Isabella Nguyen
Fall 2021 COMP 1406Z

OOP Designed Web Crawler Report Analysis

This project is a re-implementation of the web crawler from COMP 1405 except in a well-thought-out object-oriented design, with the addition of a GUI application to display and perform the search.

Instructions for running the GUI

The GUI should be run only after the crawl is complete. Once ready, click run on the `ProjectApp` class (the controller), and a window will pop up. There is a query text field at the top where the user can enter their search, and a toggle button directly under it for the user to indicate whether they want `boost` to be on or off. The toggle button will show “page rank OFF” or “page rank ON” depending on the user’s choice. Once the user has entered their desired query and page rank option, the user must click the “search” button beside the query for the search results to appear in the `ListViews`. There is a `ListView` for the titles and another for the corresponding scores.

Outline of all the classes

Page class

The `Page` class represents all the pages that are found during the crawl. This class extracts all the necessary data from the HTML string of the page and stores all the information. Examples of the data stored in this class would be the title, URL, words, outgoing and incoming links, corresponding file number, etc. This class also calls the TF and TF-IDF methods from the `Word` class to calculate the values for each word on the page. Additionally, the `Page` class contains methods to save and load information to and from files.

Crawl class

The `Crawl` class does exactly what the title implies. It performs the crawl by calling methods from the `Page` class to extract all the necessary info for each page found in the crawl by using a queue to keep track of all the pages needed to be parsed. This class also keeps track of all the `Page` objects (`allPages`) during the crawl to calculate all the TF, TF-IDF, and page rank values by calling on methods from the `Page` and `CalcMath` classes. There is also an instance variable that keeps track of all the words found during the crawl (`allWordsFound`) to calculate the IDF values through methods from the `Word` class. After the crawl is done, the class saves all the info into files for further use.

Word class

The `Word` class represents all the words found on the pages. This class contains methods that perform the calculations for the TF, TF-IDF and IDF values for each word. This class also saves the IDF values to files for every unique word found in the entire crawl and loads the TF, TF-IDF and IDF values from files.

CalcMath final class

This final class contains several methods that perform calculations, such as calculating the page rank and cosine similarity, used in other classes. This class is final because it should not be instantiated nor extended from, it is only here for simplifying and reducing code using abstraction.

<u>ProjectTesterImp class</u>
This class implements the `ProjectTester` interface and calls on methods from other classes to return the correct results.

<u>RetriveInfo final class</u>
The `RetriveInfo` class is similar to the `CalcMath` class in the way that it contains static methods as a form of simplifying and reducing code by abstraction. Differently, the `RetriveInfo` class' methods are used to abstract the details of retrieving information from files to make the code easier to follow and to reuse code through generalized methods.

<u>Search class</u>
This class takes in the query, boost and X value to find the vectors and cosine similarity scores by calling on methods from the `PageSearch` and `QueryVector` classes to rank the pages from highest to lowest score in a `TreeSet`.

<u>PageSearch class</u>
This class takes in a URL, calculates the page vector for that URL and calculates its score to be used in the `Search` class.

<u>QueryVector class</u>
This class takes in a query string and performs the necessary computation to create a query vector to be used to calculate the cosine similarity score for each page.

<u>SearchResults class</u>
This class implements the `SeachResult` interface and `Comparable<SearchResults>` interface so that the `TreeSet` in the `Search` class knows how to sort the objects of this class.

<u>ProjectView class</u>
The "view" aspect of the MVC paradigm. This class creates all the necessary components of the GUI and performs the necessary updates to be displayed to the user.

<u>ProjectApp class</u>
The "controller" aspect of the MVC paradigm. This class connects the model (`Search`) and the view (`ProjectView`) to be able to display the correct results and updates to the user.

