Simplified PageRank Algorithm

In late 90's as the number of webpages on the internet was growing exponentially different search engines were trying different approaches to rank the webpages. At Stanford, two computer science Ph.D. students, Sergey Brin and Larry Page were working on the following questions: *How can we trust information*? *Why are some web pages more important than others*? Their research led to the formation of the Google search engine.

In this project, you are required to implement a simplified version of the original PageRank algorithm on which Google was built by representing the web as a graph and implementing this graph using an Adjacency List or an equivalent data structure. The PageRank algorithm is an algorithm that is used to order or rank different web pages on the internet.

Representing the Web as a Graph

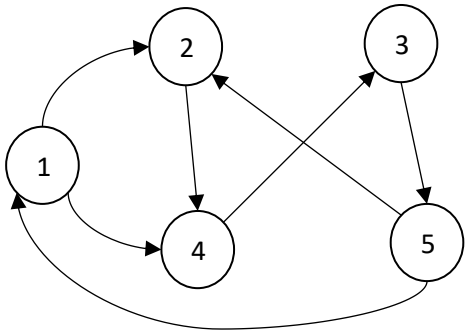
The entire internet consists of different webpages that can be represented as a graph. Each node represents a webpage and each edge represents a link between two webpages. This graph can be implemented as an Adjacency Matrix or an Adjacency List. In this assignment, you are supposed to implement the **Adjacency List representation.** If you use an Adjacency Matrix representation, you will be penalized by 50% of your implementation points and will also fail the large test cases.

Adjacency Matrix Representation

Now for the sake of simplicity, we are explaining the project in the form of an Adjacency Matrix, M. We represent the internet in the form of |V|x|V| matrix where |V| is the total number of vertices in this graph or the total number of webpages in the internet. Each vertex, V_i is a webpage in the entire internet. In the below graph, we have five vertices or webpages. Within our graph, if there is an edge from V_i to V_j (the *from_page* points *to_page*), we have the value in our adjacency matrix M_{ij} = 1 and 0 otherwise.

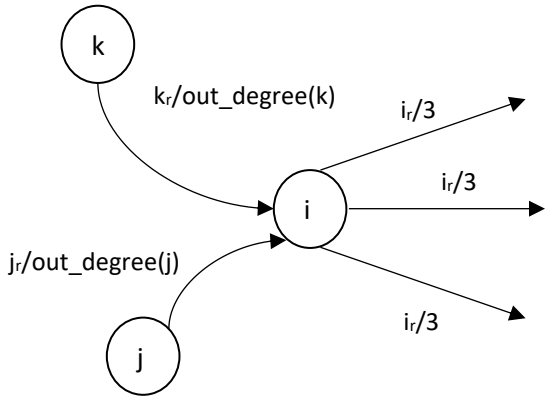
M=

	1	**2**	**3**	**4**	**5**
1	0	1	0	1	0
2	0	0	0	1	0
3	0	0	0	0	1
4	0	0	1	0	0
5	1	1	0	0	0



Each webpage is thus a node in the directed graph and has incoming edges and outgoing edges. Each node has a rank, r. According to PageRank, this rank, r is *equally* split among the node's outgoing links. In the below figure, rank of node i, is denoted by i_r and this rank is equally split among node i's three outgoing edges.

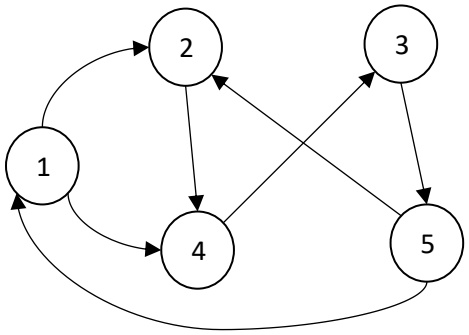
Rank(i) = Rank(j)/out_degree(j) + Rank(k)/out_degree(k)



According to PageRank, this rank, r is equal to the sum of the incoming ranks. In the above figure, rank of node i, $i_r = k_r/\text{out\_degree}(k) + j_r/\text{out\_degree}(j)$; Thus, the rank is based on the indegree (the number of nodes pointing to it) and the importance of an incoming node. This is important considering let's say you create your personal website and have a million links to other pages of importance, then you might artificially inflate your website's ranking if this was not the case. If for calculating the rank, we used out links, we could have easily duped the algorithm. Therefore, the rank is only based on in-links.

