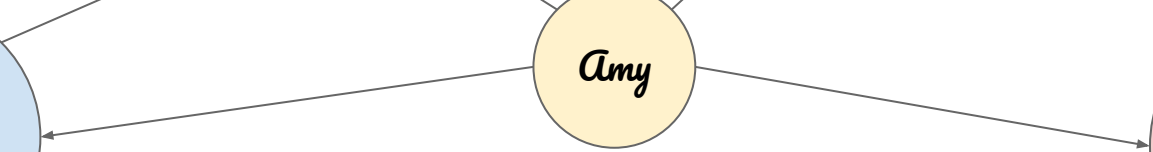
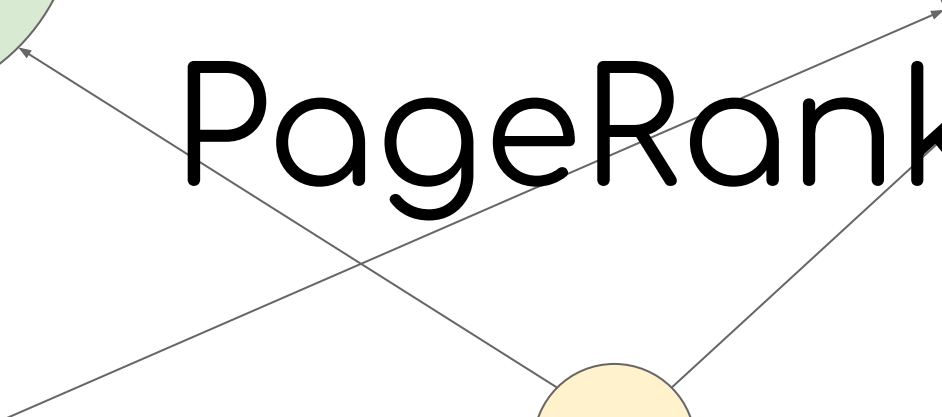
*Lionel*



*Amy*



*Lisa*



# Overview

- Description
- Simplified Algorithm
- Example
- Complete Algorithm - *What you'll be implementing!*
- Sinks
- Damping Factor
- Convergence
- Tips/Roadmap
- Ethical Considerations
- Testing

# Description

- Algorithm developed by L. Page and S. Brin (Google co-founders) used to determine the order of pages returned in response to a query
- **Main idea:**
  - ◆ More important websites = more links from other websites
  - ◆ Ex: Wikipedia oftentimes is first result from Google search - this is because many pages reference Wikipedia

# What is the “pagerank” of a page?

- The pagerank of a page represents its importance
- A page’s rank is a value between 0 and 1
- Each page starts with some amount of rank
- Think of pagerank as a “fluid” that is distributed among pages
  - ◆ the “pagerank” of a page is its total amount of “fluid”
  - ◆ the sum of all page’s pageranks is 1


# Basic PageRank **\*not implementing this\***

$$PR(u) = \sum_{v \in in(u)} \frac{PR(v)}{|out(v)|}$$

- “the PageRank value for a page  $u$  is dependent on the PageRank values for each page  $v$  contained in the set  $in(u)$  (the set of all pages linking to page  $u$ ), divided by the number  $|out(v)|$  of outgoing links from page  $v$ ”

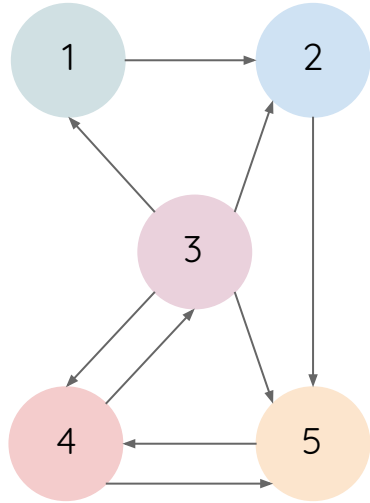
# Steps for Simplified Algorithm

A decorative network diagram in the top right corner, consisting of various sized circles (nodes) connected by thin lines (edges). Some nodes are highlighted with a darker shade or a dashed border.

1. **Iteration 0:** Initialize all ranks to be  $1/(\text{number of total pages})$ .
  2. **Iteration 1:** For each page  $\mathbf{u}$ , update  $\mathbf{u}$ 's rank to be the sum of each incoming page  $\mathbf{v}$ 's rank from the previous iteration, divided by the number total number of links from page  $\mathbf{v}$ .
- 
- A decorative network diagram in the bottom left corner, similar to the one in the top right, showing nodes and connecting lines.