Discussion of OOP Principles

The `Crawl` class is very abstracted because it mainly calls on methods from the `Page` class, making the code very short and easy to read. For example, to extract all the necessary information from each page in the queue, only one line is called in the `Crawl` to perform this, `nextPage.extractAllHTMLInfo();`. This method extracts and saves the title, words, incoming and outgoing links of the page. This abstracted view leads to encapsulation, as in the `Page` class, the `extractAllHTMLInfo()` method calls on several private methods to perform the computation. This makes it very easy to read, as one can clearly see exactly what the `extractAllHTMLInfo()`method does. The encapsulated private methods force other classes to only call `extractAllHTMLInfo()`, improving robustness by making sure other classes don't forget to, for example, extract all the words from a page. If the words were not extracted from the page, that will cause many errors later on in the project. This also prevents classes from accidentally calling the method to extract words *after* they already extracted all the info, because the method is private so there is no way another class would be able to access that method to call it individually, improving robustness. Another way this improves robustness is that this ensures that the methods are called in the correct order. For example, the `addIncomingPage()` method depends on the `extractOutgoingPages()`, therefore must be called *after* the `extractOutgoingPages()` method. This abstracted and encapsulated view of methods in both classes also improves maintainability, as one could easily change the implementation in several methods in the `Page` class without having to change the `extractAllHTMLInfo()` method nor anything in the `Crawl` class. Extensibility is also improved here because it would be very easy to add new functionality, such as a new method in the `Page` class since everything is abstracted by being divided up into several methods. Therefore, one could add another feature to be extracted and saved in a `Page` by simply making a new method and then calling it in the `extractAllHTMLInfo()` method without having to change the `Crawl` class or anything else at all.

In the `Page` class, `extractContent()` extracts the HTML string from the page and catches some exceptions. The exceptions are handled by returning different integers based on what the exception was or if there was no exception. These returned values are then used in the `extractAllHTMLInfo()` method to determine whether or not to extract the rest of the info from the page. The `extractAllHTMLInfo()` method returns the same results as the `extractContent()` method so that the `Crawl` class knows what to do next. This is a form of abstraction, as we are simply calling methods to determine functionality, instead of writing the whole code. This increases readability and makes it easier to understand and modify as it shows that we want to do the following things ONLY if this occurs (shown by what the method returns).

The `CalcMath` and `RetriveInfo` classes are final classes with several static methods. This is because they contain methods to abstract away the details of certain longer or reused calculations/computations in other classes to improve modularity and readability. One could think of these classes as utility classes similar to Java's `Math` class. Java's `Math` class is great for abstraction, as we do not need to know how Java calculates, for example, the absolute value of a number. We just want the result. That makes our code much easier to write and understand. This was the idea I was aiming for while making these helper classes. Some calculations, such as the page rank, were really long, while some computations, such as retrieving information from a file, were used multiple times throughout many classes. Therefore, it made sense to simplify and abstract the code in these classes. These two classes were originally abstract, but then I noticed that it would make no sense if another class were to extend from one of these classes, as that would not portray a proper "is a" relationship, therefore changing it to a final class.

The `CalcMath` class contains a method to calculate page rank. The method that calculates page rank calls on several private static methods that serve as helper methods to make the code easier to read and for abstraction. For example, `multMatrix(double[][] a, double[][] b)` is one of these methods. It is simply called in the page rank method and abstracts away the details. This improves modularity as one could easily change the implementation of how to multiply matrixes and it would not affect other parts of the code. In addition, all the helper methods are private. Therefore, other classes do not have access to them and are forced to use the method that calculates page rank. This ensures an abstracted and encapsulated view of the class to make sure that page rank is calculated correctly, improving robustness. We don't want other classes trying to calculate page rank themselves using the helper methods and calculating it incorrectly. The `CalcMath` class also contains a method to calculate cosine similarity that uses the same OOP principles as the page rank method.

The `RetrieveInfo` class contains generalized methods to easily retrieve information from files. They are not specific to what they are retrieving. The methods are simply general methods that code for opening a file, reading the information, closing the file and returning the necessary information. This abstracts almost all the code for extracting information to make the code very easy to read. This implementation improves maintainability because if you wanted to change the implementation of how you extracted data from a file, you would only need to do it once. This is a benefit of reusing code and cutting down on duplicated code. I also encapsulated the mapping variable to make sure it is not changed somehow (even though it's final) in another class as if it changes that would mess up the whole extracting process, improving robustness.