M=

	1	2	3	4	5
1	0	1	0	1	0
2	0	0	0	1	0
3	0	0	0	0	1
4	0	0	1	0	0
5	1	1	0	0	0



Core Idea of PageRank

- Important web pages will point to other important webpages.
- Each page will have a score and the results of the search will be based on the page score (called page rank).

Goal

In this assignment, you need to compute the rank of the webpages using a Simplified PageRank Algorithm explained in the example below. **You are supposed to implement an Adjacency List** data structure to represent the graph.

Input
Line 1 contains the number of lines (*n*) that will follow and the number of power iterations (*p*) you need to perform. Each line from 2 to *n+1* will contain two URL's – *from_page to_page* separated by a single space. This means that the *from_page* points to the URL *to_page*.

Output
Print the PageRank of all pages after *p* powerIterations in ascending alphabetical order of webpage. Also, round off the rank of the page to two decimal places.

Constraints

- 1 <= p <= 10,000
- 1 <= n <= 10,000
- 1 <= Unique webpages or |V| <= 10000

Explanation of PageRank through an Adjacency Matrix Example

Stepik Test Case 1 Explanation (*p*=2):

Input

7 2
google.com gmail.com
google.com maps.com
facebook.com ufl.edu
ufl.edu google.com
ufl.edu gmail.com
maps.com facebook.com
gmail.com maps.com

Output

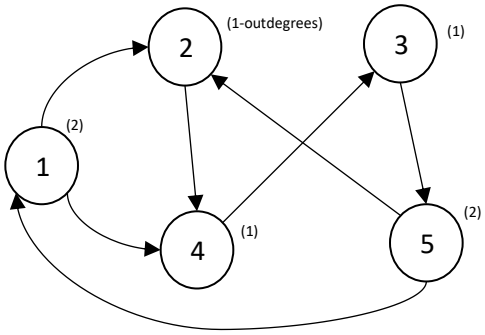
facebook.com 0.20
gmail.com 0.20
google.com 0.10
maps.com 0.30
ufl.edu 0.20

Note: Here, p = 2, n = 7, |V| = 5

Graph Representation of Above Input

Data Structure 1

1	google.com
2	gmail.com
3	facebook.com
4	maps.com
5	ufl.edu



Algorithm

Step 1: Mapping for Simplicity (Optional but you will need a mechanism to store unique vertices**)**

Use a map/associative array to map the URL's with a unique id

1 google.com
2 gmail.com
3 facebook.com
4 maps.com
5 ufl.edu

Step 2: Graph Representation and Page Rank

In PageRank, the equation to calculate the rank for your graph is given as follows:

Rank of a Page, r = M.r where,

M is the matrix with values given by the following:

$M_{ij} = 1/d_j$ if there is an edge from V_j to V_i (d_j is the outdegree of node j)

0 otherwise

For our graph, the adjacency matrix, M will look like:

	1	**2**	**3**	**4**	**5**
1	0	0	0	0	$1/d_5$
2	$1/d_1$	0	0	0	$1/d_5$
3	0	0	0	$1/d_4$	0
4	$1/d_1$	$1/d_2$	0	0	0
5	0	0	$1/d_3$	0	0

Step 3: Power iteration, r(t+1) = M.r(t)

This means that a rank of the webpage at time *t+1* is equal to the rank of that page at time *t* multiplied by matrix, M. To achieve this, we create our matrix M based on input. Next, we initialize r(t) which is a matrix of size |V|x1 and consists of the ranks of every webpage. We initialize r(t) to 1/|V|. Next, we compute *power_iterations* based on our input, *p.* There is a mathematical proof that the matrix r converges, i.e. r(t+1) = r(t) at which point the algorithm stops. However, this is difficult to test and we therefore will be testing your algorithm on a fixed power iteration value, *p*.

Example r and M matrices for our input:

r(0)

	1
1	1/5
2	1/5
3	1/5
4	1/5
5	1/5

M

	1	2	3	4	5
1	0	0	0	0	$1/d_5$
2	$1/d_1$	0	0	0	$1/d_5$
3	0	0	0	$1/d_4$	0
4	$1/d_1$	$1/d_2$	0	0	0
5	0	0	$1/d_3$	0	0

r(1) = M.r(0) =

	1	2	3	4	5			1			1
1	0	0	0	0	1/2		**1**	1/5		**1**	1/10
2	1/2	0	0	0	1/2		**2**	1/5		**2**	1/5
3	0	0	0	1	0	x	**3**	1/5	=	**3**	1/5
4	1/2	1	0	0	0		**4**	1/5		**4**	3/10
5	0	0	1	0	0		**5**	1/5		**5**	1/5
			M		**x**			**r(0)**	**=**		**r(1)**

Note: In our input case, the number of power_iterations, *p* is 2. Therefore we print r(1) of the urls sorting in alphabetical order. If *p* was 1 then return the initializing rank matrix or r(0). If *p*>2, the process repeats where you multiply the matrix, M with the new rank matrix r(t+1) at the next iteration.