**\*\*using basic algorithm (not implementing this)\*\***

## Example 1



	Iteration 0	Iteration 1
P <sub>1</sub>	1/5	1/20
P <sub>2</sub>	1/5	5/20
P <sub>3</sub>	1/5	1/10
P <sub>4</sub>	1/5	5/20
P <sub>5</sub>	1/5	7/20

1. Iteration 0: Initialize all pages to have rank  $\frac{1}{5}$ .
2. Iteration 1:
3. **P<sub>1</sub>**: has 1 link from P<sub>3</sub>, and P<sub>3</sub> has 4 outbound links, so we take the rank of P<sub>3</sub> from iteration 0 and divide it by 4, which results in rank  $(\frac{1}{5})/4 = 1/20$  for P<sub>1</sub>

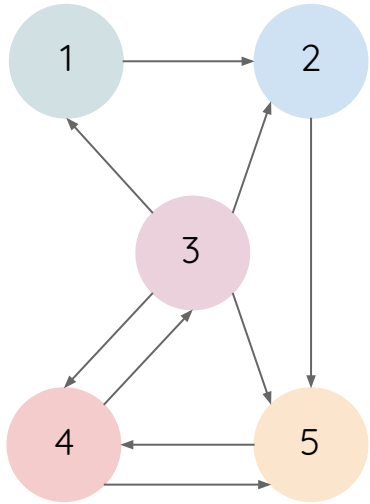
$$\mathbf{PR(P_1) = (\frac{1}{5})/4 = 1/20}$$

4. **P<sub>2</sub>**: has 2 links from P<sub>1</sub> and P<sub>3</sub>, P<sub>1</sub> has 1 outbound link and P<sub>3</sub> has 4 outbound links, so we take (the rank of P<sub>1</sub> from iteration 0 and divide it by 1) and add that to (the rank of P<sub>3</sub> from iteration 0 and divided that by 4) to get  $\frac{1}{5} + 1/20 = 5/20$  for P<sub>2</sub>

$$\mathbf{PR(P_2) = \frac{1}{5} + (\frac{1}{5})/4 = 5/20}$$

\*\* using basic algorithm (not implementing this)\*\*

## Example 1: After 2 iterations



	Iteration 0	Iteration 1	Iteration 2	Final Rank
$P_1$	$1/5$	$1/20$	$1/40$	5
$P_2$	$1/5$	$5/20$	$3/40$	4
$P_3$	$1/5$	$1/10$	$5/40$	3
$P_4$	$1/5$	$5/20$	$15/40$	2
$P_5$	$1/5$	$7/20$	$16/40$	1

### Food for thought

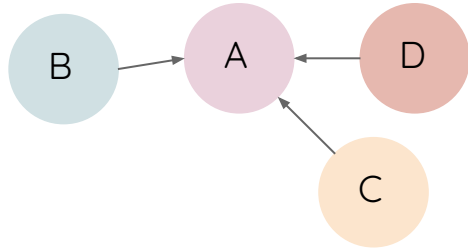
- Why do we calculate the pageranks multiple times?
- When should we stop?  
(to be explained later)

$$PR(P_5) = \frac{1}{5} + \frac{1}{5} * \frac{1}{4} + \frac{1}{5} * \frac{1}{2} = \frac{7}{20}$$



\*\*using basic algorithm (not implementing this)\*\*

## Example 2



- Say we have four pages: **A**, **B**, **C** and **D**.
- Links from a page to itself are ignored (there aren't any in this example).
- Multiple links from one page to another are treated as a single link.
- In this example, every page would start out with a rank of 0.25

\*\* using basic algorithm (not implementing this)\*\*

Example 2: Okay, so I've initialized the ranks. Now what?

Note: this is the same equation as earlier, just with the summation expanded

$$PR(A) = \frac{PR_{prev}(B)}{|out(B)|} + \frac{PR_{prev}(C)}{|out(C)|} + \frac{PR_{prev}(D)}{|out(D)|}$$

