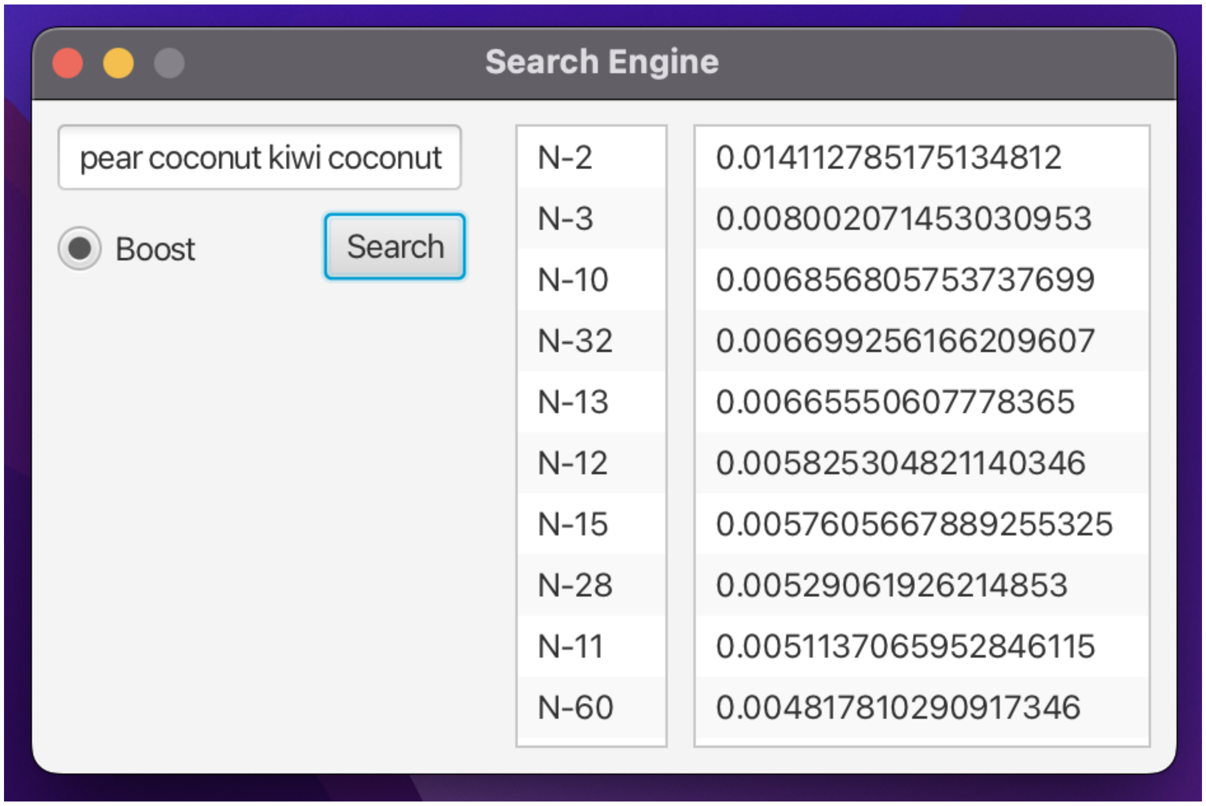
COMP 1406 Course Project Report

How to run the GUI

The GUI can be run from the controller.java class. Simply set up java fx accordingly on your device and run the program, and a simple GUI should appear as so:



N-2	0.014112785175134812
N-3	0.008002071453030953
N-10	0.006856805753737699
N-32	0.006699256166209607
N-13	0.0066555060778365
N-12	0.005825304821140346
N-15	0.005760566788925532 5
N-28	0.005290619262148 53
N-11	0.005113706595284611 5
N-60	0.004817810290917346

Type in your search query, toggle the boost radio button as you wish, and press the search button.

List of Functionality

All required functionality has been implemented.

Outline of Classes

This outline will be done through a top-down approach. I will start by discussing the broadest class, ProjectTesterImp.java, and work down towards the most specific classes, such as Result.java.

ProjectTesterImp.java

What/How?

This class implements the ProjectTester.java interface and thus has all the required methods needed to run tests on the web crawler and search engine. This class has three key objects, a Crawler, a Getter, and a Searcher (which is a local variable of the search() method). These three objects do practically all of the heavy lifting for all of the methods in this class, with most methods being single-line return statements calling methods from those objects. The only exception to this is the initialize() function, which calls a recursive helper function deleteDirectory that deletes all previous crawl data.

Why?

I wanted this class to look as clean and readable as possible, and I found that delegating all of the heavy lifting to other objects was the best way to do this. This method also resulted in more modular and extensible code that ended up improving other classes, especially Searcher.java. I also found that for my file system, a recursive function worked best to parse through every file and folder and delete where necessary.

