# COSC 420 - High-Performance Computing
# Project 2

Dr. Joe Anderson

Due: 15 November

## 1 Description

**Working in groups of two or three**, you will implement a document search engine using the PageRank algorithm, and also provide other methods to "explore" a large dataset of documents.

1. You will create a search engine to search scientific articles stored on `arXiv` (pronounced "archive").
2. This will consist of two components:
   (a) An index of text search terms, to know which documents are appropriate results to be returned
   (b) A ranking algorithm to determined which papers are the most "important", relative to the given search term. This will use the PageRank and HITS algorithms, as discussed in lecture and the textbook.
3. You will use two data files: `arxiv-metadata.txt.gz` and `arxiv-citations.txt.gz`. Note that these are provided as compressed text files, which can be extracted by the `gunzip` utility.
   (a) The first one provides a list of every paper in the arXiv database, along with id, title, authors, and abstract.
      i. Each paper is separated by a line containing the string `++++++`
      ii. The first line is the id of the paper (note that these are *not* numerical ids, but strings)
      iii. The second line is the title of the paper
      iv. The third line is the authors of the paper
      v. The third line is the abstract text of the paper
   (b) The second file provides, for each paper in the database, a list of all the other papers which it cites.
      i. Each paper is in a block of text separated from the rest by `+++++`
      ii. The first line of the block is the id of the paper. It is then separated fro the other ids with `-----`
      iii. All subsequent lines (if any) are papers cited by that paper (i.e. outgoing connections in the graph)
   (c) For reference, these were generated by the software provided at https://github.com/mattbierbaum/arxiv-public-datasets
4. Your project will need two main "modules":
   (a) A module to create and maintain the document search index

i. The program should read through the `arxiv-metadata.json` file and build a structure that creates a "backward index" which maps key words to documents that contain those key words – in our case, instead of using the full document text, we will just use the document abstract. In other words, this creates a table where the word "eigenvalue" maps to a list of all documents whose abstracts contain that keyword.

ii. This index should be built in parallel, using MPI. One way to do this would be to split the document database, build separate indexes, then merge them into one at the end.

iii. Implement an advanced storage mechanism to hold the document index (search tree, hash table, etc.), to make your lookup times faster. Remember: linear search is bad, if we can avoid it! For example, if you store each found document word in a balanced binary search tree, the worst case time to find the corresponding documents is at most $\log 500,000 \approx 19$.

A. In this model, the "key" of the BST nodes would be a word, e.g. "eigenvalue" and the words on its left would be ones that come earlier in the dictionary and on the right would be words that come later.

B. The nodes will have some "satellite data", which would be a list of document IDs that contain that search term.

C. Note that this can play fairly nicely with parallelization, if you start by having a sorted list of search terms, which then gets divided into sub-trees of a BST.

iv. You will need to write that index to a file, which can be quickly read back into the data structure when a user makes a query.

(b) A module to create and maintain the edge graph of the network, and update the hub scores and authorities of each page, according to the HITS algorithm, as well as the rank given by the PageRank algorithm.

i. Using the `arxiv-citations.txt` file, build an adjacency graph for the paper citation network.

ii. You may create any extra metadata files you might find useful (e.g. a separate file containing each paper id along with its row index in the matrix)

iii. Entry $A_{ij}$ of the matrix should be 1 if paper $i$ cites paper $j$, and 0 otherwise. Note that this will very likely be a sparse matrix!

iv. Using this matrix, compute PageRank, hub score, and authority score, of each paper, as described in the course lecture and textbook (Section 9.4.2).

v. Store these scores somewhere (probably in a file of your own formatting choice), to be used by the search engine later.

5. The actual usage of this system will of course be done by a single program, with a user interface that allows a user to enter a search query such as "gaussian mixtures" and then it returns a list of papers (and their authors), ordered by their rank. For multi-word searches, your program should also take into account which articles have "all" the words, versus only some of the words. You may design your own system to break ties, or deal with interesting edge cases (e.g. a very high rank page matches only one word, but only low rank pages match both). Be sure to document these solutions – that's where the software design gets fun!

6. Thoroughly document each module of your search engine, along with a performance study of each.

(a) How long does it take to process the raw data into your index format?

(b) How long does it take to load your database into memory?

(c) How long does it take to return results to a user?

(d) Where is there room for improvement?

(e) Would your program make a decent "Google for research papers" ?

Include a `README` file to document the problem, its solution, your code, and other required information. Include a `Makefile` to compile the project. Be sure to include full and thorough documentation.

## 2 Submission

Submit a `.zip` file called `Project2[GroupName].zip` (with `[LastName]` replaced with your own last name) containing your source code, `Makefile`, and documentation, then upload it to the course MyClasses submission page. Include printouts/logs of successful operations of your program.

## 3 Bonus

If any bonus are completed, be sure to note it in the README file and provide appropriate output to demonstrate its correctness.

1. (10 pts) Implement sparse matrix encodings and their corresponding usage, to speed up the program and reduce its memory overhead. Thoroughly benchmark this to see the performance gains you get.

2. (5 pts) Calculate and store some high-level statistics of your network: are there only a few highly-cited works, or is it more evenly distributed? Proportionally, what trends can you find within topics (looking at the id to group by topic)? What are the most popular papers? Who are the most prolific authors?

3. (10 pts) Implement a similar ranking system, but for authors! Create a network where authors are the nodes and there is a link from author $i$ to author $j$ if $i$ ever cites *any* of $j$ papers. Create another network where the links are defined by co-authorship: $i \to j$ if $i$ and $j$ are authors on the same paper. What is the structure of these graphs?

4. (10 pts) Implement the Floyd-Warshall algorithm to calculate shortest paths between all nodes in the graph. Present some example paths between highly-ranking papers.