At first, the `Search` class stored arrays of all the page vectors, cosine similarity scores, etc, sorted them in a `TreeSet` and then took the top `X SearchResults`. I realized this was not a good design choice as it was not very OOP designed. All the information gathered, such as all the page vectors, was stored in arrays with each index representing a page. This approach was very much like the COMP 1405 search implementation, which made the search implementation quite messy and not as easy to understand and modify because everything was lumped together in one class. One's code becomes much easier to deal with when it is separated into classes that are less intertwined. Therefore, I decided to redesign the implementation of the `Search` class and divided it up into different classes; `Search`, `PageSearch` and `QueryVector`. The `QueryVector` class takes in the query string and calculates the TF-IDF values to create a query vector used to calculate cosine similarity scores later. The `PageSearch` class finds the vector for a page along with its appropriate score. The `Search` class would then create instances of these classes and call their public methods to determine the correct scores of each class to return the top `X SearchResults`. This made the `Search` class much more readable and gave it a more defined functionality by abstracting away most of the details. Another advantage of this implementation is that the classes make it easy to store information without the need for multiple arrays containing data for each page. Now, you can simply call a method from an object that will automatically store the necessary data in instance variables, adding to the idea of increased readability due to abstraction. This means you only need to store the objects, as those objects will contain all the data you need. This design improves modularity due to the encapsulated methods and variables in the classes causing a looser coupling between the classes. Therefore, it is a lot easier to modify implementations of various methods within the classes without disrupting other classes.

In the `QueryVector` class, there is a method called `findUniqueWords()`. This method parses the query string and determines all the unique words in the string along with an `ArrayList` of all words (including duplicates). To ensure the method is only run once (if it is run more than once there's the possibility of error such as the `ArrayList` of words being duplicated), I encapsulated it by making it private. This improves robustness by restricting other classes from initializing or re-initializing certain variables in the `QueryVector` class.

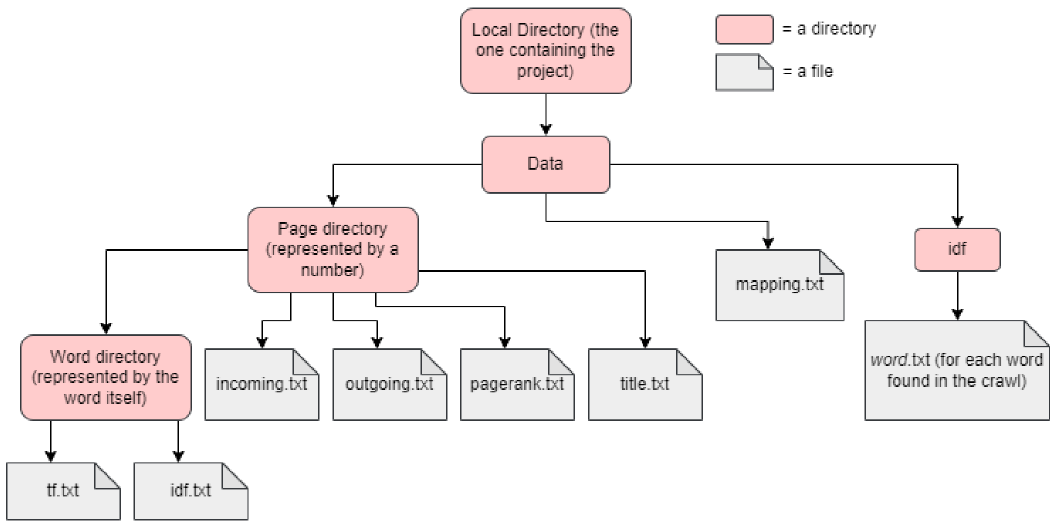
In the `Word` class, the `calcFreq(List<String> wordList, HashMap<String, Word> uniqueWords)` method's purpose is to abstract the calculations and decrease runtime complexity (refer to the runtime complexity analysis section). This is a static method because it does not belong to a certain instance of `Word`. It calculates the frequency of *all* words in a list. Polymorphism is used to make the method more generalized, improving modularity and robustness. The arguments take in a `List`, meaning that any type of list, whether it's an `ArrayList`, `LinkedList`, etc, will be able to use this method. As long as the argument is a child of `List`, the code will not break. This is ideal as if, for example, in one class we want to find the frequency of words in an `ArrayList`, but in another class, we want to use a `LinkedList`, that will not be a problem as this method is polymorphic.