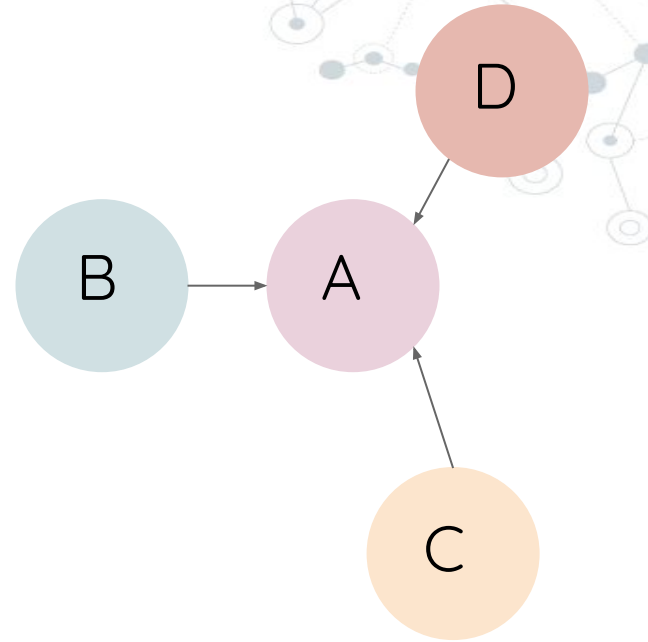
- Apply the PageRank calculation, over and over until the ranks converge.
- $PR(A) = (PR(B) \text{ from previous iteration}) / (\text{number of outbound links from B})$   
+  $(PR(C) \text{ from previous iteration}) / (\text{number of outbound links from C})$   
+  $(PR(D) \text{ from previous iteration}) / (\text{number of outbound links from D})$

\*PR(x) = PageRank of x

\*\* using basic algorithm (not implementing this)\*\*

## Example 2

- Since B, C, and D all have outbound links to A, the Pagerank of A will be **0.75** upon the first iteration
  - ◆ (B with rank of 0.25) + (C with rank of 0.25) + (D with rank of 0.25) would transfer all of those ranks to A
- But wait! What about ranks of pages B,C, and D? Because B, C, and D have no incoming edges and they give all their rank to A, they will all end up with a rank of 0. This doesn't add up to 1 . . .
- ***This is why we are not using the simplified algorithm for our PageRank!***



## Real PageRank Algorithm \*implementing this\*

We know this looks scary, but the core algorithm is the same and we're just adding a few tweaks! Check out the next few slides.

$$PR(p_i) = \frac{1-d}{N} + d \cdot \sum_{p_j \in in(p_i)} \frac{PR(p_j)}{|out(p_j)|}$$

- The PageRank of a page  $P_i$  is equal to  $(1 - \text{damping factor}) / (\text{number of pages}) + \text{damping factor} \cdot (\text{the sum of each page } P_j\text{'s PageRank, where } P_j \text{ links to } P_i, \text{ divided over the number of outbound links of } P_j)$
- $P_1, P_2, \dots, P_n$  are the pages under consideration,
- $in(P_i)$  is the set of pages that link to page  $P_i$
- $|out(P_j)|$  is the number of outbound links on page  $P_j$
- $N$  is total number of pages
- $d$  is damping factor

# Breaking Down the New Update Rule

→ The PageRank of a page  $P_i$  is equal to **(1 - damping factor)/(number of pages) + damping factor \* (the sum of each page  $P_j$ 's PageRank, where  $P_j$  links to  $P_i$ , divided over the number of outbound links of  $P_j$ )**

$$\frac{1 - d}{N}$$

→ The PageRank of a page  $P_i$  is equal to **(1 - damping factor)/(number of pages) + damping factor \* (the sum of each page  $P_j$ 's PageRank, where  $P_j$  links to  $P_i$ , divided over the number of outbound links of  $P_j$ )**

$$d \cdot \sum_{p_j \in \text{in}(p_i)} \frac{PR(p_j)}{|\text{out}(p_j)|}$$

# Damping Factor

- For certain graphs, the simple update rule can cause pagerank to accumulate and get stuck in certain parts of the graphs
- We fix this by having each node (at every round)
  - ◆ Give a  $d$  fraction of its pagerank to its neighbors
  - ◆ Give a  $(1-d)$  fraction of its pagerank to everyone in the graph
- This also means that pages with no incoming links will get some pagerank
- $d$  is called the damping factor
  - ◆ It is usually set to .85

$$PR(A) = \frac{1-d}{N} + d \left( \frac{PR_{prev}(B)}{|out(B)|} + \frac{PR_{prev}(C)}{|out(C)|} + \frac{PR_{prev}(D)}{|out(D)|} + \dots \right)$$

# Sinks

- If a page has no outgoing links to other pages, it is a **sink**
- Sinks can create problems for the PageRank algorithm
  - ◆ The pagerank (i.e., fluid) can accumulate and get stuck at sinks
- In the end, our goal is to distribute the rank of sink pages among all pages in the graph. There are two ways we can do this (**only choose one**):
  - ◆ We can add outgoing edges from sinks to all pages
  - ◆ We can distribute a sink's pagerank evenly among all pages
- Functionally, the two methods should yield identical results



# Handle Sinks - Adding Edges Method



1. After you have determined which pages are sinks, add outgoing edges from each sink to every page in your graph, including the sinks. (This should be done before the main loop and should only be done once).
2. Remember to update the sinks' number of outgoing edges. (This should also be done before the main loop).
3. Within the main body of your loop, after you clone your **\_current** array of pageranks into your **\_previous** array, be sure to reset the entirety of **\_current** to be 0.0.




