Project 1

COMP 424 – Parallel & Distributed Computing

Spring 2025

**Part A: Due Friday, March 14 by 11:55 pm**

**Part B: Due Wednesday, April 2 by 11:55 pm**

# Collaboration Policy

You will complete this assignment in groups of two. If you do not self-enroll in a group by the deadline, you will be assigned a partner.

You can discuss ideas pertaining to the project with classmates outside your group (e.g., What are we supposed to do? What is PageRank? or What is the TF-IDF scoring formula?), but not how they translate into code. That is, do not show code to anyone outside your group, nor look at anyone's code who is outside your group. If you get stuck, stop by office hours to ask the professor.

Following this policy is not only the expected ethical behavior, it also helps you prepare for the exam.

# Learning Objectives

For this project, you will be creating a web search engine. You will gain experience with several aspects of Scala and parallel systems, including the following:

* Inheritance
* Scala collections and parallel collections
* Map-Reduce

# Introduction

Saying that we use web search engines regularly would be a bit of an understatement. For this project, you get to implement your own search engine!

The text you type into a search bar is called the query. When you enter the query, the search engine searches its database of web pages to find results that are relevant to your query. That database is built long before you enter the query, but it is constantly being updated.

For this project, your database of web pages will be limited to a pre-scraped collection of approximately 700 Wikipedia pages about various programming languages. These are provided to you as a Map[String, WebPage] object loaded by main in your starter. Your search engine will have three primary components. In each of these components, particularly ranking and search, you should use parallel collections to improve speed. Finally, you will report on the quality of the search results.

1. Main Loop (Rank – Search – Sort – Display)
2. Page Ranking — Which pages are most important regardless of search terms?
3. Page Search — How closely does each page’s content match the search query?
4. Search Quality Report

# Submission

For this assignment you will submit the following items in a zip archive.

1. Your entire project directory with all source components and any unit tests you have written.
2. A report describing and demonstrating search results improvements from different techniques.

# [Part A] Main Loop — 10 points

## Core Loop Outline

*Most but not all of these steps are completed for you — make sure you understand where things are happening!*

1. Load WebPage map from resources
2. Rank pages using a method of the PageRank object (defined in the section PageRank)
3. Normalize ranks to fall in the range [0.0, 1.0] using min-max (Appendix A – completed for you)
4. While the user enters a query other than “:quit” . . .
   1. Accept the user query as a string
   2. Compute how well each page matches using the methods of the PageSearch object (defined in the section PageSearch)
   3. Normalize match ratings to fall in the range [0.0, 1.0] using min-max (Appendix A)
   4. Compute the overall match as a mean of the pages rank and text-match. For your report you will be comparing and thus must implement each of the following means.
      1. Arithmetic Mean
      2. Geometric Mean
      3. Harmonic Mean
   5. Sort pages based on their overall match using scala.math.Ordering to support multiple options for computing the mean
   6. Display the name and url of the top 10 results

## [Part A] Ordering[SearchedWebPage] Objects — 15 points

This part of the project will be completed in SortNorm.scala

You are required to sort your SearchedWebPage objects in the main loop using three different subclasses of Ordering[SearchedWebPage]. These subclasses need only implement a single method named compare. The details of how this method should work are essentially identical to the functionality of the compareTo method of Java’s Comparable interface.

<https://www.scala-lang.org/api/3.1.0/scala/math/Ordering.html>

<https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html>

The three comparison options you must implement are Harmonic, Arithmetic, and Geometric means. Each of these will use those means as part of the comparison. In each case you will take the appropriate mean of each pages importance (from PageRank) and textual match (from PageSearch) and compare the averages. The page with the higher average is considered greater than the other.

A simple example sorting pages based on their names has been provided for reference.

# [Parts A & B] Page Ranking — 30 points

One of the most important aspects of modern search is the relative weighting of pages. Certain pages are likely far more authoritative and desirable than others even if they do not match quite as well with the provided search term. For example, should a search query “Harvard” yield “harvard.edu” or “proudharvardgrad.cheaphosting.net”? Clearly the answer is the first result even if the second contains far more occurrences of the search term. That leaves the question of how do we know what pages are important?