During the COMP 1405 project, I noticed that I had to find the TF and TF-IDF for the words on a page, and then again for the query words. I thought this was a little redundant, but at the time I had no way of reusing code from the other file. Therefore, when I started this project, I made sure to fix this issue and made a superclass `Word` with two subclasses, `PageWord` and `QueryWord`. I figured some behaviour or instance variables that came with words on a page and words in a query would vary, but the general calculations of the TF, IDF and TF-IDF would be the same. I wanted to reduce and reuse code while increasing modularity and extensibility through inheritance. This would have worked quite well, except I realized that `PageWord` and `QueryWord` were starting to look almost identical. It turns out that the words on a page and in a query had very similar attributes and behaviours to the point where the inheritance implementation was unnecessary. Now, all words, whether they are from a page or query, are simply instances of the same class, `Word`. Fortunately, this was an easy fix because of the great use of OOP principles such as abstraction and polymorphism, making maintainability simple. For example, some of the variables were already in the form of the former superclass `Word`. But for the variables that were not already cast, the loose coupling between the classes and abstracted methods made the transition effortless because there was minimal "ripple effect" of errors across many methods and classes when I took away the `PageWord` and `QueryWord` classes. This improved `Word` class is now a generalized class for all words, which still solves the original goal of reducing and reusing code throughout the project to improve maintainability and extensibility, as you could change the implementation of methods or add new behaviour for words in the project by modifying the `Word` class only once.

For the MVC paradigm, encapsulation and abstraction are used to provide a looser coupling between the components of the MVC GUI model to make it much easier to read and modify. For example, the `update()` method in the view, updates the interface to display changes to the user. This method is called multiple times throughout the controller class, reducing duplicated code and making it very easy to understand. It also improves modularity and extensibility as you could change what you wanted to display to the user or add something new and you would only have to do that once. This increases development and productivity.

The MVC paradigm is also very nice in the way that the view and controller are almost completely separate from the model. Therefore, the model doesn't depend on the view, which makes it very easy to modify the model without changing the view. This increases maintainability and productivity, as you can work on one part of the project at a time and break everything up into pieces, making the development process a lot quicker and easier.

Description of How the Data Is Saved in Files


Local Directory (the one containing the project)
= a directory
= a file
Data
Page directory (represented by a number)
mapping.txt
idf
word.txt (for each word found in the crawl)
Word directory (represented by the word itself)
incoming.txt
outgoing.txt
pagerank.txt
title.txt
tf.txt
idf.txt

All the data found during the crawl is saved in a directory called "data". This makes it easier to delete all the crawl-related data when initializing the crawl because you just have to delete everything in the directory.

Within the "data" directory, there is a directory for each page with its corresponding file number assigned when the `Page` object was created, a directory for IDF values containing text files for each unique word found during the crawl, and a text file called "mapping.txt" that contains the file number corresponding to its URL.

Within each page directory, there is a folder for each word (in the code this would be each `Word` object on each page). There is also a text file containing the title, outgoing, incoming and page rank for that page.

Within each word directory, there is a text file for the TF and TF-IDF values of that word on the corresponding page.

You can think of each directory as an object and the content inside is its instance variables. Ex. Inside a page's directory has all its variables (title, etc) in text files as well as its word objects that also contain its own variables (TF, TF-IDF).

To save the desired data to files, I made a method that saves the necessary information for each page (in the `Page` class) and for each word (in the `Word` class). This way, in the `Crawl` class, I could easily iterate over each method for each `Page`/`Word` to save all the data. This creates a very abstract view of saving information, which makes it very easy to follow and understand. It also improves maintainability and extensibility, as you could easily change the implementation within the method or add something else to save in the method encapsulated in each class.

The abstracted methods also show us the hierarchy of the files saved. For example, in the `Crawl` class, everything that is called is being saved directly to the "data" folder. As you look into the method that is being called in the `Crawl` class, you will see more files/directories being saved. This will tell you that those files/directories (found in the same method) will all be saved in the same directory. This makes the design easy to understand and implement.

Runtime complexity analysis

In the `Crawl` class, I made an improved queue by using a `HashSet` that contained all the URLs that have ever been added to the queue. This way, when determining whether or not to add a `Page` to the queue, we can call the `HashSet`'s `contains()` function which runs in O(1) time. In contrast to a `LinkedList`'s search time which has a worse case of O(n) time, this improved queue can greatly reduce runtime complexity, especially as a large number of URLs are added to the queue.