Stepik Test Case 2 Explanation (*p*=3):

r(t+1) = r(2) = M.r(1) =

	1	2	3	4	5			1			1
1	0	0	0	0	1/2		**1**	1/10		**1**	1/10
2	1/2	0	0	0	1/2		**2**	1/5		**2**	3/20
3	0	0	0	1	0	x	**3**	1/5	=	**3**	3/10
4	1/2	1	0	0	0		**4**	3/10		**4**	1/4
5	0	0	1	0	0		**5**	1/5		**5**	1/5
			M		**x**			**r(1)**	**=**		**r(2)**

Optional Template

You are allowed to use your own template but make sure your code passes the sample test cases. An example template to think about the problem is:

```
class AdjacencyList {
        private:
                //Think about what member variables you need to initialize
        public:
                //Think about what helper functions you will need in the algorithm
};

void AdjacencyList::PageRank(int n){ }        // prints the PageRank of all pages after p powerIterations in
                                              ascending alphabetical order of webpages and rounding rank to two
                                              decimal places

// This class and method are optional. To accept the input, you can use this method:


int main()
{
         int no_of_lines, power_iterations;
         std::string from, to;
         std::cin >> no_of_lines;
         std::cin >> power_iterations;
         for(int i = 0;I < no_of_lines; i++)
        {
                std::cin >> from;
                std::cin >> to;
                // Do Something
        }
        //Create a graph object
        Created_Graph.PageRank(power_iterations);
}
```

Testing

- Test your code on Gradescope before submitting your implementation. You have five available test cases and you can submit any number of times.
- Create your own tests and test as much as possible. Our recommendation is you spend **at least 1-2 hours** on testing extensively.
- **We will stick to the input format.** No need to test for off-input statement such as inputting one url instead of two in a line or testing whether a url is valid or not.

Grading

- **Implementation [75 points]**
  - **You are supposed to implement an Adjacency List** data structure to represent the graph. **Failure to implement this will incur a 25 points deduction.**
  - Your code will be tested on 15 test cases each worth 5 points:
    - 5 publicly available test cases.
    - 10 test cases that will be added by the course staff and are not shown to the students.

- **Documentation [15 Points]**
  - Submit a document addressing all these prompts:
    - Describe the data structure you used to implement the graph and why? [2.5 points]
    - What is the computational complexity of each method in your implementation in the worst case in terms of Big O notation? [5 points]
    - What is the computational complexity of your main method in your implementation in the worst case in terms of Big O notation? [5 points]
    - What did you learn from this assignment and what would you do differently if you had to start over? [2.5 points]

- **Code Style and Design [10 Points]**
  - 5 points for design decisions, a well-designed Graph API, and code modularity
  - 2 points for appropriate comments
  - 1 point for white spaces
  - 2 points for consistent naming conventions

- **Catch Tests Bonus [5 Points]**
  - You can score 5 bonus points if you submit a separate file containing 5 test cases (1 point/test) using the Catch Framework. These tests should be different than the public test cases. Your score is however capped to 100 points for this project. This means that if you pass 14 tests and submit bonus test files, you will get a 100 provided you score full points on the documentation. Also, if you pass 15 tests and score 23 on documentation and design, + 5 points on bonus, you will still score 100 points [your score is 103 but is capped to 100 in this case].

Submission

- One or more .cpp or.h files that have your implementation. **Test on gradescope early and often**.
- One pdf file that has your documentation. **Upper limit – 3 pages with 20% deduction in report per extra page.** Cover page is not required, just your name on Page 1 is suffice.

Frequently Asked Questions

Based on the number of repeated questions we got in Project 1 on Slack, the course staff will now maintain an active FAQ Google document to answer your questions. Post your questions in Slack, but this time we will answer in this document and send you the link. The link to the document is:
https://docs.google.com/document/d/1a9hR1Ep2IYK-MnsXwl2VxyotO1Yd-XCC8NWUkdp24JM/

Additional Optional Resources

- Page Rank Paper: http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf
- Videos on Page Rank (The assignment is based on these videos): Lectures 5-8
- Extended Videos (Not required for Project): Lecture 9-11