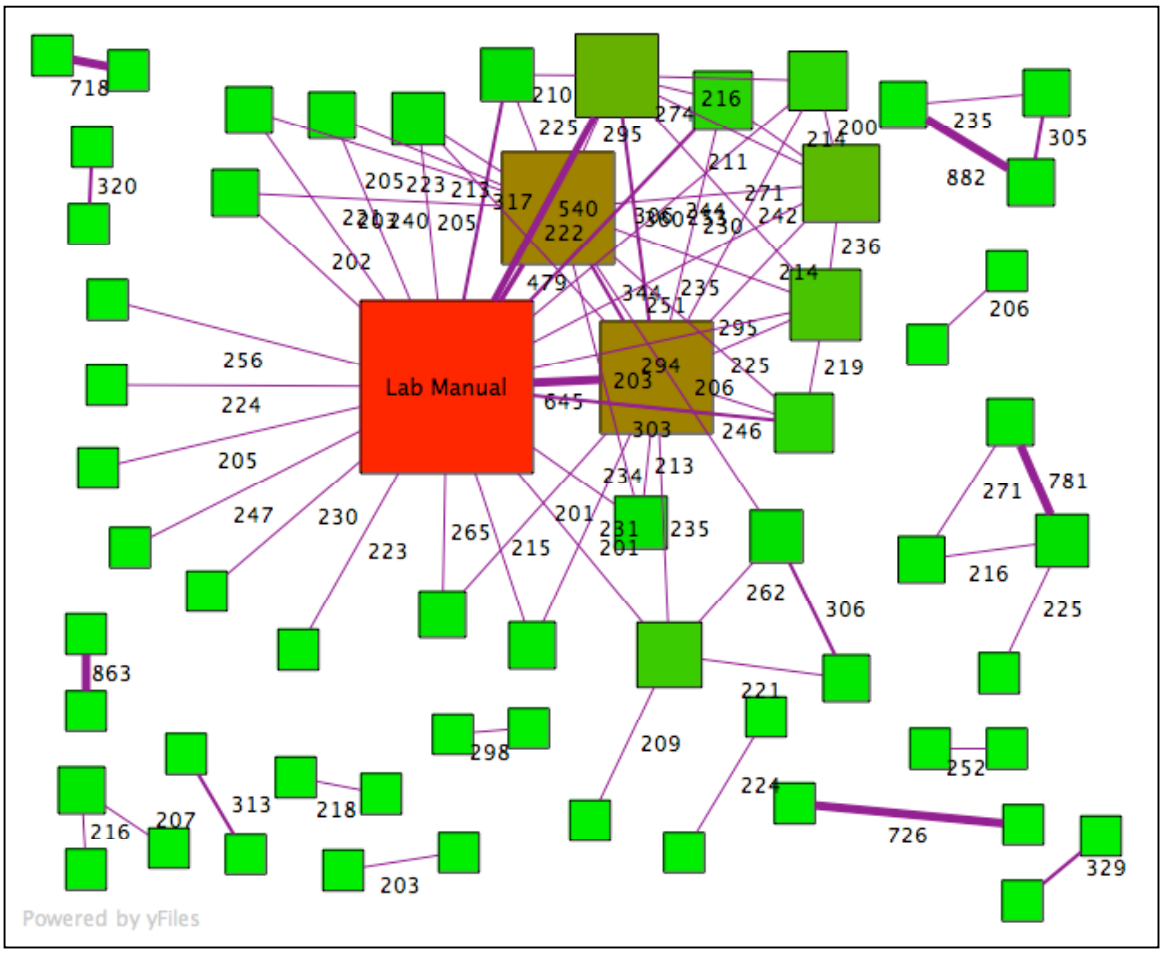
Lab 21 : Catching Plagiarists

This lab presents a real problem that requires a software solution. Your goal is to try to (quickly) determine the similarities between documents in a large set to see if you can find out if plagiarism is going on within the group.

Background:

Below is an actual graph of lab reports submitted for Intro. Physics at a large University. This graph represents the data collected for about 800 lab reports. Each node in the graph represents some document. Each edge indicates the number of 6-word phrases shared between the documents it connects. To reduce "noise" a threshold of 200 common phrases has been set – so a document that shares fewer than 200 6-word phrases with all other documents is not shown. The "Lab Manual" is a sort of style-guide for the lab report and the two brown boxes are sample lab reports that were distributed. (Many people apparently "borrowed liberally" from these help materials). Particularly suspicious are clusters like the one in the top-right corner: those documents have an inordinate number of 6-word phrases in common with each other. It is likely that those people turned in essentially the same lab report or copied large portions from each other.



Assignment:

Your task is very similar to the one described and shown above: find the common word sequences among documents in a closed set. Simply put, your **input** will be a set of plain-text documents, and a number n; your **output** will be some representation showing the number of n-word sequences each document has in common with every other document in the set.