Crawler.java

What/How?

This class handles crawling and all associated calculations and storing of data. There is only one public function, crawl(), that crawls a network from a seed URL. Here is some basic pseudocode of how crawl() works.

1. Create the main directory “crawlFolder” and subdirectory “urlFolders”
2. Create an UrlScanner object and add the seed URL to a queue of URLs that will be scanned
3. While that queue is not empty:
    a. Pop the first URL and scan it
    b. Update the hashmap with the scanned URL and its folder, and add the scanned URL to the list of scanned URLs
    c. For each outgoing URL from the scanned URL, update the incoming URL hashmap and add the URL to the queue of URLs to be scanned, if appropriate
4. Serialize the hashmap that stores each URL and its corresponding folder name
5. Update all incoming link information and store the data accordingly

6. Calculate all PageRank scores and store the data accordingly
7. Calculate all IDFs and store the data accordingly
8. Calculate all TFIDFs and store the data accordingly

Many private functions are used throughout crawl(), and they are all pretty straightforward (calculations, fetching data, etc.)

Why?

Practically all crawling functionality is done from this class, with the exclusion of actually scanning each URL. That, combined with the use of many helper functions, makes crawl() very readable and easy to follow and allows me to debug effectively since I can usually track down my issues to a specific method. I also considered making the class a utility class, however, I considered that multiple crawls may need to be done at once with different seeds, so I decided to keep it as is. The use and serialization of a hashmap to store the URL folder information were also proven to be useful within the crawl and with several classes as well.

UrlScanner.java

What/How?

This class takes care of the data collection for a URL. It has one public method, scan(). Here's some basic pseudocode:

1. Read the URL using the WebRequester class
2. Create the folder for the URL, as well as the words subfolder
3. Stores the URL address
4. For each line
    a. Collect all the word data and calculate and store tf values for all of them
    b. Collect all outgoing URLs and store them
5. Return all outgoing URLs

Why?

Parsing each URL line by line was the most effective way to go about scanning it, and from there it was simply a matter of string manipulation, calculation, and data storage.

Getter.java

What/How?

This class is used to retrieve any information needed from any URL/word. It works by using a serialized hashmap that maps each URL to its respective file name within the crawl folder. It makes very good use of FileOperations.java to read the files that contain the requested information. There is also an update() function that rereads the hashmap, typically called after a crawl.

Why?

Making use of serialization was crucial in providing quick mapping to each respective URL folder, and from there it was simply a matter of opening the needed files.

FileOperations.java

What/How?

This is a utility class used for file input and output. It has five methods that can deal with different types of files (single line, multi-line), and return different outputs (string, double, list).

Why?

I decided to make this utility class since I found I was writing a lot of duplicate code throughout multiple other classes. Not to mention, this code doesn't look very pretty and makes the classes look more cluttered and messy. I found it was best to abstract the input and output operations into one class to improve simplicity and readability. Also, this class is useful for any kind of project involving file input/output, so I may use it in future projects.

Searcher.java

What/How?

This class handles all searches. It uses a Getter object to handle all data retrieval, the entire search process is done from the search() method. Here's some basic pseudocode for this method:

```
1. Use the Getter object to get the URL folder map, and initialize a priority queue and a results list.
2. Create the query vector
3. For every URL in the database
    a. Create the vector for the URL
    b. Calculate the cosine similarity between the two vectors
    c. Create a result object and add it to the priority queue
4. Add the first X elements from the priority queue to the results list, and return it
```

Why?

While the actual algorithm of the search is pretty straightforward, I found that when I originally made the method (without a Getter object), each search took about twenty seconds. After some investigating, I determined this was because I was repeatedly deserializing the URL folder map, which was an expensive operation. This is what led to the creation of the Getter class, which can deserialize the URL folder map once, greatly improving the efficiency of each search. After this change, each search took about 0.1 seconds.

Result.java

What/How?

This class implements SearchResult.java and is used to store search results. There are two get methods, for the title and score, and a compare method. The compare method compares the two scores up to three decimal places, and if they are equal, compares the two titles lexicographically.

Why?

The get methods are straightforward, however, I decided to use a priority queue in my search algorithm which required me to implement a compare method for this class. Doing the dirty work of comparing from this class allowed the search code to become much cleaner and readable.

View.java

What/How?

This class handles the view portion of the GUI. The view itself is fairly simple, a text box for the search query, a radio button for boost, and a search button. The results appear on the right, with the names and scores appearing in separate columns. The update function clears these columns and replaces them with whatever input is received from the controller.

Why?

I didn't want to spend too much time making the GUI look super nice since I was more concerned about making a good OOP project and improving efficiency. It gets the job done and displays all relevant information.

Controller.java

What/How?

This class handles the user interaction with the view. It uses a Searcher object to execute the searches and uses an event handler to detect when the search button is pressed to run the test. The view is updated with each search.