# Handle Sinks - Distribute Rank Method

Pseudocode for `handleSinks()` - you should call this at the start of every iteration of the algorithm and set that to be each current page's rank before updating it:

1. For each sink, set its updated rank to be the sink's previous rank divided by the number of total pages and sum all of the sinks' updated ranks.
2. For every page in the graph, set the new rank to be the sum of the sinks' updated ranks.


# Convergence



- Is calculating the rank of each page one time enough?
  - We iterate because pageranks can fluctuate. One round is not enough to tell how important a page is.
  - We stop when the pageranks become steady or converge:
    - ◆ each page's pagerank from the previous iteration differs from the current iteration by less than or equal to the margin of error (which we have set to 0.01)
    - ◆ OR once we have iterated 100 times, whichever comes first
- 

# Big Ideas

A decorative network graph in the top right corner, consisting of various sized nodes (some solid, some hollow) connected by thin lines, representing a web structure.


- Calculate pageranks of each page over and over again based on its incoming links!
  - Don't forget damping factor.
  - Take sinks into account.
  - Stop converging either at 100 iterations or when margin of error is at 0.01.
- 
- A decorative network graph in the bottom left corner, similar to the one in the top right, with nodes and connecting lines.

# Tips!

- Use the graph's `vertices()` method to return an iterator of vertices!
- You should have some way to keep track of and store the vertices so that you can map them to their respective pageranks.
- Similarly, you should store the pageranks of each page! But wait... you'll have to update them, but if you update the pagerank of page A, you might need the old pagerank for page B.... think about how to handle this!
- Take care of sinks! Make sure to distribute their ranks accordingly.
- Damping factor set to 0.85 and margin of error set to 0.01. **You should not adjust these parameters.**
- Make sure to stop calculating at 100 iterations or when margin of error hits 0.01 (whichever comes first).

# RoadMap!



- Start out by figuring out what data structures and variables you need to store your ranks and vertices.
    - ◆ You will need to create two arrays to calculate ranks of pages: prev and curr.
    - ◆ Prev will contain the ranks of each page from the previous iteration and take into account pages that are sinks.
    - ◆ What other things will you need to keep track of?
      - Think about outgoing edges!
    - ◆ Make sure to initialize them in `calcPageRank(Graph<V> g)`!
- 

# RoadMap cont.


- Get the number of pages and store your vertices.
  - ◆ Remember that the `AdjacencyMatrixGraph`'s `vertices()` method returns an iterator of vertices - which is in no particular order.
  - ◆ You may want to make sure when storing the ranks of each page that they match up with the indexing in your vertices array.

# RoadMap cont.

- Tackle the main method performing the PageRank calculation - `calcPageRank(Graph<V> g)`!
  - ◆ There is no constructor (reason has to do with visualizer) so you'll want to initialize all your variables here.
  - ◆ Initialize the rank of each page to be  $1/(\text{number of pages})$ .
  - ◆ Enter the repeated rank calculation!
    - Food for thought: would a while loop, for loop, or do-while loop work best?
    - Considering an ending condition of the iterations is looking at the difference between a current and previous iteration, which would fit most?

# Main calculation



- At every iteration, you want to make sure your previous array is updated correctly.
    - ◆ **For distributing rank sink method only:** You want your sinks to be updated for the current array.
  - You want to then update the rank of each page using the previous array.
    - ◆ Maybe have a method to update the rank?
    - ◆ Don't forget the damping factor!
  - Write up a helper method to help check when you should stop iterating!
- 




# Testing

- All PageRanks should add up to 1!
- Pages with more links should have higher pageranks than pages with fewer links.
- Sinks should not have a pagerank of 0.
- Comparing to the visualizer in the demo can be a powerful testing tool!
- First make sure your algorithm works on small, simple graphs and then test it on the large olympics graph!

# Ethical Considerations



- Do we always want to return the ‘true’ pagerank of a website, or would it ever be beneficial to alter it?
  - There is a conceptual question and multiple ethical questions relating to this!
  - The conceptual question asks how you could go about reducing the rank of a ‘bad’ page without censoring (deleting) the vertex or manually setting its rank to 0. Make sure you have a good understanding of the pagerank algorithm before attempting this!
- 

# Citations

- <https://en.wikipedia.org/wiki/PageRank>
- <http://www.cs.princeton.edu/~chazelle/courses/BIB/pagerank.htm>
- <https://www.quora.com/How-is-PageRank-calculated-What-was-the-initial-PageRank-How-does-the-algorithm-begin>