Finally, you should identify "suspicious" groups of documents that share many common word-sequences among themselves but not with others.

DETAILS:

- *Output*:

You can think of processing everything into an NxN matrix (where N is the number of total documents) with a number in each cell representing the number of "hits" between any pair of documents.

For example: below is a small table showing the comparisons between 5 documents:

	A	B	C	D	E
A	-	4	50	700	0
B	-	-	0	0	5
C	-	-	-	50	0
D	-	-	-	-	0
E	-	-	-	-	-

From this table we can conclude that the writers of documents A, C and D share a high number of similar 6-word phrases. We can probably say A and D cheated with a high degree of certainty.

For a large set of documents, you may only want to print a matrix for those documents with a high number of hits above a certain threshold.

Printing an NxN matrix may be unmanageable for large sets. You could instead produce a list of documents ordered by number of hits. For example:

```
700:  A, D
50:   A, C
50:   C, D
5:    B, E
4:    A, B
```

You could also produce a graphical representation like the one shown above. If you want to discuss strategies for how to accomplish this please see me.

- *The documents:*

Some sets of documents will be provided. One set will be small (25 or so documents) for testing purposes. The other sets will be larger (one has 75 documents, the other over

1300 documents) which you should use to test the scalability of your solution. (The documents came from www.freeessays.cc, a repository of *really bad* high school and middle school essays on a variety of topics).

Your program should be able to process all of the documents in a given folder/directory.

- *Strategy:*

How are you going to do this? We'll it's up to you. The straightforward matrix solution (comparing each six-word sequence, say, to all other six-word sequences) gives an $O(w^2)$ solution – where w is the *total number of all words in all documents*. For a large set of documents w^2 grows very large, very fast. It will work though – it will just take a while. For perspective, if the 25-document set takes 10 seconds to process this way, the 1300-document set will take over 6 hours…if you can actually hold the necessary data in memory which you probably can't.

There may be a clever way to use a hash table or to leverage some ideas from sorting algorithms that will, in theory, do better than $O(w^2)$. The problem with the hash table strategy and some sortings is not the time complexity but the space complexity. For a large number of documents the amount of memory required to compute this is too large to hold in memory all at one time. If you want this solution to scale to large sets of documents, you'll have to do even more clever things, probably by creating your own supplementary data files that you can store and load on demand.

One way to gain ultimate control over the processing is to write your own specialized data structures. However, you're free to use anything in the java API.

Getting started, Grading, and Milestones:

Your grade will consist of points you earn for meeting each milestone on time and for the final product you submit. Each milestone will be submitted and checked against the criteria described below.

Milestone I :: Processing the Documents (50 points)

You need to be able to process a set of documents in a directory and produce all possible n-word sequences. You should be able to change n relatively easily. Proof of this milestone consists of demonstrating you can print all n-word sequences to the console for a given n.

Write a class with a main method called FileProcDemo. I should be able to call main({"path/to/text/files", "6"}) and see a printing of all 6-word sequences of all the files contained in the folder "path/to/text/files".

Milestone II :: A proof-of-concept data model (50 points)

You need to have some sort of model for how you're going to handle all of the data you're going to generate. You can create this separately, and test it with a small set of data to produce a "proof of concept."

Basically, I need to see some proof of how you are going to compute the similarities between documents and also how you're going to locate suspicious cases. I'll need to look at what you've done and see a demonstration of the basic concepts. Good design counts for a lot here! Not just for your grade, but for the functionality of the program.

Turn in Milestone II with a demonstration class clearly labeled and ready to run. You do not need to be working with the actual data at this point, but you may find it helpful to try it out. You may supplement your work with a README if you think it requires further explanation.

Milestone III :: Final Submission (100 points).

The minimum here is essentially a synthesis of Milestones I and II plus the last step of identifying the suspicious cases. A nice product would be a simple console application that accepts 3 command-line arguments. For example:

```
./plagiarismCatcher path/to/files 6 200
```

which would churn and then produce a list (in order ☺) of all the pairs of files in path/to/docs that shared more than 200 6-word sequences in common.

Your final program should be able to produce meaningful output for at least the small set of documents (25 or so).

Lastly, with Milestone III you will submit a short document (the project README) about what your program does, how to use it, what works, what doesn't work and any other features, bugs I should know about when I'm looking at your code.

Beyond Milestone III :: Bigger sets

- Make your solution stable for medium and/or large sets of documents. The sample large set provided should give you something to play with.
- A more dynamic user interface: allow the user to repeatedly re-process the documents by allowing her to change the word-sequence length and "noise" level.
- Produce a GUI for user interaction
- Produce a graphical representation of your results (like the box-and-line one shown above).
- …you could take this all the way to producing a full-fledged piece of software that would be quite helpful!