One of the greatest benefits of the web for this task is the presence of hyperlinks. Presumably pages that are linked to by more pages are more important. This might even be extended iteratively to say that pages that are linked to by important pages are themselves even more important. We will use these ideas in our page ranking system.

You will implement a PageRank object with methods for the following three ranking methods.

**NOTE:** All listed methods are required, but you are permitted to write as many others as you need. You should not use any mutable variables anywhere in these methods. After all methods are working, you should attempt to use parallel collections to provide a speed-up where appropriate.

## [Part A] Equal Weighting — 10 points

def equal(pages: Map[String, WebPage]): Map[String: Double]

This method is the simplest naïve approach of unweighted results. This should be implemented both for comparison to other methods and to help make progress possible on other parts of the project. This method should return a Map where each string of the input map of pages is mapped to a value of 1.0.

## [Part B] Indegree Weighting — 10 points

def indegree(pages: Map[String, WebPage]): Map[String: Double]

This method should map the id of each page in pages to the number of other pages that link to this page. That is, it should count the number of other pages for which this page’s id appears in that page’s list of links. This will need to be converted to a Double for compatibility with the general interface.

## [Part B] PageRank Weighting — 10 points

def pagerank(pages: Map[String, WebPage]): Map[String: Double]

This method is an extension of the indegree method that accounts for differences in importance between the pages that link to a given page. Most simply a link from an important page should provide more weight than a link from a relatively obscure page. This as you might imagine can be difficult to compute and is usually done with linear algebra. For our purposes we will take the random walk approach ***outlined in Appendix B***. In short, we drop a large number of independent “walkers” on random pages. These walkers then repeatedly choose a page the current page links to and go to the new page. The distribution of where these walkers are positioned after a large number of steps is the distribution of the importance of the pages in our system.

# [Parts A & B] Page Search — 30 points

Finally, the most familiar aspect of search — determining how closely a query matches a page’s content. There are numerous ways this can be done. We will follow an approach that starts from the simplest solutions and gradually introduce improvements until we have one of the most widely used algorithms. In each case we will divide our query into terms. Terms refer to individual components of the query which for our simple example will be the component words e.g. “Hello World” 🡪 List(“Hello”, “World”).

The earliest and simplest approach to matching queries to content is to simply count the total number of occurrences of all of the search terms in the content. As one might imagine, this approach heavily favors longer documents. Our first improvement upon this will be to normalize this value by dividing by the length of the content in characters. This is still not very robust. Suppose our user types “The Bible”, should we return the document that says either “The” or “Bible” the most total times per character? Of course not — this would lead to identifying the pages that most commonly use the word “The”. Our solution will be something called TF-IDF. This technique gives higher weight to terms that appear in only a few documents and lower weight to words that appear everywhere.

You will implement a PageSearch object with methods for the following three search methods.

**NOTE:** All listed methods are required, but you are permitted to write as many others as you need. You should not use any mutable variables anywhere in these methods. After all methods are working, you should attempt to use parallel collections to provide a speed-up where appropriate.

## [Part A] Simple Count — 10 points

def count(pages: List[WebPage], query: List[String]): List[Double]

This method should compute for each page the number of times any of the terms in the query appear as substrings of the page content. You may implement this to be case-sensitive or case-insensitive as you see fit. Your output list should correspond to your pages list such that with ith value of your output list is the number of term matches in the ith page of pages.

## [Part B] Term Frequency — 10 points

def tf(pages: List[WebPage], query: List[String]): List[Double]

This method is an enhanced version of the count method. Each value of the output should be equal to the result of the count method for that page divided by the number of characters in the page text.

## [Part B] TF-IDF — 10 points

def tfidf(pages: List[WebPage], query: List[String]): List[Double]

This method is an enhanced version of the term frequency method. In this case we add an additional metric — inverse-document frequency. In this method the tf result for a single term on a page is multiplied by the inverse frequency with which the term appears in the text of other pages. (Appendix C)

# [Part B] Search Quality Report — 15 points

Demonstrate improvements gained (or not gained) by using certain page ranking or query matching techniques. Be sure to include example queries and results to demonstrate your point. This report should describe the results of several experiments using different combinations of ranking and text matching functions for search. For each experiment you should provide the following information.