For the queue, I made sure to use a `LinkedList` instead of an `ArrayList`. The queue will be constantly removing values from the front of the list. In an `ArrayList`, this function will take O(n) time because the `ArrayList` will need to shift every single value after the removed value to fill up the vacant spot. But in a `LinkedList`, removing a value from the front of the list will simply cause the `LinkedList` to change the location of the head, making this function O(1) time. Therefore, using a `LinkedList` as the queue instead of an `ArrayList` will greatly reduce the runtime complexity of the queue and crawl overall.

In the `Page` class, I used `HashMaps` to quickly search in O(1) time. The `wordToWord` and `urlToPage HashMaps` contain `Word` and `Page` objects, respectively, for the entire crawl (which is why they are static). The `urlToPage HashMap` is to make sure that we are not creating another object for a `Page` we already created one for in a previous page. Several `Page` instances for the same link may be more prone to errors later on while taking up more memory than necessary. As for the `wordToWord HashMap`, that one is used to keep track of all unique words throughout the entire crawl, requiring searching through a data structure to see if the program has encountered a specific word before, which is why a `HashMap` that searches in O(1) time is ideal.

In the `Page` class, the `HashSet outgoingPages` used to be a `LinkedList`, but then I noticed that in the page rank algorithm, we will need to find if a page links to a certain page. This requires searching multiple times through some form of ADT that contains all the outgoing pages. Therefore, since a `HashSet` is able to search in O(1) time while a `LinkedList` may take O(n) time, a `HashSet` would greatly reduce the runtime complexity of the page rank algorithm, especially for links with a large set of outgoing pages. This change does not affect the functionality of the crawl either because there is no need to store the outgoing pages in a specific order, nor do we need to store duplicates in `outgoingPages`.

For calculating the TF and TF-IDF values of words on a page, I made sure to calculate those values for unique words on a page instead of every word on a page, as the duplicate values would be the same, making the extra computation unnecessary. This helps reduce the runtime complexity to O(m) instead of O(n), where m can be a lot less than n with multiple duplicates on a page.

Similarly to above, I calculated the IDF values of every unique word found throughout the entire crawl instead of every time I needed to calculate the TF-IDF value of a word on a page. This way, I can simply get the correct IDF value already calculated in another object to calculate the TF-IDF value of the current word. This reduces the number of times needed to calculate an IDF value significantly, as there are much less unique words throughout the entire crawl than the number of times we need to calculate the TF-IDF value throughout the project.

In the `RetrieveInfo` class, I made a final variable called `mapping` that contains the `HashMap` that links the page URLs to their corresponding filename. This variable will be used every time we want to extract page-specific information. Therefore, it is very useful to have a final static variable that can be used anytime we need to look up the filename when given a URL. Without this variable, we would need to retrieve the mapping from the file and convert it into a `HashMap` each time. The process of converting the data into a `HashMap` is O(n), n being the amount of URLs found in the crawl. By having the `mapping` variable at hand, we only need to do this process once, reducing the time complexity of searching up a filename from O(n) to O(1), as searching in a `HashMap` is O(1) time.

At first, there was a non-static method in the `Word` class to calculate the frequency of an instance of a `Word` in a list. It would iterate through the list and count the frequency of a certain word. But, I realized that I would need to go through the list in O(n) time for each word, making it O(n^2) time when repeating the same thing for every word. To minimize runtime complexity, I made a static method where all the frequencies would be counted at once in O(n) time by adding one to each word's freq variable every time the word was present in a list. Therefore, we only need to iterate over the list of words once, instead of for every word, greatly reducing runtime complexity.

In the `Search` class, I made sure to check if `boost` was true or false only once, as its value will never change. Instead of making one method in the `PageSearch` class that calculates the cosine similarity score, checks if `boost` is true and returns the final score, I broke up the two calculations into two methods. This way, I can calculate the cosine similarity scores and then check if `boost` is true once. This reduces the runtime complexity of checking if `boost` is true or not from O(n), n being the number of pages that were found during the crawl, to O(1).

Functionality completed successfully

All.

Functionality completed unsuccessfully

None.