Why?

Not much complexity was needed for the controller since the view itself is simple and the search is completely abstracted away. All that was needed was one event handler. However, I could've added more if I put more emphasis on the GUI (colors for search results, selecting the same row of each column, etc.)

Controller.java

What/How?

This class handles the user interaction with the view. It uses a Searcher object to execute the searches and uses an event handler to detect when the search button is pressed to run the test. The view is updated with each search.

Why?

Not much complexity was needed for the controller since the view itself is simple and the search is completely abstracted away. All that was needed was one event handler. However, I could've added more if I put more emphasis on the GUI (colors for search results, selecting the same row of each column, etc.)

Overall Design

When I first started to create the Java web crawler, my goal was to essentially translate my Python crawler into Java. As I was starting to implement this, I started to notice that I wasn't really hitting any of the marks of creating a good OOP project and wasn't taking advantage of the fact that Java is statically typed. It was almost as if I was wrestling with Java, trying to get it to be just like Python which made my code buggy and slow. After a while of trying this method, I decided to completely restart from scratch since it simply was not working.

I first decided on a new file system. My Python file system wasn't a system at all, it was one file that heavily relied on the JSON package which we no longer have access to. In hindsight, I could've used a single Hashmap to collect all the data I needed and serialized it, but in the end, I believe my new file system was much better at reducing space complexity and improving the organization of data. The file structure is as follows:

```
crawlFolder/
├── urlFolder/
│   ├── 1/
│   │   ├── words/
│   │   │   ├── apple/
│   │   │   │   ├── tf
│   │   │   │   └── tfIdf
│   │   │   ├── banana/
│   │   │   │   ├── tf
│   │   │   │   └── tfIdf
│   │   │   └── etc.
│   │   ├── incomingUrls
│   │   ├── outgoingUrls
│   │   ├── pageRank
│   │   ├── title
│   │   └── urlAddress
│   ├── 2/
│   │   ├── words/
│   │   │   ├── apple/
│   │   │   │   ├── tf
│   │   │   │   └── tfIdf
│   │   │   └── banana/
│   │   │       ├── tf
│   │   │       └── tfIdf
│   │   ├── incomingUrls
│   │   ├── outgoingUrls
│   │   ├── pageRank
│   │   ├── title
│   │   └── urlAddress
│   └── etc.
└── wordFolder/
    ├── apple
    ├── banana
    ├── peach
    └── etc.
```

This was essentially how the data was organized in the JSON file, but instead, it was spread out across multiple folders and files. I decided to name each URL folder using numbers because it is the simplest naming convention and easiest to map. These folders each have their own subfolder of words that have their own subfolder containing their tf and tfIdf values. There are also files for incoming URLs, outgoing URLs, PageRank, site title, and address. Outside of the URL folders, there is a folder of words that contain each of their idfs. This file system presented no issues for me and I found it was the simplest and most effective way of storing a large amount of data.

The second time around, my plan was to create a crawler class, and whenever I felt as if I would benefit from a new class, I would create it. I would also encapsulate everything that was appropriate to encapsulate and provide getter and setter methods when needed.

This first led to the creation of the UrlScanner class. This class abstracted the actual data collection from each URL away from the crawler itself and made the classes more extensible and readable. I made most classes and variables/objects within the Crawler and UrlScanner objects private in order to make the code more robust. Those were the only two classes I needed to make the crawler itself, so from there I shifted my focus to the ProjectTesterImp class.

My original plan was to write all the code for retrieving data within each method of this class, and while this worked at first, I noticed that there was a lot of code repetition, it was messy, there was no abstraction, and it wasn't extensible. My first step to improve the situation was to create the FileOperations class, which greatly relieved some of the messiness of the code and provided me with a utility class that I could use throughout the rest of my project and for future projects as well.

After doing that, I was planning on starting the GUI, but I noticed that I was only able to run a search through a ProjectTesterImp object, which didn't really make much sense. So I decided to make a Searcher object, but I noticed that it was running very slowly due to the repetitive deserialization of the URL folder map for each search. My way around this was to create a Getter class that deserializes the URL folder map once and can grab data many times without deserializing it over and over again. This worked excellently with the Searcher class, so I decided to use the Getter class throughout the entire ProjectTesterImp class. This resulted in super clean, readable code that can easily be debugged and extended as needed. At this point, I could delete the entire ProjectTesterImp.java file and still have all the classes necessary to perform a crawl and search, which is exactly what I wanted.

At that point, I decided that my crawler and search engine were pretty solid in terms of OOP functionality. However I will say that I did not use inheritance and polymorphism at all, and those are two powerful aspects of OOP that could've provided benefits to the program. Overall, I was satisfied with my implementation of OOP principles throughout the project, especially the abstraction and encapsulation aspects.