1. A statement of which combinations will be compared. — e.g. “For our first experiment we compared the results using indegree ranking with term frequency and TF-IDF matching.”
2. A statement of what queries you will test and why you think they will show differences between your chosen configurations.
3. A table showing the search results for each query on each configuration.
4. An analysis of the results stating whether the outcome matched your expectations and analyzing why that might be the case — particularly if the results were unexpected.

You should perform at least three such experiments and include an introductory and concluding paragraph summarizing the overall experimental set and the results.

Your reports will be graded according to the following rubric.

* Writing tone, quality, and formatting are professional (3 points)
* At least three experiments were performed and described according to the outline (12 points)

# Appendix A — Min-Max Normalization

One common requirement in taking a meaningful mean of two number requires that they exist on the same scale. Unfortunately, half of our numbers are page importance rankings, and the other half are page search scores. Furthermore, within these categories we will get dramatically different results — raw count will be at a totally different scale than count divided by character length. Consider for a moment the consequences — if we average a number in [0.0, 0.1] with a number in [1000, 10000], the result will always be approximately half of the second number with almost no influence from the first. There are many solutions to this problem, each with advantages and drawbacks. We will use min-max normalization.

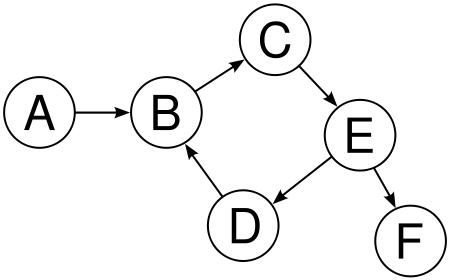
When using min-max normalization we first compute the minimum and maximum values in our collection. These values will map to 0.0 and ~1.0, respectively. Every other value will fall linearly on the scale depending on its relationship with the minimum and maximum values. A very small value is added to the denominator to avoid division by zero in the edge case where all values are the same.

Using the above formula, we can convert any raw value to its normalized counterpart .

# Appendix B — PageRank

PageRank is the original algorithm used by Google to rank the importance of web pages in their search results. As of 2019 all patents have expired, and the algorithm is public domain. The algorithm works by estimating or computing the probability of landing on a given page by clicking random hyperlinks. This can be accurately computed in several ways. We will estimate this probability distribution using the Monte Carlo method for simplicity.

Your PageRank implementation will simulate 10,000 random users clicking randomly on links for 100 steps. In order to account for leaf nodes in our directed page graph, we will use a damping factor of 85%. Below is a walkthrough of a simple example of a single random walk using the graph below.

1. Choose a random starting vertex: C
2. With weighted probability do one of the following
   1. 85% randomly follow a link (E is the only option)
   2. 15% randomly jump to a page (uniform probability)
3. We randomly choose to follow the link to E now we repeat.
   1. 85% chance to follow a link
      1. 50% chance we choose F
      2. 50% chance we choose D
   2. 15% randomly jump to a page (uniform probability)
4. We randomly choose to follow the link to F now we repeat
   1. No outgoing links so 100% chance to jump to a random node
5. We randomly choose B

. . .

Using the above method have 10,000 such random walks for 100 steps each. The raw weight of each page is then the percentage of the 10,000 random walks that ended on that page. We will normalize this slightly so that every page has a chance. Assuming we have random walks over pages and the number of random walks that stopped on page is then the normalized weight of page is as follows.

# Appendix C — TF-IDF

TF-IDF or term frequency – inverse document frequency is used to measure how important a word is within a collection of documents. Some words such as “a”, “an”, and “the” will almost never be important. Others such as “computer” might be important in a general collection but cease to be important in a collection of computer science textbooks. Simply put, a word rarely indicates a distinctive topic of any given document if all documents contain the word. The way to incorporate this is to consider not only whether a document contains a word, but also what percentage of others contain it. There are many definitions for term frequency and inverse document frequency. For this assignment, term frequency is the number of times a term appears in a document divided by the length of the document. Inverse document frequency is defined below where is the number of documents in the corpus and is the number of documents in the corpus that contain the term. TF-IDF is then simply the product of the term frequency inverse document frequency.